КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА

В.В. Зубенко, Л.Л. Омельчук

ПРОГРАМУВАННЯ.
Поглиблений курс

Рекомендовано Міністерством освіти і науки України як навчальний посібник для студентів вищих навчальних закладів

КИЇВСЬКИЙ УНІВЕРСИТЕТ ВГЦ 2002

Київ 2011

УДК 519.85(075.8)

ББК 32.973.2-018я73
З 91

Рецензенти: *М.М.Глибовець*, д-р фіз.-мат.наук, проф.,
А.Ю.Дорошенко, д-р фіз.-мат.наук, проф.,
С.О.Лук'яненко, д-р тех. наук, проф.,
О.С.Бичков, канд.фіз.-мат. наук, доцент

Схвалено Вченою радою факультету кібернетики Київського національного університету імені Тараса Шевченка (протокол № 12 від 24. 06. 2008)

Рекомендовано Міністерством освіти і науки України як навчальний посібник для студентів вищих навчальних закладів (лист № 14/ 18-Г-2030 від 29. 08. 08)

В.В. Зубенко, Л.Л. Омельчук Програмування. Поглиблений курс. – К.:Видавничо-поліграфічний центр "Київський університет", 2011. - 623 с. Бібліогр.: с. 602-609.
ISBN 978-966-439-380-2

Навчальний посібник містить поглиблений курс з основ інформатики та програмування, який у відповідності до Рекомендацій по викладанню інформатики в університетах Computing Curricula 2001-2010: Computer Science має зв'язувати в єдине ціле ідеї програмування, дискретної математики, основ архітектури ЕОМ, теорії алгоритмів та математичної логіки. Перші дві частини присвячені ознайомленню з основними логіко-алгебричними дескриптологічними засобами та уточненню на їх основі основних понять інформатики та програмування. У третій описуються мови програмування С/С++. У четвертій розглядається низка класичних алгоритмів, які складають кістяк сучасного програмування. Викладення матеріалу супроводжується великою кількістю прикладів і задач для самостійної роботи.

Для студентів та аспірантів університетів відповідних спеціальностей.

ПЕРЕДМОВА

Незважаючи на те, що дисципліна "Програмування" є однією з базових у підготовці фахівців з інформаційних технологій, сьогодні в Україні (і не тільки) фактично немає сучасного підручника в цій галузі. Відповідно до "Рекомендацій із викладання інформатики в університетах Computing Curricula 2001: Computer Science" [111] такий підручник мав би бути синтетичним і поєднувати ідеї теорії програмування, дискретної математики, основ архітектури ЕОМ, теорії алгоритмів і математичної логіки. Пропонований посібник є спробою зробити певний крок у цьому напрямі. Він містить поглиблений курс програмування з максимальним охопленням матеріалу, що відповідає програмам $CS101_B$, $CS102_B$ та CS103, які поєднують процедурне та об'єктно-орієнтоване програмування.

Метою посібника є уточнення основних понять інформатики та програмування, формування на їхній базі певного лексикону, ознайомлення з фундаментальними логіко-алгебраїчними дескриптологічними засобами й мовами програмування високого рівня, деякими класичними алгоритмами.

Посібник складається зі вступу й чотирьох розділів. Останні поділяються на логічно завершені підрозділи. У вступі пропонуються означення інформатики та програмування як наукових дисциплін. На основі введеної термінології зроблено стислий огляд історії становлення інформатики та програмування. Окремий підрозділ присвячено історії інформатики в Україні. У двох перших розділах розглянуто фундаментальні логіко-алгебричні універсалії та елементи інформатики, у третьому – мови C/C++, які є важливими інструментами процедурного та об'єктно-орієнтованого програмування, у четвертому – класичні алгоритми, що становлять кістяк сучасного програмування.

Кожний підрозіл закінчується переліком додаткової навчальної та спеціальної літератури для поглибеної роботи над відповідними темами, а також контрольними запитаннями та вправами. Символ "*" позначає вправу підвищеної складності, а пара символів "**" – вправу дослідницького характеру.

Робота з посібником вимагає ґрунтовної математичної підготовки принаймні в обсязі середньої школи, а краще – перших двох курсів ВНЗ, елементарного знайомства з обчислювальною технікою й мовами програмування. Для кращого засвоєння матеріалу бажане паралельне вивчення нормативних курсів дискретної математики й алгебри з відповідним тематичним наповненням.

Посібник побудовано за принципом замкненості – у ньому можна знайти означення всіх численних допоміжних і базових понять математики та інформатики, що використовуються. Для цього передбачено спеціальний індексний покажчик.

Наукові погляди авторів формувалися під впливом світоглядних ідей акад. В. Н. Редька в галузі програмування та програмології, без яких поява даного посібника була б неможливою.

Автори висловлюють щиру подяку всім колегам з кафедри теорії та технології програмування факультету кібернетики Київського національного університету ім. Тараса Шевченка за плідні дискусії, які дали поштовх до написання посібника, і створення відповідних умов для роботи над ним; студентам – за їхні зауваження та пропозиції; М. М. Глибовцю, А. Ю. Дорошенку, С. О. Лук'яненку й О. С. Бичкову – за наукове рецензування; А. Б. Ставровському та О. В. Кривді – за допомогу й цінні поради, що сприяли суттєвому поліпшенню змісту та якості подання матеріалу.

Автори будуть вдячні за всі зауваження й побажання, які можна надсилати за адресою: 03022, Київ-22, просп. Акад. Глушкова, 2, корпус 6, факультет кібернетики, або електронною поштою: vvz@unicyb.kiev.ua.

ВСТУП

- Предмет вивчення інформатики
- Програмування
- Електронні обчислювальні системи
- З історії інформатики
- Становлення інформатики в Україні

Ключові слова: *комунікативний процес, комунікативна система, запит, дескриптивна система, екстенсіональні, інтенсіональні й мішані дескриптивні системи, інформаційна система, інформатика як наука, програмування, мова програмування, дані, програма, ЕОМ, програмне забезпечення, електронна обчислювальна система, інформаційні технології, парадигма програмування, програмна інженерія, теорія програмування, програмологія.*

0.1. Предмет вивчення інформатики

Незважаючи на стрімкий розвиток інформатики протягом останніх кількох десятиріч, процес її самовизначення як науки все ще не можна вважати завершеним. Перебігає він досить суперечливо між кількома напрямами – інженерним, математичним, комунікативним тощо. Кожен із них має своє підґрунтя й зорієнтований на ті чи інші аспекти процесів обробки інформації.

Інженерний підхід розглядає інформатику як науку про комп'ютерні системи. Такий погляд домінував у період становлення інформатики, коли проектувались і створювались перші потужні комп'ютерні системи. Наприклад, відомі американські вчені А. Ньюел, А. Перліс і Г. Саймон проголосили предметом вивчення інформатики обчислювальні машини. Сьогодні даний підхід охоплює комплекс технологічних проблем, пов'язаних із проектуванням, розробкою й технічною експлуатацією обчислювальних систем, і має

загальну назву *програмна інженерія*. У *комунікативному підході* першорядними є проблеми взаємодії між суб'єктами всередині комунікативних процесів.

Математичний підхід виникнув у останні десятиріччя, коли розпочалося тотальне проникнення інформаційних технологій у всі сфери життя суспільства та постала проблема різкого підвищення продуктивності й надійності праці в галузі продукування інформаційних технологій, а також суттєвого зниження їхньої вартості. Фактично вже йдеться про запровадження індустріальних методів не тільки в масове виробництво комп'ютерів, а й у виробництво програмного забезпечення для них. Подібне виробництво, з його всебічною автоматизацією, вимагає ґрунтовної математичної підтримки.

Наразі інформатику обслуговують багато розділів математики й математичної логіки, семіотики й математичної лінгвістики, і така тенденція буде зберігатись у майбутньому. Однак поряд із процесом обробки та засвоєння результатів суміжних дисциплін в інформатиці відбувається активний пошук і формування її власних основ. Створення такого фундаменту є однією з найактуальніших задач сучасної інформатики.

Що вивчає інформатика. На теренах СРСР під терміном *інформатика* розуміли спочатку так звану "бібліотечну" інформатику – спеціальну наукову дисципліну, яка вивчає структуру й загальні властивості наукової інформації, а також закономірності всіх процесів наукової комунікації – від неформальних процесів обміну такою інформацією до формальних за допомогою наукової літератури, а сьогоднішню її проблематику вважали належною до кібернетики як науки про керування складними системами. Д. Кнут у 1974 р. визначав інформатику як *науку, що займається вивченням алгоритмів*. Академік АН СРСР А. Єршов вважав інформатику *фундаментальною природничою наукою, яка вивчає процеси передавання й обробки інформації*. Французькі спеціалісти Б. Майер і К. Бодуен наводять два варіанти означення інформатики: перший – як *комп'ютерної науки*, другий – як *теорії обробки інформації*. Відомі німецькі вчені Ф. Бауер і Г. Гооз посилаються на означення Французькою академією інформатики як *науки про здійснювану за допомогою автоматичних засобів цілеспрямовану обробку інформації, що розглядається як подання знань і повідомлень у технічних, економічних і соціальних галузях*. Близьким за змістом є ухвалене на сесії річних зборів АН СРСР у 1983 р. визначення інформатики як *комплексної наукової та інженерної дисципліни, що вивчає всі аспекти розробки, проектування, створення, оцінки, функціонування заснованих на ЕОМ систем*

переробки інформації, їхнє застосування та дії на різні галузі соціальної практики. У сучасних джерелах основні напрями інформатики пов'язують із розробкою спеціальних комп'ютерних методів розв'язання складних дослідницьких і практичних задач у різних галузях, у тому числі й гуманітарних, з розвитком інформаційних технологій і соціально-комунікативних процесів (роботи І.В. Сергієнка, В.С. Михалевича, Ю.М. Канигіна, В.І. Гриценка, А.В. Анісімова, С.В. Симовича та ін.). Таке різноманіття в підході до визначення предмета інформатики актуалізує проблему її самоідентифікації як наукової дисципліни.

Усі вищезгадані підходи можна було б спробувати об'єднати на основі поняття моделі, виходячи з того, що будь-яка точна наука має справу з певними моделями природних чи суспільних явищ. Наприклад, фізика вивчає фізичні моделі природних явищ, хімія – хімічні, кібернетика – моделі систем керування тощо. Цілком резонно постає питання: а які моделі мали б цікавити інформатику?

Усі означення інформатики тим чи іншим чином апелюють до процесів обробки інформації та систем, що їх реалізують. У подібних процесах обов'язковою є наявність (рис. 0.1):

1) певної сукупності інформаційних об'єктів (повідомлень та інформації, яку вони містять);

2) суб'єкта, що формує, передає та приймає певну вхідну й вихідну інформацію (суб'єкта-ініціатора);

3) суб'єкта, що приймає вхідну інформацію, за допогою певної внутрішньої процедури перетворює її та повертає оброблену вихідну інформацію (суб'єкт-обробник).

Подібні процеси й системи, що їх реалізують, називаються *комунікативними*[1]. Кожний такий процес розгортається в певному часовому просторі та складається з *етапів*. Етапи утворюють життєвий цикл процесу. Комунікативний процес має *предметну область*, яку складають первинні інформаційні об'єкти й певні співвідношення між ними. Розпочинається такий процес суб'єктом-ініціатором, який формує й передає спеціальними засобами (*засобами зв'язку*) суб'єкту-обробнику повідомлення, що містить *запит* на обробку певної вхідної інформації з предметної області. Запит, поряд із вхідною інформацією, містить інформацію про мету обробки. Мета може бути сформульована або неявно – як вимоги до вихідної інформації, або явно – як детальний опис процесу обробки.

[1] У кібернетиці подібні системи називають *системами зі зворотним зв'язком.*

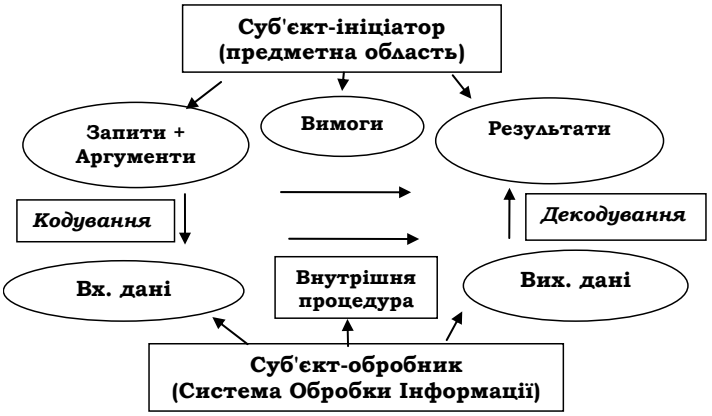


Рис. 0.1

Запит оформлюється у вигляді певного набору *дескрипцій*[2], зрозумілих суб'єкту-обробнику (напр., як сукупність алгоритмів чи програм). Останній приймає запит, виконує в межах певного часового інтервалу відповідну внутрішню процедуру для реалізації мети й повертає суб'єкту-ініціатору за допомогою засобів зв'язку оброблену інформацію. Зазвичай суб'єкт-обробник має власну систему інформаційних об'єктів, відмінну від предметної області, яка в кожний момент часу перебуває в певному стані. Тому сам запит містить не вхідні об'єкти обробника, а їхні прообрази, які після кодування утворюють вхідні дані обробника. Це стосується й вихідної інформації, поданої в заключному стані процесу обробки. Вона потребує декодування, щоб отримати дійсно результати запиту.

Комунікативні системи теж мають свій життєвий цикл, що складається з певних етапів. Він розпочинається з початкового аналізу вхідної інформації та специфікації запиту і включає такі етапи, як проектування й побудову системи, обґрунтування її коректності, підготовку документації та, нарешті, безпосередньо експлуатацію системи.

На підставі сказаного можна було б у цілому визначити *інформатику як науку, що вивчає моделі комунікативних процесів і систем.* Однак, зважаючи на дискусійний характер обговорення, зробимо кілька зауважень щодо даного означення. Насамперед зазначимо, що в такому вигляді воно здається досить загальним і багатоаспектним. Моделі комунікативних процесів і систем можуть дуже різнитися й вимагати принципово різних підходів. Достатньо послатися на різноманіття кі-

[2] Дескрипція (від англ. description – опис) – мовна конструкція, що замінює назву предмета, виражаючи його зміст іншими мовними засобами.

бернетичних моделей систем (від автоматичного регулювання, імовірнісних і статистичних моделей до нейрофізіологічних моделей керування тощо). Тому було б доречно якимось чином обмежити їх.

Зауважимо, що інформація в комунікативній системі існує не сама по собі, а передається й отримується суб'єктами комунікативного процесу у вигляді певних дескрипцій. Це означає, що комунікативні процеси й системи за своєю суттю не можуть вивчатися поза межами їхніх дескриптивних засобів. Здається, що в центрі уваги інформатики мають бути не просто моделі комунікативних процесів і систем, а саме дескриптивні моделі. *Дескриптивність* моделі означає, що всі її інформаційні елементи можуть бути подані (описані) у межах певної дескриптивної системи (ДС).

ДС, у свою чергу, розподіляються на три класи – екстенсіональні, інтенсіональні й мішані. *Інтенсіональні* використовують індивідуальні засоби – вони або прямо описують конкретну будову об'єкта (його сутність), або параметризований кістяк такої будови, який шляхом конкретизації (означення) параметрів перетворюється на об'єкт. Для *екстенсіональних* ДС характерні предикативні (об'ємні) засоби опису об'єктів і співвідношень. Різницю між екстенсіональними та інтенсіональними засобами опису яскраво ілюструють натуральні числа. З одного боку, їх можна визначати як кардинальні числа через потужність множин (екстенсіональний підхід), а з іншого – індуктивно, за допомогою числа 0 і відповідної сукупності операцій збільшення на 1 (інтенсіональний підхід). Наприклад, число 3 інтенсіонально подається як $(((0+1)+1)+1)$ тощо. Можна сказати, що при екстенсіональному підході здійснюється перехід на вищий рівень абстракції, при якому нівелюються індивідуальні риси конкретних об'єктів.

Мішані дескриптивні моделі систем можуть використовувати обидва засоби. Моделі систем будемо називати *екстенсіональними, інтенсіональними* або *мішаними,* якщо вони описуються відповідними ДС. Щоб підкреслити вагомість інтенсіональних елементів у мішаних моделях комунікативних систем, останні теж будемо називати інтенсіональними за умови, що суб'єкт-обробник у них описується інтенсіонально, а для кожного запиту (навіть якщо він екстенсіональний) існує своя внутрішня процедура в моделі Виконавця, яка його реалізує (як у системах обробки інформації на базі обчислювальних машин). Виходячи з вищезазначеного, запропонуємо таке визначення предмета інформатики:

Інформатика – це наука, що вивчає інтенсіональні моделі комунікативних процесів і систем.

За аналогією з математичними, фізичними та іншими моделями, моделі, що розглядаються в інформатиці, логічно було б назвати *інформа-*

тичними або, що більш звично – *інформаційними.* З урахуванням інтенсіональності будемо говорити й про *інформаційні комунікативні процеси* та *системи* й називати їх просто *інформаційними.* Інтенсіональність інформаційних моделей є їхньою принциповою рисою. Вона фіксує границі інформатики та вберігає її від надмірного узагальнення й ототожнення з іншими науками[3], особливо з кібернетикою як наукою про загальні закони перетворення інформації в комунікативних системах та інформологією – узагальнюючою наукою про інформацію в цілому, усі її вияви, властивості й види інформаційних процесів.

Взаємовідносини між кібернетикою та інформологією, з одного боку, та інформатикою – з іншого, приблизно такі, як між загальним поняттям і його конкретизацією. Як приклад можна навести взаємовідносини між загальною теорією груп і теорією скінченновизначених груп, між геометрією та скінченною геометрією тощо. Таким чином, не можна вважати, що кібернетика просто поглинається інформатикою, чи намагатися звести інформатику до певної технічної галузі (програмної інженерії), яка тільки обслуговує кібернетику. Не йдеться й про ігнорування інформатикою взагалі екстенсіональних дескриптологічних засобів. Навпаки, вони можуть (і повинні!) широко використовуватись як допоміжні, наприклад при описі предметних областей, формулюванні запитів тощо. Однак подібні дескрипції не є основою інформаційних систем. Водночас вони можуть бути центральними в тих самих кібернетичних моделях.

0.2. Програмування

Розглянемо тепер одне з головних понять інформатики – програмування. Зазвичай при визначенні програмування апелюють до тих чи інших його аспектів. Найчастіше його трактують як процес написання (конструювання) або побудови програм для розв'язання певної задачі за допомогою ЕОМ. Останній варіант хоч і більш вдалий, але теж не може вважатися цілком задовільним, оскільки потребує, у свою чергу, з'ясування, що таке задача тощо. Змістовнішим виглядає таке означення:

Програмування – це процес реалізації життєвого циклу інформаційних систем.

[3] У зв'язку з цим знову доречно нагадати про алгоритми як об'єкт вивчення інформатики за Д. Кнутом.

Отже, програмування – це не просто написання окремої програми чи сукупності програм, а цілеспрямований процес побудови інформаційних систем (їхніх апаратних і програмних засобів), що реалізують той чи інший клас комунікативних процесів із подальшою підтримкою їхньої життєздатності.

Різноманіття поглядів на предмет інформатики знайшло своє відображення й у виявленні найбільш значущих аспектів програмування. По-перше, це *виробничий* аспект, пов'язаний із програмуванням як специфічним ремеслом (мистецтвом) продукування інформаційних систем. У цьому ремеслі застосовується багато різних прийомів і засобів. Вивченням, систематизацією й узагальненням їх займається *методологія* програмування. З погляду методології програмування поділяється на певні напрями (парадигми). *Парадигма програмування* – це сукупність ідей і понять, що визначають певний стиль програмування. Розрізняють процедурне, функціональне, логічне, об'єктно-орієнтоване, агентне програмування тощо. Кожна парадигма орієнтується на певні моделі програм і даних, відповідні засоби їхнього опису й методи реалізації. Другий аспект – *науковий.* Теорія програмування пов'язана з вивченням програмування в загальному контексті інформатики як наукової дисципліни. У даному посібнику зроблено акцент на *програмологію* – напрям теорії програмування, що вивчає інформаційні системи та програмування в контексті їхніх дескриптологічних основ.

Третій аспект програмування – *користувацький,* пов'язаний із застосуванням інформаційних технологій і проблемами, які при цьому виникають.

З урахуванням дескриптологічного аспекту інформаційних систем надзвичайно важлива роль у програмуванні належить дескриптивним засобам. Серед них одне із чільних місць посідають мови програмування.

Мова програмування – це спеціальна дескриптивна система, призначена для формування запитів у інформаційних системах та їхніх моделях, а також для опису внутрішніх процедур суб'єктів-обробників таких систем[4].

Мови програмування можуть бути як практичними, так і суто теоретичними. Перші використовуються в реальних інформаційних системах, другі – в їхніх теоретичних моделях.

Центральними поняттями мов програмування є дані та програми. *Даними* називаються мовні конструкції, які зображують інформаційні об'єкти, а *програмами* – дескрипції, що подають запити на обробку даних і

[4] Як неодноразово зазначав В.Н. Редько, цей термін не є вдалим. Точнішим був би термін *мова програм,* а справжня мова програмування мала б підтримувати сам процес програмування в усій його повноті.

алгоритми. Розробка та реалізація мов програмування, поряд із підготовкою й виробництвом самих суб'єктів-обробників (людських, апаратних тощо), є важливою необхідною складовою індустрії інформатики.

0.3. Електронні обчислювальні системи

Революційним проривом у розвитку інформатики (індустрії, науки) стало створення в середині XX ст. перших електронно-обчислювальних машин (ЕОМ) і на їхній основі – інформаційних систем із суб'єктами-обробниками – електронними обчислювальними системами (ОбС). Поява вже перших, недосконалих ЕОМ продемонструвала широкі можливості нових інформаційних систем і дозволила значно розширити та прискорити розв'язок задач у різних сферах, насамперед у галузі обчислювальної математики. *ЕОМ,* або *комп'ютер,* – це реалізація автомата з програмним керуванням (АПК) на електронній елементній базі. На загальному рівні АПК складається з процесора, пам'яті та пристроїв введення-виведення інформації. Функціонування АПК зводиться до автоматичного виконання процесором послідовностей машинних команд (програм), що зберігаються в його пам'яті. У свою чергу, обчислювальна система складається з ЕОМ (мережі ЕОМ) і програмного забезпечення (ПЗ). Якщо обчислювальна система базується на мережі ЕОМ, то вона називається *розподіленою.*

ОбС = ЕОМ (мережа ЕОМ) + ПЗ.

Програмне забезпечення поділяється на системне й прикладне. Основою системного програмного забезпечення є *операційна система* – комплекс програм, призначених для автоматизованого керування ресурсами комп'ютера й автоматизації процесу програмування. Прикладне програмне забезпечення застосовується для реалізації запитів інформаційних систем на базі даної ОбС.

Для створення інформаційної системи на базі ОбС необхідно розв'язати три задачі: 1) визначити апаратуру, зокрема засоби зв'язку для забезпечення комунікації між суб'єктами системи; 2) вибрати системне програмне забезпечення; 3) створити (або адаптувати до умов задачі вже існуючу) прикладну частину програмного забезпечення. Саме розв'язок третьої задачі спричиняє найбільше проблем при програмуванні інформаційних систем.

Обробка інформації в інформаційних системах на базі сучасних ОбС отримала назву *інформаційних технологій* (*ІТ*). ІТ охоплює весь комплекс робіт з інформацією в інформаційних системах – від фор-

мування й кодування запиту, передавання його лініями зв'язку обчислювальній системі до реалізації запиту й повернення, декодування й осмислення його результатів.

ІТ = обробка інформації в інформаційних системах на базі ОбС.

Сучасний соціум уже немислимий без застосування ІТ. Не безпідставно вважається, що ІТ значною мірою визначають не тільки рівень, але й напрям його розвитку [5].

0.4. З історії інформатики

Будь-яка нова наукова дисципліна, відповідаючи на певні актуальні виклики сучасності, не виникає на порожньому місці. Вона на щось спирається, має певні (якщо не прямі, то опосередковані) історичні витоки. У цьому сенсі інформатиці поталанило. Незважаючи на свій юний вік (а сучасна інформатика, як уже наголошувалось, набула свого бурхливого розвитку порівняно недавно – у другій половині XX ст.), її дескриптологічні корені простягаються далеко в минуле, коли вперше виникла ідея формалізації розумової діяльності людини. Перший і тому, можливо, найважливіший крок у цьому напрямі зробив Арістотель (384–322 рр. до н. е.) у теорії силогізмів. Значно пізніше, тільки через півтори тисячі років, було зроблено наступний крок і висунуто загальну ідею машинізації логічних умовиводів. Вона належала іспанському логіку Р. Луллієму (1235–1315), який поставив задачу на основі арістотелевої логіки розробити універсальний метод пізнання й механізувати його за допомогою спеціальної машини, яка б моделювала логічні умовиводи. Хоча Луллієму не вдалося до кінця реалізувати свій задум і побудувати таку машину, сама спроба механізації моделі стала піонерською в галузі створення штучних суб'єктів-обробників інформації.

Наступний важливий крок зробив Г. Лейбніц (1646–1716), який намагався створити універсальну дескриптивну платформу для всіх наук – прообраз сучасних формальних систем числення. Він першим зрозумів роль двійкової системи числення в механізації та організації обчислень. Ним же був розроблений (і частково реалізований) проект механічної обчислювальної машини, оснований на двійковій арифметиці.

[5] За образним висловом американського вченого Дж. Вейценбаума, комп'ютер зі знаряддя праці людини поступово перетворюється на її розумовий протез, без якого вона все більше втрачатиме свою дієздатність. І ця всезростаюча залежність стосується не тільки окремої особи, а й суспільства в цілому.

Саме із числових обчислень розпочалась ера механізації й автоматизації інформаційних систем. Протягом XVII–XIX ст. з'явилася ціла низка арифмометрів і калькуляторів для механічної обробки числової інформації. Основи такої обробки базувались на винайдених ще в Стародавній Індії позиційних системах числення й правилах виконання в них чотирьох основних арифметичних дій. Ці правила набули поширення в Європі приблизно в 820–825 рр. завдяки трактату хорезмського математика й астронома аль Хорезмі. Звідси й походять такі словосполучення, як "алгоритм додавання", "алгоритм множення" тощо. Пізніше термін "алгоритм" став застосовуватись у ширшому сенсі, означаючи будь-яке правило для обробки інформації, у тому числі й символьної.

Наприкінці XIX та на початку XX ст. ідеї Лейбніца про універсальну платформу знайшли свій подальший розвиток у формалізації класичної математики, яка завершилася створенням прикладного числення предикатів (ПЧП). ПЧП – це мішана дескриптологічна система, побудована з метою формального уточнення й дослідження таких фундаментальних понять, як математичне твердження, його доведення, і як універсальний інструмент для опису математичних об'єктів і роботи з ними. До появи ПЧП доклали зусилля багато математиків. Зазначимо серед них Дж. Буля (1815–1864) – числення висловлювань, булева алгебра; Э. Шредера (1841–1902) – "Лекції з алгебри логіки"; Дж. Пеано (1858–1932) – аксіоматика арифметики; Г. Фреге (1848–1925) – аналіз первинних математичних понять, основи арифметики. У сучасному вигляді ПЧП описане у "Принципах математики" Б. Рассела (1872–1970) та А. Уайтхеда (1861–1947). Книга була надрукована у 1910–1913 рр.

Принциповим кроком на шляху до інформатики було створення в надрах математичної логіки й основ математики спеціальних інтенсіональних ДС, призначених для уточнення й вивчення загальних властивостей інтуїтивного для цього поняття алгоритму та обчислюваності. Серед таких систем були λ-числення А. Чорча (1936), машини Тьюрінга (1936), алгоритми Поста (1936) тощо. Згодом було доведено, що в певному сенсі всі ці моделі алгоритмів еквівалентні. Уточнення поняття алгоритму дозволило виділити клас алгоритмічно розв'язних задач. Для багатьох задач була доведена їхня алгоритмічна нерозв'язність. Перший приклад такої задачі навів А. Чорч (1903–1995), який довів нерозв'язність чистого ПЧП (без символів операцій і констант). Модель Тьюрінга мала суттєву перевагу над іншими – вона припускала природну машинну інтерпретацію. Використовуючи її, Тьюрінг теоретично довів існування універсального суб'єкта-обробника для інформаційних систем, потенційно здатного реалізувати запити будь-якої інформаційної системи. Цей обробник отримав назву *універсального АПК.* Конце-

пція універсального АПК та її реалізація у вигляді ЕОМ відіграли вирішальну роль у подальшому розвитку інформатики.

Символьні (нечислові) маніпуляції з інформацією були також відомі дуже давно й пов'язані з виникненням природних мов, тайнописом і шифруванням текстів. Виникла ціла наука – *криптографія*. Її батьком вважається Л. Альберті (1404–1472), який написав першу книгу з криптографії. Сьогодні без застосування методів криптографії та криптографічних систем немислима жодна серйозна інформаційна система. Іншим важливим елементом комунікативних систем є *системи зв'язку*, призначені для кодування й декодування інформації та передавання її на відстані. Вони теж мають свою багату історію.

Узагалі кажучи, сама ідея АПК належала не Тьюрінгу, а Ч. Беббіджу (1791–1881), який розробив проект Аналітичної машини – певний механічний варіант АПК. На жаль, через свою складність він не був реалізований. Тільки майже через сторіччя з'явилися перші успішні спроби створення діючих версій АПК (машина Z3 К. Цузе в 1941 р. та ін.), але вони вже були не механічними, а на електромеханічній основі. Першою діючою ЕОМ вважається машина, побудована в проекті ENIAK під керівництвом Д. Мочлі та П. Еккерта в 1946 р. у Принстонському університеті (США). Вона використовувала 18000 електроламп, виконувала біля 3000 оп./с і керувалася програмою, команди якої встановлювались за допомогою механічних перемикачів. Таке введення програми обмежувало можливості автоматизації обчислень, тому в наступному проекті цих вчених – EDWAK (1951) – уже передбачалось зберігання команд програми разом із даними безпосередньо в оперативній пам'яті. Принцип побудови подібних ЕОМ отримав назву *неймановського* – за прізвищем відомого математика Дж. фон Неймана (1903–1957), який у 1946 р. разом із Г. Голдстайном і А. Берксом у спеціальному звіті узагальнив набутий на той момент досвід розробки ЕОМ.

Однак історія появи першої ЕОМ на цьому не закінчилась, а навпаки, почала обростати із часом новими подробицями, іноді досить цікавими. Це викликано тим, що роботи зі створення перших зразків комп'ютерної техніки проводились одночасно в різних країнах (Британія, Німеччина, США, СРСР) напередодні й під час Другої світової війни (звісно, у режимах секретності). Тому інформація про них у науковій літературі протягом тривалого часу була обмежена. Матеріали про вже згаданий звіт фон Неймана, Голдстайна та Беркса з'явилися в пресі тільки в 50-х рр. XX ст.[6]. Нова сторінка історії знов пов'язана

[6] Така сама ситуація мала місце і в СРСР. Перша в країні монографія з елементами теорії ЕОМ і програмування [113] мала гриф секретності й видавалася в бібліотеках лише за наданням посвідчення про допуск до державних секретів.

з діяльністю Тьюрінга, який на початку 40-х рр. XX ст. очолював групу зі створення у Великобританії першої у світі спеціалізованої ЕОМ "Колос". Робота була успішно завершена в 1942 р., але, оскільки дана ЕОМ створювалась для британської розвідки, то результати її залишалися невідомими до 1975 р. Цікаво, що в самій Британії першою вітчизняною ЕОМ вважають машину EDSAC (М. Уілкс, 1949) – першу ЕОМ, в якій команди вже зберігались в оперативній пам'яті. Ще цікавіше, що Тьюрінг був причетний і до створення ЕОМ ENIAK (він працював у США на завершальних етапах її розробки та запуску).

0.5. Становлення інформатики в Україні

В Україні перша ЕОМ – МЭСМ (рос. – Малая Электронно-Счетная Машина) була створена в 1951 р. у Києві в Інституті електротехніки АН УРСР під керівництвом майбутнього академіка АН СРСР С.О. Лебедєва (1902–1974). Вона стала першою діючою ЕОМ, побудованою на теренах колишнього СРСР і континентальної Європи. Щоб вірно оцінити масштаб і значення цієї події, її необхідно розглядати не ізольовано, а на тлі розвитку всієї тогочасної радянської та світової науки. МЭСМ – це здобуток не тільки української науки, яким ми за правом сьогодні пишаємося, а й усієї радянської, правонаступниками якої є й українські вчені. Поява МЭСМ була логічним наслідком великого інтересу з боку тогочасного радянського керівництва до розвитку електронно-обчислювальної техніки в країні, насамперед в оборонних цілях. На ці роботи виділялись великі кошти. Значним імпульсом для розвитку теоретичних засад нового наукового напряму стала поява у 1948 р. книги Н. Вінера "Кібернетика". Справедливо було б зазначити, що ці процеси розбудови нової науки не уникли й відомих протиріч. Апофеозом їх стало офіційне проголошення певними колами партійного керівництва в СРСР у першій половині 50-х рр. XX ст. кібернетики буржуазною лженаукою. Однак, на щастя, це не мало таких згубних наслідків для неї, як для інших наук (генетики тощо). Одночасно з роботами в Києві аналогічні проекти (у значно більших масштабах) велись відразу в кількох наукових закладах держави. Свій проект зі створення ЕОМ С.О. Лебедєв висунув ще до війни, яка завадила його здійсненню. Початок робіт над МЭСМ датовано осінню 1948 р. Спочатку машина планувалась як експериментальна модель для іншої машини – БЭСМ (рос. – Большая Электронно-Счетная Машина), але в процесі роботи над БЭСМ було прийнято рішення про побудову спочатку малої діючої ЕОМ. Для цього в модель були інтег-

ровані пристрої введення-виведення інформації й деякі інші елементи. 25 грудня 1951 р. Державна комісія АН СРСР прийняла МЭСМ до експлуатації. Незважаючи на те, що пам'ять машини складали лише кілька десятків машинних слів, вона дозволила розв'язати протягом наступних кількох років багато важливих науково-технічних задач. Виступаючи на урочистому засіданні, присвяченому 25-річчю створення МЭСМ, академік АН СРСР В. Глушков, високо оцінюючи значення МЭСМ для розвитку обчислювальної техніки в Україні і в СРСР, зокрема, зазначив: "Незалежно від закордонних вчених, С.О. Лебедєв розробив принципи побудови ЕОМ. Під його керівництвом була створена перша в континентальній Європі ЕОМ, за її допомогою в стислі терміни були розв'язані важливі науково-технічні задачі й започаткована радянська школа програмування. Опис МЭСМ став першим підручником у країні з обчислювальної техніки. МЭСМ стала прототипом БЭСМ" [71]. На початку 1952 р. Лебедєв був переведений до Москви для закінчення роботи над БЭСМ. Улітку 1952 р. машина була в основному готова, розпочались її налагодження й випробування. У першому кварталі 1953 р. БЭСМ була прийнята до експлуатації.

Створення МЭСМ надало потужного поштовху розвитку кібернетики в Україні. Навколо неї згуртувалась ціла плеяда майбутніх відомих вчених-кібернетиків. У багатьох установах запрацювали наукові семінари з кібернетики. В Інституті математики АН УРСР семінар очолив акад. АН УРСР Б. Гнеденко (1912–1995). У Київському університеті ім. Т.Г. Шевченка із 1957 р. запрацював аналогічний семінар під керівництвом проф. Л. Калужніна (1914–1990). У Київському політехнічному інституті, розпочинаючи із середини 50-х рр., ідеї проектування й побудови елементів обчислювальної техніки активно просував чл.-кор. НАН України К. Самофалов (1921). Приблизно в цей самий період у Київському вищому інженерному радіотехнічному училищі (КВІРТУ) розпочали підготовку до випуску військових інженерних кадрів для роботи з автоматизованими системами керування. Активну участь у цій підготовці брали В. Глушков, чл.-кор. НАН України Є. Ющенко (1919–2001), проф. Є. Вавілов (1923–1979).

Створений у 1957 р. на базі колишньої лабораторії С.О. Лебедєва Обчислювальний центр АН України перетворюється у 1961 р. на Інститут кібернетики і носить сьогодні ім'я свого засновника – академіка В.М. Глушкова (1923–1982). Під керівництвом Глушкова ця установа стала одним із ведучих наукових центрів зі створення й упровадження ІТ не тільки в Україні, а й у Радянському Союзі.

Серед перших фундаментальних результатів вітчизняної кібернетики та обчислювальної техніки слід зазначити створення й реалізацію в 1955–1956 рр. адресної мови програмування (В. Королюк,

Є. Ющенко), появу граф-схем алгоритмів (Л. Калужнін)[7] і систем мікропрограмних алгебр (В. Глушков), які стали помітним явищем не тільки в радянській, а й у світовій науці. Адресна мова посіла місце одного з перших прообразів мов програмування високого рівня поряд із мовою Фортран 0 (1957) і першими автокодами. Деякі її ідеї набагато випередили свій час, наприклад багаторангове адресування (Алгол-68). Граф-схеми алгоритмів заклали основу графічних дескриптологічних засобів, націлених на розробку й проектування інформаційних систем. Сьогодні цей апарат отримав друге дихання у зв'язку із процесами тотальної візуалізації програмування. Він невпинно розповсюджується й на інші етапи життєвого циклу систем. Мікропрограмні алгебри Глушкова стали основою теорії дискретних перетворювачів, на якій, у свою чергу, базуються автоматизовані системи проектування обчислювальних комплексів. У 1962 р. В. М. Глушков став лауреатом Ленінської премії в галузі науки й техніки за монографію "Синтез цифрових автоматів".

Протягом 1961–1969 рр. у Інституті кібернетики були створені й передані в серійне виробництво такі ЕОМ, як "Дніпро" (1961) – перша в Радянському Союзі керуюча ЕОМ на напівпровідниках (перша подібна ЕОМ у США – RW300 – з'явилася також у 1961 р.), "Промінь" (1963), "Мир-1" (1965), "Мир-2" (1969), "Мир-3" (1969). Ці розробки не поступалися кращим світовим зразкам того часу у своєму класі, а за деякими архітектурними рішеннями навіть і випереджали їх. Колективом інституту була підготовлена до друку й видана у 1973 р. двома мовами – російською й українською – "Енциклопедія кібернетики", що містила біля 1800 статей з усіх галузей кібернетики й обчислювальної техніки. До роботи над книгою були залучені провідні вчені та спеціалісти зі 102 установ і організацій СРСР.

Наукова та практична діяльність В. Глушкова в галузі ІТ здобула й міжнародне визнання. Він був почесним членом кількох іноземних академій наук, відзначений багатьма престижними вітчизняними й закордонними нагородами. У 1997 р. Міжнародне комп'ютерне товариство (IEEE Computer Society) посмертно нагородило його медаллю "Піонер комп'ютерної техніки".

В.М. Глушкову належить видатна роль у зародженні та становленні інформатики в країні, вичленення її з надр кібернетики. Він був ініціатором створення на початку 80-х рр. у АН СРСР Відділення інформатики, обчислювальної техніки й автоматизації.

[7] Останні відомі сьогодні як граф-схеми Калужніна. Сам Л.А. Калужнін працював на механіко-математичному факультеті КДУ ім. Т.Г. Шевченка.

В. Глушков зробив значний внесок у розвиток інформатичної освіти в Україні. За його ініціативи, підтриманої академіком НАН України І.І. Ляшком (1921–2008), у Київському державному університеті ім. Т.Г. Шевченка у 1965 р. на механіко-математичному факультеті було створено кафедру теоретичної кібернетики (яку він очолював за сумісництвом до дня своєї смерті), а у 1969 р. відкрито факультет кібернетики – один із перших такого профілю в Радянському Союзі. Першим деканом факультету став І.І. Ляшко, який доклав багато зусиль для його становлення й розвитку. Сьогодні факультет кібернетики Київського національного університету ім. Тараса Шевченка є визнаним осередком передової наукової думки й активного пошуку в галузі інформатики [99, 130].

Роль В.М. Глушкова не обмежувалась лише організаційними ініціативами й заходами. Він значною мірою впливав на зміст навчально-методичної роботи, робочих програм і навчальних планів з інформатики. Велика увага приділялась фундаментальній математичній складовій у підготовці майбутніх молодих спеціалістів-кібернетиків – математичному аналізу, алгебрі, теорії автоматів, теорії алгоритмів і математичній логіці. Ця математична традиція продовжується на факультеті й сьогодні. Такий підхід знаходить усе більше розуміння й підтримки й у світовому науковому співтоваристві, про що свідчить заключний звіт спеціальної об'єднаної комісії ACM та IEEE Computer Science, який містить рекомендації з викладання програмної інженерії та інформатики в університетах [111].

Якщо звернутись до хронології, то перший в Україні Обчислювальний центр було створено у 1956 р. при Київському державному університеті ім. Т.Г. Шевченка на базі лабораторії електричного моделювання. Ця лабораторія була організована ще у 1945 р. на механіко-математичному факультеті за ініціативи чл.-кор. АН УРСР В.Є. Дьяченка. У ній у 50-х рр. XX ст. були розроблені й виготовлені унікальні сіткові електрогенератори й аналогові пристрої, за допомогою яких розв'язувалися складні обчислювальні задачі для потреб народного господарства. У 1957 р. на механіко-математичному факультеті за ініціативи П.С. Бондаренка було створено кафедру обчислювальної математики, яку тривалий час очолював чл.-кор. АН УРСР Г.М. Положій (1914–1968). Із 1956 р. у Київському політехнічному інституті (КПІ) почали готувати на окремих кафедрах спеціалістів з обчислювальної математики й обчислювальної техніки. У 1960 р. за ініціативи К.Г. Самофалова у КПІ була відкрита перша в країні кафедра обчислювальної техніки, яка з 1969 р. почала готувати інженерів-математиків зі спеціальності "0647 – прикладна математика". На початку 60-х рр. XX ст. у КВІРТУ було відкрито кафедри військової кібернетики й обчислювальної техніки та автоматизованих систем ке-

рування. Великий внесок у становлення інформатичної освіти в Україні зробили І.І. Ляшко, Б.В. Гнеденко та його учні – акад. НАН України В.С. Королюк (р. н. 1925) і чл.-кор. НАН України Є.Л. Ющенко (1919–2001), чл.-кор. НАН України К.Г. Самофалов, професори Л.А. Калужнін, Є.М. Вавілов та інші науковці й спеціалісти. Вони були авторами перших вітчизняних підручників із програмування та обчислювальної техніки [25, 51, 77, 137].

***Література для СР[8]**: предмет вивчення інформатики – [9, 37, 48, 61, 75, 86, 87, 115, 117, 147]; кібернетика та інформологія – [26, 36, 87, 116, 134]; різні підходи до визначення натуральних чисел – [121, 131]; теорія програмування і програмологія – [69, 91, 93, 106-109]; захист інформацій у інформаційних системах [6, 38]; історія інформатики та обчислювальної техніки – [9, 48, 78, 101, 116, 134, 153].

Контрольні запитання та вправи

1. Дати власні тлумачення термінів: "інформація", "обробка інформації", "запит", "інформатика".
2. Що таке комунікативна система?
3. Що таке комунікативний процес?
4. Що таке інтенсіональна та екстенсіональна ДС?
5. Дати інтенсіональне та екстенсіональне визначення: а) натуральних чисел; б) слів у довільному алфавіті.
6. Що таке інформатика як наукова дисципліна?
7. Що таке програмування?
8. Дати власне тлумачення терміну "програмування".
9. Що таке парадигма програмування?
10. Що таке програмна інженерія?
11. Що вивчає: а) теорія програмування; б) програмологія?
12. Що таке мова програмування?
13. Що таке: а) програма; б) дані програми?
14. Яку структуру має АПК?
15. Що таке обчислювальна система?
16. Що таке програмне забезпечення?
17. Що таке ІТ?
18. Коли, де й під чиїм керівництвом була створена перша ЕОМ: а) у світі; б) в Україні?
19. Коли була видана в Україні "Енциклопедія кібернетики"?

[8] СР – самостійна робота.

Розділ I

ЛОГІКО-АЛГЕБРИЧНІ УНІВЕРСАЛІЇ

У вступі наголошувалось, що інформатика широко використовує на практиці методи й засоби суміжних дисциплін. Мета даного розділу – стислий огляд фундаментальних понять алгебри й математичної логіки, які становлять основу дескриптологічних засобів, що застосовуються в інформатиці та програмуванні.

1.1. Формальна мова першого порядку

- Терми
- Елементарні формули
- Структуровані формули

Ключові слова: *предметна константа, предметна змінна, тип операції, функціональний символ, предикат, сигнатура, терм, підтерм, префіксний, постфіксний та інфіксний терми, ОПЗ, операнд, предикатний терм, умовний терм, формула, логічна зв'язка, квантор, пропозиційна змінна, пропозиційна формула, тавтологія, рівносильні пропозиційні формули.*

Серед дескриптивних математичних засобів, що застосовуються в інформатиці, чільне місце посідає *формальна мова першого порядку*, або мова *прикладного числення предикатів* (ПЧП)[9]. Її елементи становлять основу мовних конструкцій мов програмування. Це також основний на сьогодні засіб для формалізації й моделювання предметних областей у інформаційних системах. Усі неозначені тут поняття вважаються відомими. Однак із метою дотримання принципу замкненості всі вони будуть строго визначені в наступних підрозділах.

[9] У математичній логіці розглядають також мови вищих порядків, призначені для роботи з функціоналами – операціями та кванторами, визначеними на відображеннях і предикатах.

1.1.1. ТЕРМИ

Основу ПЧП становлять формули, які зображують математичні висловлювання. У будь-якій формулі фігурують певні об'єкти й конструкції, що їх описують. Кожна математична теорія має свою *предметну область*, або *універсум*, об'єктів. Наприклад, універсумом натуральної арифметики є сукупність натуральних чисел N. Математична теорія може мати кілька універсумів. Їх називають *типами* або *сортами*, а теорію – багатосортною. У математичному аналізі природно співіснують два сорти об'єктів – дійсні числа й функції, у лінійній алгебрі – сорти дійсних чисел, їхніх векторів і прямокутних матриць. Багатосортність є також характерною рисою мов програмування.

Для подання об'єктів у формулах використовують спеціальні синтаксичні конструкції – терми[10]. Найпростішими з них є константи. *Константи* – це імена конкретних об'єктів, наприклад $0, 1.25, \pi$ тощо.

Зафіксуємо певну лінійно впорядковану множину елементів $\Sigma = \{a_1, a_2, \ldots\}$ і назвемо її *алфавітом.* Елементи алфавіту будемо називати *літерами*, а скінченні послідовності літер – *словами над* Σ. Слово $y_1 y_2 \ldots y_n$ у ПЧП записують як у природній мові – без ком усередині. При цьому записи однолітерного слова та його літери збігаються (їх розрізняють за контекстом). Як імена констант та інших об'єктів ПЧП (типів, змінних, операцій і предикатів) використовують слова в певному фіксованому алфавіті Σ. Слово, навантажене значенням (смислом), називається *символом.*. Отже, *предметна константа* – це символ, за яким закріплено конкретне фіксоване значення. *Предметна змінна* – інший різновид термів. Це теж символ, але її значення, на відміну від констант, не фіксоване, і вона може в різних контекстах позначати різні об'єкти. Останні можуть належати або одному фіксованому (сильна типізація), або різним типам (слабка типізація). Кажуть, що в даному контексті змінна набула такого-то конкретного значення. Для зручності імена змінних часто індексують: наприклад, $x, y, \ldots, x_1, y_1, \ldots$.

Константи та змінні належать до *атомних* термів. У складніших випадках, коли об'єкт утворюється як результат застосування певної операції до якихось інших об'єктів, для його подання використовують *складені* терми, наприклад $\sin(\pi/2)$, $\cos 30^\circ$, $1+3, x \times y$ тощо. Якщо f – операція, що має n аргументів (*операндів*), то результат її застосування до операндів $t_1, \ldots, t_n$ записують у вигляді виразів $f(t_1, \ldots, t_n)$

[10] В елементарній алгебрі терми відомі як алгебричні вирази.

або $(t_1,...,t_n)f$, які називаються відповідно *префіксними* й *постфіксними термами*. Операндами можуть бути константи, змінні та інші терми. Вони називаються *підтермами* терму.

Операція f називається *зовнішньою,* або *опорною,* у термі. Дужки в префіксних і постфіксних термах обов'язкові. У випадку бінарних операцій може застосовуватись також *інфіксна* форма термів у вигляді $(t_1 f t_2)$, яку ми вже застосовували, але без дужок: $1+3, x\times y$.

Стосовно відсутності в цих термах дужок необхідно зробити одне важливе зауваження. Кінцева мета роботи математика – отримати той чи інший текст, що складається з конструкцій ПЧП (визначень, формул, тверджень, доведень) і пояснень до них, тобто така робота так чи інакше зводиться до роботи над текстами. Такі тексти можуть досягати значних розмірів, тому дуже важливо, щоб мовні засоби для їхнього подання були зручними, прозорими та якомога лаконічнішими. Подібні засоби називають *метамовними* відносно конструкцій ПЧП. Метамовні засоби в нашому випадку містять конструкції української мови й деякі полегшені для сприймання варіанти мовних конструкцій ПЧП. Останні (як терми без дужок) не є синтаксично правильними, але за ними легко однозначно відновити первісну конструкцію ПЧП.

Наведені вище скорочені вирази $\cos 30^\circ$ та $x\times y$ формально не є термами – вони належать метамові й замінюють "правильні" терми $\cos(30^\circ)$ та $(x\times y)$. При відновленні дужок у інфіксних термах може враховуватися пріоритет операцій. За домовленістю, наприклад, спочатку виконуються мультиплікативні операції $(\times,/)$, а потім – адитивні $(+,-)$. Так, метавираз $x\times y + x/y$ подає терм $((x\times y)+(x/y))$. Операції ж одного рівня виконуються зазвичай по порядку зліва направо. Однак може бути й навпаки. Метатерми з відкинутими дужками є в усіх мовах програмування.

Однак існують і бездужкові за означенням варіанти термів. Найвідомішими є бездужкові постфіксні, що отримали спеціальну назву – *обернений польський запис* (ОПЗ). Для термів $(1+3)$, $(x\times y)$ та $((x\times y)+(x/y))$ ОПЗ має вигляд відповідно $1\ 3\ +$, $x\ y\ \times$ та $x\ y\ \times\ x\ y\ /\ +$. В ОПЗ, незалежно від пріоритету, операції виконуються у строгому порядку тільки зліва направо, тобто черговою завжди є найбільш ліва з нерозглянутих операцій.

Вважається, що кожен з операндів операції *f*, як і змінні, має певний фіксований тип. Це стосується й значення, яке повертає операція. Такі операції називаються *типізованими.* Імена типізованих опе-

рацій будемо називати *символами операцій*, або *функціональними символами.* Наприклад, бінарна операція $\sqrt[n]{x}$ типізована, n набуває натуральних значень, змінна x і результат – дійсні числа. Типи операндів і результату складають *тип операції.* Якщо $\sigma_1,...,\sigma_n$ – типи операндів операції f, а σ – тип її результату, то тип операції f записують як $\sigma_1 \times ... \times \sigma_n \to \sigma$. Так, функція sin має тип $R \to R$, а вищезгадана операція кореня – тип $N \times R \to R$. Щоб підкреслити тип операції, його можуть додавати в явному вигляді до символу операції. Наприклад, пишуть $f : \sigma_1 \times ... \times \sigma_n \to \sigma$ або $f^{\sigma_1 \times ... \times \sigma_n \to \sigma}$. Однак у більшості випадків у цьому немає потреби, оскільки зазвичай за символом операції в конкретній ПЧП закріплюється той чи інший фіксований тип. Якщо це не так, то операцію називають *поліморфною*, а її конкретний тип визначають шляхом аналізу типів її операндів у кожному конкретному термі. Типовими прикладами поліморфних операцій є арифметичні операції на числах: у термі $2+3$ йдеться про додавання цілих чисел, а в термі $2.0+3$ – дійсних.

У мовах програмування випадки поліморфних операцій відомі під назвою *перевантаження* операцій. Кожний терм теж має свій *тип* – це тип його можливих значень. Він збігається з типом результатів його опорного функціонального символу. Отже, якщо терм t має вигляд $f^{\sigma_1 \times ... \times \sigma_n \to \sigma}(t_1,...,t_n)$, то його типом є σ.

1.1.2. ЕЛЕМЕНТАРНІ ФОРМУЛИ

Формули подають математичні висловлювання і щось стверджують про об'єкти предметної області. Кажуть, що вони можуть бути *істинними* (виконуватись), *хибними* (не виконуватись) або *беззмістовними* (не мати сенсу). Наприклад, арифметичне твердження $2+3=5$ істинне, $1+3=5$ – хибне, а твердження $(1/0)=0$ беззмістовне (частка від ділення на 0 не існує). Істину, хибу й беззмістовність будемо позначати відповідно константами 1, 0 та # [11]. Ці константи утворюють спеціальний логічний тип Bool.

Якщо $\sigma_1,...,\sigma_m$ позначають типи аргументів предиката p, то тип самого предиката записують $\sigma_1 \times ... \times \sigma_m \to$ Bool, або просто $\sigma_1 \times ... \times \sigma_m$. Щоб підкреслити тип предиката, його, як і у випадку операцій, мо-

[11] Зазвичай у формальних системах беззмістовність явно не зображують – для її подання вдаються до метазасобів.

жуть додавати до символу предиката: наприклад, $p:\sigma_1\times\ldots\times\sigma_m\to$ Bool або $p^{\sigma_1\times\ldots\times\sigma_m}$. Унарний предикат p називається *властивістю* на типі σ_1. Властивості використовуються, щоб виділити ту чи іншу підмножину A типу σ_1. Вони називаються *характеристичними* властивостями, або *характеристичними функціями,* відповідної підмножини. Якщо χ_A – характеристична властивість підмножини A, то $a\in A\Leftrightarrow\chi_A(a)=1$. Функція χ_A називається *частковою характеристичною властивістю* підмножини A, якщо $\overline{\chi}_A(a)=\begin{cases}1, a\in A\\ \#, a\notin A\end{cases}$. Вирази на зразок наведеного називаються *умовними.* Вони мають умову α і дві альтернативи – a та b – і повертають як результат одну з альтернатив залежно від того, виконується умова чи беззмістовність (якщо умова беззмістовна).

Предикати, характеристичні й частково характеристичні функції становлять основу *предикативних засобів* опису множин та інших елементів ПЧП. Введемо позначення "$\overset{def}{=}$" як скорочення фрази "покладемо за означенням". Оскільки n-арні предикати – теж операції, то для подання їхніх значень теж використовуються префіксні та інфіксні терми вигляду $p(t_1,\ldots,t_n)$, (t_1pt_2), де $t_1,\ldots,t_n$ – операнди відповідних типів. Такі терми називаються *предикатними*, або *елементарними формулами*, наприклад $3<5$, $\sin(x)=\sin(x+2\pi n)$ тощо.

Увага! Останній приклад демонструє ще один варіант скорочення записів термів і формул. Як бачимо, у них можуть пропускатись навіть символи окремих операцій. Однак треба розуміти, що йдеться про скорочений запис терму, а не сам терм. Вираз $2\pi n$ не є формально термом, а лише скорочено подає терм $((2\times\pi)\times n)$ ►

Для умовних виразів введемо лінійну форму подання. Нехай p – предикатний терм, t_1,t_2 – терми. Тоді вираз

$$(p\to t_1\mid t_2)\overset{def}{=}\begin{cases}t_1, p=1\\ t_2, p=0\\ \#, p=\#\end{cases}$$

називається *умовним термом.*

Усі формули ПЧП поділяються на *елементарні* та *структуровані*. Структуровані формули будуються з елементарних за певними стандар-

тними правилами, однаковими для всіх теорій. Тому, щоб повністю визначити ПЧП конкретної теорії, необхідно й достатньо задати сукупність елементарних формул. Для цього треба визначати всі предикатні символи, усі можливі терми й допоміжні символи. Серед останніх обов'язковими є символи "(", ")", ":", "→", "|". Введемо поняття сигнатури.

Сигнатура – це п'ятірка $\Omega = \left(\Omega_S, \Omega_C, \Omega_V, \Omega_f, \Omega_p\right)$, **де** $\Omega_S = \{s_1, s_2, \ldots\}$, $\Omega_C = \{c_1 : \sigma_1, c_2 : \sigma_2, \ldots\}$, $\Omega_V = \{x_1 : \sigma_1, x_2 : \sigma_2, \ldots\}$, $\Omega_f = \{f_1 : \tau_1, f_2 : \tau_2, \ldots\}$ **та** $\Omega_p = \{p_1 : \nu_1, p_2 : \nu_2, \ldots\}$ **– непорожні сукупності відповідно імен сортів, типізованих констант, змінних, функціональних і предикатних символів.**

Трійка $\Delta = (\sigma_1, \sigma_2, \ldots; \tau_1, \tau_2, \ldots; \nu_1, \nu_2, \ldots)$ називається *типом сигнатури* Ω, а терми й елементарні формули сигнатури Ω називають Ω-*термами* та Ω-*формулами*. Отже, кожна ПЧП має певну сигнатуру Ω. Щоб надати значення елементарним формулам ПЧП, необхідно *проінтерпретувати* (*оцінити*) усі символи її сигнатури. Для цього вибираються конкретні множини для сортів елементів; константам, функціональним і предикатним символам ставляться у відповідність певні фіксовані значення, конкретні операції та предикати згідно з їхнім типом. Після інтерпретації елементарні формули, що не містять змінних, отримують конкретне істиннісне значення (стають істинними, хибними або беззмістовними). Щоб елементарна формула зі змінними теж отримала значення, необхідно дати *оцінку* її змінним, тобто обрати для них відповідно до їхнього типу те чи інше конкретне значення в інтерпретації. Як приклад розглянемо елементарну частину ПЧП для натуральної арифметики $\Omega = (N, \text{Bool}, +, -, \times, /, 0, =, <)$.

Приклад 1.1. Елементарна частина ПЧП для натуральної арифметики . Сигнатура Ω арифметики має такий вигляд:

1) $\Omega_S = \{\sigma, \lambda\}$, де сорти σ та λ подають відповідно множину натуральних чисел $N = \{0,1,2,\ldots\}$ [12] і сукупність істиннісних значень $\text{Bool} = \{0,1,\#\}$;

2) $\Omega_C = \{0 : \sigma\}$, де константи $0 : \sigma$ інтерпретуються як число 0;

3) сукупність індексованих змінних $\Omega_V = \{x : \sigma, x_1 : \sigma, \ldots, y : \sigma, y_1 : \sigma, \ldots\}$;

4) сукупність однотипних функціональних символів $\Omega_f = \{+, -, \times, /, \% : \sigma \times \sigma \to \sigma\}$, що інтерпретуються як арифметичні

[12] Тут і далі 0 вважається натуральним числом.

операції додавання, віднімання, множення, ділення та взяття залишку від ділення;

5) сукупність однотипних предикатних символів $\Omega_p = \{<, = : \sigma \times \sigma \to \lambda\}$, що інтерпретуються як арифметичні предикати "менше" та "дорівнює".

Покладемо $\tau = \sigma \times \sigma \to \sigma$, $\nu = \sigma \times \sigma \to \lambda$. Типом сигнатури Ω є тип $\Delta = (\sigma; \tau, \tau, \tau, \tau; \nu, \nu)$. В арифметичних термах застосовується згаданий вище пріоритет операцій, а в елементарних формулах символи операцій старші, ніж предикатні. Наприклад, метаформула $2/3 = 0$ подає формулу $((2/3) = 0)$ і є істинною, формула $x < 0$ хибна в за будь-якої оцінки змінної *x,* а формула $0 < x - 1$ істинна за всіх оцінок x, окрім двох: $x = 0$ та $x = 1$. У першому випадку формула беззмістовна (результат віднімання 0 – 1 – не визначено), у другому – хибна ■

1.1.3. СТРУКТУРОВАНІ ФОРМУЛИ

Структуровані формули всіх ПЧП будуються однаково – за допомогою логічних зв'язок і кванторів. Зв'язки та квантори будемо трактувати як певні операції (*композиції*) на формулах, що дозволяють будувати нові формули з простіших. Нехай P та Q – формули. Класичними логічними зв'язками є:

1) $(P \,\&\, Q)$ – кон'юнкція (логічне "і");

2) $(P \vee Q)$ – диз'юнкція (логічне "або");

3) $(P \supset Q)$ – імплікація (логічне "випливає");

4) $(P \leftrightarrow Q)$ – еквіваленція (логічне "тоді й тільки тоді");

5) $(\neg P)$ – заперечення (логічне "ні").

У формулах 1)–3) P та Q називаються відповідно *кон'юнктивними* та *диз'юнктивними членами, засновком* і *висновком* імплікації. Прикладами структурованих формул є $(P \vee (\neg P))$, $(((P \supset Q) \supset Q) \supset Q)$, $((\neg P) \vee (\neg Q))$ тощо. Навпаки, вирази $(\neg P$ або $(P \supset)$ не є формулами. У першому випадку не вистачає закриваючої дужки, у другому – висновку імплікації.

Правила обчислення значень структурних формул задає табл. 1.1[13]. Зазначимо, що в класичних ПЧП логіка двозначна. У нашому варіанті

[13] Це тільки один з можливих варіантів правил. Він відповідає практиці мов програмування.

ПЧП логічні зв'язки поширюються й на випадок беззмістовних значень формул. Нагадаємо, що символи 1, 0 та # позначають відповідні істиннісні значення.

Згідно з табл. 1.1 формула $P \vee (\neg Q)$ для $P = 0, Q = 0$ та $P = 1, Q = \#$ істинна, для $P = 0, Q = \#$ – беззмістовна.

Вивчення алгебричних властивостей елементарних формул зводиться до вивчення так званих *пропозиційних форм*, які будуються з пропозиційних змінних за допомогою логічних зв'язок. *Пропозиційні змінні* – це змінні, які пробігають істиннісні значення 1, 0 та #. Пропозиційні форми Φ_1 та Φ_2 називаються *рівносильними* ($\Phi_1 \cong \Phi_2$), якщо їхні значення збігаються на всіх наборах значень змінних. Рівносильні форми можна замінювати одна на одну, і це не вплине на значення загальної форми, до якої вони входять. Пропозиційна форма називається *тавтологією*, якщо вона істинна за всіх змістовних (тобто 0 та 1) значень своїх змінних.

Для пропозиційних форм існує багато повних систем форм-аксіом, які дозволяють отримати з них усі форми-тавтології за допомогою певних правил виведення. Одну з таких систем наведено у вправі 19. Пропозиційні логіки використовуються для спрощення та еквівалентних перетворень структурних формул ПЧП (див. вправу 26).

Таблиця 1.1

P	Q	$P \& Q$	$P \vee Q$	$P \supset Q$	$P \leftrightarrow Q$	$\neg P$
1	1	1	1	1	1	0
1	0	0	1	0	0	
1	#	#	1	#	#	
0	1	0	1	1	0	1
0	0	0	0	1	1	
0	#	0	#	#	#	
#	1	#	#	#	#	#
#	0	#	#	#	#	
#	#	#	#	#	#	

Нехай x – довільна змінна, P – формула. *Квантор* $\forall$ ("для всіх") утворює формулу $(\forall xP)$, квантор $\exists$ ("існує") – формулу $(\exists xP)$. Кажуть, що всі входження змінної x у формули $(\forall xP), (\exists xP)$ є *зв'язаними*. Певне входження змінної x у довільну формулу Q називається *вільним*,

якщо воно не є зв'язаним. Формула може мати одночасно і зв'язані, і вільні входження однієї й тієї самої змінної.

Наприклад, у формулі $(\forall y(\exists z((\forall x(x>1))\,\&\,x+y=z)))$ (*) два перші входження змінної x зв'язані, а третє – вільне, входження решти змінних – зв'язані. Формула $(\forall xP)$ істинна в інтерпретації, якщо формула P істинна в ній для всіх можливих оцінок змінних, що містять x. Формула $(\exists xP)$ істинна в інтерпретації, якщо формула P істинна в ній принаймні для однієї оцінки змінних, що містить x. І в першому, і в другому випадках оцінка впливає тільки на вільні входження x у формулу P.

Приклад 1.2. Істинність формул із кванторами в арифметиці □.

Формули $(\forall x(2/3=0))$ та $(\forall x\neg(x<0))$ істинні в □, а формула $(\forall x(0<x-1))$ хибна. Формула $(\exists x(0<x-1))$ істинна в □, оскільки $0<x-1$ істинна для $x=2,3,\ldots$, а формула $\exists x(x=x+1)$ хибна. Вищезгадана формула (*) хибна (кон'юнктивний член $(\forall x>1)$ хибний за оцінки $x=1$). Навпаки, формула $(\forall y(\exists z(\forall x((x>1)\,\&\,x+y=z))))$ істинна. Вона не містить вільних входжень x і описує існування суми двох натуральних чисел, перший із доданків якої більший від 1 ■

Іноді можуть цікавити не всі оцінки змінних, а тільки ті, що задовольняють певну умову Q. Тоді застосовують *обмежені квантори* вигляду $(\forall xQ)$ та $(\exists xQ)$: $(\forall xQ)P \overset{def}{=} (\forall x(Q\,\&\,P))$, $(\exists xQ)P \overset{def}{=} (\exists x(Q\,\&\,P))$. Наприклад, $(\forall x>0)P$, $\exists x(x\%2=0)P$. Якщо необхідно гарантувати існування не більше одного елемента, що задовольняє певну умову *P,* то використовують квантор $(\exists 1xP)$.

Як і у випадку термів, існують певні домовленості про економне застосування дужок у структурованих формулах. По-перше, зазвичай опускають зовнішню пару дужок. По-друге, якщо формула містить входження тільки однієї бінарної зв'язки, то дужки взагалі не пишуться, а відновлюються, розпочинаючи справа наліво. По-третє, серед зв'язків установлено такий пріоритет: $\neg$ & $\vee$ $\supset$ $\leftrightarrow$. Це означає, що спочатку відновлюються дужки праворуч від заперечень, а потім дужки, пов'язані з кон'юнкцією тощо.

Приклад 1.3. Відновлення дужок.

У формулі $P\vee\neg Q\supset R\leftrightarrow P$ дужки відновлюються таким чином:

$P\vee(\neg Q)\supset R\leftrightarrow P\Rightarrow$

$(P\vee(\neg Q))\supset R\leftrightarrow P\Rightarrow$

$((P\vee(\neg Q))\supset R)\leftrightarrow P\Rightarrow$

$\Big(\big((P \vee (\neg Q)) \supset R\big) \leftrightarrow P\Big)$ ■

Для зв'язок і кванторів діє пріоритет $\neg \,\&\, \vee \left\{\begin{matrix}\forall \\ \exists\end{matrix}\right\} \supset \leftrightarrow$. Наприклад, замість формули $(\forall x(A(x) \vee B(x,y)))$ можна писати $\forall x A(x) \vee B(x,y)$, а замість формули $((\forall x(A(x)) \supset B(x,y))$ – $\forall x(A(x) \supset B(x,y)$.

Формули ПЧП дозволяють формально описувати математичні властивості об'єктів предметної області.

Приклад 1.4. Формалізуємо твердження: "Для всякого натурального числа існує просте число, більше за нього".

Введемо до сигнатури Ω ПЧП натуральної арифметики унарний предикат prime, який інтерпретується як "бути простим числом". Тоді твердження можна подати формулою $\forall n \exists p(\text{prime}(p) \,\&\, n < p)$ [14] ■

У наступних підрозділах буде уточнено загальне поняття інтерпретації ПЧП і розглянуто деякі властивості інтерпретованих ПЧП.

Наостанок зробимо кілька зауважень. По-перше, тут і далі ми зосереджуємо основну увагу на синтаксисі й семантиці ПЧП і майже не зачіпаємо суто логічну частину числення. По-друге, ми розглядаємо так званий класичний варіант ПЧП, але вже сьогодні очевидно, що за всієї його важливості він не може задовольнити всі потреби інформатики та програмування. Тому в математичній логіці та програмології активно досліджуються й нові, значно потужніші, неокласичні варіанти алгебричних систем та їхніх числень. Інформацію про них можна знайти в літературі для самостійної роботи (розд. 1.4.).

*__Література для СР:__ формальні мови – [56, 63, 69, 121]; формальна мова першого порядку – [63, 79, 84, 89, 94, 123]; пропозиційна логіка – [63, 84, 94, 123]; класична логіка як прикладна система – [89].

Контрольні запитання та вправи

1. Що таке предметна константа?
2. Що таке предметна змінна?
3. Що таке тип операції?
4. Навести приклади типізованих операцій.
5. Що таке функціональний (предикатний) символ? Проілюструвати прикладами.
6. Які елементи входять до складу сигнатури ПЧП?

[14] Насправді предикат prime(x) є арифметичним, тобто його можна виразити засобами самої ПЧП натуральної арифметики, але це потребує певних технічних навичок.

7. Що таке терм (умовний терм, підтерм)?
8. Скільки всього підтермів у таких термах: а) $(((x \times y) + (x / y)) \times 1)$,
   б) $(f^3(x, y, x) + g^2(f^3(1, x + y, g^2(x, z)), 2))$?
9. Що таке опорна операція в термі? Проілюструвати прикладами.
10. У чому полягає різниця між префіксними, постфіксними та інфіксними термами?
11. Що таке ОПЗ?
12. Яка роль формул у ПЧП?
13. Які бувають логічні зв'язки та квантори?
14. Що таке пропозиційна змінна та пропозиційна формула?
15. Дати визначення тавтології. Навести приклади тавтологій.
16. Що таке рівносильні пропозиційні формули?
17. Чи можна усунути дужки у формулах: а) $P \supset (Q \supset R)$; б) $(P \supset Q) \supset R$; в) $\neg(P \,\&\, Q)$; г) $P \vee (Q \supset R)$?
18. Відновити дужки у формулах: а) $P \supset \neg(Q \,\&\, R) \vee P \leftrightarrow R$; б) $P \supset Q \supset R \leftrightarrow \neg Q \vee P$.
19. Розглянемо систему аксіом:
    1) $P \supset (Q \supset P)$;
    2) $(P \supset (Q \supset R)) \supset ((P \supset Q) \supset (P \supset R))$;
    3) $(\neg Q \supset \neg P) \supset ((\neg Q \supset P) \supset Q)$
    і правило виведення МР (лат. – modus ponens): із формул-засновків P, $P \supset Q$ виводиться формула-висновок Q. Показати, що: а) усі аксіоми 1)–3) є тавтологіями для будь-яких формул P, Q, R, б) якщо у правилі МР засновки P, $P \supset Q$ – тавтології, то й висновок Q – тавтологія, в)** система аксіом є повною, тобто в ній виводяться усі тавтології [84].
20. Довести рівносильність пропозиційних форм 1)-6):
    1) $x \vee x \,\&\, y \cong x$; 2) $x \,\&\, (y \vee x) \cong x$; 3) $x \vee x \,\&\, y \cong x \vee y$;
    4) $\neg x \vee x \,\&\, y \cong \neg x \vee y$; 5) $x \,\&\, (y \vee x) \cong x \,\&\, y$; 6) $\neg x \,\&\, (y \vee x) \cong \neg x \,\&\, y$.
21. Відновити дужки у формулі $\forall y \exists z \forall x (P_1(x) \supset P_2(y) \,\&\, \neg P_1(z))$.
22. З'ясувати, які входження змінних x, y, z є вільними та зв'язаними у формулі $\forall y (P_2(x, y) \supset \forall z \forall x P_1(y, x, z)) \supset P_2(z, y))$.
23. Записати засобами ПЧП натуральної арифметики такі твердження:
    а) для будь-яких двох чисел існує найбільший спільний дільник;
    б) найбільший спільний дільник двох чисел ділиться на будь-який їхній спільний дільник.

24. Студент написав твердження: $\forall x(Dv(x) \supset \hat{A}^3\ddot{a}(x))$, де універсумом є академічна група, $\hat{A}^3\ddot{a}(x)$ інтерпретується як "x – відмінник", $Dv(x)$ – як "x має заборгованості". Чи може воно бути істинним?
25. Ввести необхідну сигнатуру й переформулювати твердження у вигляді формул відповідних ПЧП:
    а) тільки судді в захваті від суддів;
    б) існують судді, від яких не в захваті жоден правопорушник;
    в) кожна жінка-адвокат у захваті від якогось судді-чоловіка;
    г) деякі програми можна зрозуміти;
    д) деякі лекції не можна зрозуміти;
    е) є замки, які не відкриваються, але зачиняються;
    є) кожна програма містить помилки;
    ж) усі чесні програмісти поважають один одного.
26*. Спростити формули ПЧП, де p,q,r – унарні предикатні символи, x,y,z – предметні змінні:
    а) $p(x)\,\&\,q(y)\,\&\,r(z)\vee\ p(x)\,\&\,q(y)\,\&\,\neg r(z)\vee$
    $\vee p(x)\,\&\,\neg q(y)\,\&\,r(z)\vee\neg p(x)\,\&\,q(y)\,\&\,r(z)$;
    б) $\forall x\forall y\forall z(p(x)\vee\ p(x)\,\&\,q(y)\vee\ q(y)\,\&\,r(z)\ \vee\neg p(x)\,\&\,r(z))$;
    в) $\forall x\exists y(\,(\,p(x)\vee q(y))\&$
    $\&(\neg p(x)\,\&\,\neg q(y)\vee r(z))\vee\neg r(z)\vee(p(x)\vee q(y))\,\&\,(u(z)\vee v(z)))$.

1.2. Множини

- Поняття множини
- Алгебра множин
- Індексовані множини
- Бінарні відношення
- Еквівалентності й порядки

Ключові слова: *множина, характеристична функція, індексована множина, пряма сума множин, перетин множин, об'єднання множин, різниця множин, доповнення множини, закони булевої алгебри множин, діаграма Ейлера – Венна, кортеж, вектор, декартів добуток множин, відношення, відповідність, відображення, множення та обернення відношень, рефлективність, симетричність, транзитивність, транзитивне замикання бінарного відношення, антисиметричність, частковий порядок, лінійний*

порядок, діаграма Гессе, монотонне й неперервне відображення, нерухома точка, структура, повна структура, еквівалентність, фактор-множина, індекс еквівалентності, факторизація.

У цьому підрозділі розглядаються фундаментальні з погляду дескриптології математичні поняття – множина, індексована множина й відношення, а також деякі похідні від них поняття, важливі для застосувань.

1.2.1. ПОНЯТТЯ МНОЖИНИ

Сукупність матеріальних чи абстрактних об'єктів, об'єднаних будь-якою спільною властивістю, називається *множиною.* Об'єкти й поняття, що складають множину, називаються її *елементами.* Будемо позначати множини великими, а їхні елементи – малими літерами латинської або грецької абеток. Твердження, що елемент a є елементом множини A, подається формулою $a \in A$. Бінарний предикат $\in$ називається *предикатом приналежності.* Його заперечення записується як $\notin$, тобто формула $a \notin A$ означає, що об'єкт a не належить A. Елементами множини можуть бути й самі множини. Наприклад, студентські академічні групи утворюють курс певного року навчання й самі є множинами, що складаються з відповідних студентів. Дві множини A та B називаються *рівними* ($A = B$), якщо вони складаються з одних і тих самих елементів. Засобами ПЧП цей предикат коротко можна було б визначити так:

$$A = B \overset{def}{=} \forall x(x \in A \leftrightarrow x \in B).$$

Для того, щоб задати множину, достатньо перерахувати всі її елементи в довільному порядку. Такий перерахунок записується зазвичай усередині фігурних дужок. Наприклад, запис $\{2,0,7\}$ означає множину M, що складається із чисел 2, 0 та 7. Ця множина збігається з множинами $\{0,2,7\}$ та $\{2,0,0,7\}$. В останньому випадку елемент 0 поданий двічі, а це не суттєво для означення множини. Розрізнятимемо елемент a, множину $\{a\}$, множину $\{\{a\}\}$, єдиним елементом якої є множина $\{a\}$ тощо. Розглядають також порожню множину $\varnothing$, яка не містить жодного елемента. Множина B називається *підмножиною* множини A, а множина A, у свою чергу, – *надмножиною* множини B (записується $B \subseteq A$), якщо кожний елемент B є елементом A, тобто $B \subseteq A \overset{def}{=} \forall x(x \in B \to x \in A)$.

Підмножина B називається *власною* ($B \subset A$), якщо $B \neq A$. Нехай A – певна множина, P – деяка властивість елементів з A. Нехай формула $P(x)$ означає, що елемент x має властивість P, тобто $P(x) = 1$. Тоді запис $P(A)$ означає підмножину $\{x \in A : P(x)\}$ тих елементів A, які мають властивість P. Як уже зазначалось, P називається характеристичною властивістю (функцією) цієї підмножини. Якщо x є типізованою змінною типу T, то замість виразу вигляду $\{x \in A : P(x)\}$ пишуть просто $\{x : P(x)\}$. Так, у прикл. 1.5 змінна n вважається натуральною.

Приклад 1.5. Характеристичні властивості множин.

$\{x \in N : x < 0\} = \varnothing$,

$\{x \in N : 0 < x < 4\} = \{1,2,3\}$,

$\{x \in N : x = a_1 \vee \ldots \vee x = a_n\} = \{a_1, \ldots, a_n\}$.

$\{n : n\%2 = 0\}$ збігається з множиною парних натуральних чисел.

Тут символ "%" позначає операцію взяття залишку від ділення цілих чисел ■

1.2.2. АЛГЕБРА МНОЖИН

У зв'язку з поняттям множини цілком природно виникає питання: чим є сукупність (універсум) усіх можливих множин U ? Як з'ясувалося, сам універсум U не може належати до множин. Це першим зафіксував Б. Рассел (парадокс Рассела). Дійсно, якщо універсум визнати множиною, то має існувати й множина $M \overset{def}{=} \{x \in U : x \notin x\}$. Припустимо, що $M \in M$. Тоді за означенням $M \notin M$. Нехай тепер $M \notin M$. Тоді за означенням $M \in M$. В обох випадках маємо протиріччя. Отже, або універсум U не є множиною, або необхідно відмовитись від визначень підмножин за допомогою властивостей. Математики обрали перший шлях і вилучили з теорії універсум усіх множин. Один зі способів уникнути парадоксів, подібних парадоксу Рассела, запропонував сам Рассел у теорії типів. За цією теорією кожній константі та змінній приписується певний тип, сукупність типів має строгу ієрархічну будову, а предикат $x \in M$ має сенс тільки у випадку, коли тип M є наступним у ієрархії за типом x. Якщо це не так, то значення предиката вважається беззмістовним. Теорія типів мала певний вплив на розбудову основ математики. Однак не менше, а можливо, і ширше,

своє застосування вона знайшла при створенні штучних формальних мов різного призначення, у тому числі – мов програмування (див. принцип типізації, підрозд. 2.2.2). Принциповим у зв'язку із цим є те, що поняття типу унеможливлює появу заздалегідь помилкових мовних виразів, і перевірити виконання типових вимог можна суто формально, не звертаючи увагу на значення виразів. Визначимо тепер *основні операції* на множинах:

$A \cup B \overset{def}{=} \{x : x \in A \vee x \in B\}$ – *об'єднання множин* A *та* B,

$A \cap B \overset{def}{=} \{x : x \in A \,\&\, x \in B\}$ – *перетин множин* A *та* B,

$A \setminus B \overset{def}{=} \{x : x \in A \,\&\, x \notin B\}$ – *різниця множин* A *та* B,

$\overline{A} \overset{def}{=} \{x : x \notin A\}$ – *доповнення множини* A,

$B(A) \overset{def}{=} \{X : X \subseteq A\}$ – *булеан множини* A,

$|A| \overset{def}{=}$ – *потужність множини* A.

Булеан множини A також позначають 2^A.

Нехай A – довільна множина. Сигнатуру Ω_2 булевої алгебри множин $\mathrm{B}(A) = \{2^A; \cup, \cap, \setminus, \bar{}, \subseteq\}$ над A складають:

1) сорти σ, ρ та λ, що подають відповідно базову множину A, булеан 2^A і множину істиннісних значень $\mathrm{Bool} = \{1, 0, \#\}$;

2) сукупність Ω_C – константа $\varnothing$ типу ρ, що інтерпретується як порожня підмножина;

3) індексована сукупність змінних

$\Omega_V = \{x : \sigma, x_1 : \sigma, \ldots, y : \sigma, y_1 : \sigma, \ldots, X : \rho, X_1 : \rho, \ldots, Y : \rho, Y_1 : \rho, \ldots\}$;

4) бінарні функціональні символи $\cup, \cap, \setminus : \rho \times \rho \to \rho$ і унарний символ $\bar{} : \rho \to \rho$, що інтерпретуються відповідно як операції об'єднання, перетину, різниці й доповнення множин;

5) предикатні символи $\subset : \rho \times \rho \to \mathrm{Bool}$ та $\in : \sigma \times \rho \to \mathrm{Bool}$, що інтерпретуються як предикати включення й приналежності.

Покладемо $\tau = \rho \times \rho \to \rho$, $\nu = \rho \to \rho$, $\varsigma = \rho \times \rho \to \mathrm{Bool}, \mu = \sigma \times \rho \to \mathrm{Bool}$. Типом сигнатури Ω_2 є $\Delta = (\rho; \tau, \tau, \tau, \nu; \varsigma, \mu)$.

Для операцій об'єднання, перетину й доповнення виконуються *закони булевої алгебри*, перші два з яких проілюструємо за допомогою діаграм Ейлера –Венна:

1) *комутативності:* $A \cup B = B \cup A$, $A \cap B = B \cap A$ (рис. 1.1);

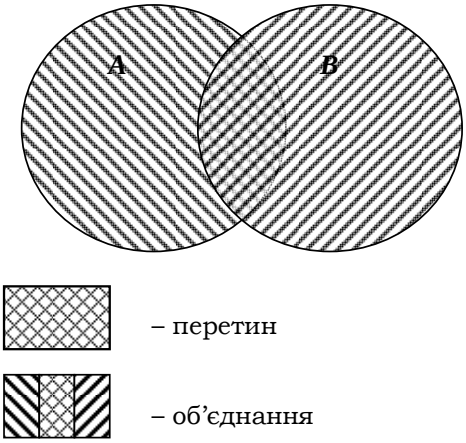


Рис. 1.1

2) *асоціативності* об'єднання й перетину множин:
$A \cup (B \cup C) = (A \cup B) \cup C$, $A \cap (B \cap C) = (A \cap B) \cap C$ (рис. 1.2);

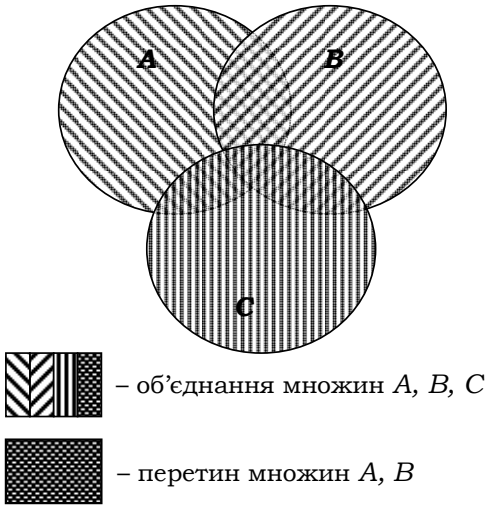


Рис. 1.2

3) *дистрибутивності:*

а) перетину відносно об'єднання:
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$;

б) об'єднання відносно перетину:
$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$;

4) *ідемпотентності:*
$A \cup A = A$, $A \cap A = A$;

5) *дії з універсальною й порожньою множинами:*
$A \cup \varnothing = A$, $A \cap \varnothing = \varnothing$, $A \cup U = U$, $A \cap U = A$, $A \cup \overline{A} = U$, $A \cap \overline{A} = \varnothing$;

6) *де Моргана:*
а) $\overline{A \cap B} = \overline{A} \cup \overline{B}$;
б) $\overline{A \cup B} = \overline{A} \cap \overline{B}$;

7) *подвійного доповнення:* $\overline{\overline{A}} = A$.

Наведені закони дозволяють за допомогою тотожних перетворень спрощувати складні вирази над множинами.

Приклад 1.6. Спростимо вираз, скориставшись законом булевої алгебри підмножин над множиною $\mathbf{A}$:

$$(A \cap B \cap C) \cup (\overline{A} \cap B \cap C) \cup (\overline{B} \cup \overline{C}) = [(A \cup \overline{A}) \cap B \cap C] \cup (\overline{B} \cup \overline{C}) =$$
$$= [\mathbf{A} \cap B \cap C] \cup \overline{B \cap C} = B \cap C \cup \overline{B \cap C} = \mathbf{A} \blacksquare$$

1.2.3. ІНДЕКСОВАНІ МНОЖИНИ

Розглянемо загальні засоби структурування елементів множин. Першим і найфундаментальнішим є *індексування* (ми його вже застосовували в підрозд. 1.1). Нехай I та M – довільні множини. Елементи I будемо називати *індексами*. *Проіндексувати* множину M означає поставити у відповідність усім або деяким її елементам певний індекс. При цьому можливі два варіанти: 1) різні елементи отримають обов'язково різні індекси (*сильна індексація*); 2) індекси різних елементів можуть збігатися (*слабка індексація*). В обох випадках зворотне не вимагається, тобто елемент може мати кілька індексів. Якщо i – індекс елемента a, то індексований елемент записується a_i. Важливо, що, приписуючи індекс елементу, ми тільки вводимо для нього нове ім'я, а сам елемент при цьому не змінюється. Проіндексовані елементи збігаються тільки тоді, коли вони однакові й у них збігаються індекси. Отже, якщо $a_i = b_j$, то $a = b$ та $i = j$. Зокрема, $a_i \neq a_j$ для будь-якого a та $i \neq j$. Слабка індексація використовується, наприклад, для побудови прямих сум множин. При звичайному об'єднанні однакові елементи в операндах "склеюються" в один екземпляр. Цього можна уникнути, якщо перед об'єднанням

множини спочатку проіндексувати. Щоб побудувати *пряму суму множин* $A \oplus B$, проіндексуємо всі елементи множин A та B відповідно індексами 1 та 2 і покладемо

$$A \oplus B \stackrel{def}{=} \{a_1 : a \in A\} \cup \{b_2 : b \in B\}.$$

У прямій сумі множин індекс кожного елемента свідчить про приналежність його до першого чи другого операнда й саме за цим показником однакові елементи можуть різнитися. Наприклад, для $A = \{a,b,c\}$, $B = \{b,c\}$ пряма сума $A \oplus B = \{a_1, b_1, c_1, b_2.c_2\}$, тоді як об'єднання $A \cup B = A$. Значення елементів b_1 і b_2 та c_1 і c_2 однакові (відповідно b та c), але як індексовані елементи вони різні й тому не "склеюються" в сумі. Надалі під індексацією будемо розуміти сильний її варіант.

У математиці як індекси найчастіше використовують натуральні числа (усі або початкові їхні відрізки). Нехай M – довільна множина. Сильно проіндексовані натуральними числами множини називаються *послідовностями на множині* M, але не всі, а тільки ті, у яких для індексації елементів використані всі без винятку натуральні числа (або всі числа їхнього початкового відрізка). Кількість елементів у послідовності називається її *довжиною.* Наприклад, для множини $A = \{a,b,c\}$ послідовностями будуть $\{a_1\}$ (довжиною 1), $\{a_1, b_2\}$ (довжиною 2), $\{c_i : i \geq 0\}$ (нескінченна послідовність), а сукупності $\{a_1, b_1\}$ та $\{a_1, b_3\}$ – ні. У першому випадку індексація несильна, у другому – пропущено елемент з індексом 2. У другому випадку індексована сукупність називається *підпослідовністю.* Таким чином, на відміну від послідовностей, у підпослідовності можуть бути пропущені ті чи інші індекси.

Індексація дозволяє переносити бінарні операції на випадок індексованих сімей множин. Наприклад, нехай $P_1,...,P_n$ та $A_1,...,A_n$ – довільні проіндексовані формули й множини, а $\{B_i : i \in I\}$ – індексована сім'я множин. Домовимось про такі позначення: $0...n \stackrel{def}{=} \{0,...,n\}$, $1...n \stackrel{def}{=} \{1,...,n\}$, вирази $1 \leq \exists i \leq n$ та $1 \leq \forall i \leq n$ замінюють обмежені квантори $\exists i \in 1...n$ та $\forall i \in 1...n$. Тоді вже відомі нам операції можна узагальнити:

$$\&_{i=1}^{n} P_i \stackrel{def}{=} 1 \leq \forall i \leq n P_i; \qquad \vee_{i=1}^{n} P_i \stackrel{def}{=} 1 \leq \exists i \leq n P_i;$$

$$\cup_{i=1}^{n} A_i \stackrel{def}{=} \{x : 1 \leq \exists i \leq n \ x \in A_i\}; \qquad \cap_{i=1}^{n} A_i \stackrel{def}{=} \{x : 1 \leq \forall i \leq n \ x \in A_i\};$$

$\bigcup\limits_{i\in I} A_i \overset{def}{=} \{x : \exists i \in I \ x \in A_i\}$; $\bigcap\limits_{i\in I} B_i \overset{def}{=} \{x : \forall i \in I \ x \in B_i\}$;

$\bigoplus\limits_{i=1}^{n} A_i \overset{def}{=} \bigcap\limits_{i=1}^{n} \{a_i : a \in A_i\}$; $\bigoplus\limits_{i\in I} A_i \overset{def}{=} \bigcup\limits_{i\in I} \{a_i : a \in A_i\}$;

$\sum\limits_{i=1}^{n} a_i = a_1 + \ldots + a_n$; $\prod\limits_{i=1}^{n} a_i = a_1 \times \ldots \times a_n$.

Подібні узагальнені операції розглядаються, наприклад, у [62, 105] і мають численні застосування у програмуванні. Скінченні послідовності отримали окрему назву *кортежів*. Індекси елементів кортежу називаються їхніми *порядковими номерами,* або *координатами*. Кортежі записуються у вигляді $[a_0,\ldots,a_{n-1}]$, $[a_1,\ldots,a_n]$. Кортеж довжиною 0 є порожньою сукупністю й записується []. На відміну від множин, елементи в кортежі можуть повторюватись, але координати при цьому в них будуть різні. Сукупність усіх кортежів з елементами з A позначається A^*. Два кортежі рівні, якщо вони мають однакову довжину й рівні покоординатно:

$$[a_0,\ldots,a_{n-1}] = [b_0,\ldots,b_{m-1}] \overset{def}{\Leftrightarrow} n = m \,\&\, \&\limits_{i=0}^{n-1} a_i = b_i.$$

Визначимо деякі операції на кортежах:

1) $|x| \overset{def}{=}$ *довжина* кортежу x;

2) $\forall i \in 1..n \ \ \mathrm{pr}_i[a_1,\ldots,a_n] \overset{def}{=} a_i$ – *проекція* кортежу за i-ю компонентою;

3) $[a_1,\ldots,a_n] \circ [b_1,\ldots,b_m] \overset{def}{=} [a_1,\ldots,a_n, b_1,\ldots,b_m]$ – *конкатенація* кортежів;

4) $\mathrm{pre}(a,[a_1,\ldots,a_n]) \overset{def}{=} [a,a_1,\ldots,a_n]$ – додавання компоненти в початок кортежу;

5) $\mathrm{app}([a_1,\ldots,a_n],a) \overset{def}{=} [a_1,\ldots,a_n,a]$ – додавання компоненти в кінець кортежу.

Кортежі x та z – це відповідно *префікс* $(x \lhd y)$ і *закінчення* $(y \rhd x)$ кортежу y, якщо існують такі кортежі v та w, що $y = x \circ w$ та $y = v \circ z$. Кортеж z є *суфіксом* кортежу y $(x \propto y)$, якщо існують такі непорожні кортежі v та w, що $y = v \circ z \circ w$.

Індексація є основним інструментом структурування даних у програмуванні. Тут як індекси використовуються спеціальні імена – *l*-вирази[15].

[15] Найпростіші з них – ідентифікатори (див. 3.1.2).

1.2.4. БІНАРНІ ВІДНОШЕННЯ

Кортежі фіксованої довжини $n \geq 0$ називаються *векторами*. Вони позначаються $(a_1,....,a_n)$. Елементи вектора називаються *компонентами*. *Декартовим добутком* сукупності множин $A_1,...,A_n$ називається множина $\prod_{i=1}^{n} A_i \overset{def}{=} \{(a_1,...,a_n) : \forall i \in 1...n \ \ a_i \in A_i\}$.

Декартів добуток n співмножників A називається *декартовим степенем* A й позначається A^n. За означенням $A^1 = A$, $A^0 = \{[]\}$, де [] – *порожній* кортеж. Якщо принаймні одна з координат добутку A_i порожня, то $\prod_{i=1}^{n} A_i = A^0$. *Відношенням* між множинами $A_1,...,A_n$ називається довільна підмножина декартового добутку $R \subseteq \prod_{i=1}^{n} A_i$. Якщо вектор $(a_1,....,a_n) \in R$, то кажуть, що елементи $a_1,....,a_n$ *задовольняють* відношення R, і записують $R(a_1,....,a_n)$. У протилежному випадку кажуть, що дані елементи *не задовольняють* R, і записують $\neg R(a_1,....,a_n)$. Відношення зручно задавати предикатами. Предикат $p : \prod_{i=1}^{n} A_i \to \text{Bool}$ називається *характеристичною властивістю* відношення R, якщо $R(a_1,....,a_n) \Leftrightarrow p(a_1,....,a_n) = 1$.

Вектори довжиною 2 називаються *упорядкованими парами* й позначаються (a_1,a_2). Елементи a_1 та a_2 називаються відповідно *першою* й *другою компонентами* пари.

Бінарним відношенням, або *відповідністю* між множинами A та B, називається довільна підмножина декартового добутку $R \subseteq A \times B$.

Серед бінарних відношень особлива роль належить *відображенням.* Це такі відношення $F \subseteq A \times B$, для яких $\forall x \in A \exists 1 y \in B \quad (x,y) \in F$. Поняття відображення є центральним у більшості розділів математики – математичному й функціональному аналізах, алгебрі, теорії диференційних рівнянь тощо. Ми повернемося до них у підрозд. 1.4.

Відношення вигляду $R \subseteq A \times A$ називається *бінарним відношенням* на множині A. Серед усіх бінарних відношень на A виділяють: *порожнє* ε, *тотальне* $A \times A$ та *одиничне* (*рівність*) $i_A = \{(a,a) : a \in A\}$.

Якщо $(a,b)\in R$, то кажуть, що елемент a *перебуває у відношенні* R з елементом b. Цей факт також записують aRb. *Образом* елемента a за відповідності R називається сукупність $R(a)=\{b\in B:(a,b)\in R\}$. R *визначено на* a, якщо образ $R(a)\neq\varnothing$. У протилежному випадку R *не визначено на* a. Образ підмножини $X\subseteq A$ за відповідності R визначається як $R(X)=\underset{x\in X}{\cup}R(x)$. Прообразом елемента b за відповідності R називається сукупність $R^{-1}(b)=\{a\in A:(a,b)\in R\}$. Прообраз підмножини $Y\subseteq B$ за відповідності R визначається як $R^{-1}(Y)=\underset{y\in Y}{\cup}R^{-1}(y)$. З відношеннями як множинами можна виконувати всі згадувані теоретико-множинні операції. Однак існують і специфічні операції. *Оберненням відношення* $R\subseteq A\times B$ називається відношення $R^{-1}=\{(b,a):b\in B\,\&\,a\in A\,\&\,aRb\}$. За означенням $R^{-1}\subseteq B\times A$.

Добутком відношень $P\subseteq A\times B$ та $Q\subseteq B\times C$ називається відношення

$$P\circ Q\overset{def}{=}\{(a,c):\exists b\in B(a\in A\,\&\,c\in C\,\&\,aPb\,\&\,bQc)\}.$$

За означенням $P\circ Q\subseteq A\times C$. Сама операція $\circ$ називається *множенням відношень.* Наведемо важливі властивості операції множення:

$(P\circ Q)\circ R=P\circ(Q\circ R)$ – асоціативність;

$R\circ i_B=i_A\circ R=R$ – діагональ;

$(P\circ Q)^{-1}=\left(Q^{-1}\circ P^{-1}\right)$ – обернення добутку.

$\underbrace{R^n\overset{def}{=}R\circ\ldots\circ R}_{n}$ називається n-м *степенем* відношення R.

Нехай $\Leftrightarrow$ є скороченням фрази "тоді й тільки тоді". Довільне відношення R на множині A називається:

1) *рефлексивним* $\Leftrightarrow\forall x\in A\ xRx\ \Leftrightarrow i_A\subseteq R$;

2) *симетричним* $\Leftrightarrow\forall x,y\in A(xRy\to yRx)\ \Leftrightarrow\ R^{-1}=R$;

3) *транзитивним* $\Leftrightarrow\forall x,y,z\in A(xRy\,\&\,yRz\to xRz)\Leftrightarrow\ RR\subseteq R$;

4) *антисиметричним* $\Leftrightarrow\forall x,y\in A(xRy\,\&\,yRx\to x=y)\Leftrightarrow R^{-1}\cap R\subseteq I$.

Із транзитивністю пов'язане таке важливе поняття, як транзитивне замикання. *Транзитивним замиканням* відношення $R\subseteq A\times A$

називається найменше транзитивне відношення $R^\infty \subseteq A \times A$, що містить R. Має місце

Лема 1.1. 1) $R^\infty = \bigcup_{i=1}^{\infty} R^i$; 2) R^∞ збігається з перетином усіх транзитивних надвідношень R.

Доведення. Дійсно, $\bigcup_{i=1}^{\infty} R^i$ містить R і є транзитивним, оскільки якщо $(a,b) \in R^n$ та $(b,c) \in R^m$, то $(a,c) \in R^{n+m}$. Будь-яке транзитивне надвідношення R має містити R^n. Це легко довести за індукцією (див. вправу 22). Отже, $\bigcup_{i=1}^{\infty} R^i$ – найменше транзитивне надвідношення R. Далі, перетин усіх транзитивних надвідношень R є транзитивним, а враховуючи, що в даному перетині бере участь і найменше з транзитивних надвідношень R, перетин збігається з ним ■

Наприклад, для $R = \{(n,n+1) : n \in N\}$ транзитивне замикання R^∞ збігається зі стандартним числовим порядком <.

1.2.5. ЕКВІВАЛЕНТНОСТІ Й ПОРЯДКИ

Довільні відношення R на множині A називаються:

1) *частковим порядком,* якщо R є рефлексивним, транзитивним і антисиметричним;

2) *лінійним порядком,* коли R є частковим порядком таким, що $\forall x,y \in A(xRy \vee yRx)$;

3) *еквівалентністю,* якщо R є рефлексивним, транзитивним і симетричним.

Частково впорядковану множину будемо називати *чумом,* або *ґратами,* а її порядок позначати $\prec$. Якщо $a \prec b$, то кажуть, що елемент a *менший ніж* b, а елемент b *більший ніж* a. Якщо при цьому $a \neq b$, то кажуть, що a *строго менший ніж* b, а b *– строго більший ніж* a.

Елемент a_0 чуму A називається *найбільшим* (*найменшим*), якщо $\forall a \in A(a \prec a_0)(\forall a \in A(a_0 \prec a))$. Найбільший і найменший елементи чуму називаються його *одиницею* (1) та *нулем* (0). Елемент a_0 чуму A називається *максимальним* (*мінімальним*), якщо $\forall a \in A \neg(a_0 \prec a\ ) \forall a \in A \neg(a \prec a_0)$. Отже, довільний елемент $a \in A$ або непорівнянний із максимальним елементом a_0, або менший за нього. *Вер-*

хнім (*нижнім*) *конусом* підмножини $B \subseteq A$ називається сукупність усіх таких елементів $a \in A$, що $\forall b \in B(b \prec a)\,(\forall b \in B(a \prec b))$. Найменший (найбільший) елемент верхнього (нижнього) конуса підмножини B називається її *верхньою* (*нижньою*) *гранню* й позначається $\sup B$ ($\inf B$).

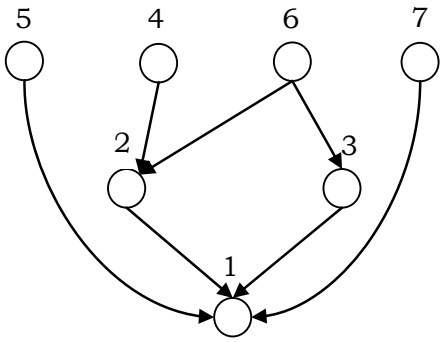
Чум можна подавати графічно за допомогою *діаграм Гессе*. У діаграмі Гессе елемент y більший за елемент x, якщо в ній існує шлях від y до x за стрілками, напрямленими вниз. З урахуванням транзитивності не має сенсу зображати всі стрілки між елементами.

Розглянемо приклад чуму й діаграму Гессе для нього.

Приклад 1.7. Діаграма Гессе.

Нехай множина $A = \{1,2,3,4,5,6,7\}$, а відношення порядку $x \prec y$ означає $y \,\%\, x = 0$, тобто y ділиться на x без залишку. Тут A – рефлексивний чум, але він не є лінійно впорядкованим (прості числа 3 і 5 непорівнянні).

Діаграма Гессе для чуму A має вигляд:



Тут 1 – найменший елемент в A; 4, 5, 6 і 7 – максимальні. Найбільшого елемента в чумі A немає. Верхня грань $\sup(2, 3) = 6$, нижня – $\inf(2, 3) = 1$, а верхньої грані $\sup(2, 5)$ не існує ■

Довільний чум A називається *індуктивним*, якщо будь-який його зростаючий ланцюг $a_1 \prec a_2 \prec \ldots a_n \prec \ldots$ має найменшу верхню грань $a = \bigcup_{i=1}^{\infty} a_i$. Відображення $f : A \to A$ називається *монотонним*, якщо для всіх $x, y \in D_f$ з умови $x \prec y$ випливає $f(x) \prec f(y)$, і *неперервним*, якщо для будь-якого ланцюга елементів $a_1 \prec a_2 \prec \ldots a_n \prec \ldots$ $f(\bigcup_{i=1}^{\infty} a_i) = \bigcup_{i=1}^{\infty} f(a_i)$. З неперервності відображення випливає його моно-

тонність (див. вправу 25). Найменший елемент в A (якщо він існує) називається його *нулем*. Елемент a називається *нерухомою точкою* відображення $f: A \to A$, якщо $f(a) = a$. Має місце

Теорема 1.1 (про нерухому точку). Нехай A – індуктивний чум із нулем ε, а $f: A \to A$ – неперервне відображення. Тоді елемент $a_0 = \bigcup_{i=0}^{\infty} f^i(\varepsilon)$ – найменша нерухома точка відображення f.

Доведення. За означенням $f(a_0) = \bigcup_{i=0}^{\infty} f^{i+1}(\varepsilon) = \bigcup_{i=1}^{\infty} f^i(\varepsilon) = \bigcup_{i=0}^{\infty} f^i(\varepsilon) = a_0$. Отже, a_0 є нерухомою точкою f. Те, що вона найменша, випливає з монотонності f (див. вправу 26) ■

Аналогічна теорема (про нерухому точку) має місце і для n-арних неперервних відображень $f: A^n \to A$ таких, що $f(x_1,\dots,x_n) = (f_1(x_1,\dots,x_n),\dots,f_n(x_1,\dots,x_n))$, де відображення f_i неперервні, а порядок на векторах визначається покомпонентно: $(a_1,\dots,a_n) \prec (a_1',\dots,a_n') \Leftrightarrow a_1 \prec a_1',\dots,a_n \prec a_n'$. Найменшою нерухомою точкою відображення f буде вектор $\bar{a}_0 = (\bigcup_{k=0}^{\infty} f_1^{(k)}(\varepsilon,\dots,\varepsilon),\dots,\bigcup_{k=0}^{\infty} f_n^{(k)}(\varepsilon,\dots,\varepsilon))$, де $f_i^{(0)}(x_1,\dots,x_n) = x_i$, $f_i^{(k+1)}(x_1,\dots,x_n) = f_i(f_1^{(k)}(x_1,\dots,x_n),\dots,f_n^{(k)}(x_1,\dots,x_n))$ для $i = \overline{1,n}$ (див. вправу 28).

Теореми про нерухому точку використовуються при дослідженні й розв'язанні рівнянь і систем рівнянь у різних алгебрах, у тому числі в алгебрах відношень, мов слів тощо.

Чум, в якому для кожних двох елементів існують верхня й нижня грані, називається *структурою*. Структура називається *повною* (*булевою алгеброю*), якщо верхня й нижня грані існують для будь-якої її підмножини. Прикладами повної структури є булеві алгебри: а) істиннісних значень $\{0,1\}$ з операціями диз'юнкції та кон'юнкції (взяття верхньої й нижньої граней $\sup$ та $\inf$, відповідно); б) множини, яку ми розглянули вище. В останньому прикладі нижню й верхню грані двох і довільної кількості множин задають операції перетину й об'єднання. Не випадково в літературі часто для довільних чумів терми $\sup(a,b)$ та $\inf(a,b)$ записують відповідно $a \cup b$ та $a \cap b$. Для повних структур виконуються всі закони булевої алгебри множин (див. вправу 24).

Фактор-класами еквівалентності R на A називаються сукупності еквівалентних між собою елементів A. Клас еквівалентності, якому

належить елемент a, позначається $[a]_R$, або просто $[a]$, для фіксованого R. Сукупність $A/R = \{[a] : a \in A\}$ називається *фактор-множиною* множини A, а її потужність – *індексом* R. Процес переходу від довільної множини до її фактор-множини називається *факторизацією* й відіграє надзвичайно важливу роль у процесах інформаційного моделювання систем.

*** Література для СР:** множини й відношення – [2, 66, 95, 97, 105, 123]; теорія нерухомої точки – [14, 52, 82, 118, 148].

Контрольні запитання та вправи

1. Дати означення множини.
2. Що таке характеристична й частково характеристична функції?
3. Сформулювати закони булевої алгебри множин.
4. Довести за допомогою діаграм Ейлера – Венна закони 3)–7) булевої алгебри.
5. Довести за допомогою діаграм Ейлера – Венна, що такі три твердження попарно еквівалентні: а) $A \subseteq B$; б) $A \cap B = A$; в) $A \cup B = B$.
6. Спростити вирази:

а) $\overline{A \cap \bar{B}} \cup B$;

б) $(A \cap B \cap C \cap \bar{X}) \cup (\bar{A} \cap C) \cup (\bar{B} \cap C) \cup (C \cap X)$,

в) $(A \cap B \cap X) \cup (A \cap B \cap C \cap X \cap Y) \cup (A \cap X \cap \bar{X})$.

7**. Нехай A – певна множина. Рівнянням у булевій алгебрі множин $B(A) = \{2^A; \cup, \cap, \setminus, \bar{}, \subseteq\}$ з одним невідомим X називається формула $R(A_1, \ldots, A_n, X) = \varnothing$ (*), де вираз у лівій частині – Ω_2-терм булевої алгебри множин, побудований за допомогою символів операцій алгебри з констант-підмножин $A_1, \ldots, A_n \subseteq A$ і невідомого X. Наприклад, $\overline{B \cup C \cap X} = \varnothing$. З'ясувати, за яких умов рівняння (*) має розв'язок і знайти всі його розв'язки [123].

8. Що таке індексована множина?
9. У чому полягає різниця між сильним і слабким індексуванням?
10. Дати визначення прямої суми множин.
11. Навести приклади застосувань слабкого й сильного індексувань.
12. Чим відрізняються: а) послідовності від підпослідовностей і кортежів; б) кортежі від векторів?
13. Що таке відображення?

14. У чому полягає різниця між відображенням і відповідністю? Проілюструвати прикладами.
15. Яке відношення називається: а) рефлективним; б) симетричним; в) транзитивним; г) антисиметричним; д) частковим порядком; е) лінійним порядком; є) еквівалентністю; ж) структурою, повною структурою? Навести відповідні приклади.
16. Дати означення монотонного й неперервного відображень. Навести відповідні приклади.
17. Що таке нерухома точка відображення?
18. Що таке фактор-множина та індекс еквівалентності?
19. Що таке факторизація множини?
20. Визначимо відношення $\square$ на словах: $u \square v$, якщо слово u є префіксом слова v. Чи буде це відношення: еквівалентністю, лінійним або частковим порядком, структурою на множині всіх слів X^* у алфавіті X?
21. Чи буде сукупність $\{a, aab, aa, bb, bbc, ab, bbb, d; \square\}$ чумом? Якщо так, то побудувати відповідну діаграму Гессе.
22. Нехай R – довільне бінарне відношення. Довести, що будь-яке транзитивне надвідношення R містить R^n, $n \geq 0$.
23. Показати, що булева алгебра множин є повною структурою відносно включення.
24. Довести закони 1)-7) булевої алгебри множин для: а) алгебри істиннісних значень з операціями кон'юнкції та диз'юнкції й запереченням; б) довільної повної структури з нулем і одиницею, доповнення $\bar{a}$ елемента a в якій визначається системою рівностей $a \cap \bar{a} = 0$ та $a \cup \bar{a} = 1$.

25*. Показати, що з неперервності відображення випливає його монотонність.

26*. Показати, що нерухома точка a_0 з доведення теореми 1.1 є найменшою. Скористатися монотонністю відображення f.

27. Нехай A – індуктивний чум із нулем ε. Показати, що декартовий степінь A^n є індуктивним чумом із нулем $(\varepsilon, \ldots, \varepsilon)$.

28*. Довести теорему про нерухому точку для n-арних неперервних відображень.

1.3. Функції як обчислювальні процедури

- Функції-процедури
- Традиційні способи специфікації функцій
- Алгебричні специфікації
- Функції як обчислювальні процедури

Ключові слова: *функція-процедура, сюр'єкція, ін'єкція, бієкція, графік функції, композиція рангу 1, композиції множення, розгалуження, обхід, ітерація, повторення, недетермінований вибір, детермінований вибір, регулярна функція, перестановка, ототожнення й параметризація аргументів, введення фіктивних аргументів, підстановка (суперпозиція), ітерації, повторення, регулярна n -арна операція, обчислювальний простір, обчислювальна процедура, функція керування, обчислення за процедурою, функція як обчислювальна процедура.*

Поняття функції належить до найважливіших понять інформатики. Відомі два різні тлумачення даного поняття: процедурне[16] й теоретико-множинне. У першому випадку функція (від лат. functio – виконання, звершення) розглядається як спосіб для отримання певного результату за аргументами, у другому – як синонім теоретико-множинного відображення.

Ці два терміни ввійшли в математичну практику не випадково. Вони мають принципово різний смисловий відтінок. Коли говорять про певне відображення f, то цікавляться тільки тим, чи є пара (a,b) елементом підмножини $f \subset A \times B$, тобто чи існує відповідний зв'язок між елементами $a \in A$ та $b \in B$ (екстенсіональний підхід). Питання про природу цього зв'язку, яким чином він забезпечується, вважається непринциповим. У випадку ж функції-процедури f її теж розглядають як можливість поставити у відповідність елементам множини A певні елементи множини B, але на перший план виходить спосіб реалізації такого зіставлення при визначенні f (інтенсіональний підхід).

Процедурний підхід сформувався історично першим (термін запровадив Г. Лейбніц). Однак він був майже витіснений теоретико-множинним на початку XX ст. у зв'язку з тотальним переходом

[16] Процедура – офіційно встановлений чи узвичайнений порядок здійснення чого-небудь, низка певних цілеспрямованих дій.

математики на нову основу. Достатньо послатись на самі назви сучасних класичних математичних дисциплін – функціональний аналіз, теорія функцій комплексних змінних тощо. Сьогодні процедурний підхід поступово повертає свої позиції завдяки бурхливому розвитку конструктивної математики та інформатики. Поняття функції-процедури є центральним у теорії алгоритмів і програмології, мовах програмування тощо.

Даний підрозділ присвячено функціям як процедурам. Ключовими будуть засоби подання таких функцій.

1.3.1. ФУНКЦІЇ-ПРОЦЕДУРИ

Нехай U – довільна множина.

Функція f – це процедура, яка може бути застосована до одного або кількох елементів (аргументів) множини U з метою знаходження інших елементів (результату) з U. Застосування функції полягає у проведенні обчислення – послідовності дій, яке завершується побудовою результату функції.

При цьому функція не завжди гарантує однозначність і навіть існування результату. Це означає, що застосування функції до аргументів може не привести до результату, а застосування функції повторно до тих самих аргументів може приводити до іншого результату. У цьому сенсі всі функції розподіляються на *детерміновані* й *недетерміновані*. Перші гарантують однозначність своїх результатів, усі решта належать до других.

Увага! Надалі нас будуть цікавити в основному детерміновані функції. Випадки, коли це не так, будуть спеціально зазначатись ►

Існує тісний взаємозв'язок між функціями й алгоритмами. Можна розглядати загальне поняття – функцію та його конкретизацію – алгоритм. Будь-який алгоритм є функцією-процедурою, а чи буде остання алгоритмом, залежить від способу її визначення. Це обговорюватиметься детально в підрозд. 2.1.4.

Терми $f(x)$, fx та xf використовуються для запису результату застосування функції f до аргументу x. Ураховуючи, що результат такого застосування за означенням може бути відсутній, будемо додавати до вказаних термів праворуч символ $\downarrow$, якщо даний результат гарантовано існує. З кожною функцією f пов'язані дві області – *визначення* $D_f = \{x \in U : f(x)\downarrow\}$ і *значень* $E_f = \{y \in U : y = f(x) \& x \in D_f\}$. Щоб показати цей зв'язок, функцію f подають у вигляді $f : D_f \to E_f$

і називають *усюди визначеною* на D_f. На практиці буває простіше описати не самі області визначення та значень даної функції, а певні їхні надмножини, наприклад T і R. Щоб підкреслити, що функція f розглядається над цими загальнішими областями, її записують як $f: T \to R$ і називають *частковою*, а пару (T, R) – *типом* функції. Будемо вважати вираз $f(x)$ беззмістовним (тобто рівним #) для всіх аргументів поза областю визначеності функції. *Графіком функції f* називається відповідність Γ_f між множинами D_f і E_f, що складається з усіх можливих пар вигляду (x, fx), тобто $\Gamma_f = \{(x, fx) : x \in D_f\}$. Графіки функцій використовуються для їх екстенсіонального опису. Функції $f: A \to B$ та $g: C \to D$ називаються *еквівалентними* $(f \cong g)$, якщо $\forall x \in A \cup C \quad fx = gx$ (*). Наприклад, функції $f_0: N \to N$ та $f: N \to N$, що визначаються термами $f_0 x \overset{def}{=} x / x$ та $fx \overset{def}{=} 1$, нееквівалентні. Їх розмежовує точка $x = 0$. Дійсно, $f_0 0 = \#$, а $f0 = 1$. Якщо область визначення D_g функції g є надмножиною області визначення D_f функції f $(D_f \subset D_g)$ і виконується умова $\forall x \in D_f (fx = gx)$, то кажуть, що функція g є *еквівалентним накриттям* f, або просто *накриттям* f. В останньому прикладі f є накриттям f_0. Еквівалентні функції називаються *рівними*, якщо їхні типи збігаються. Усюди визначена функція $f: A \to B$ називається:

1) *однозначною* (*ін'єкцією*) $\Leftrightarrow \forall x, y \in A\ (fx = fy \to x = y)$;
2) *сюр'єкцією* $\Leftrightarrow \forall y \in B \exists x \in A\ y = fx$;
3) *взаємооднозначною* (*бієкцією*), якщо вона є ін'єкцією та сюр'єкцією.

1.3.2. ТРАДИЦІЙНІ СПОСОБИ СПЕЦИФІКАЦІЇ ФУНКЦІЙ

У програмуванні означення (дескрипції) функцій називають *специфікаціями*. Розрізняють теоретико-множинні та процедурні специфікації. Перші пов'язані з описом графіків функцій, а другі – безпосередньо з описом процедур. Зазвичай з опису графіка функції не випливає її специфікація як процедури. Не завжди за ним, наприклад, можна практично віднайти значення функції на тому чи іншому аргументі чи зробити це ефективно. Однак наявність такого опису є важливою передумовою для подальшого уточнення функції, тому ро-

зробник інформаційних систем повинен володіти обома способами специфікації – як теоретико-множинним, так і процедурним.

Оскільки графіки детермінованих функцій є відображеннями, то для їхньої специфікації можна використовувати загальні способи специфікації останніх.

Позначатимемо відображення малими грецькими літерами $\alpha, \beta, \gamma, \ldots$. За означенням образ відображення $\alpha(x)$ завжди є одноелементним, тому будемо ототожнювати його із цим елементом, а сам елемент називати значенням відображення α на x і позначати, як і у випадку функцій, αx та $x\alpha$. Продовжуючи аналогію з функціями, відображення $\alpha \subseteq A \times B$ також будемо записувати як $\alpha : A \to B$ і говорити про області визначення й значення відображення, а також про ін'єктивні, сюр'єктивні та бієктивні відображення. Перші два ще називають *відображеннями в* та *відображеннями на*, відповідно.

Розпочнемо розгляд зі *словесного* (нематематичного) способу подання відображень. Він може вважатися найбільш універсальним і полягає в тому, що зв'язок між аргументами й результатами відображення формулюється в термінах природної мови. Словесні специфікації використовуються насамперед на етапах аналізу вхідних систем, обміну проміжними результатами та інформацією в процесі роботи над проектами тощо. Однак за всієї зручності вони мають і суттєві недоліки. Існує небезпека неповноти, нечіткості й неоднозначності таких специфікацій. До того ж виникають серйозні проблеми при спробі їхньої процедуризації, тобто при намаганні перейти від словесно поданого відображення до еквівалентної йому функції, особливо при автоматизації цих процесів.

Одним із найпростіших і найуживаніших на практиці способів подання відображень є *табличний*. Нехай область визначення відображення α скінченна й має вигляд $D_\alpha = \{x_1, x_2, \ldots x_n\}$, а $E_\alpha = \{y_1, y_2, \ldots y_n\}$, $y_i = \alpha x_i$ – область значень α. Тоді відображення α можна задати у вигляді таблиць:

$$\begin{pmatrix} x_1 x_2 \ldots x_n \\ y_1 y_2 \ldots y_n \end{pmatrix}$$

або

x_1	x_2	…	x_n
y_1	y_2	…	y_n

Лише раз створивши й запам'ятавши таку таблицю, далі можна вже не обчислювати значення відображення, а його результати брати безпосередньо з таблиці. Таким чином, маємо випадок, коли теоретико-множинне подання прямо входить до специфікації відповідної функції. Воно дуже зручне при проведенні обчислень, зокрема без використання

технічних засобів. Досить пригадати відомі ще зі школи таблиці Брадіса для обчислення елементарних функцій – степенів, коренів, десяткових логарифмів, тригонометричних функцій тощо. Таблиці широко використовуються в обчислювальній математиці й програмуванні, наприклад при так званій інтерполяції функцій, яка дозволяє за відомими табличними значеннями функцій у певних фіксованих точках наближено обчислювати їхні значення на довільних аргументах. Моделі таблиць є основним засобом для організації та збереження інформації в пам'яті ЕОМ.

Інший відомий спосіб подання відображень – *геометричний*. Він полягає в розташуванні за певним правилом (масштабом) усіх елементів областей визначення та значень на двох перпендикулярних прямих, які перетинаються в точці O і називаються осями абсцис (вісь OX) і ординат (вісь OY). При цьому аргументи розташовуються на першій осі, а результати – на другій. Тоді кожній парі $(x,\alpha x)$ відображення α відповідає точка на квадратній площині, якій належать осі OX та OY (рис. 1.3).

Геометричний спосіб використовується для наближеного подання й обчислення значень функцій. Він зручний при вивченні інтегральних властивостей функцій, таких як монотонність, опуклість тощо.

Для подання функцій можуть застосовуватись і графи, зокрема дерева (див. підрозд. 1.4.4). Такий спосіб називається *графічним*. Типовим прикладом є кодові дерева, які за двійковим кодом дозволяють знайти відповідний символ кодової таблиці, графи скінченних автоматів тощо. На рис. 1.4 зображено кодове дерево для двійкових кодів шістнадцяткових цифр.

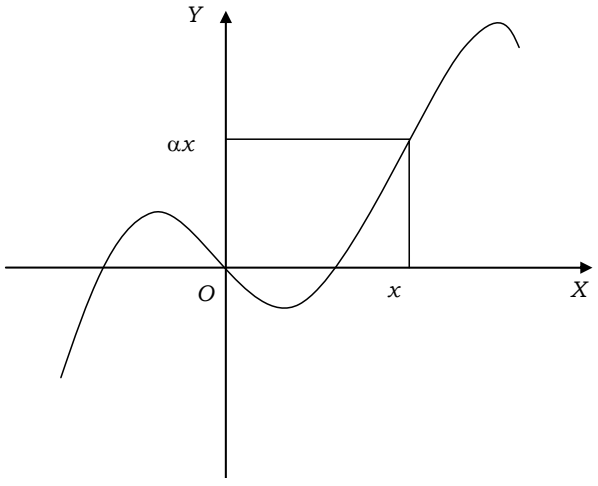


Рис. 1.3

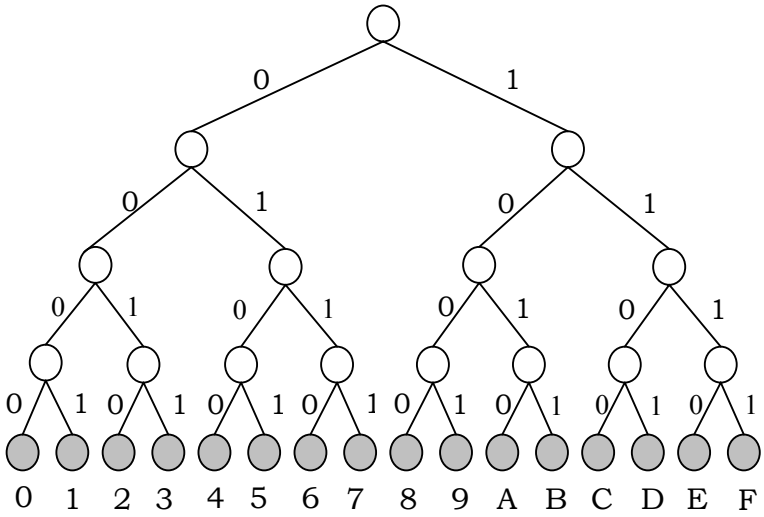


Рис. 1.4

Зазначимо, що конкретні табличні, геометричні й графічні специфікації відображень тим цікавіші на практиці, чим ефективніше вони алгоритмізуються. Наприклад, щоб знайти значення табличного відображення для даного аргументу, достатньо простим перебиранням знайти в таблиці стовпчик із даним аргументом і взяти другий його компонент. Однак якщо рядків багато, то це може бути не дуже зручно. У таких випадках пошук можна зробити більш прийнятним, якщо аргументи в таблиці розташувати в певному лінійному порядку (див. підрозд. 4.1.2).

1.3.3. АЛГЕБРИЧНІ СПЕЦИФІКАЦІЇ

Універсальним і потужним засобом подання відображень є *алгебричний*. Розрізняють явні й неявні алгебричні специфікації. В *явних* відображення подають як конструктивні об'єкти. Для цього фіксують певні сукупності атомних (базових) відображень і конструкторів на відображеннях (композицій), що дозволяють будувати з базових відображень складніші. Наприклад, композиція суперпозиції дозволяє будувати складні тригонометричні функції на основі базових тригонометричних функцій і арифметичних операцій. Нові функції подаються за допомогою спеціальних термів – алгебричних виразів.

Розглянемо певний набір класичних композицій, що найчастіше вживаються для явної специфікації відображень. Вони цікаві тим, що зберігають процедурність аргументів, тобто якщо їх застосувати до графіків

не просто відображень, а функцій, то в результаті будуть отримані теж графіки певних функцій. Спочатку розглянемо композиції унарних відображень. *Композицією рангу* 1 *над множиною* A будемо називати будь-яку операцію вигляду $K: A \times ... \times A \times (A \to A) \times ... \times (A \to A) \to (A \to A)$. Звичайні n-арні операції на A належать до композицій рангу 0. Дві композиції *рангу* 1 уже зустрічалися вище – *множення* $(\circ)$ та *обернення* $(^{-1})$[17].

Зазначимо, що результат обернення є відображенням тільки для сюр'єктивних відображень. Нехай p – предикат на A, $\alpha: A \to A, \beta: A \to B, \gamma: A \to C$ – довільні відображення.

Обмеженням відображення β за предикатом p називається відображення $\beta\downarrow_p \overset{def}{=} \beta \cap p(A) \times B$. Якщо $p(a) = 1$, то результатом обмеження $\beta\downarrow_p$ на елементі a буде значення βa, якщо ж $p(a) = 0$, то значення не визначене.

Розгалуженням за предикатом p відображень β, γ називається відображення $(p \to \beta \mid \gamma) = \beta\downarrow_p \cup \gamma\downarrow_{\neg p}$. Таким чином, якщо $p(a) = 1$, то результатом розгалуження $(p \to \beta \mid \gamma)$ на елементі a буде значення βa, якщо ж $p(a) = 0$, то значення γa.

Обходом за предикатом p відображення β називається відображення $(p \to \beta) = \beta\downarrow_p \cup i_{\neg p}$. За межами області істинності предиката p обхід $(p \to \beta)$ діє як одиничне відображення.

Звуженням відображення β за предикатом p на B називається відображення $\beta\uparrow^p: A \to B$ таке, що $\beta\uparrow^p = \beta \cap A \times p(B)$.

Позначимо через α_p^* замикання відображення α відносно предиката p на A, тобто $\alpha_p^* = ((\alpha\downarrow_p)\uparrow^p)^*$.

Ітерацією за предикатом p відображення α називається відображення $\{p \to \alpha\} \overset{def}{=} ((\alpha_p)^* \circ \alpha)\uparrow^{\neg p}$. За означенням $\{p \to \alpha\}(a) = b$, якщо існує $m \geq 0$, за якого послідовність $b_0 = a, b_i = f(b_{i-1})$, $i = \overline{1,m}$, є такою, що $b_m = b$ і виконується умова $\overset{m-1}{\underset{i=0}{\&}} p(b_i) \,\&\, \neg p(b)$.

[17] Обернення не зберігає процедурність.

Приклад 1.8. Функція $\mathrm{Gcd} = \{q \to (p \to \alpha \mid \beta)\} \circ \mathrm{pr}_1$ обчислює найбільший спільний дільник двох натуральних чисел, де $\alpha(n,m) = (n-m,m)$, $\beta(n,m) = (n,m-n)$, $p(n,m) = m < n$ та $q(n,m) = n \neq m$ для всіх $n,m \in N$. Про коректність функції Gcd див. прикл. 1.10 ■

Повторенням за предикатом p відображення α називається відображення $[p \to \alpha] \overset{def}{=} \alpha\{p \to \alpha\}$.

Недетермінованим вибором відображень $\alpha_1,...,\alpha_n : A \to A$ за предикатами $p_1,...,p_n$ називається відповідність $(p_1 \to \alpha_1 \,||\, \ldots \,||\, p_n \to \alpha_n) \overset{def}{=} \bigcup_{i=1}^{n} \alpha_i \downarrow_{p_i}$.

Предикат p_i іноді називають *охороною* відображення α_i. Якщо охорони $p_1,...,p_n$ попарно не перетинаються, то результат недетермінованого вибору $(p_1 \to \beta_1 \,||\, \ldots \,||\, p_n \to \beta_n)$ буде теж відображенням.

Приклад 1.9. Нехай $\mathrm{sign}(x) = (x < 0 \to -1 \,||\, x = 0 \to 0 \,||\, x > 0 \to 1)$ та $\pm\sqrt{x} = (x > 0 \to \sqrt{x} \;||\; x{>}0 \to -\sqrt{x})$. Тоді перший вибір є відображенням, а другий – ні ■

Нехай $p_1,...,p_n$ – довільні охорони, деякі з яких виділені як заключні. *Детермінованим вибором* відображень $\alpha_1,...,\alpha_n : A \to A$ за охоронами $p_1,...,p_n$ називається відображення

$$(p_1 \to \alpha_1 \mid \ldots \mid p_n \to \alpha_n) \overset{def}{=} \bigcup_{i=1}^{n} \Big(\&_{k=1}^{i-1} \neg p_k(a) \,\&\, p_i(a) \to \gamma_i \mid \#\Big),$$

де $\gamma_i = \alpha_i$, якщо охорона p_i є заключною, і $\gamma_i = \alpha_i \ldots \alpha_n$, якщо ні. Якщо всі умови на певному елементі одночасно хибні, то значення вибору на ньому не визначене.

Наведені композиції разом із множенням і логічними зв'язками називаються *регулярними*[18]. Як уже зазначалось, вони зберігають функціональність аргументів. Функції, що отримують за їхньою допомогою, теж називаються регулярними. Процедури обчислення значень регулярних функцій формулюються досить просто. Наприклад, щоб знайти результат добутку fg на елементі a, достатньо за правилом g знайти результат $g(a)$ і до нього застосувати правило функції f. Щоб

[18] Обернення не є регулярною композицією

знайти результат b ітерації $\{p \to f\}$ на елементі a, необхідно спочатку перевірити предикат p на a. Якщо він виконується, то необхідно обчислити за правилом f значення $a_1 = fa$, у протилежному випадку покласти $b = a$; потім перевірити предикат p на a_1. Якщо він виконується, то необхідно обчислити за правилом f значення $a_2 = fa_1$, у протилежному випадку покласти $b = a_1$ тощо. Даний процес побудови послідовності $a_0 = a$, $a_1 = fa_0$, $a_2 = fa_1, \ldots$ або завершиться на якомусь n-му кроці, $n \geq 0$, з результатом $b = a_n$, або буде продовжуватись до нескінченності з беззмістовним результатом. Аналогічно формулюються правила й для решти композицій.

Нехай A – довільна множина. Відображення вигляду $A^n \to A$ називаються *n-арними операціями.* Визначимо регулярну алгебру n-арних операцій.

Нехай $\alpha = (i_1, i_2, \ldots, i_n)$ – довільна перестановка чисел $(1, 2, \ldots, n)$. Композиція *перестановки аргументів* довільній операції $f : A^n \to A$ ставить у відповідність операцію $f^\alpha : A^n \to A$ таку, що $f^\alpha(a_1, \ldots, a_n) = f(a_{i_1}, \ldots, a_{i_n})$.

Композиція *ототожнення аргументів* довільній операції $f : A^n \to A$ ставить у відповідність операцію $f^\tau : A^n \to A$ таку, що

$$f^\tau(a_1, a_{2,}, \ldots, a_n) = f(a_1, a_1 \ldots, a_n).$$

Композиція *параметризації аргументів* параметру $a \in A$ й операції $f : A^n \to A$ ставить у відповідність операцію $f_a : A^{n-1} \to A$ таку, що

$$f_a(a_2, \ldots, a_n) = f(a, a_{2,}, \ldots, a_n).$$

Композиція *введення фіктивних аргументів* довільній операції $f : A^n \to A$ ставить у відповідність операцію $f^\phi : A^{n+1} \to A$ таку, що

$$f^\phi(a_1, a_2 \ldots, a_n, a_{n+1}) = f\left(a_2, \ldots, a_{n+1}\right).$$

Композиція *підстановки* (*суперпозиції*) операції $f : A^m \to A$ й операціям $g_1, \ldots, g_m : A^n \to A$ ставить у відповідність операцію

$$S(f, g_1, \ldots, g_m) : A^n \to A : S(f, g_1, \ldots, g_m)(a_1, \ldots, a_n) = f(g_1(a_1, \ldots, a_n), \ldots, g_m(a_1, \ldots, a_n)).$$

Нехай $pr_i^n : A^n \to A$, $1 \leq i \leq n$ – функція проектування за i-м компонентом, тобто $\mathrm{pr}_i^n(a_1, \ldots, a_n) = a_i$ для $\forall < a_1, \ldots, a_n > \in A^n$. Підстановку $S\left(g, g_1, pr_n^2 \ldots, pr_n^m\right)$ умовимось позначати коротко: $S(g, g_1)$.

Позначимо через f_B^* замикання операції $f: A^n \to A$ за першим компонентом відносно підмножини $B \subseteq A$, тобто $f_B^* = \bigcup_{n=0}^{\infty} f^n$, де

$$f^0 = E^{n+1} = \{(< a, a_2, ..., a_n >, b) \in A^n \times A : a = b\},$$

$$f^n(a_1, ..., a_n) = f \downarrow_{B \times A^{n-1}} (f^{n-1}(a_1, ..., a_n), a_2 ..., a_n),\ n > 0.$$

Композиція *ітерації* предикату p на A^n та операції $f: A^n \to A$ ставить у відповідність операцію $\{p \to f\} = ((f_{p(A^n)})^*) \uparrow^{\neg p(A^n)}$. За означенням $\{p \to f\}(a_1, ..., a_n) = b$, якщо існує $m \geq 0$, за якого послідовність $b_0 = a_1,\ b_i = f(b_{i-1}, a_2, ..., a_n),\ i = \overline{1, m}$, є такою, що $b_m = b$ і виконується умова $\&_{i=0}^{m-1} p(b_i, a_2, ..., a_n) \& \neg p(b, a_2, ..., a_n)$.

Композиція *повторення* предикату p на A^n і операції $f: A^n \to A$ ставить у відповідність операцію $[p \to f] = S(\{p \to f\}, f)$.

Наведені вище композиції n-арних операцій разом із композицією розгалуження й логічними зв'язками називаються *регулярними*. Позначимо їхню сукупність Ξ_0.

Як і у випадку регулярних композицій унарних відображень, регулярні композиції n-арних операцій теж зберігають процедурність результуючої операції, тобто якщо їхні аргументи є функціями, то досить просто сформулювати правило для обчислення й результуючої операції. Нехай $F = \{f_1 ..., f_n\}$ та $P = \{p_1 ..., p_m\}$ – довільні сукупності операцій і предикатів на A^n. Операції, отримані з елементів $F \cup P$ за допомогою регулярних композицій, називаються *регулярними n-арними операціями* над базисом $F \cup P$.

Регулярні n-арні операції над достатньо простими базисами є універсальними алгоритмічними системами (див. вправу 15).

Для подання регулярних операцій використовують регулярні терми у спрощеній префіксній та інфіксній формах. Наприклад, похідну регулярну операцію $S(\times, 2, \sin(x))$ записують у вигляді звичайного інфіксного терму $2\sin x$, операцію $[x > 0 \to S(\ln, x/2) | 0]$ – як $[x > 0 \to \ln x / 2 | 0]$ тощо.

Неявні засоби специфікують відображення (відповідності) за допомогою спеціальних формул ПЧП чи формул числень вищих поряд-

ків, які описують зв'язок між аргументами й значеннями відображень у певній алгебрі. Такі формули називаються *рівняннями* або *системами рівнянь*. Це можуть бути як звичайні рівняння з предметними змінними, так і (що важливіше для застосувань) функціональні, у яких невідомими є не предметні змінні, а функціональні зі значеннями – відображеннями чи відповідностями. Наприклад, звичайне рівняння $xy = 1$ із невідомим y визначає відображення-гіперболу, а рівняння $y^2 - x^2 = 1$ відносно y – двозначну відповідність $y = \pm\sqrt{x^2 + 1}$. Функціональні рівняння зазвичай мають багато розв'язків. Тому окрім проблеми пошуку розв'язків рівняння існує й додаткова проблема – вибору необхідного розв'язку.

У теорії найбільш вивчені канонічні функціональні рівняння вигляду $f = \Phi(f)$, де Φ – певна композиція на множині відображень**.** Наприклад, функціональне рівняння $f(x) = \left(x > 100 \to x - 10 \mid f\left(f\left(x + 11\right)\right)\right)$, де x пробігає цілі числа, а f – функціональна змінна типу $Z \to Z$, має кілька розв'язків, найменшим серед яких (відносно теоретико-множинного включення) є відображення $F_p(x) = \left(x > 100 \to x - 10 \mid 91\right)$, відоме як 91-ша функція Мак-Картні. Найменшим розв'язком іншого рівняння – $f(n) = \left(n = 0 \to 1, n \times f\left(n - 1\right)\right)$, де n пробігає натуральні числа, а f – функціональна змінна типу $N \to N$, – є функція-факторіал $n!$.

Як ще один приклад розглянемо канонічну систему рівнянь у регулярній алгебрі відповідностей: (*) $x_i = R_i(x_1, \ldots, x_n), i = \overline{1,n}$, де $R_i(x_1, \ldots, x_n)$ – регулярні вирази відносно невідомих відповідностей $x_1, \ldots, x_n$ і, можливо, деяких заданих відповідностей, побудовані за допомогою чотирьох композицій: множення $\circ$, об'єднання $\cup$ (частковий випадок недетермінованого вибору двох відповідностей $f \cup g = (1 \to f \mid\mid 1 \to g)$), ітерації * за тотожно-істинним предикатом і суперпозиції. Сукупність $B(A \times A)$ усіх відповідностей на множині A є чумом відносно звичайного теоретико-множинного включення: $f_1 \prec f_2 \Leftrightarrow xf_1y \Rightarrow xf_2y$. Найменшою верхньою гранню зростаючого ланцюга відповідностей є їхнє множинне об'єднання, а нулем – порожня відповідність ε. Композиції $f_1 \circ f_2, f_1 \cup f_2$ та f^* неперервні за кожним з аргументів (див. вправу 16). Ураховуючи, що

суперпозиція неперервних відображень неперервна, до системи (*) можна застосувати теорему про нерухому точку (див. підрозд. 1.2.5) і отримати таку теорему.

Теорема 1.2. Система рівнянь (*) має найменший розв'язок $(\bigcup_{k=0}^{\infty} R_1^{(k)}(\varepsilon,\ldots,\varepsilon),\ldots,\bigcup_{k=0}^{\infty} R_n^{(k)}(\varepsilon,\ldots,\varepsilon))$, де за означенням $R_i^{(0)}(x_1,\ldots,x_n)=x_i$, $R_i^{(k+1)}(x_1,\ldots,x_n)=R_i(R_1^{(k)}(x_1,\ldots,x_n),\ldots,R_n^{(k)}(x_1,\ldots,x_n))$ для $i=\overline{1,n}.$

Наслідок. Розв'язки лінійних функціональних рівнянь з одним невідомим x вигляду $x=f\circ x\cup g$ та $x=x\circ f\cup g$ є регулярними відносно коефіцієнтів f і g.

Доведення. З теореми 3.1 випливає, що найменший розв'язок x_0 першого рівняння збігається з $\bigcup_{k=0}^{\infty} R^{(k)}(\varepsilon)$, де

$$R(x)=f\circ x\cup g, R^{(k)}(x)=f^k\circ x\cup f^{k-1}\circ g\cup\ldots\cup f\circ g\cup g=$$

$$=f^k\circ x\cup(\bigcup_{i=0}^{k-1} f^i\circ g).$$ Тоді $x_0=\bigcup_{k=0}^{\infty} f^k\circ g=(\bigcup_{k=0}^{\infty} f^k)\circ g=f^*\circ g$.

Аналогічно для другого рівняння отримуємо найменший розв'язок $x_0=g\circ f^*$ ■

1.3.4. ФУНКЦІЇ ЯК ОБЧИСЛЮВАЛЬНІ ПРОЦЕДУРИ

Повернемось до означення функції та спробуємо його формалізувати. Як випливає з означення, для уточнення поняття функції необхідно (і достатньо) уточнити поняття процедури й обчислення. Розпочнемо з обчислень.

Виберемо майже гранично можливий рівень абстракції (який умовно можна було б назвати *пропозиційним*): усі об'єкти трактуються як теоретико-множинні чорні скриньки, усередину яких ми не заглядаємо.

Кожне окреме обчислення не є одноразовим актом: розпочавшись у певний фіксований момент часу з певною початковою інформацією на вході, воно розгортається в часовому просторі й супроводжується виконанням певних елементарних дій. Обчислення може бути скінченним або нескінченним у часі, але в обох випадках існує механізм для визначення його результату або визнання беззмістовним. Таким чином, можна констатувати, що

обчислювальний простір, в якому розгортаються обчислення, є, як мінімум, двовимірним – із часовою та інформаційною координатами.

Нехай $<\Gamma,<>$ – довільна лінійно впорядкована сукупність, яку будемо трактувати як часовий простір[19]. Позначимо τ^+ та τ^- безпосередньо наступний і попередній моменти часу відносно моменту τ. Вважається, що в часовому просторі немає найменшого й найбільшого елементів.

Інформаційну складову процесів обчислень подають *стани.* На даному етапі нас не буде цікавити їхня структура. Нехай S – довільна множина станів.

Двійка $\Pi = \langle \Gamma, S \rangle$ називається *обчислювальним простором,* а пари $(\tau, s) \in \Gamma \times S$ – *конфігураціями* простору.

Нехай $\tau,\tau_1,\tau_2 \in \Gamma$. Покладемо $\Gamma_\tau = \{\tau' \in \Gamma \mid \tau \le \tau'\}$, $\Gamma_\tau^+ = \Gamma_\tau \setminus \{\tau\}$, $\Gamma_\tau^- = \Gamma \setminus \Gamma_\tau$, $[\tau_1,\tau_2] = \{\tau : \tau_1 \le \tau \le \tau_2\}$. *Абстрактним процесом* в обчислювальному просторі Π із початком у момент часу τ називається довільне відображення вигляду $p : \Gamma_\tau \to S$, а *абстрактним процесом на інтервалі* $[\tau_1,\tau_2]$ – довільне відображення вигляду $p : [\tau_1,\tau_2] \to S$. Стани $p\tau$ ($p\tau_1$), $p\varsigma, \tau < \varsigma$ ($\tau_1 < \varsigma < \tau_2$) та $p\tau_2$ називаються *початковим, проміжними* й *заключним* станами процесу p.

Візьмемо довільну функцію $f : S \to S$ та її аргумент s. Якщо процес p розпочинається у стані s і його заключний стан збігається з fs, то кажуть, що він *подає значення* функції f на аргументі s. Зафіксуємо певний момент часу τ_0 як початковий. Сукупність процесів Ξ, що розпочинаються в момент τ_0: 1) *подає* функцію f, якщо для кожного її аргументу з області визначення ця сукупність містить процес, який подає її значення на даному аргументі; 2) *строго подає* f, якщо вона подає f і не містить "зайвих" процесів (тих, що не подають значення f).

Тепер серед усіх абстрактних процесів у обчислювальному просторі Π виділимо процеси-обчислення та їхні сукупності. Для цього зафіксуємо певну сукупність $\Delta = \{f_i : S \to S \mid i \ge 0\}$ *елементарних операцій* на станах і визначимо *функцію керування* процесами. Остання пов'язує з кожним моментом часу процесу одне або кілька можливих еле-

[19] У деяких випадках часовий простір може мати складнішу структуру.

ментарних перетворень його поточного стану. Покладемо її як таку, що має вигляд $\delta : \Gamma \times S \to 2^{\Delta}$ [20].

Нехай $\Pi = \langle \Gamma, S \rangle$ – довільний обчислювальний простір.

Обчислювальною процедурою над простором Π із сукупністю елементарних перетворень Δ і функцією керування δ називається трійка $P = \langle \Pi, \Delta, \delta \rangle$.

Кожна обчислювальна процедура породжує певну сукупність обчислювальних процесів у просторі Π. Нехай $\tau \in \Gamma$ та $s \in S$ – довільні момент часу і стан. *Обчислення* $p : \Gamma_\tau \to S$ за процедурою P із початком у момент часу τ і початковим станом s_0 визначається рекурентно: $p\tau = s_0$ і для всіх $\varsigma > \tau$ $p\varsigma = f_\varsigma(p\varsigma^-)$, де $f_\varsigma \in \delta(\varsigma, p\varsigma^-)$. Функція керування за поточним моментом часу ς і станом $p\varsigma^-$ у попередній момент часу визначає сукупність елементарних операцій, які можуть бути застосовані для отримання стану процесу в момент ς. Вважатимемо, що обчислення p у момент часу ς *переходить* від попереднього стану $p\varsigma^-$ до наступного стану $p\varsigma$ і позначатимемо цей факт $p\varsigma^- \to_P p\varsigma$. Варіантів таких переходів може бути кілька, оскільки функція переходу багатозначна. Насправді їх може бути: а) два і більше (у випадку *недетермінованих* процедур); б) рівно один (у випадку *детермінованих* процедур); в) жодного (обчислення зупиняється).

Обчислення p за процедурою P на інтервалі $[\tau_1, \tau_2]$ визначається як обмеження відповідного обчислення p за процедурою P із початком у момент τ_1 на часовий інтервал $[\tau_1, \tau_2]$. Казатимемо, що обчислення p у цьому разі розпочинається станом $p\tau_1$ і закінчується станом $p\tau_2$, і позначатимемо цей факт $p\tau_1 \Rightarrow_P p\tau_2$.

Зазвичай інтерес становлять не всі можливі обчислення в процедурі, а насамперед ті, що розпочинаються в певний фіксований момент часу з певних виділених вхідних станів і закінчуються певними виділеними вихідними станами. Тому введемо поняття *ініціальної обчислювальної процедури* $P(S_0, S_{\text{fin}}, \tau_0)$ над простором Π як четвірки

[20] У загальному випадку функція керування може задавати й перетворення часового компонента (при цьому в процесі обчислень можливі стрибки в часовому просторі як уперед, так і назад) [42].

$\langle P, S_0, S_{\text{fin}}, \tau_0 \rangle$, де P – певна процедура над простором Π, $S_0 \subseteq S$ та $S_{\text{fin}} \subseteq S$ – виділені підмножини відповідно *вхідних* і *вихідних* станів, $\tau_0 \in \Gamma$ – початковий момент часу. *Результативним* назвемо обчислення p у процедурі P, що розпочинається в момент часу τ_0 із певного початкового стану $p\tau_0 = s_0 \in S_0$, закінчується у вихідному стані, а всі його проміжні стани невихідні. Для результативного обчислення p на інтервалі $[\tau_0, \tau]$ останній його стан $p\tau$ називається *заключним* і є *результатом.* Результат позначається $P^*(s_0)$ і є єдиним, якщо функція керування однозначна. У деяких випадках за результат обчислення береться не сам заключний стан, а значення на ньому певного *результуючого* часткового відображення $Val : S_{\text{fin}} \to B$. Також вважають заключними ті скінченні обчислення, на останніх станах яких не визначена функція керування.

Якщо в результативному обчисленні зняти умову, щоб усі проміжні стани були невихідними, то таке обчислення називатимемо *гіперрезультативним.* Його вихідні стани можуть зустрічатися й серед проміжних, а результат позначається $P^{**}(s_0)$.

Якщо для певного обчислення значення результуючого відображення Val на заключному стані не визначене, то воно вважається *безрезультатним.* Безрезультатними є зазвичай нескінченні, а також ті скінченні обчислення, що не мають жодного результативного продовження.

Ми уточнили поняття процедури й обчислення в ній. Це дозволяє уточнити й поняття функції як процедури.

Функціями-процедурами називаються ініціальні обчислювальні процедури над простором Π.

Після такого уточнення словосполучення "функція-процедура" та "ініціальна обчислювальна процедура" стають синонімами.

З кожною функцією-процедурою $P(S_0, S_{\text{fin}}, \tau_0)$ пов'язані два її графіки відповідності на вхідних станах $P^* : S_0 \to S_{\text{fin}}$ та $P^{**} : S_0 \to S_{\text{fin}}$, значення яких на початковому стані $s_0 \in S_0$ складають результати всіх результативних і гіперрезультативних обчислень із початком у момент τ_0 на s_0. Про графіки P^* та P^{**} кажуть, що їх *подає,* або *обчислює, функція* P.

Багато прикладів функцій-процедур можна знайти в математиці, зокрема в алгебрі й геометрії. Наприклад, задачі з планіметрії на побудову за допомогою циркуля й лінійки є фактично задачами на побудову певних функцій. Станами тут є сукупності фігур на площині, які можуть бути побудовані за допомогою циркуля й лінійки та операцій виділення довільної точки площини, точок перетину фігур, операцій іменування точок і фігур, а також умовних варіантів подібних операцій. Часовий простір тут дискретний і збігається з натуральними числами. Цікава ситуація виникає у випадку існування кількох розв'язків задачі. Розв'язок може забезпечити недетермінована функція, але її можна детермінізувати. Якщо, наприклад, для розв'язку задачі певний Виконавець використовує дошку й необхідно побудувати два різні розв'язки, то після побудови першого дошка зазвичай витирається і функція повертається в один із попередніх станів, але не в попередню конфігурацію – момент часу інший! Завдяки цьому процес обчислення може бути продовжений детерміновано, як того вимагає побудова другого розв'язку.

Функції-процедури, керування діями в яких реально залежить від часової компоненти, будемо називати *темпоральними* на відміну від *автоматних*, коли таке керування не залежить від часу. Класичними прикладами автоматних процедур є скінченні й магазинні автомати (див. підрозд. 1.4.4), машини Тьюрінга (підрозд. 2.1.4), деякі ігрові числення тощо. Для останніх часові фактори можуть відігравати важливу роль, і тоді вони набувають ознак темпоральних процедур. У багатьох іграх важливою є черговість ходу. У грі в шахи (шашки), наприклад, у всі непарні моменти часу ходять білі, а в парні – чорні. Однак є й інші принципові обставини, пов'язані з часом – рокіровка короля залежить від того, чи рухався він до цього моменту. У деяких моделях на стратегію вибору чергового ходу може впливати обмеженість часу (гра в цейтноті). У такому випадку з часом пов'язують певну міру.

Звернемось до натуральної арифметики й розглянемо обчислювальну процедуру GCD для знаходження найбільшого спільного дільника НСД(a,b) двох натуральних чисел.

Приклад 1.10 (алгоритм Евкліда). Нехай $\Pi = \langle \Gamma, S \rangle$ – обчислювальний простір із натуральним часом $\Gamma = N$, множиною станів $S = N^+ \times N^+$, $N^+ = N \setminus \{0\}$. Набір операцій $\Delta = \{\alpha, \beta\}$ і функцію керування δ визначимо так. Для будь-яких $a, b \in N^+$, $\tau \geq 0$ покладемо: $\alpha(a,b) = (a-b,b)$, $\beta(a,b) = (a,b-a)$, $\delta(\tau,(a,b)) = (a \neq b \to (a > b \to \alpha \mid \beta))$.

Розглянемо функцію $GCD = \langle \Pi, \Delta, \delta \rangle$ та її ініціалізований варіант $GCD\,(S_0, S_{\text{fin}}, 0)$ із результуючою операцією проектування pr_1, де $S_0 = N^+ \times N^+$, $S_{\text{fin}} = i_N$. Нескладно перевірити, що повне обчислення функції GCD на будь-якому початковому стані $s_0 = (a,b)$ є результативним. Дійсно, безпосередньо з означення GCD і співвідношень:

1) $\gcd(d,d) = d$,

2) $\left(b > a \Rightarrow \gcd(a,b) = \gcd(a, b-a)\right)$,

3) $\left(b < a \Rightarrow \gcd(a,b) = \gcd(b, a-b)\right)$

випливає, що останній стан у обчисленні функції GCD із початковим станом $s_0 = (a,b)$ матиме вигляд (d,d) для деякого числа $d = \gcd(a,b)$, тому що рано чи пізно компоненти чергового стану стануть рівними, а операції α та β не змінюють найбільший спільний дільник компонентів стану. Кінцевий результат обчислення формує операція проектування $\text{pr}_1\,(d,d) = d$. Таким чином, $GCD^*(a,b) = \gcd(a,b)$ ■

Окремого розгляду заслуговують функції, що закінчують обчислення винятково за часовими критеріями. У цьому випадку можуть навіть не фіксуватись заключні стани, а замість них на станах вводиться, наприклад, певний частковий порядок, і за результат обчислення (у тому числі й нескінченного) береться найменша верхня грань його станів.

Інший варіант становлять функції, що закінчують обчислення в певний момент часу τ або після деякого моменту τ тощо.

Функції P та Q називаються *еквівалентними* ($P \equiv Q$), якщо відповідності P^* та Q^*, які вони обчислюють, еквівалентні. Стан функції $P(S_0, S_{\text{fin}}, \tau_0)$ назвемо *припустимим*, якщо він зустрічається серед станів деякого її результативного обчислення. Покладемо $S_{acc} \subseteq S$ – підмножину всіх припустимих станів процедури P. Для графіків функцій стани за межами підмножини S_{acc} зайві. Нехай $P'\left(S'_0, S'_{fin}, \tau_0\right)$ – ініціальна функція над простором Π' із множиною станів S_{acc} і обмеженими на Π' елементарними операціями, функцією переходу та вхідними й вихідними станами процедури P. Очевидно, що функції P та P' еквівалентні. Функція, усі стани якої припустимі, називається *зведеною*. Наприклад, наведена вище функція GCD є зведеною.

Має місце така теорема.

Теорема 1.3. Клас графіків функцій-процедур замкнений відносно усіх регулярних композицій.

Доведення – див. вправу 34 ■

Увага! Надалі в тексті слово "функція" буде вживатися в обох смислах – і в традиційному теоретико-множинному, і як обчислювальна процедура. Яке з них мається на увазі – буде видно з контексту. Наприклад, у розд. 3 функції – це С- та С++-функції, а у формулі $O(\log_2 n)$ $\log_2 n$ – це традиційна логарифмічна функція-відображення ►

* **Література для СР:** функції-процедури – [1, 51, 65, 77, 80, 129]; рівняння в алгебрі відношень – [52]; обчислювальні процедури – [42, 44, 45, 85]; ігрові числення – [1, 100, 126].

Контрольні запитання та вправи

1. Що таке функція як процедура на змістовному рівні?
2. Пояснити різницю між функцією як процедурою і як відповідністю.
3. Що таке графік функції-процедури?
4. Яка різниця між ін'єкцією й сюр'єкцією?
5. Що таке бієкція?
6. Яка роль графіків у поданні функцій?
7. Перерахувати традиційні способи подання функцій. У чому їхні переваги й недоліки?
8. Дати визначення всіх регулярних композицій: а) для унарних функцій; б) для n -арних функцій.
9. Що таке регулярна: а) функція; б) n -арна операція?
10. Довести регулярність функцій над арифметичним базисом: а) $\max(x,y)$; б) $\min(x,y)$; в) $\max(x+y+z, xyz)$; г) $\max^2(x+y+z/2, xyz)+1$. При цьому: 1) дозволяються базові предикати-порівняння; 2) базові предикати відсутні, але є базова функція $\mathrm{sign}(x)$; 3) те саме, що й у 2), але $\mathrm{sign}(x)$ замінити на $|x|$. У 2)-3) використовується лише композиція суперпозиції.
11. Поле шахової дошки визначається парою натуральних чисел, кожне з яких не перевищує вісім: перше число – номер вертикалі (при підрахунку зліва направо), друге – номер горизо-

нталі (при підрахунку знизу вгору). Дано натуральні числа $1 \le k,l,m,n \le 8$. Потрібно з'ясувати:

а) чи є поля (k,l) і (m,n) одного кольору;

б) чи загрожує ферзь на полі (k,l) полю (m,n);

в) чи загрожує кінь на полі (k,l) полю (m,n).

Розв'язок оформити у вигляді регулярних функцій $f(k,l,m,n)$ над арифметичним базисом (як ходять фігури, див. у підрозд. 2.1.4).

12*. З'ясувати, чи містить десяткове подання числа n входження цифри 3. Знайти кількість таких входжень. Розв'язок оформити у вигляді регулярних предиката й функції над арифметичним базисом.

13*. Довести, що функція $f(n) = \underbrace{\sqrt{2+\sqrt{2+\ldots+\sqrt{2}}}}_{n}$ є регулярною в дійсній арифметиці, доповненій операцією $\sqrt{\ }$.

14. Довести: а) регулярність степеневої функції a^n у натуральній арифметиці; б)* будь-який регулярний терм для функції a^n є циклічним, тобто містить ітерацію або повторення.

15**. Композиція мінімізації μ n-арній функції $f(x_1,\ldots,x_n)$ ставить у відповідність $(n-1)$-арну функцію $g(x_1,\ldots,x_{n-1}) = \mu(f(x_1,\ldots,x_n) = 0)$, яка повертає найменше значення a таке, що $f(x_1,\ldots,x_{n-1},a) = 0$, і при цьому всі значення $f(x_1,\ldots,x_{n-1},i)$ для $i = \overline{0,a-1}$ визначені. Базовими називаються функції-проекції по i-му компоненту pr_i^n, $n > 0$, $i = \overline{1,n}$, тотожний нуль $0(x) = 0$ і функція $s(x) = x+1$. Функція називається частково-рекурсивною, якщо вона належить замиканню множини базових функцій відносно композицій суперпозиції, примітивної рекурсії (див. підрозд. 2.6.2) і мінімізації. Показати, що будь-яка частково-рекурсивна функція є регулярною n-арною операцією над базисом, що включає всі наведені вище базові функції та предикати порівняння "<" і "=" [15, 94].

16*. Довести неперервність відповідностей $f_1 \circ f_2, f_1 \cup f_2$ та f^* за кожним з аргументів.

17. Що таке обчислювальний простір?
18. Що таке процес у обчислювальному просторі?
19. Що означає: сукупність процесів подає функцію?
20. Дати означення обчислювальної процедури й обчислення за нею.
21. Яка роль функції керування в обчислювальній процедурі?
22. Що таке обчислення й гіперобчислення?
23. Що таке детермінована й недетермінована функції?
24. Що таке зведена функція-процедура?
25. Що означає: дана процедура обчислює певну відповідність на станах?
26. Побудувати функцію для поділу відрізка навпіл за допомогою циркуля й лінійки. У цій і зад. 23-26 для нотації функцій використовувати українську мову.
27. Побудувати функцію для знаходження найбільшої спільної міри двох сумірних відрізків за допомогою циркуля й лінійки.
28. Написати функцію для побудови кола за трьома його точками за допомогою циркуля й лінійки.
29. Написати функцію для переправи через річку вовка, кози й капусти. Припустимими операціями є переправа одного з них з одного берега на інший. При цьому на березі не повинні залишатися вовк із козою чи коза з капустою.
30. Нехай x – числова змінна і припустимими є операції множення з можливим іменуванням результату й зупинки stop.
    Яке значення обчислює така послідовність операцій:
    а) знайти значення $x * x$ і позначити його $x2$;
    б) знайти значення $x2 * x2$ і позначити його $x4$;
    в) знайти значення $x2 * x4$ і позначити його $x6$;
    г) зупинитись: значення $x6$ – результат?
    Написати функцію для даного обчислення.
31. Написати функції для обчислення значень степенів. Припустимі операції – ті самі, що й у вправі 23. Дію "знайти значення a і позначити його v" будемо формально записувати як $v \leftarrow a$. Бажана послідовність операцій із мінімальною кількістю множень:
    а) x^{10}; д) a^5 та a^{19}; з) a^4 та a^{20}; к) a^{28}.
    б) x^{12}; е) a^{15}; и) a^5 та a^{13};
    в) x^{16}; є) a^{21}; і) a^2, a^5, a^{17};
    г) x^{32}; ж) a^{64}; й) a^4, a^{12}, a^{28};

32. Сформулюємо поняття копії та реалізації функцій. Нехай $\Pi_1 = \langle \Gamma, S_1 \rangle$ і $\Pi_2 = \langle \Gamma, S_2 \rangle$ – довільні обчислювальні простори, $P_1 = \langle \Pi_1, \Delta_1, \delta_1 \rangle$ та $P_2 = \langle \Pi_2, \Delta_2, \delta_2 \rangle$ – довільні процедури відповідно над Π_1 та Π_2, $\varphi : S_1 \to S_2$ – певне відображення (функції кодування). Кажуть, що функція P_2 є φ-*копією* (просто копією) функції P_1, а функція P_1 – φ-*реалізацією* процедури P_2 (просто реалізацією), якщо для будь-якого стану $s \in S_1$ і часової константи $\tau \in \Gamma$ виконується умова: (*) для будь-якого елементарного перетворення $f \in \delta_1(\tau, s)$ у сукупності $\delta_2(\tau, \varphi s)$ є таке перетворення g, що $\varphi f(s) = g(\varphi s)$. Функція P_2 називається *дублікатом* функції P_1, якщо вона є φ-*копією* P_1 для певного бієктивного відображення φ.

Показати: 1) для кожного обчислення $(\tau_0, s_0) \Rightarrow_{P_1} (\tau_n, s_n)$ процедури P_1 в її φ-копії P_2 існує обчислення вигляду $(\tau_0, \varphi s_0) \Rightarrow_{P_2} (\tau_n, \varphi s_n)$; 2)* переходячи від окремих станів до фактор-класів за ядерною еквівалентністю функцій кодування, будь-яку наближену реалізацію певної функції можна стандартним чином перетворити на її дублікат.

33**. Нехай Σ^* – вільна півгрупа над алфавітом Σ. Покладемо $\Delta(\Sigma)$ сукупність усіх операцій $app_a(u) = app(a, u)$, $a \in \Sigma$. Функції над простором $\Pi = \langle \Gamma, \Sigma^* \rangle$ із сукупністю елементарних операцій $\Delta(\Sigma)$ називаються *вільними.* Показати, що для будь-якої функції існує її вільна реалізація [42].

34**. Дано довільні функції $P_1(S_0, S_1, \tau_0)$ та $P_2(S_1, S_2, \tau_0)$ над простором Π і предикати p_1 та p_2 на множині станів S. Побудувати функції з графіками: 1) $P_1^* P_2^*$, 2) $(p \to P_1^* \mid P_2^*)$; 3) $(p \to P_1^*)$; 4) $\{p \to P_1^*\}$; 5) $(p_1 \to P_1^* \mid\mid p_2 \to P_2^*)$ [42].

35**. Ввести на станах певний частковий порядок і за результат обчислення (у тому числі й нескінченного) узяти найменшу верхню грань його станів. Розглянути теорему про замкненість для отриманих функцій-процедур.

1.4. Алгебричні системи

- Поняття алгебричної системи
- Моделі систем
- Багатосортні Ω-системи
- Деякі прикладні багатосортні Ω-системи

Ключові слова: *алгебричні системи, алгебра, замкнена підмножина, підсистема, твірна множина, похідні операція та предикат системи,* методом структурної індукції *ізоморфізм, автоморфізм, гомоморфізм, функція кодування, наближена, сильна й точна моделі системи, еквівалентні моделі, ядерна еквівалентність, конгруенція, фактор-система, типізована операція, багатосортна Ω-система, вільна Ω-система, Ω-алгебра слів, лексикографічний порядок, регулярна Ω-система мов, регулярний вираз, регулярна мова, системи алгоритмічних алгебр, матриці, матрична алгебра, список, лінійний список, стек, черга, орієнтований та неорієнтований графи, зважений граф, мультиграф, мережа, бінарне дерево, дерево пошуку, збалансоване дерево пошуку, регулярний вираз, скінченний X-автомат, КВ- та РКВ-граматики, контекстновільна мова, дерево виведення,* $\mathrm{LL}(k)$*-граматика, автомат із магазинною пам'яттю* (*МП-автомат*), *розширена БНФ, синтаксична діаграма.*

Алгебричні системи й багатосортні Ω-системи є центральними поняттями першого розділу. Вони уточнюють інтерпретації ПЧП і дозволяють розглянути деякі загальні їхні властивості.

1.4.1. ПОНЯТТЯ АЛГЕБРИЧНОЇ СИСТЕМИ

Нехай A – довільна множина. Нагадаємо, що *n-арною операцією на A* (m*-арним предикатом на A*) називається відображення вигляду $F: A^n \to A$ (предикат вигляду $P: A^m \to \mathrm{Bool}$). Якщо $D_F \subset A^n$ ($D_P \subset A^m$), то операція (предикат) називається *частковою* (*частковим*). Число n (m) визначає *тип* операції (предиката). Щоб підкреслити тип операції, будемо писати F^n. Результат операції F^n на векторі $(a_1,....,a_n)$ позначається термами $F(a_1,....,a_n)$, $(a_1,....,a_n)F$, а у випадку унарної операції – термами Fa, aF.

0-арні операції на A називаються *константами.* Область визначення константи становить порожній вектор $\varepsilon : F^0\ \varepsilon = a$ для певного виділеного елемента $a \in A$.

Алгебричною системою називається трійка $\boldsymbol{A} = (A;\Omega_F;\Omega_P)$, де A – носій системи, $\Omega_F = \left\{F_1^{n_1}, F_2^{n_2}, \ldots\right\}$ – сукупність основних операцій, $\Omega_P = \left\{P_1^{m_1}, P_2^{m_2}, \ldots\right\}$ – сукупність основних предикатів на A.

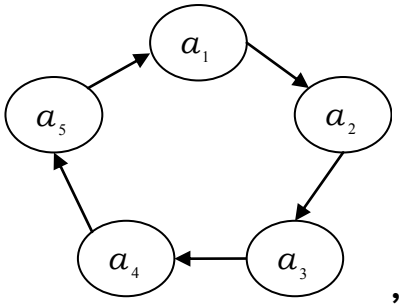
Ураховуючи, що тип Bool є стандартною частиною кожної алгебричної системи, його опускають при визначенні систем. *Типом* τ алгебричної системи називається пара $\tau = (n_1, n_2, \ldots; m_1, m_2, \ldots)$, утворена з типів її основних операцій і предикатів. Алгебрична система називається: 1) *алгеброю*, якщо $\Omega_P = \varnothing$; 2) *реляційною системою*, якщо $\Omega_F = \varnothing$. Алгебра й реляційні системи подаються у вигляді відповідних пар.

Згадувана натуральна арифметика $\square = (N, \text{Bool}; +, -, \times, /, 0, 1; =, <)$ є алгебричною системою типу $(2,2,2,2,0,0;2,2)$. Система опуклих замкнених багатокутників $\boldsymbol{A} = (V;\cap)$ є алгеброю типу 2, а система кругів $\boldsymbol{B} = (C;\subseteq)$ – реляційною системою типу 2. Одним із найпростіших класів алгебр є *півгрупи.* Вони мають тип 2. Сигнатура їх складається з єдиної асоціативної бінарної операції – множення. Півгрупами є натуральні числа відносно операцій додавання та множення, слова в певному алфавіті з операцією конкатенації.

Кожну алгебричну систему $\boldsymbol{A}$ можна подати у вигляді певної реляційної системи $\tilde{\boldsymbol{A}}$, якщо операції замінити предикатами, що відповідають їхнім графікам. Наприклад, натуральну арифметику $\square$ подає реляційна система $\tilde{\boldsymbol{N}} = \left(N; P_1^2, P_1^2, P_1^2, P_1^2, P_5^1, P_6^1; =, <\right)$, де $P_1^2(x,y) \Leftrightarrow \exists z(x+y=z)$, $P_5^1(x) \Leftrightarrow x = 0(x)$ тощо.

Підсистеми. Зупинимось тепер на важливому в практичному сенсі питанні: яка мінімальна інформація є необхідною й достатньою для опису носія алгебри? Непорожня підмножина $B \subseteq A$ системи $\boldsymbol{A} = (A;\Omega_F;\Omega_P)$ називається *замкненою* в $\boldsymbol{A}$, якщо вона замкнена відносно кожної основної операції системи, тобто якщо результат кожної такої операції на елементах із B теж належить A. Система $\boldsymbol{B} = (B;\Omega'_F;\Omega'_P)$, де Ω'_F і Ω'_P – сукупності основних операцій і предикатів системи $\boldsymbol{A}$, обмежених на замкнену підмножину B, називається

підсистемою системи $\boldsymbol{A}$. Щоб не ускладнювати позначення, будемо там, де це можливо, сукупності основних операцій і предикатів підсистем і систем позначати однаково – Ω_F та Ω_P, відповідно. Якщо $\boldsymbol{A}$ є алгеброю, то в цьому випадку говорять про *підалгебру* алгебри $\boldsymbol{A}$. Алгебра $\boldsymbol{A}_5 = \left(\{a_1, a_2, a_4, a_3, a_5\}; \lambda\right)$, де операція λ задана графом



,

не має *власних* підалгебр, тобто таких, що не збігаються із самою алгеброю, а алгебра $\boldsymbol{N}^+ = (\{1,2,...\};+)$ має зліченну кількість власних підалгебр, наприклад вигляду $(\{k, k+1,...\};+)$, $k > 1$.

Сукупність $\Re$ непорожніх власних підсистем даної системи має важливу властивість – вона замкнена відносно перетину систем (вправа 2.7). Перетин усіх непорожніх власних підсистем системи $\boldsymbol{A}$ є найменшою (*головною*) підсистемою $\boldsymbol{A}$. Якщо взяти арифметику цілих чисел $\boldsymbol{Z}_{10}$, то головною її підсистемою є $\{0\}$, а алгебра $\boldsymbol{N}^+ = \left(\{1,2,...\};+\right)$ не має головної підалгебри.

Нехай $\boldsymbol{B}$ – довільна підмножина системи $\boldsymbol{A}$. Серед підсистем $\boldsymbol{A}$ обов'язково є й надсистеми $\boldsymbol{B}$ (напр., сама система $\boldsymbol{A}$). Нехай $\Re$ – сукупність усіх підсистем системи $\boldsymbol{A}$, що включають $\boldsymbol{B}$. Тоді перетин $\boldsymbol{B}^*$ усіх систем із $\Re$ є найменшою підсистемою $\boldsymbol{A}$, що містить усі елементи B. Система $\boldsymbol{B}^* = \left(B^*; \Omega_F; \Omega_P\right)$ називається *підсистемою системи* $\boldsymbol{A}$, *породженою множиною* B, множина B^* *– замкненням множини* B *відносно операцій* із Ω_F, а елементи множини B *– твірними елементами замкнення* B.

Якщо $\boldsymbol{B}^* = \boldsymbol{A}$, то множина B називається *множиною твірних* системи $\boldsymbol{A}$. Система може мати не одну множину твірних. Наприклад, в алгебрі $\boldsymbol{A}_5$ кожен елемент a_i сам є твірним алгебри. Якщо в алгебрі $\boldsymbol{A}_5$ вибрати за твірний елемент a_1, то решту її елементів можна задати термами $a_2 = \lambda(a_1)$, $a_3 = \lambda(\lambda(a_1))$, $a_4 = \lambda(\lambda(\lambda(a_1)))$, $a_5 = \lambda(\lambda(\lambda(\lambda(a_1))))$.

Множина твірних називається *незалежною,* якщо вона мінімальна. Отже, щоб визначити довільну алгебру, достатньо описати її основні операції й вибрати одну з незалежних множин твірних. Наприклад, для означення алгебри $\boldsymbol{N}^+$ достатньо взяти за твірну число 1 і визначити операцію додавання. Усі її елементи містяться серед значень термів: $(1+1)$, $((1+1)+1)$, $(((1+1)+1)+1)$ тощо[21].

Похідні операції алгебричної системи. Нехай $\boldsymbol{A}=(A;\Omega_F;\Omega_P)$ – довільна алгебрична система. Окрім основних операцій і предикатів системи, у ній розглядають похідні операції та предикати. Зазвичай до них відносять суперпозиції основних операцій і предикатів. Ми ж будемо трактувати їх розширено і вважати похідними всі регулярні n-арні операції та предикати, породжені основними операціями системи.

Похідні операції цікаві тим, що в узагальненому вигляді дозволяють подати різноманітні варіанти скінченних комбінацій застосувань основних операцій і предикатів. Класичними є операції, отримані за допомогою регулярних композицій перестановки, ототожнення й параметризації аргументів, введення фіктивних аргументів, суперпозиції й розгалуження. Похідні операції та предикати подаються регулярними функціональними термами. Наприклад, терм $\left[p^1 \to \mathrm{S}\left(f^3, g_1^2, g_2^2, g_3^2\right) \mid \left\{q^2 \to f_1^1\right\}\right]$ подає похідну операцію, що є розгалуженням за певною унарною умовою відповідних суперпозиції та ітерації функцій. Регулярний терм $\mathrm{S}\left(f^3, g_1^2, g_2^2, g_3^2\right)$ зазвичай записують у вигляді префіксного Ω-терму $f^3\left(g_1^2, g_2^2, g_3^2\right)$.

Структурна індукція. Важливою властивістю замкнень алгебричної системи є те, що вони припускають доведення методом структурної індукції. Нехай $P(x)$ – певний предикат на замкненні B^*.

Правило структурної індукції для замкнення B^*:

(Б) База індукції: $\forall x \in B : P(x)$.

(І) Індуктивний перехід: для всіх основних операцій $F_i^{n_i}$

$$\forall x_1,\ldots,x_n \in B^* : \bigwedge_{k=1}^{n} P(x_k) \Rightarrow P\left(F_i^{n_i}(x_1,\ldots,x_n)\right).$$

(П) Повнота: $\forall x \in B^* : P(x)$.

[21] Насправді достатньо часткового випадку додавання – збільшення числа на 1.

Лема 1.2 (про структурну індукцію). З бази індукції та індуктивного переходу правила структурної індукції випливає повнота.

Доведення. Дійсно, нехай $R = \{x \in B^* : P(x)\}$, тоді з бази індукції випливає $B \subseteq R$, а за індуктивним переходом – $B^* \subseteq R$. Однак $R \subseteq B^*$, отже, $R = B^*$ ■

Таким чином, щоб установити повноту, достатньо перевірити базу індукції та правило індуктивного переходу. Звичайна математична індукція для натуральних чисел є частинним випадком структурної. Розглянемо приклад доведення за правилом структурної індукції. Визначимо сукупність слів Σ^* в алфавіті $\Sigma = \{a_1,...,a_n\}$ як замкнення порожнього слова ε відносно операції $\text{pre} : \Sigma \times \Sigma^* \to \Sigma^*$ – приписування ліворуч літери до слова: $\text{pre}(a,u) \overset{def}{=} au$.

Структурна індукція для слів має вигляд:

(Б) $P(\varepsilon)$;

(І) $\forall w \in \Sigma^*, \forall a \in \Sigma : P(w) \to P(aw)$;

(П) $\forall x \in \Sigma^* : P(x)$.

Приклад 1.11. Доведемо методом структурної індукції асоціативність операції конкатенації. Покладемо

$$P(u) \Leftrightarrow \forall w,v \in \Sigma^* : (u \cdot v) \cdot w = u \cdot (v \cdot w).$$

Перевіримо спочатку базу індукції. Нехай w,v – довільні слова над Σ. Тоді з означення порожнього слова випливає, що $(\varepsilon \cdot v) \cdot w = v \cdot w$ (1) і $\varepsilon \cdot (v \cdot w) = v \cdot w$ (2). Отже, (1)=(2) і $P(\varepsilon)$ виконується. Розглянемо індуктивний перехід. Нехай u,v,w – довільні слова над Σ, a – літера алфавіту та $P(u)$ виконується. Розглянемо $P(au)$:

$(au \cdot v) \cdot w = (auv) \cdot w = a(uv)w$ /*означення операції pre*/ =

$= a((uv)w)$ /*означення конкатенації*/ $= a(u \cdot (v \cdot w))$ /* $P(u)$ */ =

$= (au \cdot (v \cdot w))$ /*означення операції pre*/.

Таким чином, має місце індуктивний перехід, а отже, і повнота ■

1.4.2. МОДЕЛІ СИСТЕМ

Коли кажуть, що певна система є моделлю іншої (первісної) системи, то мають на увазі. що її елементи копіюють (можливо, з деяким наближенням) первісні об'єкти, а основні операції та предикати імітують поведінку основних операцій і предикатів первісної системи. Уточнимо це поняття.

Ізоморфізми й гомоморфізми систем. *Відображенням* $\phi : \boldsymbol{A} \to \boldsymbol{B}$ системи $\boldsymbol{A}$ в (на) систему $\boldsymbol{B}$ називається відображення $\phi : A \to B$ носія системи $\boldsymbol{A}$ в (на) носій системи $\boldsymbol{B}$.

Ізоморфізмом системи $\boldsymbol{A} = \left(A; F_1^{n_1}, F_2^{n_2}, \ldots; P_1^{m_1}, P_2^{m_2}, \ldots\right)$ **в (на) однотипну систему** $\boldsymbol{B} = \left(B; G_1^{n_1}, G_2^{n_2}, \ldots; Q_1^{m_1}, Q_2^{m_2}, \ldots\right)$ **називається взаємооднозначне відображення** $\phi : A \to B$**, яке зберігає головні операції та предикати системи** $\boldsymbol{A}$**, тобто для всіх** $a_1, a_2, \ldots \in A$ **та всіх** $i, j \in N$**:**

$$F_i^{n_i}\left(a_1, \ldots, a_{n_i}\right)\downarrow \phi = G_i^{n_i}\left(a_1\phi, \ldots, a_{n_i}\phi\right)\downarrow; \tag{1}$$

$$P_j^{m_j}\left(a_1, \ldots, a_{m_j}\right) \Leftrightarrow Q_j^{m_j}\left(a_1\phi, \ldots, a_n\phi\right). \tag{2}$$

Ізоморфізм системи на себе називається *автоморфізмом.*

Гомоморфізмом системи $\boldsymbol{A}$ **в (на) однотипну систему** $\boldsymbol{B}$ **називається відображення системи** $\boldsymbol{A}$ **в (на) однотипну систему** $\boldsymbol{B}$**, яке зберігає головні операції, і для всіх** $a_1, a_2, \ldots \in A$ **та кожного** $j \in N$ **виконується**

$$P_j^{m_j}\left(a_1, \ldots, a_{m_j}\right) \Rightarrow Q_j^{m_j}\left(a_1\phi, \ldots, a_{m_j}\phi\right). \tag{3}$$

Кожний ізоморфізм є гомоморфізмом. *Гомоморфізм* системи $\boldsymbol{A}$ *в* (*на*) однотипну систему $\boldsymbol{B}$ називається *сильним,* якщо він зберігає головні операції та для всіх $b_1, \ldots, b_{m_j} \in B$ і кожного $j \in N$ виконується: із $Q_j^{m_j}\left(b_1, \ldots, b_{m_j}\right)$ випливає існування таких прообразів $a_1, \ldots, a_{m_j}$ елементів у A, що $P_j^{m_j}\left(a_1, \ldots, a_{m_j}\right)$.

Усякий гомоморфізм скінченної системи на себе є автоморфізмом. Дійсно, якщо для деякого вектора $\left(a_1, \ldots, a_{m_j}\right)$ $Q_j^{m_j}\left(a_1\phi, \ldots, a_{m_j}\phi\right) = 1$, то із (3) випливає $Q_j^{m_j}\left(a_1\phi^k, \ldots, a_{m_j}\phi^k\right) = 1$ для $k = 2, 3, \ldots$. Оскільки ϕ є вза-

ємооднозначним відображенням A на себе, то для деякого k степінь ϕ^k буде тотожним і $P_j^{m_j}\left(a_1,....,a_{m_j}\right)=1$.

Поняття моделі систем. Поняття ізоморфізму й гомоморфізму належать до фундаментальних понять алгебри та інформатики. На них базується поняття моделі систем.

Система *B* є наближеною моделлю системи *A* відносно функцій кодування й декодування ϕ і ϕ^{-1}, якщо ϕ є гомоморфізмом системи *A* в систему *B*.

Взаємозв'язок між первісною системою ***A*** та її моделлю ***B*** ілюструє рис. 1.5, де $\bar{a},\bar{b},\bar{c},\bar{d}$ – відповідні вектори-аргументи операцій.

Як бачимо, щоб знайти значення певної операції системи, достатньо закодувати її аргументи (перейти до моделі), застосувати до них відповідну модельну операцію й розкодувати результат. У випадку, коли гомоморфізм ϕ сильний, говорять про *сильну модель*. Коли ϕ взаємооднозначний, то про моделі говорять як про *точні*. Коли ж ϕ є бієкцією, а система ***A***, у свою чергу, – моделлю системи ***B*** відносно функції декодування ϕ^{-1}, то говорять про *еквівалентні*, або *ізоморфні*, моделі.

Розглянемо кілька прикладів моделей систем. Кодування векторів елементів при моделюванні зводиться до кодування їхніх компонентів, тобто кодом вектора елементів є вектор кодів компонентів.

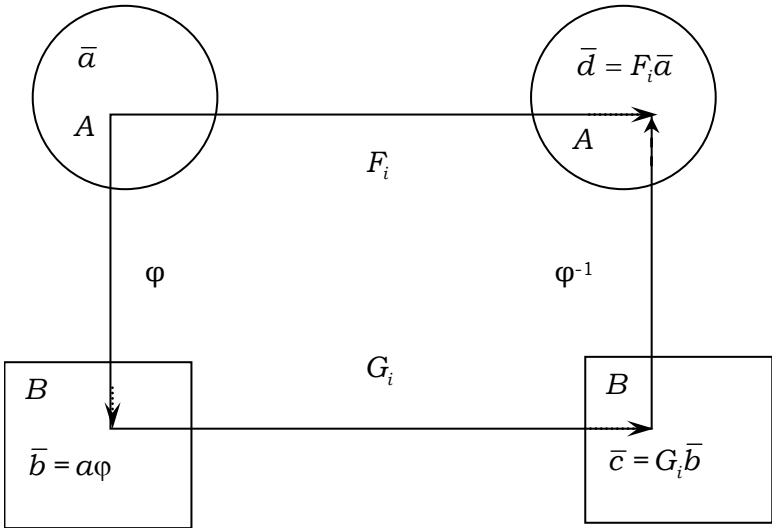


Рис. 1.5

Приклад 1.12. Нехай $\text{Boolean} = \left(\{\text{false}, \text{true}\}; \text{OR}, \text{AND}, \text{NOT}, =, <\right)$ – булевий тип мови програмування Pascal із відповідними логічними операціями й відношенням $\text{false} < \text{true}$, $\boldsymbol{B} = \left\langle \{0,1\}; \vee, \wedge, \neg, =, < \right\rangle$ – двоелементна булева алгебра з логічними операціями диз'юнкції, кон'юнкції, заперечення, відношенням "менше", з найменшим (0) і найбільшим (1) елементами. Тоді $\boldsymbol{B}$ є точною (ізоморфною) моделлю типу Boolean відносно функцій кодування $\phi(\text{false}) = 0$, $\phi(\text{true}) = 1$ і декодування ϕ^{-1} ■

Приклад 1.13. Десяткова $\boldsymbol{Z}_{10} = \left\langle Z_{10}; +, -, \times, /; =, < \right\rangle$ і двійкова $\boldsymbol{Z}_2 = \left\langle Z_2; +, -, \times, /; =, < \right\rangle$ арифметики цілих чисел еквівалентні відносно функцій кодування й декодування, що переводять десяткові числа в рівні їм двійкові, і навпаки ■

Приклад 1.14. Розглянемо систему $S_3 = \left\langle R^+; \times \right\rangle$ додатних дійсних чисел з операцією множення й систему $\boldsymbol{I}_3 = \left\langle R\ ; + \right\rangle$ усіх дійсних чисел з операцією додавання. Вони еквівалентні відносно функцій кодування $\phi_3(x) = \lg(x)$ і декодування $\varphi_3^{-1}(x) = 10^x$. Дійсно, досить згадати тотожність $\lg(x \times y) = \lg x + \lg y$ і пропотенціювати її ■

Приклад 1.15. Нехай $\boldsymbol{S}_4 = \left\langle Z^3; +, -, = \right\rangle$ – система точок тривимірного цілочислового простору з операціями покоординатного додавання й віднімання точок, $\boldsymbol{I}_4 = \langle Q_0; \times, /, = \rangle$ – підсистема $Q_0 \subset Q^+$ додатних раціональних чисел зі звичайними операціями множення й ділення, де $Q_0 = \left\{ 2^a \times 3^b \times 5^c : a, b, c \in Z \right\}$. Нескладно перевірити, що системи ізоморфні відносно функцій кодування $\phi_4\left(\langle a, b, c \rangle\right) = 2^a \times 3^b \times 5^c$ і декодування ϕ_4^{-1}. При цьому додавання моделюється операцією множення, а віднімання – операцією ділення.

Розглянемо дію $\langle 4, -1, 2 \rangle + \langle 2, 2, 2 \rangle$ у системі $\boldsymbol{S}_4$ та її аналог у моделі $\boldsymbol{I}_4$:

$$\langle 4, -1, 2 \rangle + \langle 2, 2, 2 \rangle = \langle 6, 1, 4 \rangle;$$

$$\left(\phi_4\left(\langle 4, -1, 2 \rangle\right) \times \phi_4\left(\langle 2, 2, 2 \rangle\right)\right) = \left(2^4 3^{-1} 5^2 \times 2^2 3^2 5^2\right) = 2^6 3^1 5^4.$$

Рівність $\phi_4 \langle 6, 1, 4 \rangle = 2^6 3^1 5^4$ показує, що відповідні обчислення в системах узгоджені вірно. Зауважимо, що модель $\boldsymbol{I}_4$ на практиці може бути неефективною, оскільки потребує надвеликих цілих чисел ■

Приклад 1.16. Розглянемо алгебру $\boldsymbol{S}_5 = \langle 2^A; \cup, \cap, \neg, \varnothing; =, \subset \rangle$ усіх підмножин певної скінченної множини $A = \{a_1, a_2, ..., a_n\}, n > 0$, зі звичайними множинними операціями – об'єднанням, перетином, доповненням, порожньою множиною, предикатами рівності та включення – і систему двійкових векторів довжиною n $\boldsymbol{I}_5 = \langle \{0,1\}^n; \bar{\vee}, \bar{\wedge}, \neq, \bar{\theta}, =, < \rangle$ із покоординатними операціями диз'юнкції, кон'юнкції й заперечення, нульовим вектором і бінарним предикатом, який повертає значення за правилом $e_1 < e_2 = ((e_1 \bar{\wedge} e_2) = e_1)$. Зафіксуємо певний лінійний порядок елементів у A та визначимо функцію кодування $\phi_5 : 2^A \to \{0,1\}^n$ таким чином: для $M \subseteq A$ покладемо $\phi_5(M) = (b_1, b_2, ..., b_n)$, де $1 \le \forall i \le n (b_i = (a_i \in M \to 1 \mid 0)$. Системи $\boldsymbol{S}_5$ та $\boldsymbol{I}_5$ еквівалентні. Модель $\boldsymbol{I}_5$ використовується в компіляторах мов програмування для реалізації множинних типів ■

Моделі й конгруенції. У зв'язку з поняттям моделі цілком природно постають питання: якою є сукупність усіх можливих моделей даної системи, чи є серед них моделі з тими чи іншими властивостями тощо? Спробуємо дати певну відповідь на ці й подібні запитання.

Кожна функція кодування $\phi : \boldsymbol{A} \to \boldsymbol{B}$ породжує на елементах первісної системи відношення *ядерної* еквівалентності $\sigma_\phi \subseteq A \times A : a \sigma_\phi b \overset{def}{\Leftrightarrow} a\phi = b\phi$. Фактор-класами ядерної еквівалентності σ_ϕ є повні прообрази $b\phi^{-1}$ усіх кодів b. Фактор–клас, що містить елемент a, будемо позначати $[a]$. Якщо кожному коду $b \in B$ поставити у відповідність повний його прообраз $b\phi^{-1}$, то отримаємо канонічні взаємооднозначні відображення типів $A/\sigma_\phi \to B$ та $B \to A/\sigma_\phi$, відповідно. Кажуть, що відношення σ *стабільне* відносно даної n-арної операції F на A, коли для будь-яких $a_1, ..., a_n$, $b_1, ..., b_n$ таких, що $\forall 1 \le i \le n \ (a_i \sigma b_i)$, виконується $F(a_1, ..., a_n) \sigma \ F(b_1, ..., b_n)$. Стабільними є, зокрема, відношення рівності й тотальне відношення на A.

Еквівалентність σ у системі $\boldsymbol{A}$ називається конгруенцією, якщо вона стабільна відносно кожної головної операції системи.

Нескладно впевнитись, що ядерна еквівалентність гомоморфізму $\phi: A \to B$ системи $\boldsymbol{A}$ в систему $\boldsymbol{B}$ є конгруенцією (вправа 9). Ця обставина дозволяє побудувати на фактор-класах A ядерну *фактор-систему* $\boldsymbol{A}/\sigma_\phi = \left(A/\sigma_\phi; \left\{\tilde{F}_i^{n_i}\right\}; \left\{\tilde{P}_j^{m_j}\right\}\right)$, яка є моделлю системи $\boldsymbol{A}$. Для $a_1,....,a_{n_i} \in A$ покладемо:

1) $\tilde{F}_i\left([a_1],....,[a_{n_i}]\right) \overset{def}{=} [F_i^{n_i}\left(a_1,....,a_{n_i}\right)]$;

2) $\tilde{P}_j^{m_j}\left([a_1],....,[a_{m_j}]\right) \overset{def}{\Leftrightarrow} \exists a_1' \in [a_1]....\exists a_{m_i}' \in [a_{m_i}]\ P_j^{m_j}\left(a_1',....,a_{m_j}'\right)$.

Означення $\tilde{\mathrm{F}}_i$ коректне. Дійсно, якщо розглянути інші представники класів $a_1' \in [a_1],....,a_{n_i}' \in [a_{n_i}]$, то результат операції $b' = \mathrm{F}_i^{n_i}\left(a_1',....,a_{n_i}'\right)$ на них буде в тому самому фактор-класі, що й $b = \mathrm{F}_i^{n_i}\left(a_1,....,a_{n_i}\right)$:

$\mathrm{F}_i^{n_i}\left(a_1,....,a_{n_i}\right)\phi = \mathrm{G}_i^{n_i}\left(a_1\phi,....,a_{n_i}\phi\right) = \mathrm{G}_i^{n_i}\left(a_1'\phi,....,a_{n_i}'\phi\right) = F_i^{n_i}\left(a_1',....,a_{n_i}'\right)\phi$.

Отримали $b\phi = b'\ \phi$. Отже, має місце

Лема 1.3 (про ядерну еквівалентність гомоморфізму). Ядерна еквівалентність гомоморфізму $\phi: A \to B$ системи $\boldsymbol{A}$ в систему $\boldsymbol{B}$ є конгруенцією. Канонічне відображення $[]: A \to A/\sigma_\phi$ таке, що $a[\ ] = [a]$, є гомоморфізмом системи $\boldsymbol{A}$ на фактор-систему $\boldsymbol{A}/\sigma_\phi$, а остання система – наближеною моделлю системи $\boldsymbol{A}$. У свою чергу, $\boldsymbol{B}$ є наближеною моделлю системи $\boldsymbol{A}/\sigma_\phi$. Якщо $\phi: A \to B$ – сильний гомоморфізм, то $\boldsymbol{B}$ є точною моделлю системи $\boldsymbol{A}/\sigma_\phi$.

Доведення. За означенням $F_i^{n_i}\left(a_1[\ ],....,a_{n_i}[\ ]\right)[\ ] = \tilde{F}_i^{n_i}\left([a_1],....,\left[a_{n_i}\right]\right)$ та $P_j^{m_j}\left(a_1,....,a_{m_j}\right) \Rightarrow \tilde{P}_j^{m_j}\left([a_1],....,\left[a_{m_j}\right]\right)$ і обидві умови гомоморфізму виконуються. За побудовою канонічне відображення $\phi: A/\sigma_\phi \to B$ таке, що $[a]\ \phi = a\phi$, є гомоморфізмом фактор-системи $\boldsymbol{A}/\sigma_\phi$ на систему $\boldsymbol{B}$, тобто $\boldsymbol{B}$ – наближена модель системи $\boldsymbol{A}/\sigma_\phi$. Цей гомоморфізм є ізоморфізмом, а B – точною моделлю системи $\boldsymbol{A}/\sigma_\phi$, якщо гомоморфізм $\phi: A \to B$ – сильний ■

Отже, кожній сильній моделі $\boldsymbol{B}$ системи $\boldsymbol{A}$ відповідає певна ізоморфна їй фактор-система системи $\boldsymbol{A}$. Ураховуючи, що кожна фактор-

система $\boldsymbol{A}/\sigma$ за конгруенцією σ є сильною моделлю системи $\boldsymbol{A}$, можемо зробити важливий висновок:

Лема 1.4. Сукупність усіх сильних моделей системи $\boldsymbol{A}$ з точністю до ізоморфізму вичерпується сукупністю всіх її фактор-систем за різними конгруенціями.

Доведення. Випливає з наведеної вище леми 1.3 ■

Таким чином, задачі: 1) знайти з точністю до ізоморфізму всі сильні моделі системи $\boldsymbol{A}$; 2) знайти всі конгруенції на $\boldsymbol{A}$ – рівносильні.

Приклад 1.17. На кожній системі $\boldsymbol{A}$ є принаймні дві конгруенції – рівність і тотальна. У першому випадку кожний фактор-клас $\boldsymbol{A}/=$ одноелементний, у другому – сукупність $\boldsymbol{A}/\sigma$ одноелементна та є носієм системи ■

Приклад 1.18. Розглянемо приклад моделі алгебри $\boldsymbol{N}^+ = \left(\{1,2,...\};+\right)$. Нехай $\boldsymbol{B} = \left(\{-1,1\};\times\right)$ – двоелементна алгебра, тоді відображення $n\phi \overset{def}{=} (-1)^n$ є гомоморфізмом алгебри $\boldsymbol{N}^+$ на алгебру n, тобто остання є наближеною моделлю $\boldsymbol{N}^+$. Дійсно,

$$(n+m)\phi = (-1)^{n+m} = (-1)^n \times (-1)^m = n\phi + m\phi .$$

Сукупність $\boldsymbol{N}^+/\sigma_\phi$ двоелементна і складається з підмножин парних і непарних чисел. Окрім ядерної конгруенції σ_ϕ на $\boldsymbol{N}^+$, існують інші конгруенції ■

1.4.3. БАГАТОСОРТНІ Ω-СИСТЕМИ

Ми ознайомилися з деякими абстрактними властивостями алгебричних систем. Тепер розглянемо такі системи з практичнішого погляду. По-перше, нагадаємо, що реальні предметні області не є просто множинами елементів із певними співвідношеннями між ними. Навпаки, їхні елементи зазвичай згруповані в певні сорти подібних елементів, і співвідношення між ними мають сенс тільки в межах таких сортів чи їхніх сукупностей. По-друге: якщо ми маємо алгебричну систему з певною сукупністю твірних, то як синтаксично подати довільний елемент такої системи?

Поняття багатосортної Ω-системи. Нехай U – довільна множина, яку будемо називати універсумом, і $A_1,...,A_{n+1} \subseteq U$. n-арна операція F^n, область означення якої є підмножиною $A_1 \times ... \times A_n$, а область зна-

чення – підмножиною A_{n+1}, називається *типізованою*, вираз $A_1 \times \ldots \times A_n \to A_{n+1}$ – її *типом,* а підмножини $A_1, \ldots, A_{n+1}$ – *сортами* аргументів і значень. Аналогічно визначається типізований предикат.

Нехай $\Omega = (\Omega_s,\ \Omega_c,\ \Omega_v,\ \Omega_f,\ \Omega_p)$ – певна сигнатура типу $\Delta = (\sigma_1, \sigma_2, \ldots; \tau_1, \tau_2, \ldots;\ \nu_1, \nu_2, \ldots)$. Проінтерпретуємо сигнатуру Ω на універсумі U. Як уже зазначалося в підрозд. 1.1, для цього необхідно вибрати в U конкретні сорти для символів-сортів, константам сигнатури поставити у відповідність їхні фіксовані значення потрібного сорту, символам-змінним – відповідний тип, функціональним і предикатним символам – конкретні типізовані операції та предикати згідно з їхніми типами. Виберемо певну *інтерпретацію* сигнатури Ω і позначимо її I. Нехай $I\varsigma$ – значення сигнатурного символу ς при інтерпретації I. Покладемо $\Omega_s^I = \{Is_1, Is_2, \ldots\}$, $\Omega_c^I = \{c_1 : I\sigma_1, c_2 : I\sigma_2, \ldots\}$, $\Omega_v^I = \{x_1 : I\sigma_1, x_2 : I\sigma_2, \ldots\}$, $\Omega_f^I = \{f_1 : I\tau_1, f_2 : I\tau_2, \ldots\}$, $\Omega_p^I = \{p_1 : I\nu_1, p_2 : I\nu_2, \ldots\}$ – сукупності сортів, типізованих констант, змінних, операцій і предикатів інтерпретації I.

Багатосортною Ω-системою на універсумі U називається п'ятірка $\boldsymbol{A} = (\Omega_s^I, \Omega_c^I, \Omega_v^I, \Omega_f^I, \Omega_p^I)$, де I – довільна інтерпретація сигнатури Ω.

Сукупність сортів Ω_s^I називається *носієм* системи, а Ω_c^I, Ω_v^I, Ω_f^I, Ω_p^I – сукупності відповідно її констант, змінних, *основних* операцій і предикатів.

Багатосортні Ω-системи задають семантику ПЧП із сигнатурою Ω.

При розгляді конкретних Ω-систем інтерпретації сигнатури фіксують, тому верхній індекс I в елементах таких систем зазвичай опускають. Розглянемо кілька важливих прикладів Ω-систем.

Словарна Ω-система. Вище вже зазначалась роль слів у іменуванні математичних об'єктів. Однак їхнє значення в інформатиці цим не вичерпується. Слова становлять основу вербальних ДС, до яких належать і всі традиційні мови програмування. Вони також відіграють особливу роль при конструюванні об'єктів. Як ми побачимо далі, внутрішня будова будь-якого з алгебричних об'єктів може бути подана у вигляді слова-терму. На словах діють усі операції та предикати, які визначені в підрозд. 1.2.3 на кортежах – конкатенація, приписування літери на початок і кінець слова, відношення префіксності, суфіксності, закінчення тощо. Однак для них можуть бути визначені й власні операції та відношення. Насамперед, лінійна впорядкованість

літер усередині алфавіту X породжує лінійний *лексикографічний* порядок на множині всіх слів X^*: $a_0....a_n \prec b_0....b_m \overset{def}{\Leftrightarrow} b_0....b_n \lhd c_0....c_m \vee \exists i\left(\forall j < i \to b_j = c_j \,\&\, b_i \prec c_i\right)$. Даний порядок використовується, наприклад, у словниках.

Розглянемо один із мінімальних варіантів словарної сигнатури Ω. Нехай $\Omega = \left(\Omega_s, \Omega_c, \Omega_v, \Omega_f, \Omega_p\right)$, де $\Omega_s = \{\text{char}, \text{string}\}$, $\Omega_c = \{e : \text{string}\}$, $\Omega_p = \{=, \prec : \text{string} \times \text{string}\}$,

$$\Omega_f = \left\{\begin{array}{l} \circ : \text{string} \times \text{string} \mapsto \text{string} \\ \text{head} : \text{string} \mapsto \text{char} \\ \text{tail} : \text{string} \mapsto \text{string} \\ \text{app} : \text{string} \times \text{char} \mapsto \text{string} \\ \text{pre} : \text{string} \times \text{char} \mapsto \text{string} \end{array}\right\}.$$

Зафіксуємо певний алфавіт $\Sigma = \{a_1, ..., a_n\}$. Визначимо словарну систему W як Ω-систему з множинами сортів $\Omega_S = \left\{\Sigma, \Sigma^*\right\}$, констант $\Omega_C = \{\varepsilon\}$, де ε – порожнє слово, операцій $\Omega_F = \{\circ, \text{head}, \text{tail}, \text{app}, \text{pre}\}$ з операціями конкатенації, взяття голови й хвоста слова, приписування символу праворуч і ліворуч до слова та предикатів $\Omega_P = \{=, \prec\}$ із предикатами рівності й лексикографічного порядку. Для спрощення словарних Ω-термів опускають символи операцій конкатенації та приписування літер праворуч і ліворуч. Наприклад, замість термів $u \circ v$, $\text{pre}(a, x)$ та $\text{app}(a, x)$ уживають відповідно uv, ax та xa.

У підрозд. 2.6.2 наведено індуктивні означення всіх операцій словарної системи W.

Регулярні мови. Нехай $n \in N$. Зафіксуємо певний алфавіт $\Sigma = \{a_1, ..., a_n\}$. Довільні підмножини $L \subseteq \Sigma^*$ слів у алфавіті Σ називаються *мовами*. Сигнатура регулярної Ω-системи мов має такий склад:

$\Omega_s = \{\text{char}, \text{string}, \text{regular}\}$,

$\Omega_c = \{a_1, a_2, ..., a_n : \text{char}, \varepsilon : \text{string}, \varnothing : \text{regular}\}$,

$\Omega_v = \{a, b, c, ... : \text{char}; u, v, w, ... : \text{string}; \text{X}, \text{Y}, \text{Z}, ... : \text{regular}\}$

$\Omega_p = \{=, \subseteq : \text{regular} \times \text{regular}\}$, $\Omega_f = \left\{\begin{array}{l} \cup : \text{regular} \times \text{regular} \mapsto \text{regular} \\ \circ : \text{regular} \times \text{regular} \mapsto \text{regular} \\ * : \text{regular} \mapsto \text{regular} \end{array}\right\}$.

Визначимо регулярну систему мов $\Re$ як Ω-систему з множинами сортів $\Omega_S = \{\Sigma, \Sigma^*, B(\Sigma^*)\}$, констант $\Omega_C = \{a_1, a_2, ..., a_n, \varepsilon, \{\varepsilon\}\}$, предикатів $\Omega_P = \{=, \subseteq\}$ і операцій $\Omega_F = \{\cup, \circ, *\}$, де $\cup$ – теоретико-множинне об'єднання, $X \circ Y \overset{def}{=} \{xy : x \in X, y \in Y\}$, $X^* \overset{def}{=} \bigcup_{i=0}^{\infty} U_i$, $U_0 = \{\varepsilon\}$, $U_1 = X$, $U_i = U_{i-1} \circ X$. Ω_L-терми системи $\Re$ називаються *регулярними виразами*, а їхні значення в $\Re$ – *регулярними мовами.* Для спрощення запису регулярних виразів використовується пріоритет операцій (відповідає порядку операцій у Ω_f), можуть опускатися символ множення $\circ$ і фігурні дужки в одноелементних множинах. Наприклад, $\{c_1, c_2\} c_3 = \{c_1, c_2\} \circ \{c_3\} = \{c_1 c_3, c_1 c_3\}$.

Приклад 1.19. Регулярні мови.

Вирази: 1) $\{a\}^*$; 2) $a \cup \{ba\}^*$; 3) $\{aa\}^*$ задають регулярні мови:

1) $\{a^n : n \geq 0\}$;

2) $\{\varepsilon, a, ba, baba, bababa, ...\}$;

3) $\{a^{2n} : n \geq 0\}$ ■

Регулярні вирази широко використовуються в сучасних мовах програмування[22], при побудові лексичних аналізаторів тощо.

Системи алгоритмічних алгебр (САА). Були введені В.М. Глушковим у зв'язку із задачами синтезу й аналізу дискретних перетворювачів інформації та поклали початок застосуванню алгебричних методів у проектуванні ЕОМ і програмуванні.

Сигнатура Ω САА має такий склад:

$$\Omega_s = \{\text{func}, \text{pred}\}, \Omega_c = \{\text{id}, \text{f}_1, ..., f_n : \text{func}, p_1, ..., p_m : \text{pred}\},$$

$$\Omega_p = \{=, \subseteq : \text{func} \times \text{func}\}, \ \Omega_f = \begin{Bmatrix} \circ : \text{func} \times \text{func} \mapsto \text{func} \\ \bullet : \text{pred} \times \text{func} \mapsto \text{pred} \\ (\vee) : \text{pred} \times \text{func} \times \text{func} \mapsto \text{func} \\ \{\ \} : \text{pred} \times \text{func} \mapsto \text{func} \\ \vee, \wedge : \text{pred} \times \text{pred} \mapsto \text{pred} \\ \neg : \text{pred} \mapsto \text{pred} \end{Bmatrix}.$$

[22] Наприклад, у таких мовах, як XML, HTML, Perl, Python та інших мовах WEB-програмування.

Зафіксуємо певну множину A. Нехай $F(A)$ позначає сукупність усіх відображень типу $A \to A$, а $P(A)$ – усіх предикатів на A. Відображення та предикати в САА називають відповідно операторами й умовами. Нехай $i_A, F_1, ..., F_n$ – одиничний і довільні оператори з $F(A)$, а $P_1, ..., P_m$ – умови з $P(A)$. Визначимо *САА* як Ω-систему *FP* із множинами сортів $\Omega_S = \{F(A), P(A)\}$, констант $\Omega_C = \{i_A, F_1, ..., F_n, P_1, ..., P_m\}$, предикатів $\Omega_P = \{=, \subseteq\}$ і операцій $\Omega_F = \{\circ, (\to), \{\to\}, \bullet, \neg, \wedge, \vee\}$. В останній сукупності $\circ$ – множення операторів, далі йдуть відповідно композиції розгалуження, ітерації, множення оператора на умову, диз'юнкції, кон'юнкції й заперечення.

Як і в регулярних мовних виразах, для спрощення регулярних САА-термів використовується пріоритет операцій (відповідає порядку в Ω_f) і опускаються символи множення $\circ$ та $\bullet$. Наприклад, вираз $f_1\{p_2 \to f_2\}f_4$ скорочує мікропрограму $((f_1 \circ \{p_2 \to f_2\}) \circ f_4)$, а вираз $\neg p_1\{p_2 \to f_2\} \vee p_4$ – мікропрограму $(\neg(\{p_2 \to f_2\} \bullet p_1) \vee p_4)$.

Якщо сигнатуру САА доповнити символами чотирьох однотипних композицій $\uparrow, \downarrow, \to, [] : \text{pred} \times \text{func} \mapsto \text{func}$ – відповідно композицій звуження, обмеження, обходу й повторення, а також символом композиції недетермінованого вибору $||: \text{pred} \times \text{func} \times \text{pred} \times \text{func} \mapsto \text{func}$, то отримаємо сигнатуру $\Omega_{RAA} = (\Omega_s; \Omega_c, \Omega_v, \Omega_f, \Omega_p)$ *регулярної алгоритмічної алгебри.*

Вільні Ω-системи. При вивченні й моделюванні Ω-систем важлива роль належить вільним Ω-системам. Носієм вільної системи $\Im$ даного типу Δ є сукупність $T(\Omega)$ усіх можливих типізованих Ω-термів, що відповідають певній сигнатурі функціональних і предметних констант-твірних. Нехай $\Omega = (\Omega_s; \Omega_c, \Omega_v, \Omega_f, \Omega_p)$ – певна сигнатура типу $\Delta = (\sigma_1, \sigma_2, ...; \tau_1, \tau_2, ...; \nu_1, \nu_2, ...)$, а f_i – функціональний символ типу $\tau_i = s_{i_1} \times ... \times s_{i_n} \to s_{i_{n+1}}$. Тоді йому відповідає конструктор термів $\hat{f}_i$, який за термами $t_{i_1}, ..., t_{i_n}$ типів відповідно $s_{i_1}, ..., s_{i_n}$ будує терм $f_i(t_{i_1}, ..., t_{i_n})$ типу $s_{i_{n+1}}$, тобто $\hat{f}_i(t_{i_1}, ..., t_{i_n}) \stackrel{def}{=} f_i(t_{i_1}, ..., t_{i_n})$. Визначимо сукупність $T(\Omega)$ як замкнення множини Ω_c предметних констант ві-

дносно всіх операцій-конструкторів $\hat{f}_i$ сигнатури Ω_f. Уся сукупність $T(\Omega)$ за означенням поділена на типи. Покладемо, що T_k – сукупність усіх термів типу s_k.

Вільною Ω-системою $\Im = \left(\{T_k\}, \Omega_c, \Omega_v, \left\{\hat{f}_i^{\tau_i}\right\}, \{\hat{p}_j\}\right)$ типу Δ називається довільна Ω-система на універсумі термів $T(\Omega)$, операції та предикати якої є операціями-конструкторами термів і предикатами на термах відповідних типів.

Одним із найпростіших прикладів вільних Ω-систем є *вільна півгрупа* $\Im(\Sigma) = \left(\Sigma^*, \Sigma, \circ, =\right)$ слів певного алфавіту Σ з операцією конкатенації слів і рівністю.

Покажемо, що будь-яка Ω-система $\boldsymbol{A} = (A; \Omega_C, \Omega_F; \Omega_P)$ типу Δ із сукупністю $\Omega_C = \{\bar{c}_1 : \sigma_1, \bar{c}_2 : \sigma_2, \ldots\}$ констант-твірних і сукупностями $\Omega_F = \left\{F_1^{\tau_1}, F_2^{\tau_2}, \ldots\right\}$, $\Omega_P = \left\{P_1^{\nu_1}, P_2^{\nu_2}, \ldots\right\}$ основних операцій і предикатів є сильною моделлю вільної Ω-системи $\Im$ типу Δ, тобто система $\boldsymbol{A}$ ізоморфна певній фактор-системі $\Im / \sigma_\phi$ вільної системи $\Im$ типу Δ.

Поняття гомоморфізму багатосортних систем Ω-системи певного типу не відрізняється від аналогічного поняття для звичайних систем. Єдине, що вимагається від гомоморфізму багатосортних систем, – щоб він додатково зберігав тип елементів.

Теорема 1.4 (про вільні Ω-системи). Відображення $\phi : \Omega_c \to \Omega_C$ таке, що $c_i\phi = \bar{c}_i$, $i \geq 1$, може бути єдиним чином розширене до гомоморфізму "на" певної вільної системи $\Im = \left(\left\{T_{s_i}\right\}, \Omega_c, \Omega_v, \left\{\hat{f}_i^{\tau_i}\right\}, \left\{\hat{p}_j^{\nu_j}\right\}\right)$ на систему $\boldsymbol{A}$.

Доведення. Покладемо:

1) $\hat{f}_i^{\tau_i}\left(t_{i_1}, \ldots, t_{i_n}\right)\phi \overset{def}{=} F_i^{\tau_i}(t_{i_1}\phi, \ldots, t_{i_n}\phi)$ для кожного функціонального символу $f_i^{\tau_i}$ типу $\tau_i = s_{i_1} \times \ldots \times s_{i_n} \to s_{i_{n+1}}$ і всіх термів t_{i_k} типу s_{i_k}, $1 \leq k \leq n$;

2) $\hat{p}_j^{\nu_j}\left(t_{i_1}, \ldots, t_{i_m}\right) \overset{def}{=} P_j^{\nu_j}\left(t_{i_1}\phi, \ldots, t_{i_m}\phi\right)$ для кожного предикатного символу $\hat{\mathrm{p}}_j^{\nu_j}$ типу $\nu_j = s_{i_1} \times \ldots \times s_{i_m}$ і всіх термів t_{i_k} типу s_{i_k}, $1 \leq k \leq m$.

Покажемо, що ϕ – гомоморфізм. Дійсно, за означенням

$\hat{f}_i^{\tau_i}\left(t_{i_1}, \ldots, t_{i_n}\right)\phi = f_i^{\tau_i}\left(t_{i_1}, \ldots, t_{i_n}\right)\phi$ /*означення $\hat{f}_i^{\tau_i}$ */ $= F_i^{\tau_i}\left(t_{i_1}\phi, \ldots, t_{i_n}\phi\right)$ /*Def ϕ */.

Нехай $\varphi : T(\Omega) \to A$ – інший гомоморфізм і $c_i\phi = c_i\varphi$, $i \geq 1$. Покажемо, що $\phi = \varphi$. Застосуємо структурну індукцію для термів. База індукції виконується. Візьмемо довільний функціональний символ $f_i^{\tau_i}$ типу $\tau_i = s_{i_1} \times \ldots \times s_{i_n} \to s_{i_{n+1}}$ і довільні терми t_{i_k} типу s_{i_k}, $1 \leq k \leq n$. Припустимо, що $1 \leq \forall k \leq n$ виконується $t_{i_k}\phi = t_{i_k}\varphi$. Тоді

$\hat{f}_i^{\tau_i}\left(t_{i_1},\ldots,t_{i_n}\right)\varphi = F_i^{\tau_i}\left(t_{i_1}\varphi,\ldots,t_{i_n}\varphi\right)$ /*означення φ-гомоморфізму*/ =

$= F_i^{\tau_i}\left(t_{i_1}\phi,\ldots,t_{i_n}\phi\right)$ /*індуктивне припущення*)/ =

$= \hat{f}_i^{\tau_i}\left(t_{i_1},\ldots,t_{i_n}\right)\phi$ /*означення ϕ-гомоморфізму*/.

Відображення $\phi : T(\Omega) \to A$ є відображенням "на", тому що сукупність Ω_C за умовою є твірною системи $\boldsymbol{A}$ ■

Як свідчить теорема 1.1, вибираючи в кожному фактор-класі $\Im / \sigma_\phi$ певний Ω-терм за канонічний представник усього класу, завжди можна отримати термальну копію будь-якої системи $\boldsymbol{A}$ й перейти від абстрактних елементів носія $\boldsymbol{A}$ до конкретної множини Ω-термів. Це відкриває універсальний шлях до словарного моделювання будь-яких Ω-систем. Зрозуміло, він не завжди забезпечує необхідний рівень ефективності моделей, але важливо, що такий шлях існує і ним можна скористатися.

Нестандартні алгебричні системи. Як зазначалось у підрозд. 1.1, потреби метаматематики й програмування привели до необхідності узагальнення понять звичайної n-арної операції й алгебричної системи. Одне з найпростіших із них ми вже розглянули – поняття багатосортної операції і Ω-системи. Наступними узагальненнями є: 1) перехід від векторів фіксованої довжини до довільних послідовностей (як скінченних, так і нескінченних), а також 2) до довільних скінченних і нескінченних індексованих сукупностей елементів. Дослідження в цих напрямах були розпочаті польськими математиками й логіками всередині минулого сторіччя. Для нових узагальнених операцій було побудовано алгебричну теорію, аналогічну традиційній (теореми про гомоморфізми, конгруенції, вільні алгебри тощо). Деякі з цих результатів викладено в [105]. Потужним імпульсом для прискорення й розвитку подібних результатів стали дослідження з денотаційної та композиційної семантики мов програмування (див.

літературу для СР). Ми покажемо, як застосовуються узагальнені операції при описі семантики вихідних систем у підрозд. 2.1.2.

1.4.4. ДЕЯКІ ПРИКЛАДНІ БАГАТОСОРТНІ Ω-СИСТЕМИ

Кілька важливих прикладних Ω-систем уже було розглянуто вище. Продовжимо їхній розгляд.

Матрична алгебра. Нехай R – певна сукупність чисел. *Числовою матрицею* розміром $n \times m$ над R називається вектор A довжиною n, компоненти якого називаються *рядками* та є, у свою чергу, числовими векторами довжиною m над R. Для позначення елементів матриці використовують подвійну індексацію й пишуть $A = \left(a_{ij}\right)_{i=1\,j=1}^{n\;\;m}$, де a_{ij} означає j-й елемент у i-му рядку, або просто $A = \left(a_{ij}\right)$, якщо n та m фіксовані. Якщо $n = m$, то матриця називається *квадратною*. Основними операціями на матрицях є такі. Нехай $\lambda \in R$, $\bar{c} = \left(c_1,\ldots,c_n\right)$ – довільний вектор довжиною n на R, A,B,C – матриці розмірами відповідно $n \times m$, $n \times m$ та $m \times k$.

У результаті *множення матриці на скаляр і вектор* отримаємо матрицю $\lambda \circ A \overset{def}{=} \left(\lambda a_{ij}\right)_{i=1,\,j=1}^{n,m}$ того самого розміру $n \times m$ і вектор довжиною m $\bar{c} \times A \overset{def}{=} \left(d_j\right)_{j=1}^{m}$, $d_j = \sum_{l=1}^{n} c_l a_{lj}$, $j = \overline{1,m}$.

Додавання матриць повертає матрицю того самого розміру:

$$A + B \overset{def}{=} \left(a_{ij} + b_{ij}\right)_{i=1,\,j=1}^{n,m}.$$

Множення матриць повертає матрицю розміром $n \times k$:

$$A \times C \overset{def}{=} \left(d_{ij}\right)_{i=1\,j=1}^{n\;\;k}, \text{ де } d_{ij} = \sum_{l=1}^{m} a_{il} c_{lj}.$$

У матричній алгебрі є багато законів, що виконуються для чисел множини *R*. Наприклад, додавання цілих матриць асоціативне й комутативне, множення асоціативне тощо (вправа 13).

Реалізація й аналіз алгоритмів матричної алгебри розглядаються в підрозд. 4.2.

Списки, стеки й черги. Кортежі, елементами яких можуть бути інші кортежі, називаються *списками.* Зафіксуємо певну множину U, елементи якої називаються *атомами.* Побудуємо сім'ю множин U_i: $U_0 = U$, $U_{i+1} = U_i \cup U_i^*$, $i > 0$. Отже, U_1 складають атоми й кортежі з атомів. Останні називають *лінійними списками.* До U_2 додаються кортежі кортежів і т. д. Сукупність списків $\mathrm{Lst} = U^\infty$ визначається як

$$U^\infty \overset{def}{=} \bigcup_{i=1}^{\infty} U_i \setminus U_0 .$$

Приклад 1.20. Списки.

Нехай $U = \{A, B, C, +, -, *, /, [,]\}$, тоді

A – атом;

[ ] – порожній список;

$[A]$ – лінійний список;

$[[A]]$ – нелінійний список;

$[[A,+,B],*,C],[[*,[+,A,B],C],[A,B,+,C,*]$.

Останні три списки подають різні форми термів для арифметичного виразу $(A+B)*C$. Це відповідно $(A+B)*C$ – інфіксна форма; $*(+(A,B),C)$ – префіксна форма та $A\ B + C\ *$ – ОПЗ. Звичайна постфіксна форма цього виразу має вигляд $((A,B)+,C)*$ ■

Основними операціями на списках є: $\mathrm{cons} : U \times \mathrm{Lst} \to \mathrm{Lst}$ – додавання ліворуч елемента до списку; конкатенація списків; вставлення й вилучення елементів зі списку; пошук елементів у списку; перевірка, чи не є список порожнім. Визначимо операцію cons: для атома $u \in U$ та списку $s \in \mathrm{Lst}$ $\mathrm{cons}(u, s) = [u, s]$.

Сукупність *стеків* St утворюють лінійні списки з U_1, тобто $St = U_1 \setminus U$, але операції в них інші: $\mathrm{push} : U \times St \to St$, $\mathrm{read} : St \to U$, $\mathrm{pop} : St \to St$ і предикат $\mathrm{empty} : St \to \mathrm{Bool}$, а саме:

$\mathrm{push}(a, [a_1, ..., a_k]) = [a_1, ..., a_k, a]$ – додати елемент у стек;

$\mathrm{read}([a_1, ..., a_k]) = (k > 0 \to a_k \mid \#)$ – прочитати елемент у стеку;

$\mathrm{pop}([a_1, ..., a_k]) = (k > 0 \to [a_1, ..., a_{k-1}] \mid \#)$ – зняти елемент зі стеку;

$\text{empty}(s) \Leftrightarrow s = [\ ]$ – перевірка, чи є стек порожнім.

Як бачимо, зі стеку можна видалити лише той елемент, що був доданий останнім (*останнім прийшов – першим пішов*), тому стеки ще називають списками LIFO (англ. – Last-In, First-Out).

Сукупність Qn *черг,* як і стеків, утворюють лінійні списки з U_1, тобто $Qn = U_1 \setminus U$. Вони мають ті самі операції, що й стеки, але відрізняються означенням операцій $\text{read} : St \to U$ та $\text{pop} : St \to St$:

$\text{read}([a_1,...,a_k]) = (k > 0 \to a_1 \mid \#)$;

$\text{pop}([a_1,...,a_k]) = (k > 0 \to [a_2,...,a_k] \mid \#)$.

Із черги можна видалити лише той елемент, який перебуває в ній найдовше (*першим прийшов – першим пішов*), тому черги також називають списками FIFO (англ. – First-In, First-Out).

Програмні моделі списків, стеків, черг і реалізація основних операцій з ними розглядаються в підрозд. 4.3.

Орієнтовані графи. Мультиграфи. Мережі. Орієнтовані графи (надалі – просто графи) є певною спеціалізацією поняття бінарного відношення. До цього розглядались найбільш загальні властивості бінарних відношень. Тепер ми зробимо наступний крок і, обмежившись скінченним відношенням, розглянемо тонші зв'язки між його елементами. Саме завдяки цим зв'язкам графи знайшли широке застосування в інформатиці. Зокрема це стосується подання складноструктурованої інформації. Фактично ми хочемо перейти від загального поняття до глибшого розглядання певного його фіксованого смислового варіанта. При подібних переходах семантика понять залишається незмінною, а от синтаксис і термінологія часто підлаштовуються під нові постановки задач – достатньо згадати перехід від загального поняття кортежу до його спеціалізації у вигляді векторів, матриць, слів, списків, черг, стеків.

Графом називається пара $G = (V, R)$, де V – сукупність *вершин*, а $R \subseteq G \times G$ – скінченне бінарне відношення на V, елементи якого називаються *ребрами*. Про ребро $(a,b) \in R$ кажуть: 1) воно *зв'язує* вершину a з вершиною b; 2) a є *початком* ребра, а b – його *кінцем;* 3) ребро *виходить* із вершини a і *заходить* у вершину b. Графи та їхні фрагменти зручно подавати графічно, для чого на площині вибирають точки-вершини та з'єднують їх стрілками-ребрами (рис. 1.6):

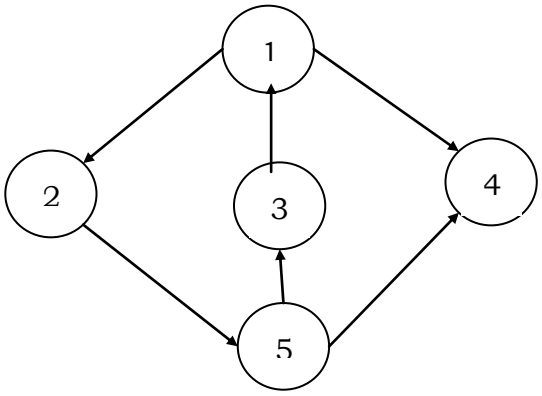


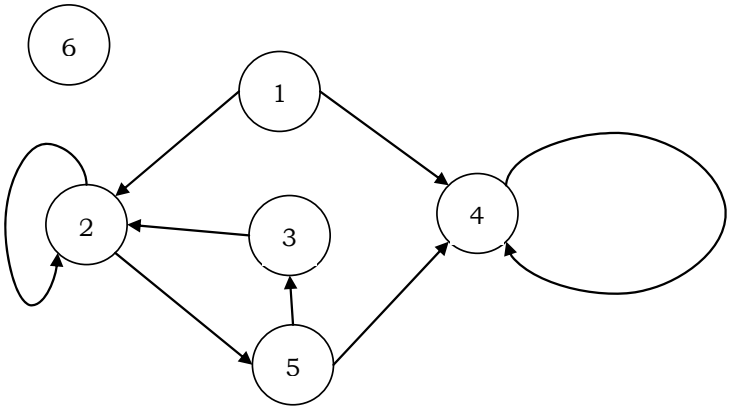
Рис. 1.6

Граф називається *порожнім*, коли порожньою є одна з двох його складових. Якщо порожня сукупність ребер, то в графічному поданні будуть відображені тільки вершини, якщо ж порожньою є й сукупність вершин, то графічне подання буде теж порожнім. Однак у будь-якому разі слід пам'ятати, що, незалежно від подання, порожній граф має такий самий статус, як і всі інші графи. *Шляхом* у графі називається послідовність ребер, в якій початок кожного наступного ребра збігається з кінцем попереднього. *Ко-шлях* – це послідовність ребер, в якій кінець кожного наступного ребра збігається з початком попереднього. *Зв'язок* – послідовність шляхів і ко-шляхів така, що початок кожного наступного члена збігається з кінцем попереднього. *Цикл* – це скінченний шлях, в якому початок і кінець збігаються. Шлях (ко-шлях) *максимальний,* якщо його не можна продовжити. *Початкова вершина* – це вершина, в яку не входить жодне ребро. З *кінцевої вершини* не виходить жодного ребра. *Ізольована вершина* є одночасно й початковою, і кінцевою. Граф *зв'язний*, якщо між кожними його двома вершинами є зв'язок. *Підграфом* графа $G=(V,R)$ називається граф $G=(V',R')$, де $V'\subseteq V$, а $R'\subseteq R$. Підграф графа є його *фрагментом,* якщо він містить усі зв'язки графа між вершинами фрагмента.

Проілюструємо введені поняття на прикладі незв'язного графа.

Приклад 1.21. Розглянемо граф, зображений на рисунку. Він має такі цикли: $2-5-3-2$; 2; 4 – останні два є *петлями*.

Початкові вершини – 1 та 6. Кінцеві вершини – 4 та 6. Ізольована вершина – 6. Підграф, що містить вершини $1-2-3-4-5$ і всі ребра, зв'язний.



■

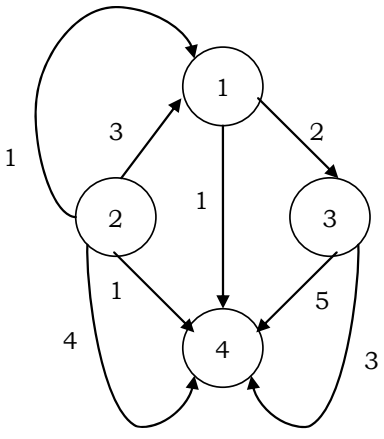
Неорієнтованим називається граф $G=(V,R)$ із симетричним відношенням R. У зображенні такого графа на площині дві стрілки, що з'єднують певну пару вершин, замінюють однією лінією без напрямку (див. прикл. 1.22).

Мережа – зв'язний граф без циклів.

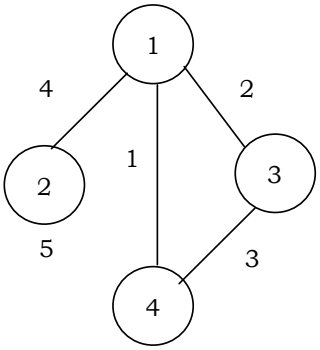
Зважений граф – трійка (G,α,β), де G – граф, α – функція, областю визначення якої є сукупність вершин G, β – функція, областю визначення якої є сукупність ребер G.

Мультиграф – це граф, ребра якого проіндексовані натуральними числами. На відміну від графа, у мультиграфі може бути кілька ребер з однаковими початками й кінцями. У графічному зображенні таким ребрам відповідають різні стрілки.

Приклад 1.22. Зважені мультиграф (а) і неорієнтований граф (б):



a)



b)

■

Розгляду й реалізації деяких важливих алгоритмів на графах присвячено підрозд. 4.6.

Дерева. *Деревом* називається граф, в якому існує початкова вершина – *корінь,* з якої в будь-яку іншу вершину існує рівно один шлях. Вершини дерева називаються *вузлами.*

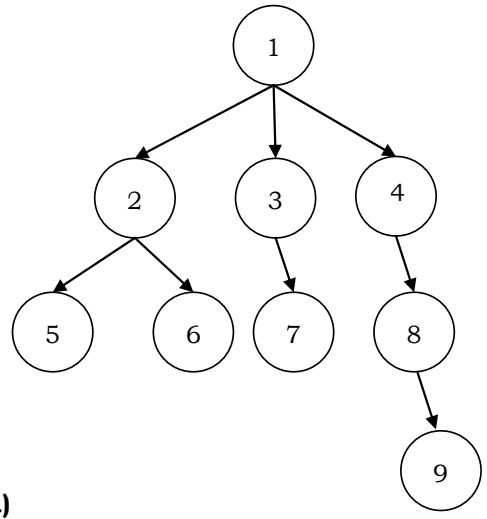
Лема 1.5 (про дерево). Граф є деревом $\Leftrightarrow$ коли виконуються умови: 1) у графі є тільки одна початкова вершина; 2) у будь-який вузол входить рівно одне ребро; 3) усі шляхи скінченні.

Доведення. Див. вправу 33 ■

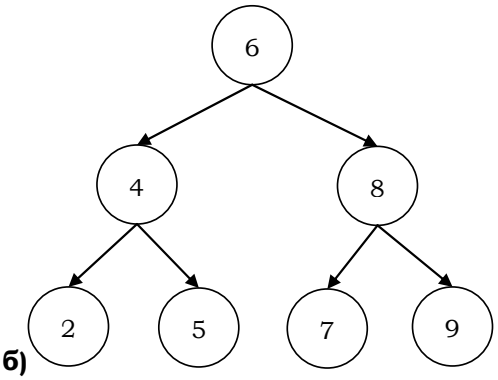
Кінцеві вершини дерева називаються *листками*. Нескладно впевнитись, що будь-який зв'язний фрагмент дерева є деревом, яке називається *піддеревом* із коренем у даному вузлі. У *бінарному* дереві кожне піддерево складається з кореня, з'єднаного з двома піддеревами, одне або два з яких (у випадку листків) можуть бути порожніми. Зазвичай ці два піддерева впорядковані між собою, одне з них називається *лівим*, а друге – *правим. Висотою* дерева називається найбільша з довжин його максимальних шляхів, тобто найбільша відстань від кореня до листка в дереві. Нехай на множині вузлів дерева задано певний лінійний порядок $(V, \prec)$. Бінарне дерево називається *деревом пошуку*, якщо в ньому для кожного вузла вершини лівого піддерева менші за нього, а вершини правого – більші. Відомо, що час перевірки, чи належить даний елемент дереву пошуку, не перевищує його висоти. *Збалансовані дерева* – це дерева, в яких для будь-якої вершини кількість елементів у лівому й правому піддеревах відрізняється не більше ніж на 1. Зрозуміло, що для збалансованого дерева в загальному випадку час пошуку кращий. Проте іноді в дереві можуть бути зовсім відсутні ліві або праві піддерева. Такі дерева називаються *виродженими*. Висота їх дорівнює кількості вузлів у дереві. Для таких дерев пошуку пошук елемента перетворюється на звичайний послідовний пошук. Тому інколи при кожному вставленні елемента в дерево пошуку дерево корегують так, щоб воно було збалансованим.

Проілюструємо введені поняття.

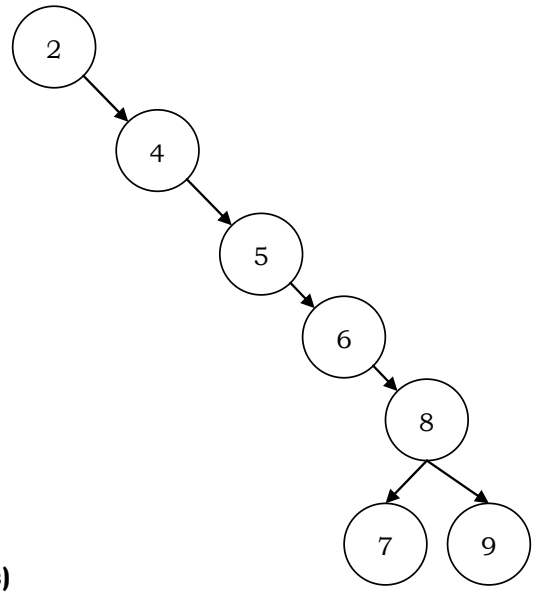
Приклад 1.23. На рисунку зображено дерево з початковою вершиною (вузлом) 1 і листками 5, 6, 7, 9 (а). Максимальні шляхи в ньому: 1–2–5, 1–2–6, 1–3–7, 1–4–8–9. Таким чином, висота дерева дорівнює 3. На рис. б), в) зображені відповідно збалансоване й незбалансоване цілочислові дерева пошуку.



а)



б)



в)

■

Деякі важливі алгоритми на бінарних деревах розглядаються в підрозд. 4.3.

Скінченні автомати. Скінченні X-автомати – це спеціальний клас обчислювальних процедур (див. підрозд. 1.3.4), призначений для синтаксичного аналізу регулярних мов. Обчислення в таких процеду-

рах відбуваються в натуральному часовому просторі, а стани мають структуру пар (q,u), де q належить певній скінченній сукупності Q внутрішніх станів, а u – слово в певному вхідному алфавіті X. Перебуваючи в стані (q_t,u_t) і читаючи першу літеру слова u_t, незалежно від моменту часу t автомат переходить у новий стан (q_{t+1},u_{t+1}), в якому друга компонента або збігається зі словом u_t, або є його закінченням без першої літери. Автомат, розпочавши свою роботу в довільній вхідній конфігурації, може закінчити її після переходу в один із заключних внутрішніх станів. Результат розпізнавання стосується прочитаного префікса початкового слова й буде для нього позитивним, якщо автомат після його прочитання перейде в заключний стан. Кажуть, що цей префікс *допускається* автоматом. Для всіх інших прочитаних префіксів початкового слова результат розпізнавання негативний. Автомат може також продовжити обчислення після переходу в заключний стан (тобто скінченні X-автомати мають справу з гіперобчисленнями). Якщо префікс, що допускається автоматом, збігається зі вхідним словом обчислення, то кажуть, що вхідне слово допускається автоматом. Заключний стан вигляду (q_t,ε) і обчислення, яке закінчується таким станом, будемо називати *термінальними.* Для префікса υ, що допускається автоматом, завжди існує результативне термінальне обчислення зі вхідним станом (q_0,υ). Таким чином, сукупності всіх префіксів і слів, які допускаються автоматом A, збігаються й називаються *мовою* $L(A)$, *що допускається* цим автоматом. Подібні мови називають *скінченно-автоматними.*

Формально *скінченний* X*-автомат* – це п'ятірка $A=(Q,X,\delta,q_0,F)$, де:

Q – скінченна множина внутрішніх станів;

X – скінченна множина вхідних символів (вхідний алфавіт);

$\delta: Q\times(X\cup\varepsilon)\to B(Q)$ – *функція переходів*, керує роботою автомата;

$q_0\in Q$ – початковий внутрішній стан;

$F\subseteq Q$ – множина заключних станів.

Перехід $\delta(q,\varepsilon)=p$ означає, що автомат у процесі обчислення переходить у новий стан без зміни другого компонента. *Недетермінізм* автомата виявляється в тому, що, перебуваючи в деякому стані й читаючи поточний символ, автомат може перейти в один із кількох можливих станів або здійснити ε-перехід. Якщо, перебуваючи в будь-якому стані й читаючи поточний символ, автомат може перейти не

більше ніж в один внутрішній стан і при цьому не має ε-переходу, то він називається *детермінованим*.

Скінченні автомати зручно задавати у вигляді орієнтованих зважених графів. Наприклад, граф (рис. 1.7) задає недетермінований автомат $A_1 = (Q, X, \delta, q_0, F)$, де $Q = \{q_0, q_1, q^*\}$, $F = \{q^*\}$, $X = \{0,1,2\}$, δ складають команди $q_0 1 \to q_1$; $q_0 2 \to q_1$; $q_1 \varepsilon \to q^*$; $q_1 0 \to q_1$.

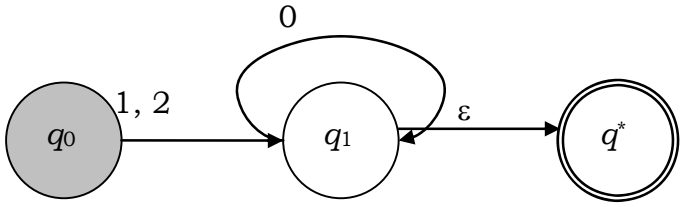


Рис. 1.7

Недетермінізм автомата A_1 виявляється в тому, що в стані q_1 він може або прочитати символ 0, або здійснити ε-перехід.

Шлях у графі автомата називається *повним*, якщо він розпочинається в початковій вершині й закінчується в одній із заключних. Кожному повному шляху в автоматному графі відповідає певне результативне термінальне обчислення автомата, і навпаки. Слово в алфавіті *X*, побудоване в результаті конкатенації символів, якими зважені стрілки в повному шляху, є таким, що допускається автоматом. Наприклад, для вищенаведеного автомата слово 100 допускається, а 11 – не допускається, тому що після переходу в стан q_1 автомат уже не може прочитати символ 1. Нескладно впевнитись, що автомат A_1 допускає мову $\mathrm{L}(A_1) = 1(0^*) \cup 2(0^*)$.

Аналогічно граф на рис. 1.8 задає детермінований автомат $A_2 = (Q, X, \delta, q_0, F)$, де $Q = \{q_0, q^*\}$, $F = \{q^*\}$, $X = \{a, b, c\}$, $\delta = \{q_0 a \to q^*; q^* b \to q^*; q^* c \to q_0\}$. Очевидно, що $\mathrm{L}(A_2) = ab^* (cab^*)^*$.

Введемо відношення безпосереднього переходу на конфігураціях скінченного автомата $A: (q, aw) \vdash (p, w)$, якщо в автоматі є команда $qa \to p$. Нехай $\vdash^*$ – рефлексивно-транзитивне замикання цього від-

ношення, тоді слово w *допускається* автоматом A, якщо $(q_0, w) \vdash^* (p, \varepsilon)$, де q_0 – початковий стан, а $p \in F$.

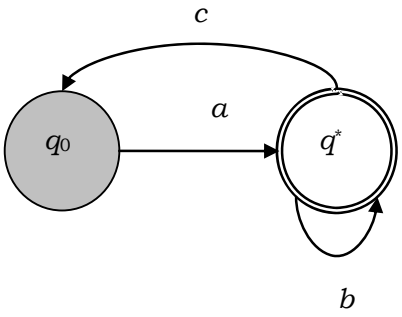
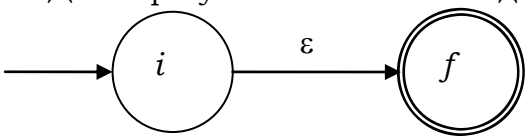


Рис. 1.8

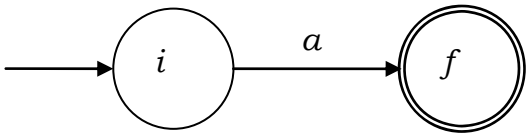
Теорема 1.5 (Кліні). Класи регулярних і скінченно-автоматних мов збігаються.

Доведення. Покажемо, як побудувати скінченний автомат A, що розпізнає регулярну мову, задану даним виразом R. Автомат A будується відповідно до структури виразу R, має строго один заключний стан, у ньому немає переходів у початковий стан та із заключного стану в інші. Нехай B та C – недетерміновані скінченні автомати для регулярних виразів T та S, відповідно.

1. Для виразу $R = \varepsilon$ A має вигляд :



2. Для виразу $R = a$ $(a \in T)$ A має вигляд :



3. Для виразу $R = T \cup S$ автомат A будується, як показано на рис. 1.9. Тут i – новий початковий стан, f – новий заключний стан. Автомат A спочатку по ε переходить у початкові стани автоматів B та C, а закінчує роботу ε-переходом із заключних станів автоматів B та C у стан f. Початковий і заключний стани автоматів B та C не є такими для нового автомата A.

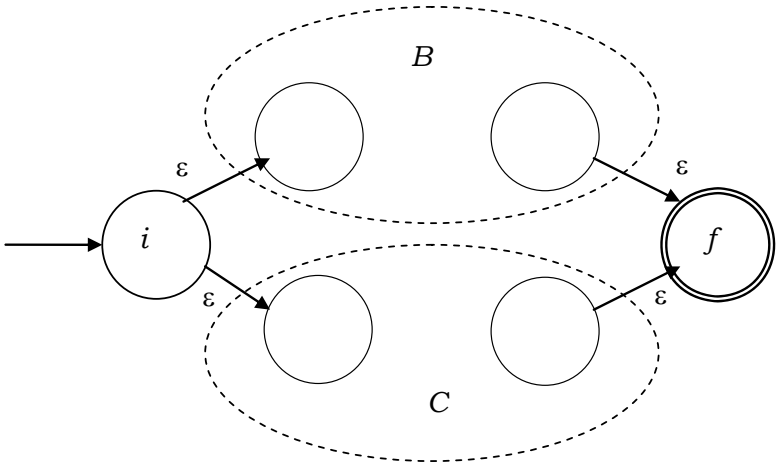


Рис. 1.9

4. Для виразу $R = TS$ автомат A будується, як показано на рис. 1.10:

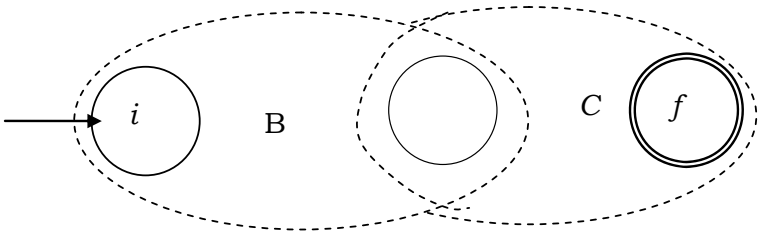


Рис. 1.10

Початковий стан автомата B стає початковим, а заключний стан автомата C – заключним для нового автомата. Початковий стан C і заключний стан B ототожнюються.

5. Для виразу $R = S^*$ автомат A будується таким чином (рис. 1.11):

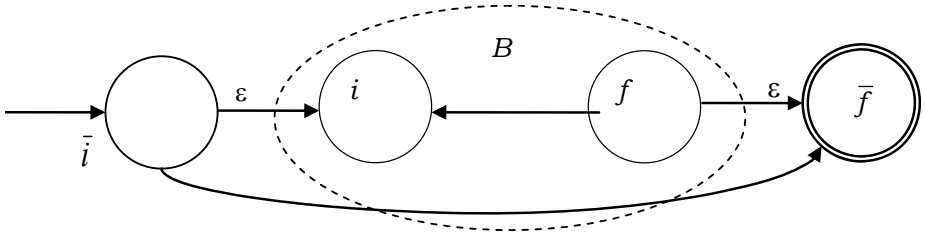


Рис. 1.11

Тут $\bar{i}$ та $\bar{f}$ – нові початковий і заключний стани. Автомат B зациклений – у ньому з'явився ε-перехід від заключного до початкового стану.

Отже, усі випадки для регулярного виразу R розглянуто.

Зворотне *доведення* можна знайти в [23, 52] (вправа 39) ■

Як показує наступна теорема, розпізнавальні можливості недетермінованих і детермінованих скінченних автоматів однакові. Автомати A та B називаються *еквівалентними*, якщо $L(A) = L(B)$.

Теорема 1.6 (детермінізація скінченних автоматів). Для довільного недетермінованого скінченного автомата $A = (Q, X, \delta, q_0, F)$ існує еквівалентний йому детермінований автомат B.

Доведення. Побудуємо детермінований скінченний автомат $B = \{2^Q, X, \delta_1, Q_0, F_1\}$. Станами його є підмножини станів недетермінованого автомата. Щоб одержати стан детермінованого автомата, в який здійснюється перехід по даному символу з даного стану $M \subseteq Q$, потрібно об'єднати всі стани недетермінованого скінченного автомата, в які існують переходи по даному символу з усіх вершин множини M, а потім приєднати до них усі вершини, досяжні зі станів M, за допомогою серій ε-переходів. Початковий стан Q_0 містить стан q_0 і всі стани, досяжні з нього за допомогою всіх можливих серій ε-переходів. Покладемо сукупність заключних станів $F_1 = \{Q' \subseteq Q : Q' \cap F \neq \varnothing\}$. За побудовою кожному обчисленню в автоматі A з довільним вхідним словом відповідає аналогічне обчислення в детермінованому автоматі, і навпаки ■

Насправді в детермінованому еквіваленті автомата A досяжними будуть не всі стани булеана 2^Q, а значно менше. Саме їх можна взяти за сукупності станів автомата B, а не весь булеан. Наприклад, з восьми станів булеана 2^Q автомата з рис. 1.7 досяжними будуть лише два: $\{q_0\}$ та $\{q_1, q^*\}$. Зведений детермінований еквівалент зображено на рис. 1.12. Він спростився за рахунок усунення ε-переходу.

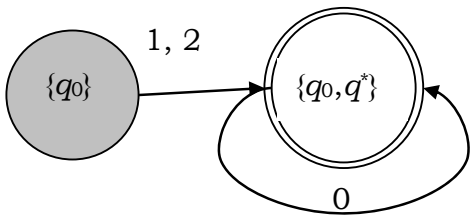


Рис. 1.12

Контекстно-вільні граматики та МП-автомати. Як і скінченні X-автомати, контекстно-вільні граматики (КВ-граматики) та МП-автомати – це спеціальні класи недетермінованих обчислювальних процедур, призначені для математичного опису синтаксису формальних мов та їхнього синтаксичного аналізу. КВ-граматики вперше були застосовані при описі мови програмування Алгол-60 у вигляді так званих форм Бекуса – Наура (БНФ). Розглянемо ці формалізми й проблему ефективного синтаксичного аналізу мов, заданих ними.

КВ-граматикою називається четвірка $G=(N,T,P,S)$, де N та S – множини *нетермінальних* і *термінальних* символів, P – множина *правил виведення* (*продукцій*) вигляду $X\to w$, де $X\in N$, $w\in(T\cup N)^*$, S – *аксіома* з множини N. Кажуть, що слово uwv *безпосередньо виводиться* зі слова uXv у граматиці G (позначається $uXv\to uwv$), якщо $X\to w\in P$. Ланцюжок слів вигляду $u_1\to u_2\to\ldots\to u_n$ називається *виведенням* (*обчисленням*) слова u_n зі слова u_1 у граматиці G. Позначимо $\Rightarrow$ рефлексивно-транзитивне замкнення відношення безпосереднього виведення в граматиці G. *Контекстно-вільною мовою* (*КВ-мовою*), породженою граматикою G, називається множина $\mathrm{L}(G)=\{x\in T^*: S\Rightarrow x\}$.

Дві *КВ*-граматики називають *еквівалентними*, якщо вони породжують одну й ту саму мову.

Приклад 1.24. Побудуємо КВ-граматику для мови $L_2=\{a^nb^n: n\ge 0\}$. КВ-граматика $G_1=(T,N,P,S)$, де $T=\{a,b\}$, $N=\{S\}$, $P=\{S\to aSb, S\to\varepsilon\}$, породжує мову L_2. Дійсно, для будь-якого натурального $n\ge 0$ у граматиці G_1 є виведення вигляду $S\Rightarrow a^nSb^n\Rightarrow a^nb^n$. Таким чином, $D\subseteq L(G_1)$. Однак інших виведень у граматиці G_1 не існує. Отже, $D=L(G_1)$ ■

Реальні КВ-граматики мов програмування мають десятки нетермінальних символів і сотні правил виведення. Для зменшення кількості правил і спрощення запису їхніх правих частин використовують розширену форму, яка не збільшує породжувальну силу КВ-граматик (див. вправу 51), але компактніша. Нові граматики отримали назву *розширених* (*РКВ-граматик*). Їх також називають *розширеними БНФ* (*РБНФ*). В останніх стрілочка в правилах замінена метасимволом ::=. У правилах РКВ-граматик використовуються метасимволи |,(,),[,],{,},+,*,-,\,<,>. Символ | читається як "або". Пара круглих дужок об'єднує кілька конструкцій в одну загальну. Конструкція у *квадрат-*

них дужках може бути відсутньою. Конструкція у *фігурних дужках* означає ітерацію записаного в них виразу (див. підрозд. 1.4.3). За нею може йти один із модифікаторів: *,+ або (n,m). *Модифікатор* ітерації означає, що вираз у дужках може повторюватися: 1) нуль або будь-яку кількість разів; 2) не менше одного разу; 3) від n до m разів.

Наприклад, розширені правила $S \to a\{S\}^*b$ та $S \to a\{S\}+b$ еквівалентні правилам зі зліченною кількістю альтернатив: $S \to ab \mid aSb \mid aSSb \mid \ldots$ та $S \to aSb \mid aSSb \mid \ldots$, відповідно, а правило $S \to a\{S\}(2,3)b$ еквівалентне правилу $S \to aSSb \mid aSSSb$.

Діапазон символів кодової таблиці скорочено записується з використанням метасимволу – : наприклад, A–Z означає сукупність усіх великих латинських літер від A до Z. *Дужки* <,> використовуються для йменування нетерміналів.

Увага! Щоб метасимвол можна було використовувати як термінальний, перед ним необхідно вжити бек-слеш \. Бек-слеш \ також застосовують для перенесення правила на наступний рядок тексту ►

Наприклад, розширене правило $S \to \backslash\{a+a\backslash\}$ еквівалентне звичайному правилу $S \to \{a+a\}$, де фігурні дужки є терміналами. Правило перенесення $S \to ABCDEFGHEIJKLMNO\backslash$ *PQRSTUVWXYZ* еквівалентне правилу $S \to ABCDEFGHEIJKLMNOPQRSTUVWXYZ$.

Іншим прикладом РКВ-граматики є граматика G_2, яка породжує сукупність арифметичних виразів. Її нетерміналами є <*вираз*>, <*доданок*> ,<*операція*1>, <*множник*> , <*операція*2>, <*змінна*> та <*число*>, аксіомою – <*вираз*>, а правилами –

<*вираз*> → <*доданок*> [<*операція*1> <*доданок*>]
<*доданок*> → <*множник*>[<*операція*2> <*множник*>]
<*множник*> → \(<*вираз*>\)|<*змінна*> | <*число*>
<*операція*1> → \+ | \-
<*операція*2> → * | /
<*число*> → {<*цифра*>}
<*цифра*> → 0 – 9
<*змінна*> → A – Z | a – z

Граматику G_2 буде використано в підрозд. 4.5. Побудуємо в ній виведення виразу $(a+12)$:

<*вираз*>→<*доданок*>→<*множник*>→(<*вираз*>)→(<*доданок*> <*операція1*> <*доданок*>)→(<*множник*> <*операція1*> <*доданок*>)→(<*змінна*> <*операція1*> <*доданок*>)→(*a* <*операція1*> <*доданок*>)→(*a* + <*дода-*

нок>) → (*a* + <*число*>) → (*a* + {<*цифра*>}) → (*a* + <*цифра*><*цифра*>) → (*a* + 1<*цифра*>) → (*a* + 12).

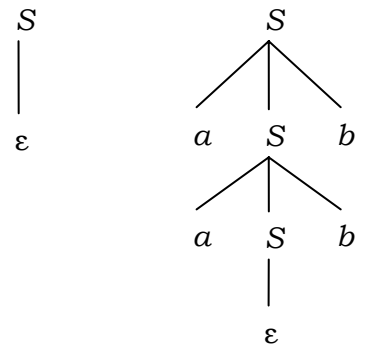
Виведення слова називається *лівостороннім,* якщо на кожному його кроці замінюється найбільш лівий нетермінал. Виявляється, що обмеження лівосторонніми виведеннями не зменшує породжувальну силу КВ-граматик (вправа 23), тому далі будемо розглядати тільки лівосторонні виведення.

Для аналізу виведень у КВ-граматиках застосовують дерева виведень. Нехай $G = (T, N, P, S)$ – довільна КВ-граматика. Зважене упорядковане дерево D називається *деревом виведення* граматики G, якщо виконуються умови:

1) Корінь дерева зважений аксіомою S.

2) Якщо $D_1 ..., D_k$ – усі піддерева дерева з коренем, зваженим нетерміналом X, а корені піддерева D_i зважені символом $X_i \in T \cup N \cup \{\varepsilon\}$, то правило $X \to X_1 ... X_k$ належить P. Якщо при цьому $X_i \in T$, то корінь піддерева D_i є листком дерева D.

Кожному виведенню з аксіоми граматики G відповідає певне дерево виведення, і навпаки, – за деревом виведення можна побудувати саме виведення (напр., лівостороннє) слова в граматиці. На рис. 1.13 зображені два дерева виведень граматики G_1:



а) б)

Рис. 1.13

Дерево а) відповідає виведенню $S \to \varepsilon$,
б) – виведенню $S \to aSb \to aaSbb \to aabb$.

З усього комплексу задач, що розглядаються в теорії КВ-граматик, зупинимось тільки на проблемі ефективного синтаксичного аналізу мов, які породжуються цими граматиками. Відомо, що алгоритм перевірки існування виведення даного слова у КВ-граматиці має складність, близьку до $O(n^3)$, де n – довжина слова, що є неприйнятним із практичного погляду []. Тому було докладено значних зусиль для пошуку класів КВ-граматик, які були б, з одного боку, достатньо потужними, а з іншого – допускали б лінійну оцінку алгоритму синтаксичного аналізу. Одним із найпростіших таких класів є клас LL(k)-граматик.

Для КВ- (РКВ)-граматики G і ланцюжка w, що складається з термінальних і нетермінальних символів, визначимо множину $\mathrm{FIRST}_k(w) = \left\{ x \in T^* \mid w \Rightarrow xv, |x| = k \text{ або } w \Rightarrow x, |x| < k \right\}$, де k – натуральне число. Множина $\mathrm{FIRST}_k(w)$ складається з термінальних префіксів довжиною k усіх слів, що виводяться з w.

Граматика G називається LL(k)-*граматикою*, якщо для кожного її нетермінала $X \in N$, пари продукцій $X \to v$, $X \to w$ і кожного лівостороннього виведення $S \Rightarrow xXu, x \in T^*$ множини $\mathrm{FIRST}_k(vu)$ та $\mathrm{FIRST}_k(vw)$ не перетинаються. LL(k)-властивість дозволяє в процесі побудови лівостороннього виведення для даного термінального слова на кожному кроці за k наступними його символами однозначно вибрати праву частину продукції для продовження виведення (звісно, якщо така взагалі існує).

Якщо взяти граматику G_1 для мови L_2, то вона є LL(1)-граматикою. Дійсно, довільне лівостороннє виведення в граматиці G_1 має вигляд $S \Rightarrow a^n S b^n, n \geq 1$ і для нього виконується LL(1)-властивість $\mathrm{FIRST}_1\left(aSb^{n+1}\right)\left(=\{a\}\right) \cap \mathrm{FIRST}_1\left(\varepsilon b^n\right)\left(=\{b\}\right) = \varnothing$.

З'ясуємо, чи виводиться в граматиці G_1 слово $aabbb$. Слово розпочинається з термінала $a \in \mathrm{FIRST}_1(aSb)$, тому на першому кроці застосовується продукція $S \to aSb$. Далі в слові знову йде літера a, тому й на другому кроці виведення має застосовуватись та сама продукція. Маємо виведення $S \to aSb \to aaSbb$. Далі в слові йде літера b. Оскільки $b \notin \mathrm{FIRST}_1(aSb)$, то продукція $S \to aSb$ застосована бути не може. Залишається продукція $S \to \varepsilon$, що приводить до виведення $S \to aSb \to aaSbb \to aabb$, яке вже не може бути продовженим. Отже, слово $aabbb$ не може бути породженим у граматиці G_1.

Для практичного застосування $\mathrm{LL}(k)$-граматик необхідно вміти знаходити множину значення функції FIRST. Нехай маємо КВ-граматику $G=(N,T,P,S)$ і ланцюжок w термінальних і нетермінальних символів. Розглянемо один із методів побудови множини $\mathrm{FIRST}_k(w)$ для КВ-граматик. Нехай x/k означає префікс слова x довжиною k. За означенням $x/k=x$, якщо $k\geq|x|$. Для мови L і натурального k покладемо, що L/k – сукупність префіксів довжиною k усіх її слів. Нехай $w=X_1X_2\ldots X_n$, $X_i\in T\cup N$, $i=\overline{1,n}$. Нескладно впевнитись, що

$$\mathrm{FIRST}_k(w)=\left(\mathrm{FIRST}_k(X_1)\mathrm{FIRST}_k(X_2)\ldots\mathrm{FIRST}_k(X_n)\right)/k.$$

Тому, щоб знайти сукупність $\mathrm{FIRST}_k(w)$, достатньо вміти знаходити сукупності $\mathrm{FIRST}_k(X)$ для $X\in N$. Зазначимо, що для $X_i\in T\cup\{\varepsilon\}$ $\mathrm{FIRST}_k(X_i)=\{X_i\}$.

Для $i\geq 0$ і кожного $X\in T\cup N$ визначимо такі мови $F_i(X)$:

1. $F_i(a)=\{a\}$, $a\in T$;
2. $F_0(X)=\left\{x: x\in T^* \;\&\; \exists X\to xu\in P\left(|x|=k\vee|x|<k\;\&\;u=\varepsilon\right)\right\}$;
3. Нехай $F_{i-1}(X)$ уже визначені для всіх $X\in N$, тоді $F_i(X)=F_{i-1}(X)\cup$ $\cup\left\{x:\exists X\to Y_1Y_2\ldots Y_m\in P\;\&\;x\in F_{i-1}(Y_1)F_{i-1}(Y_2)\ldots F_{i-1}(Y_m)/k\right\}$.

Ураховуючи, що число k фіксоване та $F_{i-1}(X)\subseteq F_i(X)$, на певному кроці l для всіх $X\in N$ сукупності $F_{l-1}(X)$ та $F_l(X)$ будуть збігатися. За побудовою $\mathrm{FIRST}_k(X)=F_l(X)$.

Приклад 1.25. Побудуємо значення функції FIRST_1 для правих частин правил граматики G_2:

$\mathrm{FIRST}_1\left(<\textit{доданок}>\;\left[<\textit{операція}1><\textit{доданок}>\right]\right)$ =
$=\{\,(,\ a,\ \ldots,\ z,\ A,\ \ldots,\ Z,\ 0,\ \ldots 9\}$;

$\mathrm{FIRST}_1(<\textit{множник}>\;[<\textit{операція}2>\;<\textit{множник}>])$ =
$=\{\,(,\ a,\ \ldots,\ z,\ A,\ \ldots,\ Z,\ 0,\ \ldots 9\}$;

$\mathrm{FIRST}_1\left((<\textit{вираз}>)\right)$ = $\{\,(\}$;

$\mathrm{FIRST}_1\left(<\textit{змінна}>\right)$ = $\{a,\ \ldots,\ z,\ A,\ \ldots,\ Z\}$;

$\mathrm{FIRST}_1\left(<\textit{число}>\right)$ = $\{0,\ \ldots,\ 9\}$.

Очевидно, граматика G_2 є $\mathrm{LL}(1)$-граматикою. Це випливає з того, що значення функції FIRST_1 для всіх альтернативних правих частин продукцій попарно не перетинаються ■

Звернемось тепер до автоматів із магазинною[23] пам'яттю (МП-автоматів) – класу процедур, призначених для розпізнавання й синтаксичного аналізу КВ-мов. Від скінченного автомата МП-автомат відрізняється наявністю в загальній структурі його станів третього компонента – стеку символів. Перші два компоненти функція переходу МП-автомата перетворює так, як і у випадку скінченних автоматів, а третю – за допомогою стекових операцій pop (зняти) і push (покласти елемент у стек).

МП-автомат – це сімка $A=(Q,X,Y,\delta,q_0,Z_0,F)$, де:

1) Q – скінченна множина внутрішніх станів;
2) Σ – скінченна множина вхідних символів (вхідний алфавіт);
3) H – скінченна множина магазинних символів;
4) $\delta: Q\times(\Sigma\cup\varepsilon)\times H\to B\left(Q\times H^*\right)$ – функція *переходів*, керує роботою автомата;
5) $q_0\in Q$ – початковий внутрішній стан керуючого пристрою;
6) $Z_0\in H$ – початковий стан стеку;
7) $F\subseteq Q\times\Sigma^*\times H^*$ – множина заключних станів.

Зазвичай множина заключних станів має вигляд $F=Q_{fin}\times\Sigma^*\times H^*$, або $F=Q_{fin}\times\Sigma^*\times\{\varepsilon\}$, для певної підмножини $Q_{fin}\subseteq Q$ заключних внутрішніх станів МП-автомата. За означенням із порожнім стеком автомат продовжувати обчислення не може.

Обчислення за МП-автоматом – це послідовність станів, в які він переходить, читаючи або не читаючи літери вхідного слова. Перебуваючи в стані $(q,au,\upsilon Z)$ і читаючи першу літеру a вхідного слова, незалежно від моменту часу t автомат A реалізує перехід $\delta(q,a,Z)=(p,\omega)$ і переходить у новий стан $(q_{t+1},u_{t+1},\upsilon_{t+1})$, в якому внутрішній стан $q_{t+1}=p$, друга компонента $u_{t+1}=u$. Якщо ж автомат не читає літеру на вході й реалізує ε-перехід $\delta(q,\varepsilon,Z)=(p,\omega)$, то $q_{t+1}=p$, $u_{t+1}=au$. В обох випадках нове значення стеку $\upsilon_{t+1}=\upsilon\omega$.

[23] Магазин – це синонім слова "стек".

При цьому $\omega = \varepsilon$ означає, що зі стеку знімається верхній елемент (операція pop), $\omega \neq \varepsilon$ – що зі стеку знімається верхній елемент і в нього послідовно вносяться символи слова ω (серією операцій push).

МП-автомат називається *недетермінованим*, якщо, перебуваючи в деякому стані й читаючи поточний символ, він може перейти в один із кількох можливих станів або здійснити ε-перехід. Якщо, перебуваючи в будь-якому стані й читаючи поточний символ, автомат може перейти не більше ніж в один стан і при цьому не має ε-переходу, то він, як і скінченний автомат, називається *детермінованим.*

Введемо відношення $\vdash$ безпосереднього переходу на конфігураціях МП-автомата. Оскільки значення функції переходу не залежить від часової компоненти, то останню можна не показувати в конфігураціях автомата. Покладемо $(q, au, \upsilon Z) \vdash (p, u, \upsilon\omega)$ для переходу $\delta(q, a, Z) = (p, \omega)$ і $(q, au, \upsilon Z) \vdash (p, au, \upsilon\omega)$ – у випадку ε-переходу $\delta(q, \varepsilon, Z) = (p, \omega)$. Нехай $\vdash^*$ – рефлексивно-транзитивне замикання відношення безпосереднього переходу.

Слово w *розпізнається* автоматом A, якщо $(q, w, Z_0) \vdash^* (p, \varepsilon, \upsilon)$, де q_0 – початковий стан, а $p \in F$. Позначимо $L(A)$*-мову, що розпізнається МП-автоматом* A.

Приклад 1.26. МП-автомат для мови L_2 із прикл. 1.24.

Покладемо $Q = \{q_0, q^*\}$, $\Sigma = \{a, b\}$, $H = \{Z_0, 1\}$, $\delta = \{(q_0, a, Z_0) \to (q_0, 1), (q_0, a, 1) \to (q_0, 11), (q_0, b, 1) \to (q^*, \varepsilon),$ $(q^*, b, 1) \to (q^*, \varepsilon)\}$, $F = \{(q^*, \varepsilon, \varepsilon)\}$. Детермінований МП-автомат $A_3 = (Q, \Sigma, H, \delta, q_0, Z_0, F)$ із даними компонентами задає мову L_2. Наприклад, він допускає слово *aabb* таким чином: $(q_0, aabb, Z_0) \vdash (q_0, abb, 1) \vdash (q_0, bb, 11) \vdash (q^*, b, 1) \vdash (q^*, \varepsilon, \varepsilon) \in F$.

На першому, другому, третьому й четвертому кроках обчислення були застосовані перше, друге, третє й четверте за порядком правила переходу функції δ, після чого обчислення перейшло в заключний стан. Узагалі, перебуваючи в початковому внутрішньому стані q_0 і читаючи на вході літери a, автомат A_3 щоразу кладе в стек символ 1 (тим самим запам'ятовується кількість підряд прочитаних літер a),

доки не зустріне першу літеру b і не перейде в заключний внутрішній стан q^*. Перебуваючи далі в стані q^* і читаючи символи b, A_3 знімає зі стеку щоразу один символ 1. Перехід у єдиний заключний стан $(q^*,\varepsilon,\varepsilon)$ свідчить, що літер b було прочитано поспіль рівно стільки, скільки спочатку літер a ■

Має місце

Теорема 1.7. Клас КВ-мов збігається з класом мов, що розпізнаються МП-автоматами.

Доведення. Нехай $G=(N,T,P,S)$ – КВ-граматика. Побудуємо для неї МП-автомат $A=(Q,\Sigma,H,\delta,q_0,Z_0,F)$, який буде моделювати всі її лівосторонні виведення й розпізнавати тільки ті слова, що належать мові $\mathrm{L}(G)$. Подібні МП-автомати для КВ-граматик називаються МП-*аналізаторами типу розгортки.* Покладемо $Q=\{q\}$, $\Sigma=T$, $H=N\cup T$, $q_0=q$, $Z_0=S$, $F=\{(q,\varepsilon,\varepsilon)\}$, а функція δ визначається так: 1) для кожного правила $X\to w\in P$ $(q,w)\in\delta(q,\varepsilon,X)$; 2) $(\delta(q,a,a)=(q,\varepsilon)$ для кожного $a\in T$. Нескладно перевірити, що за побудовою $S\Rightarrow u\in T^*\Leftrightarrow(q,u,S)\vdash^*(q\varepsilon,\varepsilon)$, тобто A є МП-аналізатором типу розгортки для граматики G (вправа 27).

Для доведення зворотної теореми можна обмежитися МП-автоматами зі спустошенням. МП-автоматом *зі спустошенням* називається МП-автомат A з множиною заключних станів вигляду $F=Q_{fin}\times\{\varepsilon\}\times\{\varepsilon\}$, де $Q_{fin}\subseteq Q$. Нехай $A=(Q,\Sigma,H,\delta,q_0,Z_0,F)$ – довільний МП-автомат і $F=Q_{fin}\times\Sigma^*\times\{\varepsilon\}$ для певної підмножини $Q_{fin}\subseteq Q$. Побудуємо для A КВ-граматику $G=(N,T,P,S)$, що породжує мову $L(A)$, таким чином, щоб будь-яке лівостороннє виведення слова $u\in\Sigma^*$ у граматиці G відповідало обчисленню автомата A зі словом u на вході. Покладемо $N=\{[qZr]:q,r\in Q,Z\in H\}\cup\{S\}$. Потрібно, щоб правила граматики забезпечували справедливість твердження (*) $[qZr]\Rightarrow u\Leftrightarrow(q,u,Z)\vdash^*(r,\varepsilon,\varepsilon)$. Необхідну сукупність правил P можна отримати так:

1) для переходу $(r, X_n \ldots X_1) \in \delta(q, a, Z)$ до P додаються всі правила вигляду $[qZr] \to a\left[rX_1 s_1\right]\left[s_1 X_2 s_2\right]\ldots\left[s_{n-1} X_n s_n\right]$ для всіх можливих послідовностей $s_1, s_2, \ldots, s_n$ станів із Q;

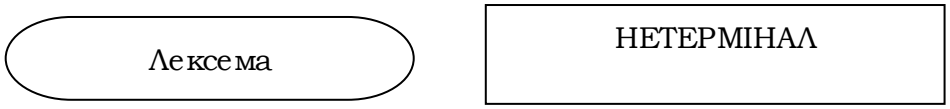
2) для переходу $(r, \varepsilon) \in \delta(q, a, Z)$ до P додається правило $[qZr] \to a$;

3) до P додаються правила $S \to [q_0 Z_0 q]$ для кожного $q \in Q$.

Нескладно перевірити, що для будь-яких $q, r \in Q$ та $Z \in H$ виконується (*) (вправа 28). Звідси випливає, що $L(G) = L(A)$ ■

МП-аналізатор типу розгортки для LL(k)-граматик досить просто детермінізувати. Для цього достатньо, щоб він: а) мав можливість перед вибором продукції забігти наперед і прочитати наступні k нерозпізнаних вхідних символів; б) умів побудувати значення функції FIRST_k для відповідних альтернатив поточного виведення. Обидві можливості реалізуються в межах скінченно-автоматного керування.

Синтаксичні діаграми. Запропоновані Н. Віртом. Можуть розглядатися як графічний варіант РКВ-граматик, орієнтований на користувача. *Синтаксична діаграма* (СД) є сукупністю певних орієнтованих графів (піддіаграм) на площині з вершинами, що мають вигляд:



і відповідають правим частинам усіх правил РКВ-граматики. Кожному входженню лексеми чи нетермінального символу в праву частину правила відповідає окрема вершина піддіаграми. Вершини з'єднуються стрілками в тій самій послідовності, в якій зустрічаються в правилі. Кожна піддіаграма має рівно одну вхідну й вихідну стрілки. При цьому, якщо права частина є альтернативною, то піддіаграма складається з піддіаграм альтернатив, об'єднаних спільними входом і виходом. Якщо це конструкція:

1) у квадратних дужках, то до її піддіаграми додається стрілка, що з'єднує вхід і вихід;

2) у фігурних дужках, то вихід її піддіаграми зациклюється на свій же вхід і додається стрілка, що з'єднує вхід і вихід. Якщо при цьому вона має модифікатор: а) *, то піддіаграма не змінюється; б) +, то остання стрілка не додається; в) (n, m), то піддіаграма будується просто як відповідна кількість альтернатив.

Конструкція-діапазон замінюється відповідною кількістю альтернатив.

Піддіаграма, що відповідає аксіомі граматики, називається головною.

На рис. 1.14 наведено СД для РКВ-граматики G_3 із сукупністю нетерміналів N ={<*ідентифікатор*>, <*літера*> ,<*цифра*>}, аксіомою <*ідентифікатор*> і правилами:

<*ідентифікатор*> → <*літера*> {<*літера*> | <*цифра*>}

<*цифра*> → 0 – 9

<*літера*> → *A – Z.*

Граматика G_3 задає клас лексем-ідентифікаторів – слів, що розпочинаються з латинської літери або літери _ (знак підкреслювання) і складаються з них і десяткових цифр.

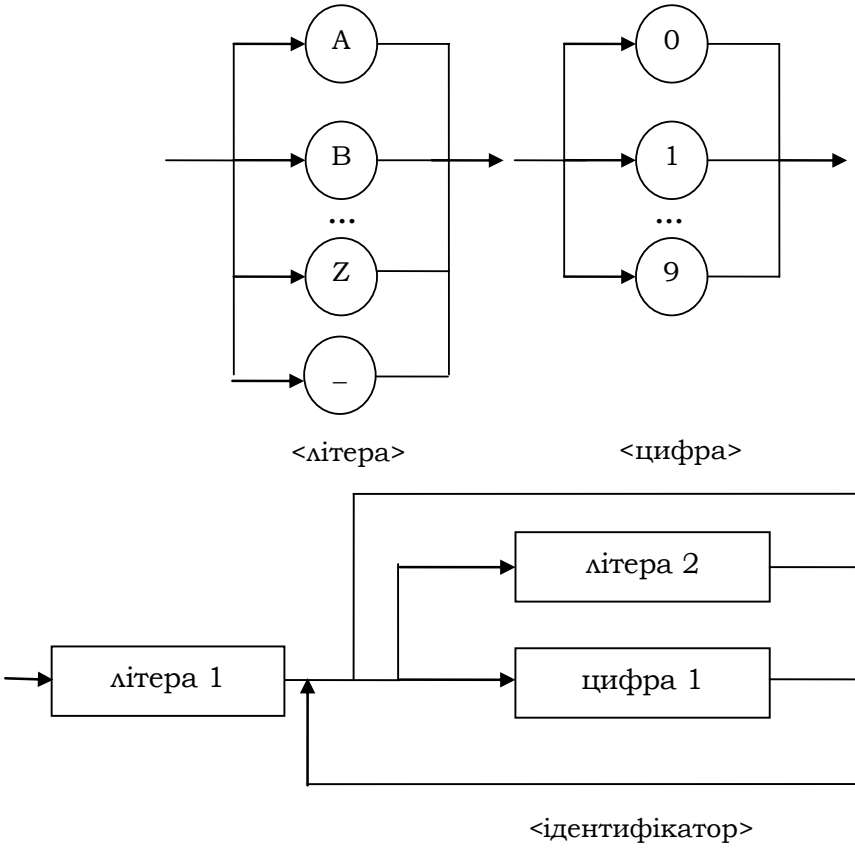


Рис. 1.14

Кожному виведенню слова з аксіоми в граматиці відповідає його виведення (шлях) у СД. Виведення в СД – це проходження в ній за стрілками піддіаграм, що розпочинається з головної діаграми з одночасною конкатенацією символів-лексем, розташованих у вершинах.

При цьому проходження вершини-нетермінала означає негайний тимчасовий перехід до її піддіаграми й повернення назад після виходу з неї. Виведення закінчується на виході з головної піддіаграми. Результатом його є конкатенація пройдених символів-лексем. Наприклад, ідентифікатору К_16 відповідає такий шлях: (літера,1) $\rightarrow$ К $\rightarrow$К (літера,2) $\rightarrow$К_$\rightarrow$ К_(цифра, 1) $\rightarrow$ К_1$\rightarrow$К_1(цифра, 1) $\rightarrow$К_15.

***Література для СР:** алгебричні системи – [62, 79, 97]; узагальнені (неокласичні) алгебричні системи й логіки для них [39, 62, 94, 105, 150]; регулярні вирази, скінченні автомати [7, 23, 52, 97]; КВ-граматики та МП-автомати – [7, 20, 23]; САА та їхні застосування – [23, 24, 32, 52].

Контрольні запитання та вправи

1. Що таке алгебрична система, алгебра й реляційна система? Навести власні приклади таких систем.
2. Що таке замкнена підмножина й підсистема алгебричної системи?
3. Що таке множина твірних алгебри? Навести приклади твірних множин алгебр.
4. Що таке незалежна система твірних алгебри? Навести приклади таких множин.
5. Чим відрізняються ізоморфізм, автоморфізм і гомоморфізм систем?
6. Що таке півгрупа й підпівгрупа? Навести власні приклади півгруп, їхніх підпівгруп і твірних множин.
7. Що означає: одна система моделює іншу?
8. Дати означення наближеної, сильної й точної моделей системи.
9. Які системи називаються еквівалентними?
10. Дати означення конгуенції системи.
11. Що таке ядерна еквівалентність системи?
12. Що означає: узгодженість операцій і предикатів двох систем?
13. Нехай $\mathfrak{R}$ – довільна сукупність непорожніх власних підсистем системи $\boldsymbol{A}$. Довести, що перетин $B = \bigcap\limits_{D\in\mathfrak{R}} D$ або порожній, або замкнений.
14. Нехай $\mathfrak{R}$ – сукупність усіх підсистем системи $\boldsymbol{A}$, що містять B. Довести, що перетин $\boldsymbol{B}^*$ усіх систем із $\mathfrak{R}$ є найменшою підсистемою $\boldsymbol{A}$, що містить усі елементи B.
15. Сформулювати лему про ядерну еквівалентність гомоморфізму.

16. Довести, що ядерна еквівалентність гомоморфізму $\phi : A \to B$ системи $\boldsymbol{A}$ в систему $\boldsymbol{B}$ є конгруенцією.
17. Довести, що класи лишків за модулем p задають конгруенцію на Z.
18. Навести інші нетривіальні конгруенції цілих чисел. Що можна сказати взагалі про їхню кількість?
19. Знайти всі конгруенції алгебри $A = \left(\{a,b,c,d\}, F\right)$, де унарна операція F задана таблицями: а) $\begin{pmatrix} abcd \\ badc \end{pmatrix}$; б) $\begin{pmatrix} abcd \\ bcdc \end{pmatrix}$.
20. Що таке типізована операція?
21. Дати визначення багатосортної Ω-системи.
22. Що таке вільна Ω-система?
23. Сформулюйте теорему про вільні Ω-системи.
24. Дати визначення Ω-алгебри слів, регулярної Ω-системи мов і системи алгоритмічних алгебр.
25. Довести, що система дійсних квадратних матриць розміром $n \times n$ є півгрупою відносно додавання та множення (у першому випадку – комутативною, у другому – ні). Що можна сказати про систему твірних у цих півгрупах?
26. Що таке список і лінійний список?
27. Сформулювати основні операції для лінійних списків.
28. Що таке стек і черга?
29. Сформулювати основні операції для стеків і черг.
30. Дати визначення графа, зваженого графа, мультиграфа й мережі.
31. Що таке дерево?
32. Що таке піддерево й листок дерева?
33. Довести лему 1.5 про дерево.
34. Що таке висота дерева?
35. Що таке дерево пошуку та збалансоване дерево?
36. Навести визначення скінченного, детермінованого й недетермінованого автоматів.
37. Дати визначення ізоморфізму й гомоморфізму скінченних автоматів.
38. Зобразити графічно автомат $Q = \left\{q_0, q_1, q^*\right\}$, $F = \left\{q_1, q^*\right\}$, $T = \{a,b,c,\ \}$, $q_0\varepsilon \to q_1$; $q_0 a \to q_1$; $q_1 b \to q_1$; $q_1 c \to q^*$. Чи є він детермінованим і яку регулярну мову розпізнає?
39*. Довести, що кожна скінченно-автоматна мова є регулярною [23, 52].
40*. Знайти найменші розв'язки: а) лінійних рівнянь з одним невідомим x вигляду $x = ax \cup b$ та $x = xa \cup b$; б) систем

таких рівнянь. Показати, що кожна скінченно-автоматна мова є найменшим розв'язком певної системи лінійних рівнянь із б) [23, 52].

41. Побудувати регулярний вираз і детермінований автомат для множини десяткових скінченних дробів вигляду $xx...x.yy...y$, де x, y – довільні десяткові цифри.
42. Побудувати регулярний вираз і за ним детермінований автомат, які описують: 1) сукупність ідентифікаторів – слів, що складаються з латинських літер і цифр і розпочинаються з літери; 2) сукупність адрес електронної пошти; 3) сукупність Інтернет-адрес.
43. Те саме, що й у вправах 37-38, але застосувати: а) розширену БНФ; б) синтаксичну діаграму.
44. Детермінізувати автомат Q із вправи 15.
45. * Довести, що мова $L_2 = \left\{a^n b^n : n \geq 0\right\}$ не є регулярною.
46. Наведіть визначення КВ-граматики.
47. Побудувати за даною КВ-граматикою відповідну до неї обчислювальну процедуру.
48. Що таке еквівалентні КВ-граматики? Навести кілька прикладів таких граматик.
49. Символ X об'єднаного алфавіту $N \cup T$ називається *недосяжним* у граматиці $G = (N, \Sigma, P, S)$, якщо в ній немає жодного виведення вигляду $S \Rightarrow uXv, u, v \in (N \cup T)^*$. Якщо в граматиці відкинути всі правила, що містять недосяжні символи, то породжувана нею мова від цього не зміниться. Як знайти всі недосяжні символи граматики?
50. Довести, що для будь-якого виведення слова в граматиці G існує його лівостороннє виведення.
51. Показати, що для будь-якої РКВ-граматики існує еквівалентна КВ-граматика.
52. Наведіть визначення $\mathrm{LL}(k)$-граматик. Які переваги мають такі граматики?
53. Мовою Діка в алфавіті $\Sigma_{2n} = \{a_1, b_1, ..., a_n, b_n\}, n > 0$ називається мова D_{2n}, породжена КВ-граматикою $G_4 = (\Sigma_{2n}, N, P, S)$, де $N = \{S\}$, $P = \{S \to SS, S \to \varepsilon, S \to a_i S b_i, i = \overline{1,n}\}$. Чи є граматика G_4 $\mathrm{LL}(k)$-граматикою для деякого $k > 0$?
54. Побудувати $\mathrm{LL}(1)$-граматику для сукупності ідентифікаторів.
55. Наведіть приклад не $\mathrm{LL}(1)$-, а $\mathrm{LL}(2)$-граматики.

56*. Нетермінал X КВ-граматики G називається *ліворекурсивним*, якщо в граматиці існує виведення $X \Rightarrow Xw$. Довести, що граматика, яка має принаймні один ліворекурсивний нетермінал, не може бути $\mathrm{LL}(k)$-граматикою для жодного $k > 0$.

57. Нехай КВ-граматика G_5 з аксіомою S має правила $P = \{S \to A \mid B, A \to aAb \mid 0, B \to aBbb \mid 1\}$. Яку мову вона породжує?

58*. Показати, що граматика G_5 не є $\mathrm{LL}(k)$-граматикою для жодного $k > 0$.

59*. Показати, що для мови $L(G_5)$ узагалі не існує $\mathrm{LL}(k)$-граматики.

60. Довести, що МП-автомат A з доведення теореми 1.4 за побудовою є МП-аналізатором типу розгортки для граматики *G*.

61. Довести твердження (*) з доведення теореми 1.4.

62. Довести, що для будь-якого МП-автомата A з множиною заключних станів $F = Q_{fin} \times \Sigma^* \times H^*$ існує еквівалентний йому МП-автомат зі спустошенням.

63. Побудувати КВ-граматику та МП-автомат для мови $L = \left\{ uu^R : u \in \{a,b\}^* \right\}$, де u^R – дзеркальне обернення слова u.

64. Побудувати МП-аналізатори типу розгортки для граматик із правилами: а) $S \to AB \mid b, A \to SA \mid a$; б) $S \to SS \mid A, A \to (A) \mid ( )$.

65. Побудувати МП-аналізатор для мови Діка (див. вправу 53).

66. Що таке БНФ та РБНФ?

67. Що таке синтаксична діаграма?

68. Побудувати синтаксичну діаграму та РБНФ для мови Діка в алфавіті (,),[,],{,}.

69. Побудувати синтаксичну діаграму та РБНФ для понять мови С (див. підрозд. 3.2.2):
   а) <*ціле_число*>;
   б) <*дійсне_ число*>;
   в) <*літерал* >;
   г) <*коментар*>;
   д) <*шістнадцяткова_константа*>;
   е) <*вісімкова_константа*>.

Розділ II

ЕЛЕМЕНТИ ІНФОРМАТИКИ

У цьому розділі розглянуто основні поняття інформатики – інформаційна система, життєвий цикл інформаційних систем, обчислювальна система, мови програмування, а також теоретичне підґрунтя для програмування циклів і рекурсивних програм. Окремий підрозділ присвячено елементам технології програмування.

2.1. Інформаційні системи

- Вхідні системи
- Вихідні системи
- Інформаційні системи в першому наближенні
- Абстрактні алгоритми
- Структурні алгоритми
- Конструктивні інформаційні системи
- Складність інформаційних систем

Ключові слова: *вхідна система, інформаційний об'єкт, інформаційне поле, X-фрейм, оператор і операція присвоювання, групове присвоювання, X-арна функція, нормальне розширення X-арної функції, еквітонна функція, еквітонна операція, еквітонна V-операція, $X-Y$-оператор, векторний аналог і параметризований векторний аналог $X-Y$-оператора, вихідна система, функції кодування й декодування, інформаційна система, узгодженість інформаційної системи, алгоритмічна мова, абстрактний алгоритм, графік алгоритму, алгоритмічно обчислювальна відповідність, масовість, темпоральність, елементарність, визначеність, розв'язна множина, частково розв'язна множина, результативність, фінітність і релятивність алгоритмів, алгоритм з оракулами, табличний алгоритм, структурна блок-схема, структурний алгоритм, конструктивна інформаційна система, алгоритмічно нерозв'язна проблема, часова та просторова складність алгоритму, порядки складності* $O(g(n))$ *та* $\Omega\left(g\left(n\right)\right)$.

Мета даного підрозділу – деталізувати поняття інформаційної системи, введене у 0.1. Нагадаємо, що кожна інформаційна система складається із суб'єкта-ініціатора, суб'єкта-обробника та функцій кодування й декодування, які їх зв'язують. У моделях комунікативних систем дані суб'єкти подаються предметними областями із сукупностями запитів (вхідними системами) і системами обробки інформації (вихідними системами).

2.1.1. ВХІДНІ СИСТЕМИ

Якщо абстрагуватись від несуттєвих деталей і зосередитися на семантиці, то суб'єкт-ініціатор можна обмежити предметною областю й сукупністю запитів.

Предметні області є наближеними моделями систем-об'єктів зовнішнього світу та взаємозв'язків між ними. Оскільки реально предметні області багатосортні, то природно звернутись для їхнього подання до багатосортних Ω-систем.

Нехай $\Omega = (\Omega_s,\ \Omega_c,\ \Omega_v,\ \Omega_f,\ \Omega_p)$ – певна сигнатура типу $\tau = (\sigma_1, \sigma_2,\ldots; \nu_1, \nu_2,\ldots)$; $\boldsymbol{A} = (\Omega_s^I, \Omega_c^I, \Omega_v^I, \Omega_f^I, \Omega_p^I)$ – Ω-система на універсумі U, породжена інтерпретацією I, а $\Theta \subset (\Omega_v^I)^U$ – довільна сукупність типізованих вхідних оцінок змінних сигнатури. *Типізованість* означає узгодженість у оцінці типу змінної із сортом її значення. Означені змінні й константи системи $\boldsymbol{A}$ будемо розглядати як *вхідні параметри* запитів. В остаточному вигляді запити вхідних систем, як ми вже знаємо, оформлюються в термінах мов програмування.

Нехай L – мова програмування для формулювання запитів у предметній області $\boldsymbol{A}$.

Трійка $S=(\boldsymbol{A},L,\Theta)$ називається вхідною системою інформаційних систем.

Така універсальна модель вхідної системи виявляється достатньо змістовною й має кілька суттєвих переваг: для специфікації предметної області може використовуватися весь потужний арсенал логіко-алгебричних засобів, а для попереднього опису запитів – Ω-формули ПЧП. У деяких випадках такий запит може бути просто варіантом Ω-формули $\Phi = \Phi(x_1,\ldots,x_n,y_1,\ldots,y_m)$, записаним мовою L, де $X = \{x_1,\ldots,x_n\}$ – сукупність вхідних предметних змінних, а $Y = \{y_1,\ldots,y_m\}$ – вихідних, і значення останніх прямо обчислюються за

формулою Φ. У загальному ж випадку необхідно написати одну або цілий комплекс програм мови L, що забезпечують пошук за значеннями вхідних змінних значень вихідних, які задовольняють формулу Φ і, можливо, відповідають певним іншим вимогам.

2.1.2. ВИХІДНІ СИСТЕМИ

Тепер звернемося до суб'єктів-обробників інформації (далі просто обробників), які отримують від суб'єкта-ініціатора повідомлення з інформацією про значення вхідних параметрів і програму із запитом, обробляють її й повертають результати обробки.

Уточнимо спочатку на загальному рівні семантику таких важливих понять, як повідомлення, інформація та її обробка (не зачіпаючи поки що самі процедури обробки).

Нехай $V = \{v_1, v_2, ...\}$ – певна сукупність *імен*. На природу імен не будемо накладати жодних обмежень, вважатимемо тільки, що серед них присутні всі натуральні числа. Таке припущення дозволить, як і при поданні вхідних областей, користуватися там, де це природно, засобами ПЧП при структуруванні інформаційних об'єктів і описі співвідношень між ними. Нехай $U = \{a_1, a_2, ...\}$ – довільний універсум *значень*, серед яких присутні всі імена, тобто $V \subset U$. В інформаційних системах імена відіграють роль повідомлень, а значення – роль інформації, що міститься в повідомленнях.

Роль імен у суб'єктах-ініціаторах і суб'єктах-обробниках суттєво різниться. Якщо в предметній області вони є лише сигнатурними елементами й тільки позначають об'єкти, то в системах обробки інформації імена не тільки подають значення, а й самі стають активними учасниками обчислювальних процесів. Їх можна обчислювати, пересилати окремо від значень тощо.

Правила означень імен у вигляді часткових відображень $\alpha : V \to U$ задають зв'язок між іменами та їхніми значеннями. Вважається, що таких імен у кожному правилі означень скінченна кількість. Нехай V^U – сукупність усіх *означень імен*. Якщо поглянути на означення імен із теоретико-множинних позицій, то фактично йдеться про вже знайому нам індексацію значень іменами.

Якщо за ім'ям закріплено тільки одне можливе значення в певному контексті (певній сукупності означень), то його називають *константою,* якщо ж кілька – то *змінною.* У кожний окремий момент комунікативного процесу змінна або має якесь конкретне значення, або не-

означена. Нехай $a = v\alpha$ – значення імені v у означенні α. Тоді про пару (v,a) будемо говорити як про *інформаційний об'єкт.* Якщо змінна v не означена в означенні α, то цей факт записується $v\alpha = \#$, або $(v,\#)$. Кажуть, що змінна в цьому означенні *беззмістовна.*

Розглянемо індексовані слова $1000_{(2)}$ та $1000_{(roma)}$ в алфавіті двійкових цифр і римській системі числення (індекс указує на систему числення). У першому випадку слово іменує десяткове число 8, тобто число 8 є значенням імені $1000_{(2)}$. У другому випадку слово не означене, оскільки воно взагалі не входить до сукупності імен римської системи.

Кожне означення породжує фіксовану скінченну множину інформаційних об'єктів з іменами з V і значеннями з U. Ця сукупність збігається з графіком означення α. І навпаки, кожне скінченне функціональне відношення між множинами V та U задає певне означення імен, тобто визначає певну множину інформаційних об'єктів над V та U. Подібні множини інформаційних об'єктів будемо називати *інформаційними полями* (*іменними множинами* за В.Н. Редьком).

Коли далі йтиметься про обробку інформації, то це означатиме перетворення обробником інформаційних полів.

Якщо дане ім'я не є формально означеним у даному полі, то вважається, що воно теж входить до нього, але із беззмістовним значенням. У програмуванні інформаційні поля називають *даними,* а функції, що утворюють і перетворюють дані – *операторами.* При цьому конкретні дані можуть складатися як з одного інформаційного об'єкта, так і зі скінченного їх набору. Результат застосування оператора f до поля α будемо позначати як звичайними термами, так і виразом вигляду $f|_\alpha$. Казатимемо, що одне інформаційне поле β *включає* (містить) більше інформації, ніж інше інформаційне поле α, якщо $\alpha \subseteq \beta$.

Зазвичай при роботі з інформаційними полями увага фокусується не на всій їхній інформації, а тільки на актуальній на даний момент її частині, що міститься в певному фрагменті поля. Це саме стосується й перетворення інформаційних полів – змінюється не вся їхня інформація, а тільки та, що належить певному фрагменту. Такі актуальні фрагменти полів отримали назву *X*-фреймів. *X-фреймом* називається довільна скінченна сукупність імен $X \subset V$ змінних, які складають області визначення або значень певного оператора.

Найчастіше змінюється значення одного інформаційного об'єкта (змінної) з певним іменем v. Тут можливий один із двох варіантів: 1) змінна v із певним значенням a вже є в полі; 2) дана змінна ще не

означена в ньому. У першому випадку на підставі аналізу X-фрейму, до якого входять ті інформаційні об'єкти, з якими пов'язана змінна v, і певних обчислень значення змінної v у полі замінюється на нове значення b. У другому випадку, знову ж таки на основі подібного аналізу, а частіше без такого (ініціалізація змінної, введення вхідної інформації), змінна v отримує початкове значення b_0 і приєднується до поточного поля.

У першому випадку заміна значень змінної в полі називається *присвоюванням. Оператор присвоювання* позначається ":=" і має три аргументи (*операнди*): вираз le, який задає ім'я v, функцію re для обчислення нового значення змінної та інформаційне поле α з області визначення функції re. Нехай $v = le(\alpha)$, $a = re(\alpha)$. Позначимо α_{-v} результат вилучення з поля α змінної v. Формально результат дії оператора присвоювання записується так:

$$le := re\,|_{\alpha} \overset{def}{=} \alpha_{-v} \cup \{(v,a)\}.$$

Вираз le та функція re називаються відповідно *l-* та *r-виразами*[16].

Загальні оператори присвоювання можна дещо структурувати, якщо ввести поняття операції присвоювання. *Операція присвоювання*, на відміну від оператора присвоювання, виконує паралельно дві дії: основну й побічну. Основна – за іменем v, значенням a й поточним станом інформаційного поля α повертає як результат a: $v \leftarrow a(\alpha) \overset{def}{=} a$. Побічна ж за тими самими аргументами оновлює значення змінної v у полі $\alpha : v \leftarrow a\,|_{\alpha} \overset{def}{=} \alpha_{-v} \cup \{(v,a)\}$.

Можна синтаксично підкреслити, про які саме дії – обидві чи тільки побічну – йдеться в операції присвоювання (так як це робиться, наприклад, у мові С). Якщо тільки про побічну, то терм операції закінчують символом ';'. У цьому випадку оператор присвоювання $le := re$ можна подати як результат підстановки у 3-арну операцію присвоювання виразів le та re на місце першого й другого аргументів із символом ';' у кінці: $le := re = le \leftarrow re$;.

Приклад 2.1. Нехай $X = \{x, y\} \subset V$ та $\alpha = \{(x,2),(y,5)\}$. Розглянемо вираз-присвоювання (*) $x \leftarrow x + 2 * y$ та оператор-присвоювання (**) $x \leftarrow x + 2 * y$;. *l*-виразом у них є ім'я змінної x, *r*-виразом – функція, яка з довільного стану X-фрейму підставляє значення змінних x та

[16] Від положення їх в операторі присвоювання: англ. – left (лівий) і right (правий).

y у терм $x+2*y$ та обчислює значення. В обох випадках ім'я x і отримане значення підставляються в операцію присвоювання. Якщо застосувати конструкції (*) та (**) до поля α, то вираз присвоювання (*) набуде значення 12, а поле α трансформується в поле $\beta=\{(x,12),(y,5)\}$. Результатом же оператора присвоювання (**) буде тільки нове поле $\beta=\{(x,12),(y,5)\}$ ■

Операція присвоювання дозволяє ввести поняття групового присвоювання. Ураховуючи, що вирази-присвоювання є одночасно й *r*-виразами, їх можна підставляти в ліву частину операцій присвоювання. *Груповими присвоюваннями* називаються складені присвоювання вигляду $x\leftarrow(y\leftarrow e)$, $x\leftarrow(y\leftarrow(z\leftarrow e))$ тощо. Дужки в них можна опускати. У результаті групового присвоювання $z\leftarrow x\leftarrow x+2*y$ поле α трансформується в поле $\beta=\{(x,12),(y,5),(z,12)\}$, а в результаті присвоювань $z\leftarrow x\leftarrow z\leftarrow 0$ та $z\leftarrow x\leftarrow 2*(z\leftarrow 1)$ – відповідно в поля $\beta=\{(x,0),(y,0),(z,0)\}$ та $\gamma=\{(x,2),(y,2),(z,1)\}$ (див. попередній приклад).

Можна піти далі та структурувати самі *r*-вирази, виділивши в них *операції читання* значень змінних у інформаційних полях:

$$r(v)|_{\alpha} \overset{def}{=} \alpha v\downarrow .$$

За означенням, якщо змінна v не входить у поле даних α, то $r(v)|_{\alpha}=\#$.

Приклад 2.2. Якщо повернутись до прикл. 2.1, то *r*-вираз у правій частині операції присвоювання можна подати у вигляді терму $r(x)+2*r(y)$, а саму операцію присвоювання – як $x\leftarrow r(x)+2*r(y)$. Обчислимо її значення в полі α:

$$(x\leftarrow r(x)+2*r(y))|_{\alpha}=(2+2*5)|_{\alpha}=12$$ ■

У мовах програмування функції читання змінних подаються скорочено – просто як їхні імена, а відрізняють їх від імен змінних за контекстом. Наприклад, в операторі $z\leftarrow z+1$; перше входження є іменем змінної (*l*-виразом), а друге – операцією читання.

Оператори присвоювання належать до фундаментальних засобів перетворення інформації в інформаційних системах. Вони є прикладом важливого класу узагальнених операцій, які в композиційній семантиці мов програмування отримали назву X-арних та Y-арнозначних функцій. Нехай $X=\{x_1,\ldots,x_n\}\subset V$ – довільний

X-фрейм. Функція f називається X*-арною,* якщо $D_f \subset X^U$. X-арна функція природним чином поширюється на всі інформаційні поля (дані), що містять даний X-фрейм.

Функція $\vec{f}$ називається нормальним розширенням X-арної функції f, якщо $\forall \alpha \in D_f \forall \beta \in V^U \left(\alpha \subseteq \beta \Rightarrow \vec{f}(\beta) = f(\alpha)\right)$.

Нормально розширені X-арні функції отримали назву *еквітонних узагальнених операцій.* Значення еквітонної X-арної операції $\vec{f}$ на певному полі α залежить лише від стану X-фрейму поля α, тобто інформаційні об'єкти поза цим X-фреймом жодним чином не впливають на значення $\vec{f}(\alpha)$.

Саме за допомогою еквітонних X-арних функцій здійснюється обробка інформації у вихідних системах.

Нехай X, Y – певні сукупності імен. За областю значень серед еквітонних X-арних функцій виділяють: 1) Y*-арнозначні оператори*, якщо $E_f \subset Y^U$; 2) X*-арні операції* (*предикати*), якщо $E_f \subseteq U$; 3) V*-операції,* якщо $E_f \subseteq V$.

Еквітонні X*-арні* операції та предикати беруть участь у підготовці нових значень змінних при перетворенні інформаційних полів. Серед них виділяють уже згадувані операції читання, які просто беруть поточні значення змінних у даному інформаційному полі. Алгебричні системи з еквітонними X-арними операціями та предикатами (*узагальнені* алгебричні системи) мають усі загальні властивості класичних алгебричних систем (див. вправу 12).

Серед X-арних операцій є операції *з побічним ефектом.* Це операції, які в процесі обчислення результату можуть змінити поточне значення певної змінної X-фрейму. Типовими прикладами операцій із побічним ефектом є операції присвоювання.

V-операції виробляють імена інформаційних об'єктів. Існує таке поняття, як доступ до даного в інформаційних полях за його іменем. При цьому розрізняють доступи для здійснення операцій *читання* (взяття значення даного) і для *запису* (заміни значення даного). Операції читання й запису називаються *інтерфейсними* й використовують V-операції для реалізації доступу до даних.

Y-арнозначні оператори виробляють нові стани Y-фрейму.

Увага! Еквітонні операції (V-операції) трактуються як $\varnothing$-арнозначні оператори з областю значень $\varnothing^U = U$ ($\varnothing^V = V$) ▶

Позначимо через α_{-Y} результат вилучення з поля α змінних Y-фрейму, у тому числі й беззмістовних. Фундаментальна роль у обробці даних належить *операції оновлення даних* $\uparrow$, яка узагальнює операцію присвоювання $\leftarrow$ і поточний стан довільного Y-фрейму в інформаційному полі α замінює на новий його стан β:

$$\alpha \uparrow \beta \overset{def}{=} \alpha_{-Y} \cup \beta .$$

Після виконання операції оновлення змінні Y-фрейму набувають або нових значень, або початкового (якщо до цього були беззмістовними в α). Нехай f – Y-арнозначний оператор. Y-арнозначним *оператором оновлення* f; називається результат підстановки в операцію оновлення оператора f на місце другого аргументу. Для довільного поля α: $f;|_{\alpha} \overset{def}{=} \alpha \uparrow f(\alpha)$.

Зазначимо, що $\varnothing$-арнозначні оператори залишають інформаційне поле без змін ($\alpha_{-\varnothing} = \alpha$).

Еквітонні X-арні Y-арнозначні оператори оновлення називаються *$X-Y$-операторами.*

Розглянуті вище оператори присвоювання є $X-Y$-операторами. Оператор присвоювання $x \leftarrow x + 2 * y$; із прикл. 2.1 є $\{x,y\}-\{x\}$-оператором.

$X-Y$- операторами здійснюють усі перетворення інформаційних полів у вихідних системах.

Якщо зафіксувати певний порядок імен у сукупності $X \cup Y = \{x_1,...,x_n\}$, то для кожного $X-Y$-оператора f існує його *векторний аналог* – відображення $\bar{f}: U^n \to U^n$ таке, що $\forall\alpha = \{(x_1,a_1),...,(x_n,a_n)\} \in D_f$ та $\forall\beta = \{(x_1,b_1),...,(x_n,b_n)\} \in E_f$:

$$f|_{\alpha} = \beta \Leftrightarrow \bar{f}(a_1,...,a_n) = (b_1,...,b_n).$$

Кажуть, що $X-Y$-оператор *обчислює* свій векторний аналог. Зазвичай цікавим є не загальний векторний аналог, а *параметризована* його проекція $\bar{f}_{X,Y}$, що відповідає на вході іменам із X, а на виході – з Y. Запити вхідних систем у вигляді формул ПЧП містять специфікації певних векторних відображень, для яких у процесі програмування будуються $X-Y$-оператори, що їх обчислюють.

Загальна схема перетворення (оновлення) інформаційного поля певним $X-Y$-оператором виглядає так: 1) береться поточна інформація з X-фрейму поля; 2) на її підставі отримується нова інформація за допомогою звичайних n-арних операцій; 3) нова інформація заноситься у Y-фрейм. Дану схему можна дещо конкретизувати:

1) спочатку V-функції формують імена змінних із X-фрейму, а операції читання за цими іменами знаходять їхні поточні значення в інформаційному полі;

2) n-арні операції за цими значеннями виробляють нові значення змінних Y-фрейму;

3) V-функції виробляють імена Y-фрейму, після чого один або кілька операторів присвоювання (або введення вхідної інформації) виробляють новий стан Y-фрейму й усього інформаційного поля.

Увага! Окрім розглянутих $X-Y$-операторів із фіксованими X-фреймами у вихідних системах зустрічаються і $X-Y$-оператори зі змінною структурою X-фрейму. У такому $X-Y$-операторі X є параметром, фактичне значення якого (сукупність імен) визначається тільки в процесі виклику оператора (як у випадку узагальнених операцій над індексованими сукупностями множин (див. підрозд. 1.2)) ►

Таким чином, для обробки інформації в інформаційних системах мають бути наявні такі елементи:

1) сукупність V^U інформаційних полів з іменами з V і значеннями з універсума U;

2) сукупності інтерфейсних V-операцій $\{h_k\}$ і операцій читання $\{g_l\}$;

3) сукупність звичайних (і параметризованих X) $X-Y$-операторів $\{\mathrm{O}_l\}$ для перетворення інформаційних полів;

4) базова алгебрична система $\boldsymbol{U}=\left(U;\{f_i\},\{p_j\}\right)$ n-арних операцій і предикатів на універсумі для обчислення нових значень змінних.

Усі чотири наведені елементи об'єднаємо в узагальнену алгебричну систему $\boldsymbol{D}$ *маніпулювання даними* вихідної системи. Отримавши повідомлення про запит і його аргументи, обробник першу інформацію переводить у $X-Y$-оператор f із $\boldsymbol{D}$, що реалізує запит, а на підставі другої формує вхідні дані й застосовує до них f. Для подання $X-Y$-операторів і всієї системи маніпулювання даними $\boldsymbol{D}$ обробник використовує спеціальну *внутрішню* мову програмування $L_{вн}$, а Виконавець B повинен уміти реалізовувати $X-Y$-оператори, подані в мові $L_{вн}$.

Пара $\boldsymbol{S}_{out}=(B,L_{вн})$, що складається з Виконавця та внутрішньої мови програмування, називається вихідною системою.

Важливим прикладом вихідних систем є конкретні ОбС. Кожна з них має певну систему команд і даних із конкретними базовими операціями, іменами, типами значень, інтерфейсними функціями й сукупність $X-Y$-операторів, які описуються внутрішньою мовою машинних програм даної ОбС, а також Виконавця – апаратну частину, що реалізує ці програми.

Найбільш універсальною вихідною системою є людина з усіма природними та штучними внутрішніми мовами, якими вона володіє.

2.1.3. ІНФОРМАЦІЙНІ СИСТЕМИ В ПЕРШОМУ НАБЛИЖЕННІ

В інформаційних комунікативних системах важливими елементами є функції кодування й декодування об'єктів, які пов'язують між собою вхідну й вихідну системи, а також засоби подання цих систем. Нехай $\boldsymbol{S}_{in}$ та $\boldsymbol{S}_{out}$ – певні вхідна й вихідна системи, Θ – сукупність вхідних оцінок змінних. Довільні однозначні відображення $\mathrm{c}:(\Omega_V^I)^U \to V^U$ та $\mathrm{d}: V^U \to (\Omega_v^I)^U$ називаються відповідно *функціями кодування* й *декодування*. Функція кодування відображає вхідні параметри запитів у відповідні інформаційні поля, які підлягають обробці у вихідній системі, а функція декодування повертає у вхідну систему результати цієї обробки у вигляді певної оцінки вихідних параметрів запиту.

Четвірка $\boldsymbol{S}=(\boldsymbol{S}_{in},\boldsymbol{S}_{out},\mathrm{c},\mathrm{d})$ називається інформаційною системою.

Інформаційна система $\boldsymbol{S}$ називається *узгодженою* відносно запиту $\Phi=\Phi(x_1,...,x_n,y_1,...,y_m)$, де $X=\{x_1,...,x_n\}$ – сукупність імен його вхідних параметрів, а $Y=\{y_1,...,y_m\}$ – сукупність імен його результуючих параметрів, якщо у вихідній системі $\boldsymbol{S}_{out}$ знайдеться $X-Y$-оператор O такий, що

$$\forall\alpha\in\Theta\exists\beta\in(\Omega_v^I)^U$$

$$\left(\left(\beta=d\left(\mathrm{O}\mid_{c(\alpha)}\right)\overset{n}{\underset{i=1}{\&}}x_i\alpha=a_i\overset{m}{\underset{j=1}{\&}}y_j\beta=b_j\right)\supset\Phi(a_1,...,a_n,b_1,...,b_m)\right)\ (*).$$

Умова (*) означає: якщо закодувати будь-який варіант вхідних параметрів запиту Φ і застосувати до отриманого інформаційного поля оператор O, а потім узяти результат роботи O і розкодувати його, то після підстановки в запит Φ вхідних і отриманих вихідних значень параметрів x_i та y_j запит буде задоволено. Назвемо оператор O *реалізацією* запиту Φ. Інформаційна система називається *узгодженою,* якщо вона узгоджена для всіх запитів вхідної системи.

2.1.4. АБСТРАКТНІ АЛГОРИТМИ

При уточненні поняття алгоритму відійдемо від прийнятої в теорії алгоритмів практики визначати якийсь конкретний клас алгоритмів (машини Тьюрінга чи алгоритми Маркова) і, посилаючись на його універсальність, пов'язувати з ним загальне поняття алгоритму. Такий підхід, прийнятний у світоглядному сенсі, на жаль, майже нічого не дає безпосередньо програмуванню. Тому розглянемо інший підхід, який спирається на загальне поняття функції як обчислювальної процедури (див. підрозд. 1.3.4) і реалізацію таких процедур.

Коли говорять про реалізацію класу функцій-процедур чи окремої функції, то мають на увазі, що існує вихідна система $\boldsymbol{S}_{out} = (B, L_{âí})$ (реальна чи гіпотетична) з Виконавцем, здатним сприймати функції-процедури й проводити за ними конкретні обчислення: фіксувати окремі стани, знаходити значення функції керування на кожному кроці обчислення й виконувати відповідні елементарні перетворення.

Увага! Внутрішня мова програмування системи, що реалізує функції-процедури, не обов'язково вербальна, але обов'язково *фінітна* – вважається, що Виконавець системи здатен розпізнавати й оперувати тільки скінченними за поданням об'єктами ►

Об'єкти внутрішніх мов вихідних систем, що реалізують функції-процедури, будемо називати конструктивними, подані в них функції – абстрактними алгоритмами (А-алгоритмами), а самі внутрішні мови й вихідні системи – алгоритмічними мовами й системами[17].

Таким чином, конструктивність об'єкта має сенс тільки в контексті певної алгоритмічної системи (класу систем), а функція-процедура стає А-алгоритмом тільки тоді, коли для неї існує конкретний Виконавець. На вибраному нами пропозиційному рівні абстракції не розглядається внутрішня структура самих Виконавців (вони теж є чорними скриньками).

Приклад 2.3. Важливий клас А-алгоритмів задають звичайні арифметичні вирази й загальні інтерпретовані Ω-терми. Кожен із них визначає певне формальне правило для обчислення значень. Виконавцем таких алгоритмів може бути, наприклад, особа, яка вміє аналізувати структуру термів і виразів і виконувати відповідні операції та предикати. Можна піти ще далі й розглянути всі інтерпретовані регулярні вирази над сигнатурою Ω (див. підрозд. 1.3.3) ■

[17] Не випадково перші мови програмування теж називалися алгоритмічними – наприклад, Алгол-60 (англ. – ALGOrithmic Language ALGOL-60).

Інші приклади А-алгоритмів – машини Тьюрінга і гра в шахи – детально розглядатимуться нижче.

Відповідності A^* та A^{**}, що обчислюються А-алгоритмом A (його графіки), називаються *алгоритмічно обчислювальними*. Як і функції, А-алгоритми можуть бути детермінованими й недетермінованими.

Наведемо основні властивості А-алгоритмів, які безпосередньо випливають з їхнього означення.

Масовість. А-алгоритм може бути застосований до будь-якого вхідного стану.

Темпоральність. Обчислення за А-алгоритмом відбуваються в глобальному часовому просторі, елементи якого можуть впливати на вибір перетворень поточного стану.

Елементарність. У кожний момент часу в обчисленні виконується одна елементарна операція з фіксованої сукупності таких операцій.

Визначеність. Порядок застосування елементарних операцій в обчисленні не довільний, а визначається функцією переходу за поточною конфігурацією обчислення.

Результативність. Є механізм завершення обчислень.

Фінітність. Скінченність подання А-алгоритму.

Релятивність. Відносність поняття А-алгоритму. Конструктивність А-алгоритму як об'єкта алгоритмічної мови прямо залежить від конкретного Виконавця. А-алгоритм даного Виконавця не обов'язково буде А-алгоритмом для іншого. Іншим аспектом релятивності А-алгоритмів є взаємовідносини елементарних операцій і Виконавця. У деяких випадках елементарні операції (усі чи їхня частина) можуть вважатися абстрактними й не підлягати виконанню. Такі операції називаються *оракулами*, а самі А-алгоритми – А-*алгоритмами з оракулами*.

Як бачимо, А-алгоритмам притаманні всі основні властивості класичних числових і словарних алгоритмів. Однак з'явилися й нові специфічні риси – темпоральність і релятивність.

У зв'язку з релятивністю виникає задача порівняння різних класів А-алгоритмів за потужністю. Будемо казати, що А-алгоритм A_1 з множиною станів S_1 є *моделлю* А-алгоритму A_2 з множиною станів S_2 відносно функції кодування $\varphi : S_2 \to S_1$, якщо для кожного із вхідних станів s А-алгоритму A_2 виконується $A_2^*(s) = \varphi^{-1}(A_1^*(\varphi s))$.

Нехай K_1 та K_2 – довільні класи А-алгоритмів над обчислюваними просторами $\Pi_1 = \langle \Gamma_1, S_1 \rangle$ та $\Pi_2 = \langle \Gamma_2, S_2 \rangle$, відповідно. Зафіксуємо певну функцію кодування $\varphi : S_2 \to S_1$. Кажуть, що клас А-алгоритмів K_1

моделює клас А-алгоритмів K_2, якщо для кожного А-алгоритму з K_2 у класі K_1 існує його модель. Якщо класи А-алгоритмів моделюють один одного відносно певних фіксованих функцій кодування, то вони називаються *еквівалентними* відносно цих функцій.

Фіксуючи конкретні обчислювальні простори П і відповідних Виконавців, можна отримати весь спектр класичних алгоритмічних та ігрових систем (машини Тьюрінга, алгоритми Маркова й Колмогорова – Успенського, дискретні перетворювачі, трансдюсери, різні класи схем програм, формальні граматики й числення Поста, ігрові числення тощо). Ми це вже бачили на прикладі скінченних і магазинних автоматів (див. підрозд. 1.4.4). Усі перелічені класи класичних алгоритмів і числень є автоматними, еквівалентними й отримали назву *універсальних*.

Для ілюстрації можливостей А-алгоритмів розглянемо машини Тьюрінга й шахи (відоме ігрове числення) як А-алгоритми.

Виконавець А-алгоритму A, що реалізує машину Тьюрінга, оперує станами, які зображують машинні конфігурації. Такі стани мають два компоненти: перший – внутрішній стан машини, другий подає стан пам'яті машини й має вигляд послідовності комірок, в якій зберігається слово в алфавіті X із зазначеною позицією робочої комірки. Часовий простір Г – натуральний. Значення функції керування на конфігурації визначається внутрішнім станом і станом робочої комірки (робочим станом). Воно жодним чином не залежить від номера кроку (часової компоненти) обчислення. Сама дія полягає у зміні Виконавцем внутрішнього й робочого станів і позиції робочої комірки. Новою позицією може стати позиція однієї з двох сусідніх комірок у послідовності. Ураховуючи, що сукупності внутрішніх станів Q і станів робочої комірки X машин Тьюрінга скінченні, функцію переходу нашого А-алгоритму A можна подати теж скінченно, наприклад таблично як відповідність $\delta : Q \times X \to Q \times X \times \{-1,0,1\}$, яка за поточними внутрішнім і робочим станами визначає нові їхні значення й нову позицію робочої комірки. Нуль означає, що вона не змінюється, –1 – зсув ліворуч, а 1 – зсув праворуч. Виділяється початковий $q_0 \in Q$ і сукупність заключних $F \subseteq Q$ внутрішніх станів. Сукупності S_0 початкових і S_{fin} заключних станів А-алгоритму складають множини $\{q_0\} \times X^*$ та $F \times X^*$, відповідно. Результатом обчислення А-алгоритму P на початковому стані є його робочий стан у заключному стані обчислення. Ураховуючи, що є лише один початковий внутрішній стан

А-алгоритму A, можна вважати, що він обчислює словарну відповідність A^* на множині робочих станів X^*.

Нехай $X = \{|, \#\}$ та $F = \{q^*\}$. Наведена таблиця задає функцію керування А-алгоритму для машини Тьюрінга, яка реалізує додавання натуральних чисел: початковий стан робочої стрічки $|^n \# |^m$, де n, m – довільні натуральні числа, перетворює на $|^{n+m}$:

Q	X	$Q \times X \times \{-1, 0, 1\}$
q_0	|	$(q_1, \varepsilon, 1)$
q_0	#	$(q^*, \varepsilon, 0)$
q_1	|	$(q_1, \|, 1)$
q_1	#	$(q^*, a, 0)$

Розглянемо таке ігрове числення, як шахи. Особливу увагу звернемо на конструктивну (дескрипційну) складову цієї гри. Гра відбувається на шаховій дошці – матриці розміром 8×8, клітини якої – поля, мають білий або чорний колір (рис. 2.1).

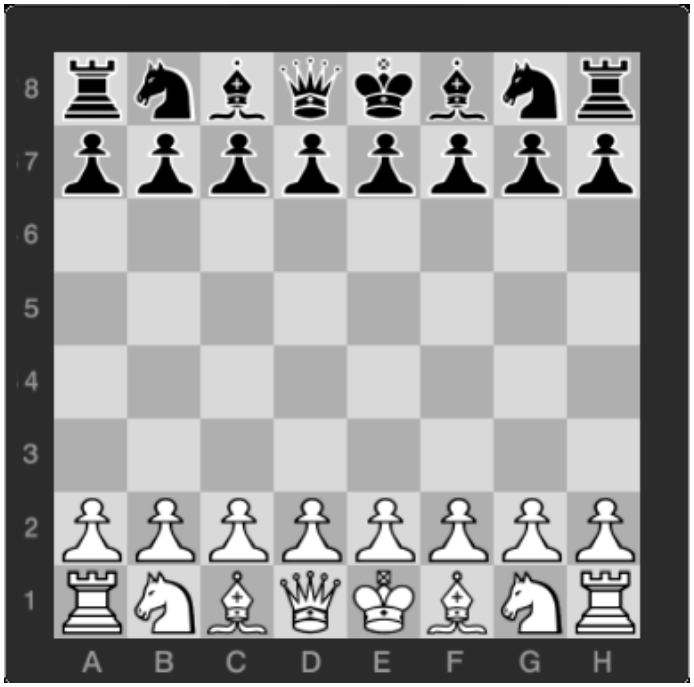


Рис. 2.1

Кожна з двох сторін має 16 фігур одного кольору (білого або чорного): 8 пішаків, 2 коней, 2 слонів, 2 тури, 1 ферзя й 1 короля. Розташування їх на дошці називається *позицією*. Можливі такі ходи фігур:

1) пішак може рухатися вперед по вертикалі на одну або дві позиції (останнє можливе тільки тоді, коли він перебуває в початковій позиції). При досягненні восьмої горизонталі (для білих) і першої (для чорних) пішак може бути замінений на будь-яку іншу фігуру (але не на короля);

2) тура може рухатися в довільному напрямку по вертикалі або горизонталі;

3) слон – те саме, але тільки по діагоналях матриці;

4) кінь ходить буквою Г: спочатку на дві позиції вверх (униз, ліворуч або праворуч), а потім на одну позицію ліворуч або праворуч, уверх або вниз. Усього можливо до восьми варіантів ходів (на краю дошки кінь має менше ходів);

4) ферзь має можливості тури й слона;

5) король – те саме, що й ферзь, але рух обмежений тільки вісьмома сусідніми клітинами. Окрім цього, у початковій позиції король може зробити рокіровку з однією зі своїх тур за умов: король і тура до цього не рухалися; король не перебуває під шахом (тобто під ударом однієї з фігур супротивника); між турою й королем немає фігур. Рокіровка полягає в пересуванні тури впритул до короля й перестрибуванні королем через неї на сусіднє поле. При цьому нові позиції короля й тури не мають бути під ударом супротивника. Рокіровок може бути рівно дві (коротка й довга). Коротка – з турою у стовпчику h, довга – у стовпчику a.

Якщо на шляху фігури (не пішака) стоїть фігура іншого кольору, то вона може бути збита – усунена з дошки. При цьому фігура, що збиває, ставиться на її місце. Пішак збиває не прямо по ходу, а по діагоналі (див. другий хід чорних у "безсмертній" партії нижче). Якщо на шляху фігури стоїть своя фігура, то через неї перестрибувати не можна.

На рис. 2.1 зображено початкову ігрову позицію. Гру починають білі. Кожна зі сторін по черзі робить один із припустимих ходів однією зі своїх фігур. Мета гри – створити заключну матову позицію (в якій король не може уникнути збиття) для короля іншої сторони. Заключними є також патові позиції, в яких одна зі сторін позбавлена будь-якого припустимого ходу, а також ті позиції, що тричі повторювалися у грі.

Обчисленнями (*партіями*) у шаховій процедурі є послідовності конфігурацій – позицій фігур на дошці у процесі гри з доданими шістьма прапорцями й одним лічильником. Кожний із прапорців сигналізує про наявність чи відсутність першого ходу в партії однієї з шести фігур (по три з кожної сторони), що потенційно можуть брати участь у рокіровці. Прапорець хибний, якщо такого ходу відповідна

фігура ще не зробила. Лічильник показує найбільшу кількість повторень однієї з позицій у партії на даний момент часу. Часовий простір – натуральний. Функція керування шаховою процедурою за поточною позицією і станом прапорців і лічильника повертає сукупність усіх припустимих продовжень гри (припустимих у даний момент ходів) і нові стани прапорців і лічильника. Ураховуючи, що кількість усіх можливих позицій скінченна (не більше $64^{16} \times 2^6 \times 3 = 3 \times 2^{102}$), обчислення рано чи пізно або перейде в заключний стан, або почне повторюватись. Якщо одна з позицій повториться тричі, то гра закінчиться. Щоб зобразити конкретну партію, достатньо задати послідовність ходів за білих і чорних. Для цього використовується спеціальна алгоритмічна мова – шахова нотація. Фігури в ній – кінь, слон, тура, ферзь і король – позначаються відповідно символами К, С, Т, Ф і Кр, пішак не має спеціального позначення. Поля дошки позначаються індексами. По горизонталі вони проіндексовані малими латинськими літерами a, b, c, d, e, f, g, h, а по вертикалі – числами 1, 2, 3, 4, 5, 6, 7, 8. Наприклад, пара індексів (a, 1) задає перше поле дошки в рядку а, і це записується a1. Один конкретний хід має вигляд:

[<фігура1>][<поле1->][:]< поле2>[<фігура2>] [+|!|!!|?|??|×].

Відсутність першої та присутність другої фігури свідчать про хід пішака. Друга фігура в цьому випадку – це фігура, на яку замінюється пішак при досягненні по горизонталі краю дошки. Перше поле (необов'язкове) показує, з якого поля йде фігура, а друге – куди. Символ ":" позначає бій фігури, а "+" – напад на короля (шах королю), "×" – матову позицію, "!" та "!!" – сильний і дуже сильний ходи, "?" та "??" – слабкий і дуже слабкий ходи, відповідно.

Наприклад, початкова позиція (рис. 2.1) зображується так:
Білі: Кр e1, Фd1, Та1, Th1, Кb1, Кg1, Сc1, Cf1, a2, b2, c2, d2, e2, f2, g2, h2; чорні: Кр e8, Фd8, Та8, Th8, Кb8, Кg8, Сc8, Cf8, a7, b7, c7, d7, e7, f7, g7, h7.

Така партія називається "дитячий мат": 1. f2–f4 e7–e6 2. g2–g4?? Фh5×.

Тепер розглянемо шахову партію, відому під назвою "безсмертна", що була зіграна в Лондоні в 1851 р. А. Андерсеном і Л. Кизерицьким:

1. e4 e5 /*зліва хід білих, справа – чорних у відповідь, обидва ходи пішаками */2. f4 ef/* хід чорних справа означає e5 : f4 – пішак на e5 б'є сусіднього білого пішака */3. Сc4 Фh4 + 4. Крf1 b5 5. С : b5 Кf6 6. Кf3 Фh6 7. d3 Кh5 8.Кh4 Фg5 9. Кf5 c6 10. g4 Кf6 11. Тg1! Сb 12. h4 Фg6 13. h5 Фg5 14. Фf3 Кg8 15. С : f4 Фf6 16. Кc3 Сc5 17. Кd5 Ф : b2

18. Cd6 C : g1 19. e5 Ф : a1 + 20. Kpe2 Ka6 21. K : g7 + Kpd8 22. Фf6 + C : f6 23. Ce7×.

Гра в шахи не є універсальною алгоритмічною системою – кількість її конфігурацій, хоч і астрономічно велика, усе ж скінченна. Однак із погляду конкретних Виконавців вона практично нескінченна, тому завдяки безлічі варіантів ходів та їхніх комбінацій шахи стали чудовою моделлю для багатьох реальних задач, пов'язаних із розробкою ефективних стратегій пошуку й обробки інформації в інформаційних системах. Це стосується насамперед перебірних задач у галузях штучного інтелекту, прийняття рішень тощо [91].

Як і для загальних функцій, має місце

Теорема 2.1 (замкненості). Клас графіків А-алгоритмів замкнений відносно всіх регулярних композицій.

Доведення – див. вправу 34 із підрозд. 1.3 ■

Як і класичні алгоритми, А-алгоритми можуть використовуватись для опису множин. Підмножина станів $R \subseteq S$ називається *розв'язною* (*частково розв'язною*), якщо існує А-алгоритм A, графік якого збігається з характеристичною функцією χ_R (частковою характеристичною функцією $\overline{\chi}_R$) підмножини R.

Важливий клас А-алгоритмів складають табличні алгоритми. Нехай часовий простір Γ ізоморфний адитивній групі Z цілих чисел і A – довільна процедура над простором $\Pi = \langle Z, S \rangle$. Факторизуємо функцію переходу за часовим аргументом і станами. Зафіксуємо деяку часову константу $k \geq 0$ і певне відношення еквівалентності π на станах із S. Будемо казати, що функція переходу δ періодична за модулем відношення π із періодом k (k-періодична за модулем π), якщо для будь-яких еквівалентних станів p та s і довільного моменту часу $t \in Z$ виконується $\delta(t, p) = \delta(t \pm k, s)$. Функція переходу δ просто періодична за модулем π, якщо вона k-періодична за модулем π для деякого k. Індексом багатозначної функції назвемо максимальну потужність сукупностей значень функції на окремих аргументах. Функція-процедура з періодичною функцією переходу, яка має скінченний індекс, а еквівалентність π – скінченний індекс $|\pi|$ і розв'язні класи, називається *табличним алгоритмом.* Табличні алгоритми (T-алгоритми) можна задавати двовимірними таблицями розміром $k \times |\pi|$: перший вимір відповідає числам від 0 до $k-1$, другий – класам еквівалентності π, а в клітинах таблиці розміщуються значення функції переходу. Традиційні алгоритмічні системи зазвичай є табли-

чними як А-алгоритми, а оскільки вони автоматні, то й 1-періодичні за модулем рівності (скінченні автомати, МП-автомати). Машина Тьюрінга є табличним алгоритмом: два її стани еквівалентні, якщо їхні внутрішні стани та стани робочих комірок збігаються.

Для табличних алгоритмів теж має місце теорема замкненості.

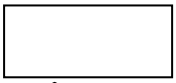
Теорема 2.2 (замкненості). Клас графіків табличних алгоритмів замкнений відносно всіх регулярних композицій.

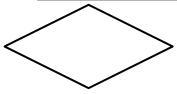
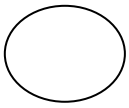
Доведення – див. вправу 34 із підрозд. 1.3 ■

А-алгоритми обчислюють і часові оператори, що відображають їхні вхідні конфігурації в заключні, тому вони можуть служити підґрунтям для уточнення й загального поняття обчислювального часового оператора в довільній області. Важливо, що цей шлях прямий і не спирається на ті чи інші універсальні числові й словарні моделі таких алгоритмів.

2.1.5. СТРУКТУРНІ АЛГОРИТМИ

Як важливий приклад А-алгоритмів розглянемо клас так званих структурних алгоритмів. Їхній конструктивний опис задають структурні блок-схеми (СБС), що є спеціальними графами на площині з трьома типами вершин – прямокутними, ромбічними та еліпсоїдними. Ці графи мають по одній вхідній і вихідній стрілці й побудовані за допомогою композицій із таких базових елементів:

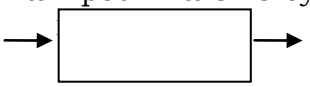
а) оператор 

б) умови:  

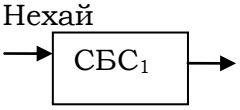
в) стрілки:

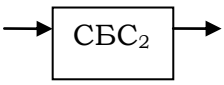
г) символи: +,- .

Найпростіша СБС будується з оператора і двох стрілок:



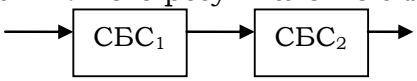
Нехай



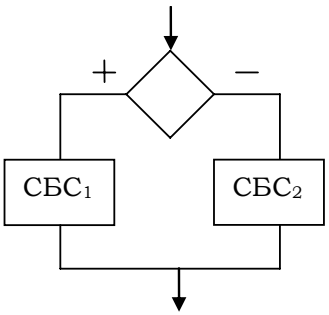


– довільні блок-схеми. Розглянемо композиції, за допомогою яких із наведених блок-схем і базових елементів будуються нові СБС.

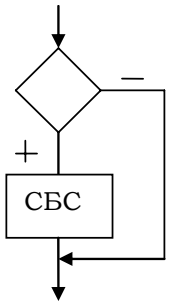
Послідовне з'єднання. Його результатом є *складена* СБС:



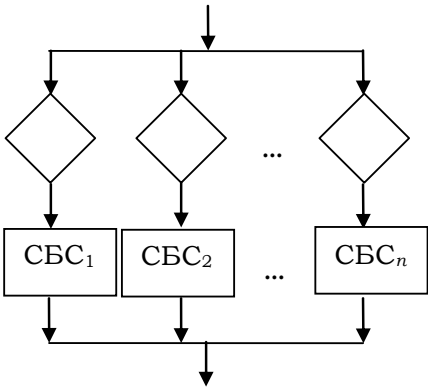
Розгалуження:



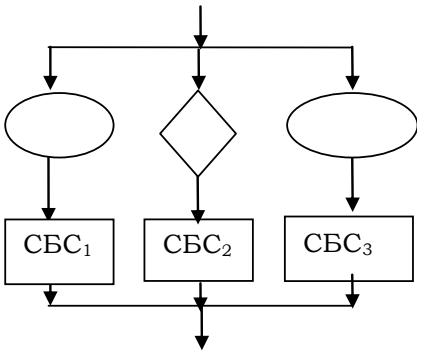
Обхід:



Вибір:

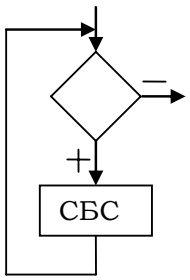


Детермінований вибір. Цю композицію можна отримати з композиції вибору, якщо деякі або всі вершини-ромби замінити на вершини-еліпси. Наприклад:

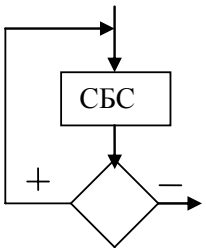


При інтерпретації вершини-еліпси будуть відповідати заключним охоронам.

Ітерація:



Повторення:



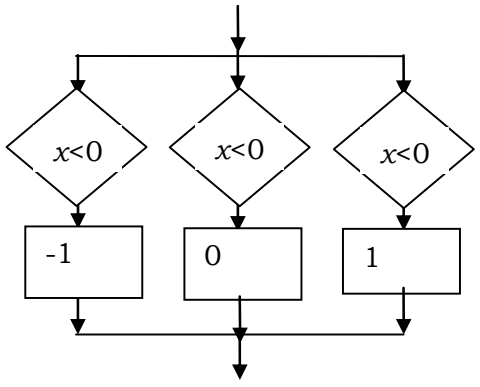
Структурні блок-схеми – це елементи замкнення базових елементів відносно наведених вище структурних композицій.

Як елементи замкнень усі СБС є скінченними об'єктами. Нехай $\Pi = \langle \Gamma, S, \Delta \rangle$ – обчислювальний простір із натуральним часом $\Gamma = N$, множиною станів S та $\Delta = \{\varphi_1, ..., \varphi_k\} \cup \{\gamma_1, ..., \gamma_l\}$ – сукупністю елементарних перетворень станів і предикатів на них, $\varphi_i : S \to S$, $i = \overline{1,k}$, $\gamma_j : S \to \text{Bool}$, $j = \overline{1,l}$. *Інтерпретацією* блок-схеми A називається пара

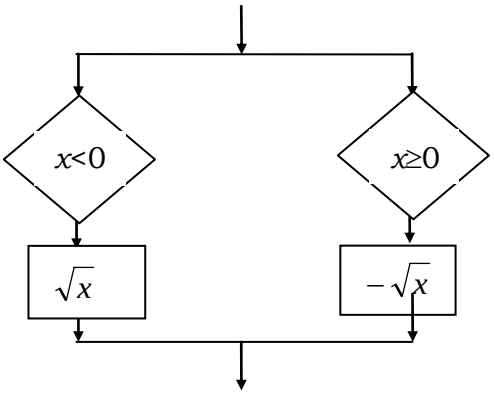
$I = (\Pi, \delta)$, де δ – функція розташування, яка кожному базовому елементу схеми A ставить у відповідність певне елементарне перетворення або предикат обчислювального простору. При цьому в елементах-прямокутниках схеми функція δ розташовує елементарне перетворення, а в елементах-ромбах – елементарний предикат.

Блок-схема A разом зі своєю інтерпретацією I і виділеною підмножиною початкових станів (входів) S_0 називається *структурним алгоритмом*, який будемо позначати $A^I(S_0)$. Кожний такий А-алгоритм задає формальне правило для обчислення значень певної регулярної функції (див. підрозд. 1.3.3). Ми не будемо строго задавати функцію керування такого А-алгоритму – це можна зробити індукцією за часом (див. вправу 34). Просто пояснимо, як гіпотетичний Виконавець виконує структурний алгоритм $A^I(S_0)$. Позначимо через $A^I(s)$ значення алгоритму A^I на вході $s \in S_0$. Правило обчислення значення $A^I(s)$ визначається структурою СБС. Складений структурний алгоритм трактується за правилом множення функцій, розгалуження – за правилом розгалуження функцій, обхід – за правилом обходу тощо. Усі ці правила можна знайти в підрозд. 1.3.3. Інтуїтивно це означає, що необхідно, розпочинаючи зі входу СБС, рухатись за стрілками й виконувати відповідні дії на поточному стані. Це можуть бути або елементарні перетворення, або обчислення значень предикатів. Даний рух або закінчиться за скінченну кількість кроків у вихідній вершині СБС, або безрезультатно зациклиться. Розглянемо кілька прикладів структурних алгоритмів.

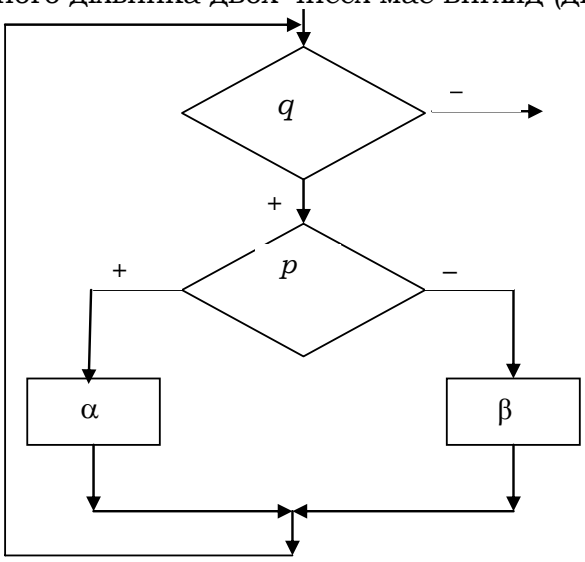
Приклад 2.4. Алгоритм вибору для обчислення відображення $\text{sign}(x)$ має вигляд:



Наступний структурний алгоритм обчислює квадратний корінь $\sqrt{x}$:



Цей алгоритм також є вибором, але, на відміну від попереднього вибору, він недетермінований. Алгоритм GCD для обчислення найбільшого спільного дільника двох чисел має вигляд (див. приклад 1.8):



■

2.1.6. КОНСТРУКТИВНІ ІНФОРМАЦІЙНІ СИСТЕМИ

Серед інформаційних систем найбільшого поширення на практиці набули конструктивні інформаційні системи.

Конструктивною називається будь-яка узгоджена інформаційна система з алгоритмічною вихідною системою.

Прикладом конструктивних інформаційних систем є інформаційні системи на базі ОбС, узгодженість яких забезпечують системи програмування, що реалізують програми-запити.

Питання про узгодженість теоретичних моделей інформаційних систем може бути далеко не простим. З теорії алгоритмів і математичної логіки відомо, що існують моделі інформаційних систем, які не можуть бути в принципі узгоджені в межах традиційних універсальних класів алгоритмів та їхніх Виконавців. Це залежить від набору запитів вхідної системи й відповідної (теоретичної) мови програмування. Як зазначалось у вступі, А. Чорч у 1936 р. навів приклад формули ПЧП, для якої не існує класичної алгоритмічної реалізації (напр., у класі машин Тьюрінга), і започаткував тим самим клас *алгоритмічно нерозв'язних проблем*. Сьогодні цей клас містить уже десятки й сотні таких проблем у різних предметних областях. Більшість із них зводиться до алгоритмічної нерозв'язності проблем зупинки й самозастосовності машин Тьюрінга (див. завдання для СР).

Які ж існують шляхи для подолання неузгодженості загальних інформаційних систем? Їх може бути кілька. Стандартний шлях полягає в обмеженні умов задач (сукупностей запитів, інтерпретацій і означень змінних вхідних систем). Глобальний же шлях може передбачати заміну Виконавця вихідної системи або перехід узагалі до іншої парадигми конструктивності інформаційної системи [106, 109, 142].

2.1.7. СКЛАДНІСТЬ ІНФОРМАЦІЙНИХ СИСТЕМ

Проблема ефективності реалізацій запитів є настільки багатогранною, що в кожному випадку до неї підходять індивідуально. З одного боку, є теорія складності обчислень, яка базується на тьюрінговій моделі обчислень (теорія NP-повних проблем), з іншого, – є реальні ОбС, далекі від цієї моделі. Деякі аспекти складності систем узагалі не можуть бути охоплені існуючою теорією складності обчислень – наприклад технологічні проблеми керування розробкою великих інформаційних систем. Є загальна проблема опису поведінки таких систем: у них можуть одночасно існувати сотні й навіть тисячі змінних і кілька потоків керування. З погляду комбінаторики кількість станів таких об'єктів астрономічно велика, що породжує особливу складність опису цих систем і керування їхньою розробкою. Проблеми ефективності інформаційних систем перебувають у центрі уваги вже згадуваної програмної інженерії й потребують окремого детального розгляду. Зупинимося тільки на одній з її складових – понятті складності алгоритмів.

Виділяють *часову* та *просторову* складності алгоритму. Для їх дослідження на обох складових обчислювального простору алгоритму – часовій і станах – вводиться певна метрика. Наприклад, якщо йдеться про машини Тьюрінга, то часовою мірою може бути довжина обчислень, мірою їхніх станів – загальна кількість внутрішніх станів або довжина робочої стрічки в конфігураціях. Просторова складність програми може визначатися розміром інформаційних полів, що з'являються в процесі її виконання, а часова – часом, необхідним для виконання програми.

Ефективність програми залежить від ефективності вибраного для неї алгоритму. Час роботи алгоритму оцінюється за найбільшою кількістю елементарних операцій, які він виконує на вхідних даних одного розміру. При цьому важливим є не стільки кількість елементарних дій, скільки характер зростання їхньої кількості при збільшенні розміру вхідних даних. Отже, *часовою складністю алгоритму* називається функція $F_A(n)$, що визначає найбільшу кількість елементарних дій при виконанні алгоритму A на даних розміром n. Можна говорити й про часову складність задачі, розуміючи при цьому найменший час її розв'язання за допомогою будь-якого з алгоритмів. *Просторова складність алгоритму* $P_A(n)$ визначається розміром пам'яті, необхідним для його застосування, а просторова складність задачі – найменшим розміром пам'яті, необхідним для її розв'язання за допомогою будь-якого з алгоритмів.

Складність алгоритмів практично всіх реальних задач є неспадною функцією. Аналітичне подання функцій $F_A(n)$ та $P_A(n)$ для реальних алгоритмів зазвичай неможливе. Тому практичне значення має *порядок* їх зростання відносно n. Наприклад, якщо при бульбашковому сортуванні (див. підрозд. 4.1.3) вхідний масив є відсортованим навпаки, то після кожного порівняння відбувається обмін – а це ще три присвоювання. Якщо нехтувати допоміжними операціями зі змінами індексів, то $F_A(n) = 4 \times n \times (n-1)/2$, тобто ми маємо функціональну залежність між розмірами n і максимальними кількостями елементарних дій, виконуваних за алгоритмом A.

Функція $f(n)$ має порядок, або верхню оцінку $O(g(n))$, якщо існують константи C і n_0 такі, що $|f(n)| \le C|g(n)|$ для всіх $n > n_0$.

Функція $f(n)$ має нижню оцінку $\Omega(g(n))$, якщо $|f(n)| \ge C|g(n)|$ для деякої константи C.

Наприклад, функція $f(n)=25n$ має порядок $O(n)$, оскільки для $C=25$ та $n_0=1$ вірно, що $f(n)=Cn$, $n>n_0$. Наведена вище часова складність бульбашкового сортування має порядок $O(n^2)$.

Якщо кількість виконаних операцій алгоритму описує функція $f(n)=\Omega(n^2)$, то це означає, що навіть у найкращому випадку буде виконано не менше порядку n^2 дій. Верхня й нижня оцінки тим змістовніші, чим вони ближчі відповідно до найменшої верхньої й найбільшої нижньої оцінок.

Якщо обидві (верхня й нижня) оцінки складності функції збігаються, то кажуть, що вона має оцінку $f(n)=\Theta(n)$. Наприклад, це виконується для наведеної вище функції $f(n)=25n$.

Приклади аналізу складності алгоритмів можна знайти в розд. 4. Для оцінки часової складності переважної більшості реальних алгоритмів достатньо логарифмічної, степеневої й показникової функцій, а також їхніх сум, добутків і підстановок. Усі вони монотонно зростають і задаються простими аналітичними виразами. Наприклад, $f(n)=n*(n-1)=O(n^2)$, оскільки для $n>2$ маємо $0.5*n^2<n*(n-1)<n^2$.

Основним способом порівняння ефективності алгоритмів є зіставлення порядків їхньої складності.

Увага! Найкращими з практичного погляду вважаються алгоритми з логарифмічною й лінійною часовою складністю $F_A(n)$. На жаль, у більшості випадків така складність недосяжна. Проте достатньо широкі класи алгоритмів мають прийнятну поліноміальну часову складність ►

Окрім часової та просторової складностей, розглядають також складності розуміння та структури алгоритму. Останню умовно можна назвати інтелектуальною складністю алгоритму.

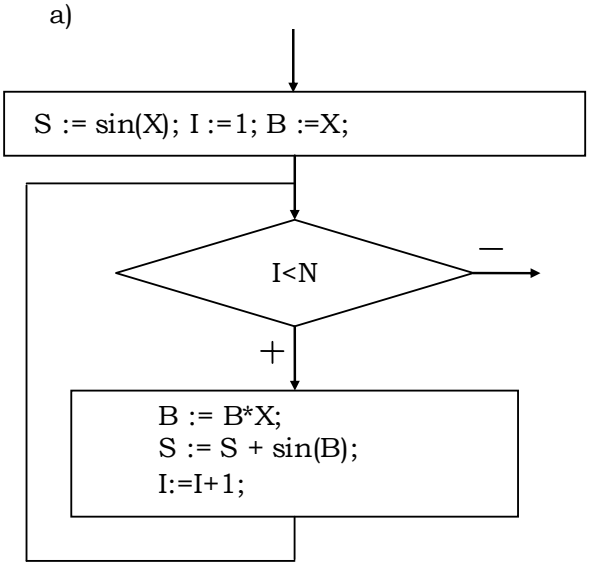
Усі форми складності взаємопов'язані. Зазвичай при поліпшенні часової складності алгоритму погіршуються його просторова та інтелектуальна складності, і навпаки.

***Література для СР:** вихідні системи – [93, 108]; алгоритми – [50, 56, 74, 80, 84, 94, 129]; А-алгоритми – [44, 45]; табличні алгоритми – [44, 45]; еквівалентність універсальних класичних систем алгоритмів, теза Чорча – [52, 80, 94]; NP-повні й алгоритмічно нерозв'язні проблеми – [29, 94, 97]; інтелектуальна складність інформаційних систем – [49, 90, 91]; шахові програми – [13, 83].

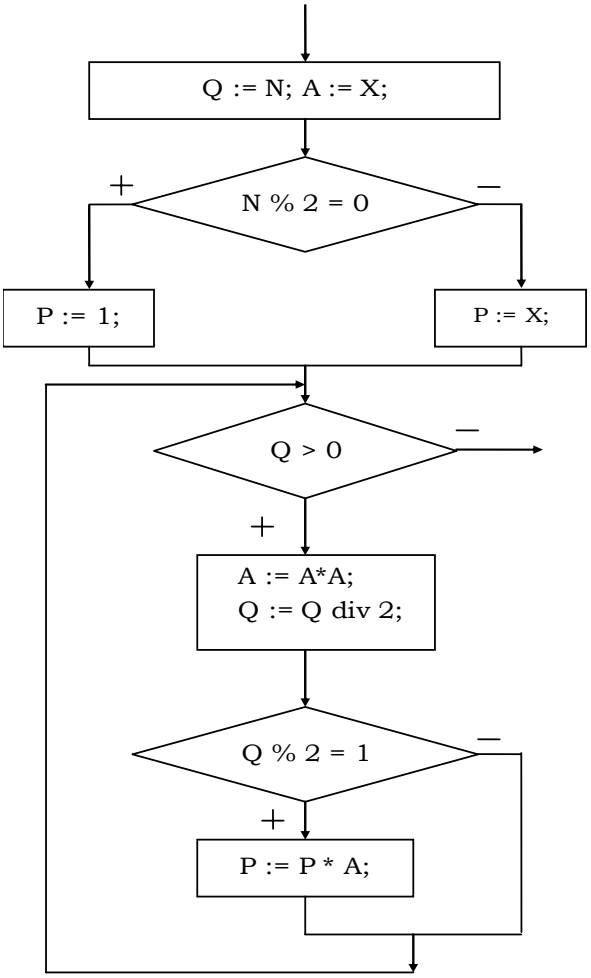
Контрольні запитання та вправи

1. Що таке вхідна система?
2. Дати визначення вихідної системи.
3. Яка роль вхідної й вихідної систем у інформаційній системі?
4. Що таке інформаційний об'єкт та інформаційне поле?
5. Що таке X-фрейм поля?
6. Що таке X-арна функція, еквітонна X-арна операція, $X-Y$-оператор?
7. Що таке оператор присвоювання?
8. Що таке операція присвоювання?
9. Що таке групове присвоювання?
10. У чому полягає різниця між операцією й оператором присвоювання?
11. Що таке векторний аналог і параметризований векторний аналог $X-Y$-оператора?
12. **. Сформулювати для узагальнених алгебричних систем з еквітонними X-арними операціями та предикатами поняття: а) замкненої підмножини й підсистеми; б) гомоморфізму та ізоморфізму; в) конгруенції. Довести аналоги лем 1.3 і 1.4 [94, 105].
13. Що таке функції кодування й декодування в інформаційній системі?
14. У чому полягає смисл узгодженості інформаційної системи? Навести приклади узгоджених і неузгоджених інформаційних систем.
15. Які інформаційні системи називаються конструктивними?
16. Що таке алгоритмічна мова?
17. Що таке конструктивний об'єкт?
18. Що таке абстрактний алгоритм?
19. Сформулюйте основні властивості А-алгоритмів.
20. У чому полягає релятивність і темпоральність А-алгоритмів?
21. Що таке алгоритми з оракулами?
22. Що таке графік А-алгоритму й алгоритмічно обчислювальна відповідність?
23. Що таке розв'язна множина?
24. Що таке частково розв'язна множина?
25. Сформулювати теорему про замкненість А-алгоритмів.
26. Що таке табличний алгоритм?
27. Написати послідовність ходів для обходу шахової дошки $n \times n$ конем і турою, які спочатку розташовані в лівому нижньому куті. Кожне поле відвідується лише один раз. Чи завжди можливий розв'язок?

28. Розв'язати шахові задачі. Розв'язок полягає в побудові відповідних шахових обчислень. Перший хід у задачах завжди в білих. Оголосити мат за три ходи:
    а) білі: Кр g8, Те1, Тg1, Кg3, пішак f4; чорні: Кр f6, Td7, Th6, пішак f5;
    б) білі: Кр f4, Тс4, Cd4, Ce4, Kf5, Kf3, пішаки: a2, b3, g4, h4; чорні: Кр b8;
    в) білі: Кр e4, пішаки: c7, d6, e7, f6, g7; чорні: Кр e6;
    г) білі: Кр e7, Са1, Kh6; чорні: Кр h8, пішаки: g7, h7;
    д) те саме, що й у г), тільки дошка перевернута – чорні пішаки прямують на восьму горизонталь.
29. Розробити нотацію для запису партій гри в; а) шашки; б) доміно. Навести приклади партій у цих іграх. Правила ігор можна знайти в [108].
30. Що таке табличний алгоритм?
31. Побудувати табличні алгоритми для функцій із вправ 12, 13, 27 із підрозд. 1.3.
32**. Довести теорему 2.2.
33. Дати визначення структурної блок-схеми та структурного алгоритму.
34*. Описати строго функцію керування структурного алгоритму.
35. Які функції обчислюють такі структурні алгоритми:
    а)



б)



36. Що таке алгоритмічно нерозв'язна проблема?
37. Що таке просторова й часова складності алгоритму?
38. Оцінити просторову й часову складності структурних алгоритмів із вправи 28.

2.2. Життєвий цикл інформаційних систем

- Основні етапи життєвого циклу інформаційних систем
- Деякі методологічні принципи програмування

Ключові слова: *життєвий цикл, технологія програмування, аналіз, зовнішня специфікація, технічне завдання, проектування, кодування, правильність і коректність системи, тестування, налагодження, верифікація, трансформаційне програмування, доказове програмування, документування, генератор звітів, дослідна експлуатація* (*атестація*), *експлуатація, технологія програмування, принципи підпорядкування, відокремлення, абстракції, декомпозиції, ієрархії, аксіоматизації, типізації, функціональності й композиційності, тип змінної, алгебра типу, конструктор типу, похідний тип, вежа типів, композиція програм, імперативна логіка програм.*

Надалі під інформаційними системами (або просто системами) будемо розуміти інформаційні системи на базі ОбС. Життєвий цикл систем уже згадувався у вступі. У цьому підрозділі ми детальніше обговоримо його з урахуванням специфіки ОбС.

2.2.1. ОСНОВНІ ЕТАПИ ЖИТТЄВОГО ЦИКЛУ ІНФОРМАЦІЙНИХ СИСТЕМ

Розпочнемо із загального визначення життєвого циклу систем. Для спрощення далі будемо називати його просто життєвим циклом.

Під життєвим циклом (ЖЦ) розуміють сукупність науково-технічних і організаційних заходів, спрямованих на розробку й експлуатацію інформаційних систем.

Бачимо, що ЖЦ охоплює не тільки процес створення інформаційних систем, а й питання, пов'язані з реалізацією відповідних комунікативних процесів.

ЖЦ систем складається з *етапів*. Зазначимо, що етапи відображають лише загальну структуру ЖЦ. Конкретна ж його реалізація може мати свої особливості й динаміку, які визначаються вибраною технологією програмування. На практиці реальні ОбС можуть бути дуже складними. Ефективно продукувати такі складні об'єкти неможливо без використання спеціальних технологій.

Основою *технологій програмування* (ТхП) є різноманітні засоби (у тому числі й автоматизовані), що реалізують ЖЦ. Будь-яка ТхП спирається на

певну методологію, тобто на сукупність певних концепцій, методів, організаційних заходів тощо. Розділ інформатики, що вивчає технологічні проблеми програмування, отримав назву *програмної інженерії.*

Охарактеризуємо коротко кожний з етапів ЖЦ.

На **етапі аналізу** вивчається актуальність задачі з побудови даної інформаційної системи й описуються (*специфікуються*) вимоги до неї. Цей етап містить:

- з'ясування актуальності задачі;
- специфікацію вхідної системи (на теоретико-множинному рівні);
- вимоги до апаратури й системної частини ОбС;
- прагматичні вимоги до прикладної частини ПЗ;
- оцінку й планування ризиків.

Перед початком розробки вивчають актуальність системи й комерційну доцільність її створення (який на неї може бути попит, скільки вона коштуватиме, які матиме переваги над існуючими аналогами тощо). Потім аналізують безпосередньо складові інформаційної системи. Як зазначалось у підрозд. 2.1, вхідні системи є наближеними моделями реальних об'єктів зовнішнього світу та взаємозв'язків між ними. Тому спочатку вивчають первинні об'єкти, операції та предикати на них. Такі первинні системи можуть бути як неперервними, так і дискретними (зліченними), але їхні вхідні моделі – обов'язково дискретними, оскільки такими є вихідні системи, а функції кодування в інформаційних системах вважаються однозначними. Фактично вхідні системи є фактор-алгебрами первинних систем. Звідси й випливає методика побудови вхідних Ω-систем. Шляхом аналізу первинні об'єкти за певними ознаками об'єднують у класи еквівалентності й отримують зліченну фактор-множину первинних об'єктів – універсум майбутньої вхідної системи. При цьому необхідно слідкувати, щоб усі операції та предикати були стабільними відносно вибраної еквівалентності. Тоді (див. підрозд. 1.4.2) на фактор-множинах можна визначити операції та предикати так, щоб отримати модель первинної системи, тобто вхідну Ω-систему. Після цього уточнюється сукупність запитів системи, а точніше, відповідність між їхніми вхідними параметрами й результатами.

Прагматичні вимоги до ПЗ стосуються ефективності використання ресурсів, швидкодії роботи прикладних програм, інтерфейсу (зручності у спілкуванні з програмою), безпеки системи тощо.

Важливою складовою аналізу системи є специфікація й тестування властивостей апаратури та операційної системи ОбС, а також оцінка й планування ризиків проекту.

Результат аналізу системи називається *зовнішньою специфікацією системи* (ЗСС). ЗСС оформлюють у вигляді *технічного завдання* (ТЗ), яке є невід'ємною складовою спеціального юридичного документа – Договору про розробку програмного продукту. Суб'єктом Договору є *Замовник* – фізична чи юридична особа, яка бажає отримати певний програмний продукт, і *Розробник* – теж фізична або юридична особа, яка бере на себе зобов'язання з виготовлення й постачання програмного продукту відповідно до вимог за умовами Договору.

Етап проектування полягає у:

- виборі згідно з вимогами вхідної специфікації системи структури й елементної бази апаратної частини системи та її ПЗ;
- виборі функцій кодування й декодування;
- розробці архітектури системи – декомпозиції її на підсистеми, установлення взаємозв'язків між ними й визначення структури керування;
- деталізації семантичної структури підсистем, розбитті їх на модулі, окремі функції тощо.

Щоб обрати функції кодування й декодування, необхідно визначити структуру вхідних і вихідних даних у програмах. Після цього відповідно до прийнятої технології програмування розробляються загальна структура та склад усього комплексу прикладних програм і зв'язків між ними, а також семантична структура кожної з них. За допомогою спеціальних засобів і методів здійснюється декомпозиція їх на простіші складові – блоки, модулі, функції, класи тощо.

На **етапі кодування** результати проектування трансформуються в конструкції мови (мов) програмування.

Обґрунтування правильності інформаційної системи (**ОПС**) полягає в перевірці того, що, по-перше, отримана вихідна система є узгодженою (*коректною*), тобто дійсно реалізує запити вхідної системи згідно з аналізом задачі відносно вибраних функцій кодування й декодування і, по-друге, вона задовольняє всі прагматичні вимоги. Обґрунтування коректності може здійснюватись як інженерними, так і математичними методами. *Інженерні* методи базуються на *тестуванні* систем. Спочатку тестуються окремі функції й модулі, а потім здійснюється комплексне тестування всієї системи. Тестування програми (функції) P полягає у проведенні контрольних обчислень за програмою на окремих спеціально підібраних варіантах вхідних даних (*тестах*) з метою виявлення можливих дефектів і помилок в її роботі. Після кожного контрольного обчислення результати порівнюють із тими, що мають відповідати запитам, тобто для кожного тесту a й кожного запиту Φ вхідної системи перевіряють умову (*) узгодженості запиту та його реалізації в системі (див. підрозд. 2.1.3). Якщо знайдено невідповідність, то шукають помилку в програмі й

намагаються її усунути. Цей процес називається *налагодженням* (англ. – debugging) програми (рис. 2.2). Тестування й налагодження є своєрідним мистецтвом. Воно полягає в умінні підібрати тести так, щоб мінімізувати наявні ресурси (матеріальні й часові) для виявлення й усунення можливих помилок ("ляпів") у програмі. На жаль, на практиці катастрофічний дефіцит ресурсів не дозволяє за допомогою тестування повністю розв'язати проблему обґрунтування коректності системи.

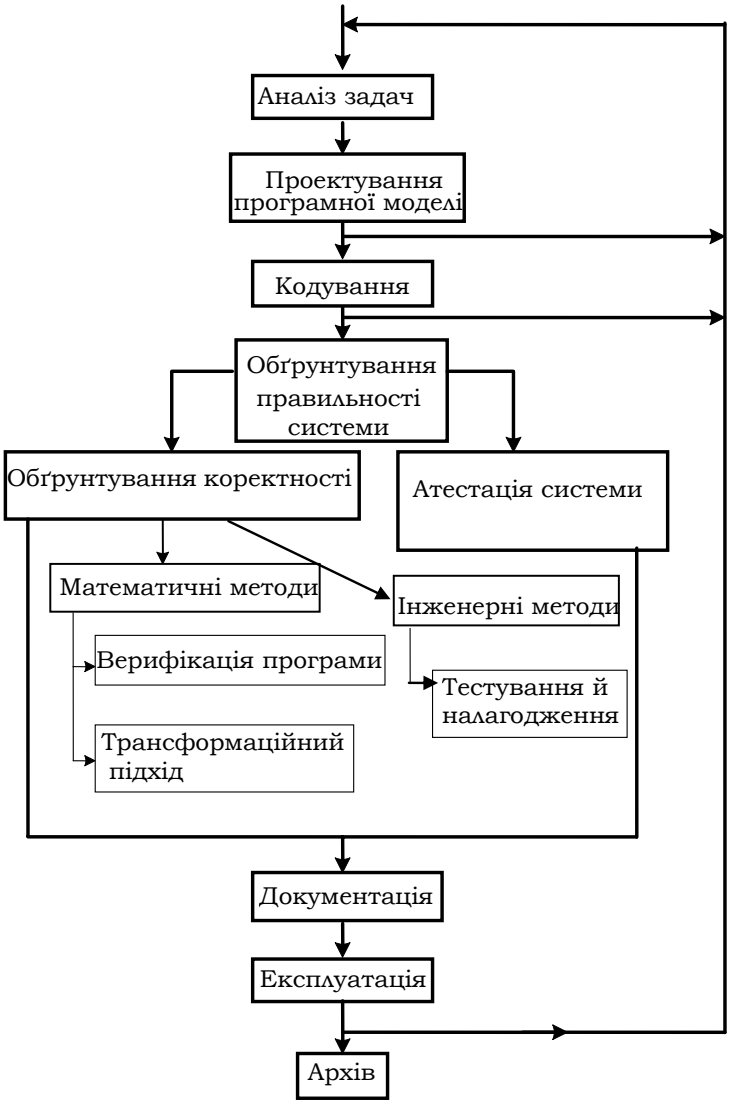


Рис. 2.2

Тестування дозволяє виявити тільки деякі (найгрубіші) помилки програми, але не гарантує того, що інші помилки не будуть знайдені в майбутньому, наприклад під час атестації та експлуатації системи. Детальніше тестування й налагодження розглядатимуться в підрозд. 2.7.5, 2.7.6.

Отже, у загальному випадку тестування лише частково обґрунтовує коректність системи, тому цілком природним є звернення до математики в пошуку інших, надійніших методів. На відміну від інженерних, *математичні* методи роблять коректність системи математичним фактом. Технології програмування, що спираються на подібні методи ОПС, отримали назву *доказового програмування*. Необхідною передумовою такого підходу є наявність формальної специфікації вхідної системи. Розглянемо кілька підходів до доказового програмування.

Історично перший підхід виник у середині 70-х рр. XX ст. і базувався на математичній *верифікації* програм. За допомогою верифікації намагалися математично довести коректність програми в межах певної математичної системи. Для цього було розроблено спеціальні математичні методи формального подання семантики програм і відповідні логічні засоби для роботи з ними. Отримані результати набули широкого розголосу в науковій спільноті та знайшли відображення в літературі, насамперед у роботах Т. Хоара, Е. Дейкстри, Д. Гріса та інших спеціалістів (див. літературу для СР). Експерименти з конкретними програмами показали, що математичні доведення коректності програм є реальними. Однак у більшості випадків ці доведення досить складні й самі потребують великих людських і матеріальних ресурсів. Текстуально вони займають (принаймні на сьогодні) набагато більший обсяг, ніж програми. У зв'язку із цим виникає нова проблема – коректності доведень. Було зроблено спроби автоматизувати процес верифікації програм для певних класів задач за допомогою спеціальних програм-верифікаторів.

Методи верифікації залишаються поки що переважно теоретичними. Проте вони дозволили краще зрозуміти природу програмування й підштовхнули до нових ідей у цій галузі. Зокрема, відомі досить успішні спроби застосування методів верифікації програм при проектуванні мікросхем тощо.

Перспективнішим на практиці виявився інший варіант доказового програмування – *трансформаційне програмування.* Воно полягає в застосуванні спеціальних мов специфікацій для опису певного спектра проміжних моделей вхідної системи та здійснення серії еквівалентних трансформацій цих моделей таким чином, щоб на завершальному етапі була отримана потрібна програмна система. При цьому методи еквівалентних трансформацій проміжних моделей ма-

ють гарантувати результативність усього процесу програмування й коректність отриманої вихідної системи. Отже, спочатку в процесі аналізу специфікується вхідна система $S_{\text{вх}}$. Потім за допомогою певних правил Tp_1 будується її модель S_1. За S_1 за допомогою спеціальних трансформацій Tp_2 отримують її модель S_2 тощо. Будується ланцюг систем $S_{\text{вх}} \xrightarrow[Tp_1]{} S_1 \xrightarrow[Tp_2]{} S_2 \xrightarrow[Tp_3]{} \ldots \xrightarrow[Tp_n]{} S_n = S_{\text{вих}}$ такий, що кожна з них є моделлю попередньої системи, а остання задовольняє необхідні вимоги. Ідея базується на тому, що трансформаційні правила мають зберігати еквівалентність моделей, а модельність є достатньої умовою для узгодженості системи. Тому за побудовою завершальна модель буде необхідною програмною моделлю вихідної системи. Такий підхід до доказового програмування досить реалістичний і стає все популярнішим. Приклади його застосування див. у підрозд. 2.5.4. Достатньо розглянути перехід від рекурентного або індуктивного описів вхідних співвідношень до відповідного набору структурних чи рекурсивних схем програм, далі – до еквівалентних С-програм і, нарешті, після роботи С-компілятора – до об'єктного й машинного кодів системи. Тут ми маємо ланцюг систем, що розпочинається з рекурентного опису функції й закінчується машинним кодом відповідної програми. Кожний крок здійснюється за чіткими формальними правилами, а останні кроки, розпочинаючи з компіляції, – узагалі автоматично.

Формальні методи специфікації програм і способи їх еквівалентних перетворень розробляє й вивчає теорія програмування. Багато таких методів і відповідних перетворень уже досить глибоко вивчені й широко застосовуються на практиці (теорія компіляції та інтерпретації мов програмування, мови моделювання та специфікацій тощо).

Чи задовольняє вихідна система прагматичні вимоги, з'ясовують під час атестації, а також шляхом аналізу її семантичної структури.

Етап документування полягає в підготовці внутрішньої програмної документації відповідно до вимог ТЗ. Існують певні міжнародні й національні стандарти на таку документацію. У сучасних технологіях програмування на цьому етапі використовують спеціальні системи – *генератори звітів.*

Етап експлуатації полягає в реалізації запитів інформаційної системи, тобто кодуванні та введенні в ОбС вхідних даних, запитів і проведенні відповідних обчислень. Розпочинається зазвичай із *дослідної експлуатації,* або *атестації,* системи, під час якої Замовник перевіряє, чи задовольняє програма вимоги ТЗ. Паралельно відбуваєть-

ся навчання майбутніх користувачів. Для Розробника ж дослідна експлуатація – це додаткова можливість продовжити тестування й налагодження системи. При цьому поліпшується інтерфейс і остаточно з'ясовується її життєздатність. У процесі експлуатації може виникнути необхідність у зміні постановки задачі, подальшому удосконаленні системи тощо. Тому важливо, щоб система проектувалась і програмувалась з урахуванням можливості її майбутнього модифікування.

У цілому, як видно з рис. 2.2, ЖЦ має циклічну природу. Для повернення назад до одного або кількох попередніх етапів може бути багато причин. Це і виявлені на етапах обґрунтування коректності або експлуатації помилки чи недоліки в роботі системи (найчастіше), і зміни Замовником вимог до системи тощо.

2.2.2. ДЕЯКІ МЕТОДОЛОГІЧНІ ПРИНЦИПИ ПРОГРАМУВАННЯ

Системний підхід до вивчення та програмування інформаційних систем вимагає певної методології, що ґрунтується на *загальних* і *спеціальних принципах*. Зупинимось на деяких найважливіших. Спочатку розглянемо **загальні принципи**.

Принцип підпорядкування. Акцентує увагу на трьох основних аспектах програм – прагматиці, семантиці й синтаксисі та фіксує відповідний пріоритет між ними, а саме: прагматичні властивості програм визначають їхні семантичні властивості, які, у свою чергу, продукують синтаксичну структуру:

Прагматика $\Rightarrow$ **Семантика** $\Rightarrow$ **Синтаксис**

Таке підпорядкування, зокрема, означає, що розробку будь-якої системи слід розпочинати з етапу з'ясування її призначення й вимог до неї, а вже потім проектувати семантичну структуру, і тільки після цього оформлювати в термінах конкретної мови програмування. На жаль, як свідчить практика програмування, цей принцип часто порушується, що негативно впливає на якість і результативність розробок систем.

Принцип відокремлення. Фіксує відносну самостійність основних аспектів програм, тобто синтаксису, семантики й прагматики. Оскільки реальні програми як об'єкти дуже складні, то на практиці кожен з аспектів може розглядатися окремо із застосуванням своїх дескриптологічних засобів. Для прагматики використовують звичайну мову та ПЧП; для семантики – спеціальні мови специфікацій, граф-схеми, різні діаграми тощо; для синтаксису – формальні грама-

тики, автомати та інші формалізми. Отже, кожен з аспектів фіксують і спочатку досліджують окремо. Після цього вдаються до їхньої інтеграції, вивчення й побудови програм як цілісних об'єктів.

Принцип абстракції. Указує на необхідність використання в процесі програмування моделей систем на різних рівнях абстракції. Наприклад, у всіх технологіях програмування етапу безпосередньої побудови машинного коду передують етапи аналізу та проектування. Перший полягає в побудові певної абстрактної моделі вхідної системи й формулюванні прагматичних вимог до вихідної. На другому за моделлю вхідної системи проектують загальну структуру вихідної та окремі її модулі. При цьому для декомпозиції системи використовують різні рівні абстракції й відповідні до них мовні та інструментальні засоби. Останні завдяки рівню абстрактності зручніші для розробників, ніж машинні мови.

Принцип декомпозиції, або "розділяй і володарюй". Обернений до нього принцип "знизу-вгору". Згідно з принципом декомпозиції при проектуванні складної програмної системи її послідовно розділяють на все менші й менші підсистеми, кожна з яких потім деталізується окремо. Суть у тому, що робота з такими окремими підсистемами значною мірою здійснюється незалежно, отже, і простіше – немає необхідності тримати в голові водночас усю інформацію про систему. При проектуванні знизу-вгору спочатку реалізують окремі базові функції, модулі або класи об'єктів (у випадку об'єктно-орієнтованого програмування), потім, за їх допомогою, – певну ієрархію крупніших підзадач, яка закінчується самою системою. Реально при програмуванні складних систем використовують обидва принципи, що доповнюють один одного.

Принцип ієрархії. Полягає в упорядкуванні по вертикалі всіх об'єктів і зв'язків усередині системи, тобто об'єкти та зв'язки між ними систематизуються за певними ознаками й розташовуються в певному ієрархічному порядку. Така ієрархія спрощує структуру системи, робить її більш прозорою і зрозумілою. Послідовну реалізацію цього принципу демонструє об'єктно-орієнтоване програмування.

Принцип типізації. Полягає в упорядкуванні по горизонталі всіх об'єктів системи. Згідно з ним кожний інформаційний об'єкт системи належить певному типу, і тільки одному. Тип визначає загальну синтаксичну й семантичну структуру об'єктів, їхні загальні властивості та прагматику. Наприклад, аналіз типу даних дозволяє, не звертаючись до значень, контролювати на суто синтаксичному рівні їхнє помилкове використання в програмах.

Тип фіксує сукупність значень змінної, операції та стандартні функції, а також засоби доступу, визначені для неї.

Сукупність значень, операцій, предикатів і стандартних функцій типу визначають його *алгебру* (насправді – алгебричну Ω-систему). Коли семантично різні операції й функції позначаються одним іменем (так звані поліморфні операції та функції), їхній конкретний тип визначається за контекстом. Типи мають ієрархічну залежність і утворюють *вежу типів* мови програмування. Розрізняють *базові* типи й *похідні*, що утворюються з базових за допомогою певних правил, які називаються *конструкторами типів*. Базові типи разом із конструкторами є фундаментальною характеристикою мови програмування.

Принцип аксіоматизації. Указує на важливість фіксації певними логічними засобами властивостей систем, щоб зробити процес їхнього програмування цілеспрямованим і контрольованим. Для цього виділяють сукупність апріорних властивостей (аксіом) та індуктивних правил, що дозволяють з аксіом отримувати інші властивості систем, необхідні для контролю й керування процесом програмування. Принцип вимагає ретельного аналізу й аксіоматизації семантики мов програмування, зокрема аксіоматизації всіх їхніх типів даних і композицій.

Спеціальних принципів може бути багато – вони визначають специфіку парадигм програмування й пов'язаних із ними технологій. Наведемо для прикладу тільки два таких принципи.

Принцип функціональності. Кожна мова програмування описує певний фіксований клас інформаційних об'єктів і множину запитів про них. Основні конструкції мов програмування – дані та програми. *Даними* називаються конструкції, які подають у них інформаційні об'єкти, а *програмами* – конструкції, що подають запити. Запити можуть бути декларативними й процедурними. У першому випадку вони мають вигляд певних формул-декларацій, що описують зв'язок між вхідними й вихідними даними, у другому – алгоритмів, що описують процедуру пошуку результатів запиту. В обох випадках програма задає відповідність, яка пов'язує вхідні дані й результати обчислень за нею (графік програми). Принцип функціональності вказує на важливість графіків програм і алгоритмів у процесі побудови програмних моделей систем та їхньої аксіоматизації. Ці графіки можуть розглядатись окремо від програми й задаватись довільними прийнятними засобами, у тому числі й непроцедурними (табличними, аналітичними або алгебраїчними за допомогою рівнянь). Цей принцип лежить в основі *функціонального програмування*, рівень абстракції якого – теоретико-множинний.

Принцип композиційності. Програма як певна функція ($X-Y$-оператор) має певну внутрішню структуру, елементами якої є інтер-

фейсні функції, базові операції та предикати. Для кожного класу обчислювальних функцій важливим є питання його конструктивного алгебричного подання (проблема аналізу), що полягає в пошуку відповідної функціональної алгебричної системи з певною сукупністю твірних елементів. Така алгебрична структуризація програм використовується на етапах проектування й верифікації систем. Операції, що описують алгебричну структуру програм, отримали назву *композицій*. Систему твірних програмних функцій і композицій називають за В.Н. Редьком *імперативною логікою програм* (*ІЛП*).

ІЛП є *повною* для даної мови програмування, якщо дозволяє адекватно побудувати функціональну структуру будь-якої її програми.

Побудова й вивчення універсальних повних імперативних логік програм є однією з важливих передумов для наукового порівняльного аналізу та класифікації мов програмування, аксіоматизації й верифікації інформаційних моделей тощо.

Принцип композиційності лежить в основі *композиційного програмування*.

***Література для СР**: ЖЦ і його основні етапи – [49, 67, 68, 72, 88, 120]; доказове програмування – [27, 31, 52]; верифікація програм – [4, 52, 76, 92, 138, 145]; трансформаційне програмування – [88, 139]; композиційне програмування – [108]; методологія програмування – [68, 91, 93, 109]

Контрольні запитання та вправи

1. З яких етапів складається ЖЦ?
2. Що таке технологія програмування?
3. У чому полягає суть етапу аналізу систем?
4. Що таке зовнішня специфікація й технічне завдання системи?
5. Хто такі Замовник і Розробник?
6. У чому полягає етап проектування?
7. Що таке програмний код системи?
8. У чому полягає різниця між правильністю й коректністю інформаційної системи?
9. Що таке тестування й налагодження програм?
10. Що таке повна система тестів?
11. Що таке верифікація систем?
12. Що таке генератор звітів?
13. Що таке атестація системи?
14. У чому полягає експлуатація системи?
15. Яка причина циклічності ЖЦ інформаційних систем?

16. Що таке принцип підпорядкування? Пояснити на прикладах.
17. У чому полягає принцип відокремлення? Пояснити на прикладах.
18. Що таке декомпозиція систем?
19. Сформулюйте принцип типізації. Проілюструйте його на прикладі будь-якої мови програмування.
20. У чому полягає принцип функціональності програм?
21. Сформулюйте принцип композиційності програм.
22. Проілюструйте на прикладах зі своєї практики програмування застосування принципів: а) підпорядкування; б) відокремлення; в) декомпозиції; г) типізації; д) функціональності; е) композиційності; є) абстракції.
23. Дайте визначення типу змінної.
24. Що таке алгебра типу?
25. Що таке конструктор типу?
26. Що таке похідний тип?
27. Що таке вежа типів мови програмування?
28. Що таке композиція програм?
29. Що таке імперативна логіка програм?

2.3. Обчислювальні системи

- Структура комп'ютера
- Структура й функції операційних систем
- Машинні типи даних
- Навчальна машина

Ключові слова: *центральний процесор, лічильник команд, регістр, оперативна пам'ять, поле оперативної пам'яті, адреса поля, зовнішні пристрої, операційна система, ядро операційної системи, драйвер, командний процесор, утиліта, система програмування, текстовий редактор, компілятор, інтерпретатор, налагоджувач, об'єктний модуль, редактор зв'язків, завантажник, числа, літери, адреси, команди, машинне слово, беззнакові цілі, цілі, дійсні, адреси, позиційні системи числення, двійкова, вісімкова, десяткова й шістнадцяткова системи числення, число з фіксованою й рухомою точками, мантиса, порядок, нормалізоване число, структура навчальної машини, структура машинної команди, команди завантаження, зберігання, пересилання адрес, порівняння, переходу, керування пристроями введення-виведення, перетворення, арифметичні команди.*

Розглянемо детальніше структуру й функції ОбС на прикладі персонального комп'ютера та його програмного забезпечення.

2.3.1. СТРУКТУРА КОМП'ЮТЕРА

Основні структурні компоненти комп'ютера: *центральний процесор* (ЦП), або системний блок – пристрій, який безпосередньо отримує з оперативної пам'яті команди програми й виконує їх, а також керує всіма іншими пристроями комп'ютера; *оперативна пам'ять* (ОП); зовнішні пристрої; четвертий (інтерфейсний) компонент – системна шина, через яку основні компоненти обмінюються даними (командами, адресами, числовою й символьною інформацією).

Усі дані зберігаються в ОП і на зовнішніх пристроях. Розрізняють власне пам'ять і способи доступу до розміщеної в ній інформації. При цьому виділяють *прямий* і *послідовний доступи* до ділянок пам'яті. При прямому доступі час звернення однаковий для всіх ділянок, при послідовному – залежить від місця їх розташування. ОП складається з послідовності комірок, кожна з яких може зберігати елемент даних і має адресу – порядковий номер у послідовності. Вона передбачає прямий доступ до комірок і призначена для тимчасового зберігання програм і даних (зазвичай це короткотерміновий період, який збігається із часом виконання певної програми).

Найменша одиниця даних – *біт* – набуває одного з двох значень – 0 або 1. Елемент даних, що зберігається в комірці, називається *байт* і складається з восьми бітів. Ділянки ОП, в яких розміщуються машинні програми й дані, називаються *полями*. Поля – це неперервні ділянки пам'яті, тому для визначення поля достатньо вказати адресу його першої комірки й довжину – загальну кількість комірок. Стандартним полем ОП є *машинне слово* – поле, в якому зберігається більшість машинних команд. Зазвичай воно складається з двох байтів і адреса його кратна 2, але може бути й більшим, наприклад, довжиною 4.

У деяких процесорах із побайтовою адресацією дані незалежно від їхнього типу можуть розташуватися в ОП за будь-якою адресою. Однак здебільше в архітектурах процесорів використовується *принцип вирівнювання*, за яким обмежують сукупності адрес полів, де можуть зберігатися дані того чи іншого типу. Наприклад, стандартні цілі завжди розміщуються в машинних словах, а 32-бітні у комп'ютерах із побайтовою адресацією – тільки в полях з адресами, кратними 4 тощо.

Зовнішня пам'ять призначена для тривалого зберігання інформації. Адреси її комірок можуть формуватися складніше, ніж для ОП. Наприклад, такий пристрій, як дискова пам'ять, має сторінкову організацію, а адреса – структуру пари, що складається з номера сторінки та зсуву – номера комірки всередині сторінки.

Зовнішня пам'ять містить також пристрої введення-виведення, які забезпечують обмін інформацією між суб'єктами інформаційної частини. Це можуть бути різного роду термінали, клавіатура, миша, пристрої друку тощо.

Центральний процесор. Зупинимось детальніше на структурі ЦП, загальну схему якого зображено на рис. 2.3.

Безпосередньо команди виконує *арифметико-логічний пристрій* (АЛП). Порядок, за яким виконуються команди, задає *пристрій керування* (ПК). Важлива роль відводиться регістру – *лічильнику команд* – PC (англ. – Program Counter), в якому формується адреса наступної для виконання команди.

Алгоритм виконання машинної програми наведено на рис. 2.4. Він досить схематичний – деякі операції можуть виконуватися паралельно тощо.

Регістр стану процесора – PS (англ. – processor status) містить інформацію про поточний стан процесора й деякі побічні результати виконання останньої команди. Ця інформація використовується для керування роботою процесора (може впливати на вибір наступної команди тощо). ЦП має також власну робочу пам'ять у вигляді загальних регістрів R0, R1, …, Rn і регістр-покажчик системного стеку SP (англ. – Stack Pointer), в якому зберігається адреса вершини стеку. *Системний стек* – це службова ділянка ОП, що використовується для підтримки процесу виконання програм.

Загальна схема ЦП:

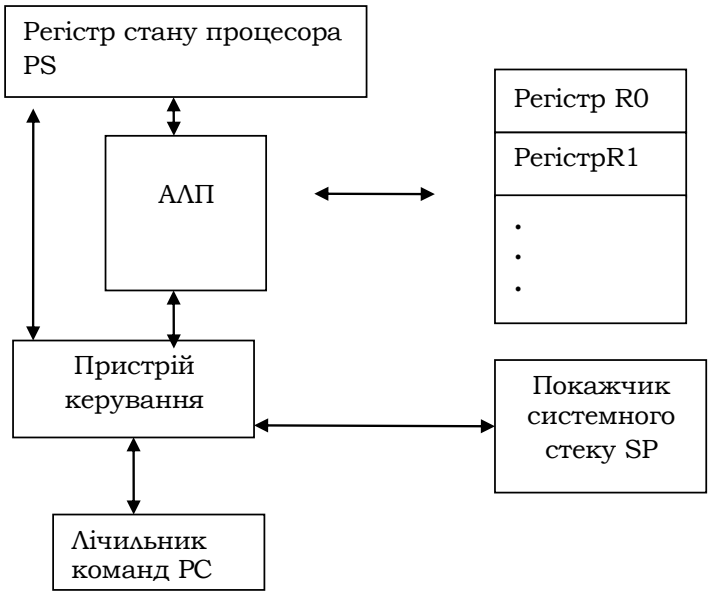


Рис. 2.3

Алгоритм виконання машинної програми подає така схема:

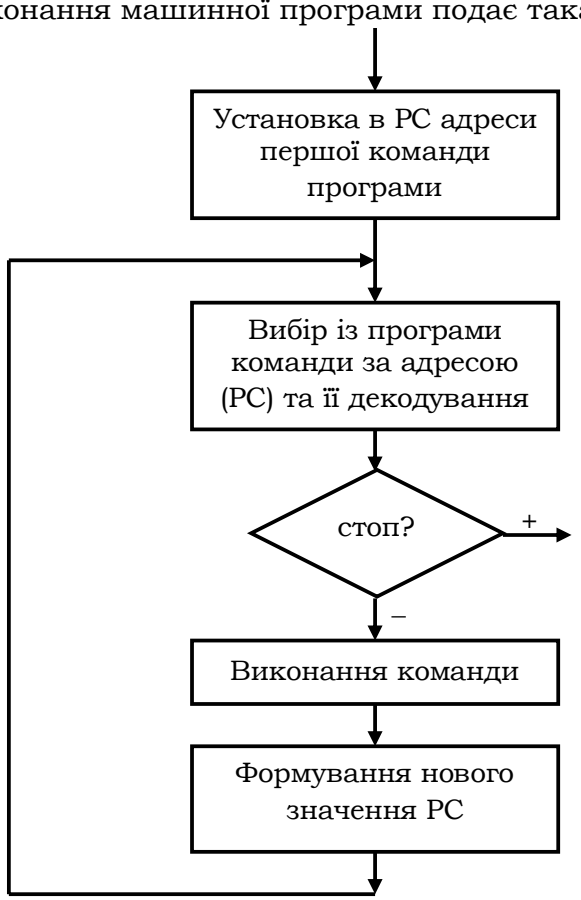


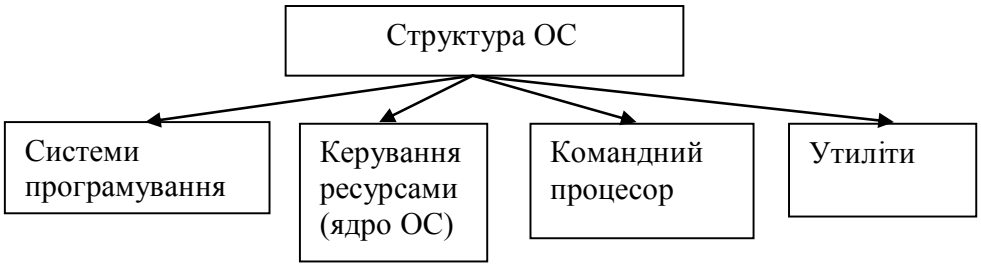
Рис. 2.4

2.3.2. СТРУКТУРА Й ФУНКЦІЇ ОПЕРАЦІЙНИХ СИСТЕМ

Процесор, пам'ять і зовнішні пристрої є основними *ресурсами* комп'ютера. Будь-яка програма потребує для підготовки й виконання певних ресурсів. Машинну програму в стадії виконання назвемо *процесом*. Керування ресурсами, зокрема планування й розподіл їх між процесами, здійснює операційна система.

Операційна система (ОС)– це комплекс машинних програм, призначений для автоматизованого керування ресурсами комп'ютера й забезпечення зв'язку між ним і користувачем.

Окрім автоматизованого керування, ОС надає користувачу засоби для оперативного керування комп'ютером, розробки програм, введення-виведення даних тощо. Структуру ОС можна зобразити так:

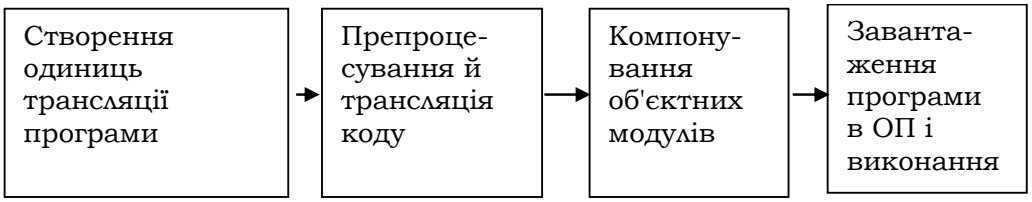


Командний процесор надає засоби для оперативного керування роботою комп'ютера і є спеціальним мовним інтерпретатором, що реалізує спеціальні команди, за допомогою яких можна змінювати параметри ОС, керувати зовнішніми пристроями, обробляти файли, розробляти програми, запускати системні й прикладні програми на виконання тощо.

Системи програмування призначені для автоматизації процесу програмування й виконання програм. Вони реалізують мови програмування і складаються з текстових редакторів, трансляторів (компіляторів та інтерпретаторів), компонувальників, завантажників програм тощо.

Утиліти – обслуговуючі програми, що здійснюють введення-виведення й реорганізацію даних. Типовими з них є програми роботи з файлами, сортування й архівації даних, фільтри, антивірусні програми тощо.

Розглянемо детальніше процес створення й виконання програм:



Програми складаються з однієї чи кількох одиниць трансляції. *Одиниця трансляції* – це програма чи її частина у вигляді текстових файлів, що допускає автономну трансляцію. Спочатку за допомогою текстового редактора формуються одиниці трансляції програми. Потім до них послідовно застосовуються препроцесор (за необхідності) і транслятор мови програмування.

Якщо процесором є *інтерпретатор*, то на вхід йому передаються вхідні дані, а трансляція (інтерпретація) полягає в побудові машинної

програми, завантаженні її (або її фрагментів) у ОП і виконанні на заданих вхідних даних.

Якщо ж процесором є *компілятор*, то в результаті компіляції на виході спочатку буде отримано *об'єктний* (*бінарний*) *код* програми, який подається на вхід *редактора зв'язків,* або *компонувальника*, який із цього об'єктного коду та об'єктних кодів стандартних функцій компонує *образ задачі* – машинну програму з відносними (віртуальними) адресами, що розміщується попередньо в зовнішній пам'яті. На відміну від інтерпретатора компілятор не завантажує й не виконує програму.

Усі компілятори мають бібліотеки стандартних функцій, призначені для виконання найважливіших загальних задач. Під час компіляції компілятор зазначає виклики стандартних функцій, потім програма-компонувальник пов'язує об'єктний код програми з об'єктними кодами бібліотечних функцій.

Щоб виконати образ задачі, його необхідно завантажити в ОП і передати керування на першу команду. Під час завантаження віртуальні адреси образу задачі замінюються абсолютними адресами ОП.

Кожна з фаз створення програми ініціалізується певною командою командного процесора ОС.

Уся пам'ять, що надається машинній програмі для її виконання, розподіляється на чотири логічні області (рис. 2.5). Перша містить машинний код програми; у другій зберігаються глобальні дані; у стеку – автоматичні дані й параметри виклику функцій, адреси повернення з виклику функцій тощо; у купі – решта програмних даних; у службовій частині ОП – ядро ОС та інша службова інформація, необхідна для керування процесами.

Рис. 2.5

У сучасних ОС системи програмування забезпечуються інтегрованим графічним інтерфейсом і називаються *інтегрованим середовищем розробки.*

2.3.3. МАШИННІ ТИПИ ДАНИХ

Машинні типи даних становлять основу базових типів мов програмування. Це дані, якими безпосередньо оперують машинні команди. До них належать: беззнакові цілі, цілі, дійсні, літери, адреси (у тому числі й адреси машинних команд). Для подання даних машинних типів зазвичай використовують двійкову позиційну систему числення.

Позиційні системи числення. *Системи числення* (СЧ) – це дескриптологічні системи, призначені для подання чисел. В інформатиці застосовують різні системи числення. Наприклад, у теорії виняткова важлива роль належить індуктивному визначенню натуральних чисел. Однак для організації конкретних обчислень перевагу надають зображенню чисел у *позиційних системах числення.* Кожна з таких систем використовує певну *основу* – довільне натуральне $p \geq 2$. Систему з основою p будемо позначати СЧ$_p$. Вона є узагальненням звичайної десяткової системи. Алфавіт СЧ$_p$ складають p літер-цифр $\Sigma_p = \{c_0, c_1, ..., c_{p-1}\}$ і допоміжні літери $\{".","+","-"\}$. У випадку $p > 10$ перші 10 цифр є звичайними десятковими від 0 до 9, для решти вибираються свої фіксовані імена. Наприклад, для запису чисел з основою 16 шістнадцяткові цифри від 10 до 15 позначаються латинськими літерами A, B, C, D, E та F.

Ціле число у СЧ$_p$ має вигляд слова в алфавіті Σ_p. Далі під цілим та іншими числами будемо розуміти залежно від контексту або їхній запис, або значення. Нехай $n = a_k a_{k-1} ... a_0$ – ціле число. Тоді $a_i \in \Sigma_p$, $1 \leq i \leq k$, називається *i-м розрядом* числа n, a_0 – *молодшою,* а a_k – *старшою* його цифрами. Щоб підкреслити приналежність числа до даної системи, будемо індексувати число її основою p. Наприклад, пишемо 25_{10} та 25_{16}, що означає відповідно число 25 у десятковій і шістнадцятковій системах. Відсутність знака перед числом, або якщо йому передує знак "+", означає додатне число. Знак "-" перед числом є ознакою від'ємного числа.

Значенням числа n_p вважається десяткова сума полінома $\overline{n}_p = \left(\sum_{i=0}^{k} c_i p^i \right)_{10}$, де c_i та p трактуються як відповідні десяткові числа. Продовжуючи попередній приклад, отримаємо:

$\overline{25}_{10} = 5 \times 10^0 + 2 \times 10^1 = 25_{10}$, $\overline{25}_{16} = 5 \times 16^0 + 2 \times 16^1 = 37_{10}$.

Раціональні числа мають дві форми подання – з фіксованою й рухомою точками. *Число з фіксованою точкою* (*ФТ-число*) має вигляд слова в алфавіті цифр, усередині якого (або на початку чи в кінці) розміщено точку. Наприклад, 0.25_{10}, $.25_{16}$ 3.1428_{10}, 101.1_2, $1111._2$ – ФТ-числа.

Нехай $x = a_k a_{k-1} \ldots a_0 \,.\, b_{-1} b_{-2} \ldots b_{-l}$ – довільне ФТ-число. Тоді слово $a_k a_{k-1} \ldots a_0$ називається *цілою,* а $.b_{-1} b_{-2} \ldots b_{-l}$ – *дробовою* його частиною. *Значенням* числа $x = a_k a_{k-1} \ldots a_0 .\, b_{-1} b_{-2} \ldots b_{-l}$ є десяткова сума цілої й дробової його частин. Значенням дробової частини числа x є сума $\left(\sum_{i=-1}^{-l} b_{-i} p^{-i} \right)_{10}$, де b_i та p трактуються як десяткові числа. Наприклад,

$$\overline{0.25}_{10} = 2 \times 10^{-1} + 5 \times 10^{-2} = 0.2 + 0.05 = 0.25_{10},$$

$$\overline{0.25}_{16} = 2 \times 16^{-1} + 5 \times 16^{-2} = 0.125 + 0.01953125 = 0.14453125_{10},$$

$$\overline{101.1}_2 = \overline{101} + \overline{.1_2} = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^{-1} = 5 + 0.5 = 5.5_{10}.$$

Як і цілим, дробовим числам може передувати знак. *Число з рухомою точкою* (*РТ-число*) складається з мантиси m і порядку. *Мантисою* може бути ціле число або число з фіксованою точкою. *Порядок* має вигляд $e \pm n$, де літера e – формальна ознака порядку, а n – натуральне число. Таким чином, РТ-число має вигляд $m\, e \pm n$. Наприклад, $1e + 2_2$, $-1.25e + 2_{16}$, $3.1428e - 100_{10}$ – РТ-числа.

Значенням РТ-числа $m\, e \pm n$ є значення мантиси, помножене на масштабний множник, який задається порядком. Якщо мантиса ціла, то вона перетворюється до еквівалентного ФТ-числа, а масштабний множник – це степінь основи із заданим порядком. Наприклад:

$$\overline{1e + 100_2} = 1.0 \times 2^4 = 16.0,$$

$$\overline{-0.25e + 2_{16}} = -0.14453125 \times 16^2 = -37.0,$$

$$\overline{-3.1428e - 50_{10}} = -0.\underbrace{00\ldots0}_{49}31428.$$

Потрібно сказати кілька слів про прагматику чисел і позиційних систем числення. З одного боку, числа служать кількісною мірою множин і окремих об'єктів, їх можна порівнювати й виконувати над ними арифметичні дії, які моделюють реальні й дуже важливі кількісні співвідношення між предметами та їхніми сукупностями. Не випадково більшість моделей, з якими мають справу природничі (а наразі все частіше й гуманітарні) науки, є числовими або зводяться до числових. З іншого боку, як було зазначено у вступі, числа в позиційних системах виявились зручними для механізації та автоматизації основних арифметичних операцій і відношень між ними.

Арифметичні операції додавання, віднімання, множення й ділення в позиційній системі з основою p виконуються за тими самими правилами, що й у десятковій системі.

Розглянемо як приклад додавання двох чисел з основою p $n_p = a_k a_{k-1}...a_0$ та $m_p = b_l b_{l-1}...b_0$.

Нехай

$$k > l,\ s_i = a_i + b_i,\ 0 \le i \le l,\ s_i = a_i,\ s'_{i+1} = (s'_i < p \to s_{i+1} \mid s_{i+1} + 1),\ l+1 \le i \le k,$$
$$t = (s'_k < p \to k \mid k+1).\ s'_t = (t = k \to s'_k \mid 1).$$

Тоді

$$\bar{n}_p + \bar{m}_p = \sum_{i=0}^{k} a_i p^i + \sum_{i=0}^{l} b_i p^i = \sum_{i=0}^{l} s_i p^i + \sum_{i=l+1}^{k} s_i p^i =$$
$$= \sum_{i=0}^{l} ((s_i < p \to s_i \mid s_i - p)p^i + p^{i+1}) + \sum_{i=l+1}^{k} s_i p^i = \sum_{i=0}^{t} s'_i p^i.$$

Відоме правило додавання у стовпчик базується саме на цих співвідношеннях і реалізує пошук цифр s'_i суми $\bar{n}_p + \bar{m}_p$. Додамо, наприклад, двійкові числа 10010_2 та 11110_2 і шістнадцяткові $F5$ та 66:

а) $+\ 10010_2$
11110_2
100000

б) $+\ F5$
66
$15B$

У другому прикладі $k = l = 1$, $s'_0 = s_0 = B(=11)$, $s_1 = F + 6 = 21$, $s'_1 = 21 - 16 = 5$, $t = k + 1 = 2$ та $k < t$. Отже, $s'_2 = 1$.

Аналогічно додаванню обґрунтовуються правила для решти арифметичних операцій.

Переведення чисел з однієї системи в іншу. У маніпуляціях із машинними даними важливу роль відіграє перекодування даних, тобто перехід від однієї СЧ $_p$ до іншої СЧ $_q$ так, щоб зберігалась інформація.

Такі перекодування отримали назву *переведення чисел* з однієї системи в іншу. Будемо їх позначати $(p)\to(q)$, де p,q – основи відповідних СЧ.

Для перетворень вигляду $(p)\to(10)$ достатньо скористатися визначенням значення числа з основою p і обчислити відповідну суму.

Для довільного переведення $(p)\to(q)$ теж скористаємося десятковою системою. Нехай $n_p = a_k a_{k-1}...a_0$ – довільне ціле в системі СЧ$_p$ і необхідно знайти його еквівалент $n_q = b_l b_{l-1}...b_0$ у системі СЧ$_q$. Значення $\overline{n}_p = \left(\sum_{i=0}^{k} a_i p^i\right)_{10}$ та $\overline{n}_q = \left(\sum_{i=0}^{l} b_i q^i\right)_{10}$ у результаті мають збігатися. Покладемо $u = \overline{n}_p$. З урахуванням того, що u та q відомі, фактично необхідно розв'язати рівняння

$$u = b_l q^l + ... + b_1 q^1 + b_0 = (b_l q^{l-1} + ... + b_1)q + b_0 \quad (*)$$

відносно b_i.

Якщо взяти до уваги, що $0 \le b_0 < q$, то з визначень цілочислового ділення $(\div)$ і операції взяття залишку (%) випливає, що $(b_l q^{l-1} + ... + b_1) = u \div q$, $b_0 = u\%q$. Аналогічно знаходять b_1. Нехай $u_1 = u \div q$. Тоді $u_1 = (b_l q^{l-2} + ... + b_2)q + b_1 = (u_1 \div q)q + u_1\%q$ та $b_1 = u_1\%q$. Процес пошуку b_i буде продовжуватися, доки u_i не перетвориться на 0.

Таким чином, можна сформулювати правило для переведення цілих $(p)\to(q)$:

Щоб перевести довільне ціле системи з основою p у систему з основою q, необхідно знайти частку й залишок від ділення даного числа на основу q. Залишок є молодшою цифрою шуканого числа. Повторювати дію для нових часток, дописуючи ліворуч черговий залишок до слова з уже знайдених залишків, і припинити, коли остання частка стане 0.

Що можна сказати про довжину нового числа $n_q = b_l b_{l-1}...b_0$, тобто $l+1$? Позначимо m цифру $q-1$. Найменшим і найбільшим числами з $(l+1)$-розрядами у СЧ$_q$ є відповідно q^l і число $r = \underbrace{mm...m}_{l+1}$ зі значен-

ням $\bar{r} = \left(\sum_{i=0}^{l} mq^i \right)_{10} = m \left(\sum_{i=0}^{l} q^i \right)_{10}$. За формулою суми геометричної прогресії маємо $\bar{r} = \left(m \frac{q^{l+1} - 1}{q - 1} \right) = q^{l+1} - 1$. Тоді $q^l \le \bar{n}_q < q^{l+!}$. Після логарифмування цієї нерівності отримаємо $l \le \log_q \bar{n}_q < l + 1$, $l \le \lfloor \log_q \bar{n}_q \rfloor < l + 1$ та $l = \lfloor \log_q \bar{n}_q \rfloor$. Отже, кількість цифр у числі n_q дорівнює $\lfloor \log_q \bar{n}_q \rfloor + 1$.

Наведемо кілька прикладів переведень чисел з однієї системи в іншу.

Приклад 2.5. Переведемо числа 25_{10} та $A7_{16}$ у двійкову систему. Після переведення ці числа матимуть $\lfloor \log_2 25 \rfloor + 1 = 5$ та $\lfloor \log_2 167 \rfloor + 1 = 8$ розрядів. Відповідні процеси переведення відображено в стовпчиках таблиці. У лівому стовпчику розташовані частки (u_i), у правому – залишки-цифри (b_i). У результаті маємо: $25_{10} = 11001_2$ та $A7_{16} = 10100111_2$:

$A7_{16} = 167 (= u_0)$	$1 (= b_0)$
$83 (= u_1 = u / 2)$	$1 (= b_1)$
$41 (= u_2 = u_1 / 2)$	$1 (= b_2)$
$20 (= u_3 = u_2 / 2)$	$0 (= b_3)$
$10 (= u_4 = u_3 / 2)$	$0 (= b_4)$
$5 (= u_5 = u_4 / 2)$	$1 (= b_5)$
$2 (= u_6 = u_5 / 2)$	$0 (= b_6)$
$1 (= u_7 = u_6 / 2)$	$1 (= b_7)$
$0 (= u_8 = u_7 / 2)$	

$25_{10} = 25 (= u_0)$	$1 (= b_0)$
$12 (= u_1 = u / 2)$	$0 (= b_1)$
$6 (= u_2 = u_1 / 2)$	$0 (= b_2)$
$3 (= u_3 = u_2 / 2)$	$1 (= b_3)$
$1 (= u_4 = u_3 / 2)$	$1 (= b_4)$
$0 (= u_5 = u_4 / 2)$	

■

Розглянемо правило переведення дробових частин раціональних чисел. Нехай $y_p = .b_{-1}b_{-2}...b_{-l}$ – довільний дріб у системі СЧ $_p$ і необхідно знайти його еквівалент $.b_{-1}b_{-2}...b_{-l}...$ у системі СЧ $_q$. На жаль, точний еквівалент дробу в новій системі може мати зліченну кількість розрядів після крапки, тобто виявитися періодичним. Наприклад, $.1_3 = (.333...)_{10}$. Тому в загальному випадку може йтися лише про наближення до еквівалента числа. Тоді або обмежуються s першими розрядами еквівалента для певного s, або роблять те саме, але з округленням. Покладемо $u = \overline{y}_p$. Знову необхідно розв'язати рівняння відносно b_{-i}, але вже інше: $u = b_{-1}q^{-1} + ... + b_{-s}q^{-s} + ...$ (**). Нехай $\lfloor x \rfloor$ та $\{x\}$ означають цілу й дробову частини дійсного числа. Помножимо обидві частини (**) на q. Тоді $uq = b_{-1} + b_{-2}q^{-1} + ... + b_{-s}q^{-s+1} + ...$ та $b_{-1} = \lfloor uq \rfloor$, $b_{-2}q^{-1} + ... + b_{-l}q^{-l+1} + ... = \{uq\}$. Покладемо $u_1 = \{uq\}$. Аналогічно знаходимо $b_{-2} = \lfloor u_1 q \rfloor$ та $u_2 = \{u_1 q\}$ і т. д., доки u_i не набуде значення 0 або будуть отримані перші s розрядів. Таким чином, можна сформулювати правило для переведення правильних дробів $(p) \to (q)$:

Щоб перевести довільний дріб $.b_{-1}b_{-2}...b_{-l}$ системи з основою p у систему з основою q, необхідно помножити дане число на нову основу q і взяти цілу й дробову частини отриманого добутку. Ціла частина є старшим розрядом шуканого дробу. Повторювати процес для дробової частини добутку, додаючи праворуч кожного разу нову цілу частину до слова з раніше знайдених цілих частин, і припинити, коли чергова дробова частина стане 0 *або будуть отримані перші s розрядів.*

Приклад 2.6. Переведемо число $.25_{10}$ у двійкову та п'ятіркову системи, а число $.A7_{16}$ – у двійкову. У наближеннях треба взяти три розряди після коми. Спочатку знайдемо значення дробів: $.\overline{25} = 0.25$, $.\overline{A7}_{16} = 10 \times 16^{-1} + 7 \times 16^{-2} = 0.625 + 0.02734375 = 0.65234375$. Відповідні процеси переведення відображені у стовпчиках таблиці, де в лівих стовпчиках розташовані дробові частини відповідних добутків (u_i), а в правих – їхні цілі частини-цифри (b_i). У результаті маємо $.25_{10} \approx .111_5$, $.25_{10} = .01_2$ і $.A7_{16} = .01001111_2$.

$.A7_{16} = 0.65234375\ (= u)$	$(10) \to (2)$
$0.3046875\ (= u_1 = \{u \times 2\})$	$0\ (= b_1)$
$0.609375\ (= u_2 = \{u_1 \times 2\})$	$1\ (= b_2)$
$0.21875\ (= u_3) = \{u_2 \times 2\}$	$0\ (= b_3)$
$0.4375\ (= u_4) = \{u_3 \times 2\}$	$0\ (= b_4)$
$0.875\ (= u_5) = \{u_4 \times 2\}$	$1\ (= b_5)$
$0.75\ (= u_6) = \{u_5 \times 2\}$	$1\ (= b_6)$
$0.5\ (= u_7) = \{u_6 \times 2\}$	$1\ (= b_7)$
$0\ (= u_8) = \{u_7 \times 2\}$	$1\ (= b_8)$
$0.25 (= u)$	$(10) \to (5)$
$0.25\ (= u_1 = \{u \times 5\})$	$1\ (= b_1)$
$0.25\ (= u_2 = \{u_1 \times 5\})$	$1\ (= b_2)$
$0.25\ (= u_2 = \{u_1 \times 5\})$	$1\ (= b_3)$
…	…
$0.25\ (= u)$	$(10) \to (2)$
$0.5\ (= u_1 = \{u \times 2\})$	$0\ (= b_1)$
$0\ (= u_2) = \{u_1 \times 2\}$	$1\ (= b_2)$

■

Переведення чисел з основою $p = 2^m, m > 0$. Для таких чисел існують спрощені правила переведень. Розглянемо їх на прикладі цілих переведень $(2^4) \to (2)$. Нехай дано $n_{16} = a_k a_{k-1} \ldots a_0$ – довільне шістнадцяткове ціле й необхідно його перевести в еквівалентне двійкове $n_2 = b_l b_{l-1} \ldots b_0$. Маємо рівняння $\sum\limits_{i=0}^{k} a_i 16^i = \sum\limits_{j=0}^{l} b_j 2^j$ відносно b_j.

Нехай $a_i = \left(b_{i3} b_{i2} b_{i1} b_{i0}\right)_2$, $i \in 1 \ldots k$. Ліву частину можна перетворити таким чином:

$$\sum_{i=0}^{k} a_i 16^i = \sum_{i=0}^{k} \left(b_{i3} 2^3 + b_{i2} 2^2 + b_{i1} 2^1 + b_{i0} 2^0\right) 2^{4i} =$$

$$= \sum_{i=0}^{k} \left(b_{i3} 2^{4i+3} + b_{i2} 2^{4i+2} + b_{i1} 2^{4i+1} + b_{i0} 2^{4i}\right) = \sum_{i=0}^{4k+3} b_{i \div 4i \% 4} 2^i.$$

Останній член у переведенні є значенням двійкового числа $\left(\underbrace{b_{k3}b_{k2}b_{k1}b_{k0}}\ldots\underbrace{b_{13}b_{12}b_{11}b_{10}}\ \underbrace{b_{03}b_{02}b_{01}b_{00}}\right)_2$, отже, саме воно є шуканим еквівалентом, тобто n_2. Сформулюємо правило для переведення $(16)\to(2)$:

Щоб перевести довільне шістнадцяткове число у двійкове, достатньо знайти двійкові чотирилітерні подання всіх його цифр і записати їх поспіль у тому самому порядку, розпочинаючи з подання старшої цифри.

Приклад 2.7. Перевести за спрощеним правилом шістнадцяткові числа у двійкові: а) $A7_{16}$ (див. прикл. 2.6); б) $ABCDEF_{16}$. Маємо: а) $\underbrace{1010}_{A}\underbrace{0111}_{7}{}_2$; б) $\underbrace{1010}_{A}\underbrace{1011}_{B}\underbrace{1100}_{C}\underbrace{1101}_{D}\underbrace{1110}_{E}\underbrace{1111}_{F}{}_2$ ■

Розглянемо тепер обернене переведення $(2)\to(16)$. Нехай дано $n_2 = a_k a_{k-1}\ldots a_0$ – довільне ціле й необхідно його перевести в еквівалентне шістнадцяткове $n_{16} = b_l b_{l-1}\ldots b_0$. Маємо рівняння $\sum\limits_{i=0}^{k} a_i 2^i = \sum\limits_{j=0}^{l} b_j 16^j$ відносно b_i, ліву частину якого, розпочинаючи з молодших розрядів, розіб'ємо на сусідні четвірки (якщо $k+1$ не кратне 4, то остання група цифр при цьому буде не четвіркою) і перетворимо:

$$\sum_{i=0}^{k} a_i 2^i = \sum_{j=0}^{k\div 4-1}\sum_{m=0}^{3} a_{4\times j+m}2^{4\times j+m} + \sum_{i=k\div 4\times 4}^{k} a_i 2^i =$$

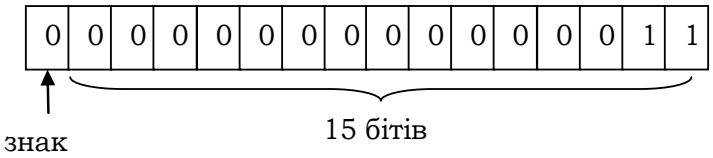
$$= \sum_{j=0}^{k\div 4-1}(16^j \sum_{m=0}^{3} a_{4\times j+m}2^m) + 16^{k\div 4}\sum_{i=k\div 4\times 4}^{k} a_i 2^{i\%4} = \sum_{j=0}^{k\div 4-1} b_j 16^j + b_{k\div 4}16^{k\div 4},$$

де $b_j = \sum\limits_{m=0}^{3} a_{4\times j+m}2^m$, $0\le j\le k$, $b_{k+4} = \sum\limits_{i=k\div 4\times 4}^{k} a_i 2^i$ – шістнадцяткові цифри. Останній член у переведенні є значенням шістнадцяткового числа $(b_l b_{l-1}\ldots b_0)_{16}$, і воно є шуканим еквівалентом, тобто n_2. Сформулюємо правило для переведення $(16)\to(2)$:

Щоб перевести довільне двійкове число в шістнадцяткове, достатньо розбити його на групи із чотирьох сусідніх розрядів, розпочинаючи з молодших, знайти їхні шістнадцяткові коди й записати поспіль у тому самому порядку.

Приклад 2.8. Перевести за спрощеним правилом двійкове число 11110011100110_2 у шістнадцяткове. Останньою групою цифр буде 0011. Маємо: 3CE6 ■

Цілі як машинний тип даних. У машинних типах даних використовується двійкова система. Стандартний розмір цілих – це машинне слово (зазвичай 2 або 4 байти). Біти (розряди) у такому слові нумеруються справа наліво, розпочинаючи з нуля. Знаковим є 15-й (31-й) біт. Нуль у ньому означає додатне, а одиниця – від'ємне число. Таке подання цілих називається *прямим кодом*. Так виглядає прямий код числа 3 у двобайтному форматі:



У прямому коді в пам'яті зберігаються цілі, а також у цьому форматі виконуються машинні множення й ділення чисел. На жаль, з адитивними операціями в прямому коді виникають певні проблеми. Для таких операцій зручнішим є додатковий код, який для додатних чисел збігається з прямим. Спочатку сформулюємо, що таке *обернений код* цілого від'ємного числа. Щоб отримати такий код, необхідно в прямому коді інвертувати всі розряди окрім знакового. Розглянемо число -256. Його прямий код становить 1000 0001 0000 0000, обернений – 1111 1110 1111 1111. Додатковий код додатного числа збігається також із прямим, а код від'ємного утворюється з оберненого шляхом додавання до нього 1. Наприклад, обернений код числа -256 збігається із сумою $1111\ 1110\ 1111\ 1111 + 1 = 1111\ 1111\ 0000\ 0000$. Зазначимо, що число 0 має два прямі коди, але тільки один обернений, а саме: обернений код від'ємного нуля $1000\ 0000\ 0000\ 0000 \Rightarrow 1111\ 1111\ 1111\ 1111 + 1 = 0000\ 0000\ 0000\ 0000$ збігається з оберненим кодом додатного нуля 0000 0000 0000 0000. Найменшим від'ємним числом є -2^{15} з оберненим кодом 1000 0000 0000 0000, а найбільшим додатним – число $2^{15} - 1$. Таким чином, існує певна несиметричність цілих відносно 0.

Перевагою додаткового коду є те, що в ньому однаково реалізуються операції додавання цілих різних знаків (алгебричне додавання), а операція віднімання зводиться до додавання заміною знака другого операнда на протилежний. При цьому знаковий розряд теж бере участь у операції. Щоб повернутися від додаткового коду до прямого,

можна повністю повторити дії, які виконувалися при отримуванні додаткового коду: спочатку всі розряди, окрім знакового, інвертувати, а потім додати 1.

Приклад 2.9. Додати числа: а) 12 та –5; б) 32767 та 1024.

а) Побудуємо обернені коди операндів: 0000 0000 0000 1100 та 1111 1111 1111 1010 + 1 = 1111 1111 1111 1011. Тепер додамо їх:

0000 0000 0000 1100
1111 1111 1111 1011
0000 0000 0000 0111.

Перенесення зі знакового розряду ігнорується. Оскільки отриманий обернений код додатний, то він є і прямим кодом суми $12 + (-5) = 7 = 111_2$.

б) Побудуємо обернені коди операндів:
32767 = 0111 1111 1111 1111 та
1024 = 0000 0100 0000 0000. Додамо їх:

0111 1111 1111 1111
0000 0100 0000 0000
1000 0011 1111 1111.

Перенесення в знаковий розряд змінило число на від'ємне. Ситуація з перенесенням у знаковий розряд називається *переповненням* і свідчить про некоректність результату операції ■

Беззнакові машинні цілі – це машинні цілі, в яких знаковий розряд розглядається як цифровий. Їхній діапазон збігається з інтервалом $[0..2^{16}-1]$.

Дійсні числа як машинний тип даних. Машинні дійсні числа подаються як РТ-числа. Для кожного дійсного числа r існує багато варіантів РТ-чисел, що збігаються з ним за значенням. Дійсно, якщо мантису поділити на основу, а порядок збільшити на 1, то значення РТ-числа не зміниться. РТ-число $me \pm n$ з основою p *нормалізоване*, якщо $1/p \le \bar{m} < 1$. Наприклад, десяткове число .02e+1 ненормалізоване (.02 < .1), а .2e+0 – нормалізоване.

Дійсні числа подаються у вигляді *двійкових нормалізованих* РТ-чисел. Їхня мантиса має вигляд .1.... Поле пам'яті для подання машинних РТ-чисел має таку структуру:

знак	m - мантиса	p - порядок

Розмір усіх складових поля фіксований. Тому існує скінченна кількість машинних РТ-чисел, яка визначається кількістю комбінацій-трійок $(\pm, m, p)$. Оскільки місцезнаходження точки в нормалізованій

мантисі фіксоване, то для подання останньої достатньо подати тільки її цифри, старша з яких завжди (!) є 1[18]. Порядок зазвичай подається беззнаковим цілим і фактично обчислюється як різниця $p-2^{l-1}$, де $l-$ довжина поля порядку (у бітах). Нехай $k-$ розмір мантиси і $q=2^{l-1}$. Тоді всі додатні РТ-числа належать діапазону

$$\left[0.\underbrace{00..01}_{k}\times 2^{-q}..0.\underbrace{11...1}_{k}\times 2^{q}\right],$$

а від'ємні – діапазону

$$\left[-0.\underbrace{11...1}_{k}\times 2^{q}..-0.\underbrace{00..01}_{k}\times 2^{-q}\right].$$

Поза межами цих інтервалів жодне дійсне число не має представників серед машинних РТ-чисел.

Тепер поговоримо про щільність розташування РТ-чисел на дійсній числовій осі. Оскільки всіх РТ-чисел скінченна кількість, то між сусідніми є певна ненульова відстань, розмір якої характеризує *щільність* розташування машинних чисел на числовій осі. Щільність тим більше, чим менше вказана відстань. Якщо зафіксувати порядок і порівняти щільність чисел на інтервалах [0.1 .. 1) (ці числа пробігають значення $m\times 2^0$, $.1\le m<1$) та [1 .. 2) (ці числа пробігають відповідно значення $m\times 2^1$, $.1\le m<1$), то щільність на другому інтервалі буде менша. Це випливає з того, що сукупність мантис чисел, які належать цим інтервалам, одна й та сама, але другий інтервал удвічі більший від першого. Це означає, що похибка обчислень для великих чисел експоненціально зростає – навіть великі цілі числа будуть подаватися неточно. Певну уяву про щільність розташування машинних РТ-чисел дає константа $\max\{x:1.0\oplus x=1.0\}$, яка називається *машинним нулем* в околі 1.0. Тут операція $\oplus$ є машинним додаванням. Якщо до 1 додавати числа менші ніж машинний нуль, то на сумі це не позначиться.

Розглянемо кілька прикладів. Для зручності обмежимося десятковою системою. Будемо вважати, що мантиса містить 4, а порядок – 2 десяткові цифри й задається прямим кодом зі знаком.

Приклад 2.10. а) Візьмемо число $10{,}35=.1035\times 10^2=.1035e+2$. Найближчим праворуч від нього машинним числом буде $10{,}36=.1036\times 10^2=.1036e+2$, а щільність між ними становитиме 0.01.

[18] Деякі процесори враховують цю обставину і старшу одиницю явно не зберігають у полі мантиси, що дозволяє додати до неї додатковий розряд.

б) Розглянемо число $0{,}001035 = .1035 \times 10^{-2} = .1035e-2$. Найближче праворуч від нього машинне число – це $.1036 \times 10^{-2} = .1036e-2$, а щільність між ними суттєво більша і становить уже 0.000001.

в) Розглянемо РТ-число $a = .1035 \times 10^{12} = 1035e+12$. Найближче ціле праворуч від нього число $b = .1036 \times 10^{12} = .1036e+12$. Щільність між ними становить

$$d = 1036\ 0000\ 0000 - 1035\ 0000\ 0000 = 1\ 0000\ 0000 = 10^8.$$

Отже, у проміжку між числами a та b, довжина якого становить 10^8, немає жодного машинного РТ-числа! ■

Машинна арифметика. Операції машинної арифметики цілих виконуються за правилами, що діють у звичайній двійковій арифметиці. Проте обмеженість діапазону машинних цілих може впливати на результат операцій (див. *переповнення,* підрозд. 2.3.3). Що стосується дійсної машинної арифметики, то тут ситуація набагато складніша. Розглянемо, як виконуються основні арифметичні дії над РТ-числами.

Якщо це адитивні операції $+$, $-$, то їм передує *вирівнювання порядків* у операндах, тобто операнд із меншим порядком зводиться до більшого. Нехай $a = m_1 e + p$, $b = m_2 e + q$ та $p > q$. Тоді після вирівнювання

$a = m_1 e + p$, $b = m_2' e + p$, де $m_2' = m_2 \times 10^{-(p-q)}$ та $a \pm b = (m_1 \pm m_2')e + p$.

Приклад 2.11. Обчислити суму $a + b$.

1) $a = .123555\text{E}+5$, $b = .155\text{E}+6$. Після вирівнювання порядків $a = .0123555\text{E}+6$, $b = .155\text{E}+6$ і $a + b = .1673555\text{E}+6$.

2) Нехай у мантисі 4 десяткові цифри. $a = .1222\text{E}+3$, $b = .15\text{E}+6$. Після вирівнювання порядків $a = .0001e+6$, $b = .15e+6$. $a + b = .1501\text{E}+6$. У сумі *втрачено* величину $.0000222\text{E}+6 = 22.2$ ■

Як бачимо, якщо порядки операндів суттєво різні, то в процесі їх вирівнювання можуть втрачатися розряди меншого з операндів, що впливає на точність результату операції.

Правила для обчислення мультиплікативних операцій *, / звичайні. При множенні порядки додаються, а мантиси перемножуються. При діленні мантиси діляться, а порядки віднімаються. При цьому у вирівнюванні порядків немає потреби. Нехай $a = m_1 e + p$, $b = m_2 e + q$. Тоді

$$a * b = m_1 * m_2 e + (p + q);$$
$$a / b = m_1 / m_2 E + (p - q).$$

Однак через обмеження діапазону порядків можлива ситуація їхнього *втрачання*. Тому ділення на маленькі числа і множення на великі може призвести до втрати порядку результату.

Приклад 2.12 (Н. Вірт). Нехай у мантисі ті самі чотири цифри. Розв'язати рівняння $x^2 - 200x + 1 = 0$.

Метод 1. Знайдемо дискримінант квадратного тричлена: $D = \text{sqrt}((-200)^2 - 4) = \text{sqrt}(0.4e + 5 - 0.00004e + 5) = 200.0$. За відомою формулою для коренів квадратного рівняння маємо

$$x_1 = 200.0, x_2 = 0.000 .$$

Метод 2. Залишимо тепер більший корінь (x_1), а другий знайдемо за теоремою Вієта: $x_2 = c/(a \times x_1)$. Дістанемо $x_2 = 1.000/(200.0 \times 1.000) = 0.005$.

Таким чином, за першим методом ми отримали значення x_2, далеке від кореня. Після підстановки його в рівняння маємо $1 \neq 0$. Другий метод дає значно кращий результат:

$$0.005 * 0.005 - 200 * 0.005 + 1 = 0.000025 - 1 + 1 = 0.000025 \neq 0 \ \blacksquare$$

Літери як машинний тип даних. Цей тип даних призначений для обміну інформацією між суб'єктами інформаційних систем. Інформацією можуть бути запити (вхідні дані й програми), результати роботи програми, команди командного процесора тощо. Такий характер інформації вимагає дуалізму в її поданні. Розрізняють зовнішню та внутрішню її форми. У зовнішньому вигляді вона існує як інформація, подана за допомогою спеціальних ідеограм – літер, у внутрішньому – кодується двійковими числами. Кожний комп'ютер має *кодову таблицю*, яка містить сукупність усіх можливих літер та їхніх двійкових кодів. Широко відомою такою таблицею є ASCII (American Standard Cod for Information Interchange) та її національні варіанти (див. підрозд. 2.4.2). Іншим прикладом кодової таблиці є кодова таблиця Навчальної Машини (див. підрозд. 2.3.4). Серед усіх літер, зображених у ній, виділяють літери *керування.* Вони призначені для керування зовнішніми пристроями машини. У таблиці ASCII ці символи розміщено на початку (31 літера). Наприклад, літера BEL із кодом 7 запускає звукогенератор, літера LF із кодом 10 здійснює перехід на новий рядок тощо. На відміну від решти літер, літери керування не мають своїх ідеограм. Їх подають або за допомогою кодів, або використовують певні комбінації інших літер (керівні або ESC-послідовності, див. підрозд. 3.1.2).

2.3.4. НАВЧАЛЬНА МАШИНА

Щоб ознайомитись детальніше з командами та програмами комп'ютера, розглянемо Навчальну Машину (НМ) – спрощений варіант першої версії машини MIX Д. Кнута без дійсної арифметики й деяких пристроїв введення-виведення. (Друга версія машини називається MMIX). Обидві належать до абстрактних машин і є спрощеними моделями реальних ЕОМ. Нині існують програмні емулятори машин MMIX і MIX на всіх відомих ЕОМ, тому всі MMIX- і MIX-програми можуть бути проінтерпретовані й виконані на них.

Структура оперативної пам'яті навчальної машини. Основною одиницею інформації НМ є байт. Розмір байта, тобто кількість інформації, яку він містить, чітко не визначається, але байт має набувати як мінімум 64 різних значень. Таким чином, довільне число від 0 до 63 включно може бути подано в одному байті. Крім того, байт може зобразити не більше 100 різних значень, тобто один байт НМ відповідає 6 двійковим розрядам (бітам), у десятковій системі – двом десятковим цифрам (у повній версії MMIX 1 байт містить 64 біти!). У двох суміжних байтах можна зобразити числа від 0 до 4095($=2^{12}$-1), у трьох – від 0 до 262143, у чотирьох – від 0 до 16777215, у п'яти – від 0 до 1073741823.

Машинне слово складається з п'яти байтів і знака (+, -):

0	1	2	3	4	5
±					

ОП складається із 4000 слів:

0000						
0001						
0002						
....						
3998						
3999						

Кожне слово має свою *адресу* – порядковий номер у пам'яті.

Полем пам'яті називається неперервна послідовність слів. *Поле слова* – це послідовність байтів усередині слова. Поле слова визначається лівим і правим номерами байта (L:R):

(0:0) – тільки знак;

(0:2) – знак і два перші байти;
(0:5) – усе слово;
(4:5) – два молодші байти.

Поле (L:R) будемо кодувати числом $N = 8L + R$. Наприклад, $N = 0:0) = 0 \cdot L + 0 = 0$, $N = (0:2) = 0 \cdot L + 2 = 2$ тощо.

Центральний процесор. ЦП НМ має два загальні регістри у форматі слова (позначаються $A та $X) і вісім спеціальних двобайтних (позначаються **R**1–**R**6, **J** та **PC**).

Регістр $A

±	A1	A2	A3	A4	A5

називається *сумatором* і використовується, у першу чергу, для арифметичних операцій із даними.

Регістр $X

±	X1	X2	X3	X4	X5

використовується як розширення $A.

Регістри **R**1–**R**6 називаються *індексними* й використовуються як лічильники, а також для формування адреси:

R1:

±	R14	R15

…

R6:

±	R64	R65

R14 та R15 означають лівий і правий байти в регістрі **R**1.

Регістр **J**

±	J4	J5

зберігає адресу команди, яка розташована за останньою виконаною командою переходу (JUMP) і застосовується передусім для виклику підпрограм. Він змінюється, коли виконується будь-яка команда переходу, і не змінюється у протилежному випадку.

Регістр **PC**:

+-	PC4	PC5

PC – лічильник команд, зберігає адресу наступної команди.

Крім цих регістрів, у НМ є:

1) однобітовий тригер переповнення **T**, який може бути в одному з двох станів: "увімкнено" або "вимкнено";

2) індикатор порівняння **B**, який має три стани: "менше", "дорівнює", "більше":

L	G	E

3) пристрої введення-виведення.

Алгоритм функціонування процесора НМ такий, як і в будь-якого комп'ютера (рис. 2.4).

Команди. Програма в НМ – це послідовність команд, записана в певному полі пам'яті. Усі команди мають формат слова:

0	1	2	3	4	5
±	A	A	I	F	C

Байт C – *код* операції, що вказує на операцію, яку необхідно виконати. Наприклад, C = 8 задає операцію LDA (англ. Load the A register – завантажити регістр A). F-байт використовується для модифікації, уточнення дії операції. Зазвичай F задає специфікацію поля (L:R) = 8L + R. Наприклад, якщо C = 8, F = 11, то виконується операція *завантажити в регістр* $A *поле* (1:3). Поле команди ±AA називається *адресою операнда* (знак є частиною адреси). Поле I називається *індексом*. Індекс – це номер індексного регістра, який використовується для модифікації адреси операнда. Якщо I = 0, то адреса ±AA використовується прямо, у протилежному випадку в полі I міститься число *i* від 1 до 6, і тоді вміст індексного регістра **R***i* алгебрично додається до значення числа ±AA. Такий процес індексування виконується для кожної команди. Будемо використовувати **M** для позначення адреси, отриманої після індексування. Якщо в результаті формування значення **M** результат виходить за межі двох байтів (> 4095), то він вважається невизначеним. Для більшості команд **M** використовується як адреса комірки пам'яті. Ми припускаємо, що є 4000 комірок, пронумерованих від 0 до 3999, тому адресу будь-якої комірки можна задати у двох байтах. Для кожної команди, в якій **M** є посиланням на комірку пам'яті, має виконуватись умова 0≤M≤3999. У цьому випадку запис ***M** буде позначати значення, яке міститься в комірці за адресою **M**.

Далі при описі команд будемо користуватись їхнім більш наочним мнемонічним поданням:

OP ADDRESS, I(F),

де **OP** – мнемонічне ім'я команди, що замінює код операції, **ADDRESS** – це ± AA, **I** та **F** подають значення полів I та F. Якщо I = 0, то воно випускається. Якщо F є стандартною F-специфікацією для даної команди, то його також можна не писати. Майже для всіх команд стандартною F-специфікацією є (0:5), тобто специфікація повного слова. Якщо стандартною є інша специфікація, то вона буде зазначена при обговоренні відповідної команди. Мнемонічне подання команди дозволяє однозначно відновити команду як конкретне слово в пам'яті. Наприклад, команда з кодом операції 8 – завантажити число в суматор – має мнемонічне ім'я LDA (див. вище).

Символьне подання	Фактична команда				
	+/-	A	I	F	C
LDA 2000, 2(0:3)	+	2000	2	3	8
LDA 2000, 2(1:3)	+	2000	2	11	8
LDA 2000 (1:3)	+	2000	0	11	8
LDA 2000	+	2000	0	5	8
LDA -2000, 4	-	2000	4	5	8

Команда LDA 2000, 2(0:3) завантажує в регістр $A інформацію з поля (0:3) комірки з адресою 2000, проіндексованою за регістром **R**2.

Команди завантаження регістрів:

- **LDA** (англ. load A – *завантажити* A), C = 8, F = поле.

Значення ***M** записується в регістр $A. В операціях, де операндом є не все слово, а тільки його частина, ураховують знак, якщо він є частиною поля; у протилежному випадку знаком вважається +. При завантаженні регістра значення розташовується в його правій частині.

Якщо у F задана стандартна специфікація поля (0:5), то в регістр завантажується абсолютна величина ***M** зі знаком +. Якщо в полі **M** є команда, то завантажується її поле ±AA та в регістрі $A буде

±	0	0	0	A	A

Приклад 2.13. Припустимо, що в комірці 2000 міститься таке слово:

-	80	3	5	4

Тоді при завантаженні різних його частин отримуємо такі результати:

Команда	Вміст $A після операції					
LDA 2000	-	80		3	5	4
LDA 2000 (1:5)	+	80		3	5	4
LDA 2000 (3:5)	-	0	0	80		3
LDA 2000 (0:3)	+	0	0	3	5	4
LDA 2000 (4:4)	+	0	0	0	0	5
LDA 2000 (0:0)	-	0	0	0	0	0
LDA 2000 (1:1)	+	0	0	0	0	7

(В останньому прикладі результат визначено не повністю, оскільки розмір байта змінний.) ■

- **LDX** (англ. load $X – завантажити X), C = 15, F = поле.

Ця операція виконується аналогічно LDA, тільки замість $A завантаження виконується в $X.

- **LD**i (англ. load **R**i – завантажити **R**i), 1≤i≤6.

C = 8 + i, F = поле.

Така сама операція, що і LDA, але замість $A завантаження виконується в **R**i. Індексний регістр містить тільки два байти (а не п'ять) і знак; байти 1–3 вважаються завжди рівними 0. Дія команди LDi вважається невизначеною, якщо в ній робиться спроба занести в байти 1–3 значення, відмінне від 0.

• **LDAN** (англ. load A negative – *завантажити в* $A *з протилежним знаком*), C = 16, F = поле.

• **LDXN** (C = 23, F = поле), **LDiN** (C = 16 + i, F = поле, 1≤i≤6)

Команди LDAN, LDXN і LDiN аналогічні відповідно командам LDA, LDX, LDi, але при завантаженні знак змінюється на протилежний.

Команди зберігання регістрів:

• **STA** (англ. store A – *запам'ятати* $A), C = 24, F = поле.

Вміст регістра A копіюється в комірку з адресою **M** на місце поля, указаного у F. Інші поля в комірці не змінюються. Із правої частини регістра обирається потрібна кількість байтів, потім вони зсуваються ліворуч, якщо це необхідно, і заносяться у відповідне поле комірки **M**. Якщо знак не є частиною поля, то він не змінюється. Вміст регістра також не змінюється.

Приклад 2.14. Припустимо, що комірка 2000 містить

-	1	2	3	4	5

У регістрі $**A** міститься

+	6	7	8	9	0

Тоді маємо:

Команда	**Вміст $A після операції**					
STA 2000	+	6	7	8	9	0
STA 2000 (1:5)	-	6	7	8	9	0
STA 2000 (5:5)	-	1	2	3	4	0
STA 2000 (2:2)	-	1	0	3	4	5
STA 2000 (2:3)	-	1	9	0	4	5
STA 2000 (0:1)	+	0	2	3	4	5

■

• **STX** (англ. store $X – запам'ятати $X), C = 31, F = поле, **ST**i (англ. store **R**i – запам'ятати **R**i), C = 24 + i, F = поле, 1≤i≤6.

Операції такі самі, як і STA, тільки запам'ятовується не $A, а $X та **R**i.

• **STJ** (англ. store **J** – запам'ятати J), C = 32, F = поле.

Така сама операція, що і **ST**i, але запам'ятовується значення **J** і знак завжди +. У команді **STJ** стандартною специфікацією для поля

F є (0:2), а не (0:5), і це зрозуміло, оскільки значення **J** майже завжди записується в адресне поле команди.

• **STZ** (англ. store zero – *обнулити поле*), C = 33, F = поле.

Така сама операція, що і STA, але в пам'ять записується нуль зі знаком + . Іншими словами, до вказаного поля комірки заноситься нуль.

Цілі арифметичні операції. В операціях додавання, віднімання, множення й ділення може мати місце специфікація поля. В арифметичних командах стандартною специфікацією поля є (0:5). Символом **V** будемо позначати специфіковане в конкретній команді поле слова з адресою **M**, а ***V** – його значення.

• **ADD** (англ. add – *додавати*), C = 1, F = поле.

Значення ***V** додається до $A. Якщо абсолютне значення результату виходить за межі $A, то у тригер переповнення заноситься 1, а в $A залишається залишок результату без старшої 1 (ніби 1 перенесення пішла в інший регістр, який розташований ліворуч від A). У протилежному випадку стан тригера переповнення не змінюється. Якщо результат дорівнює 0, то знак $A не змінюється.

Приклад 2.15. У результаті виконання наведеної нижче послідовності команд буде обчислена сума п'яти байтів регістра A:

```
STA 2000
LDA 2000(5:5)
ADD 2000(4:4)
ADD 2000(3:3)
ADD 2000(2:2)
ADD 2000(1:1) ■
```

• **SUB** (англ. subtract – *віднімати*), C = 2, F = поле.

Величина ***V** віднімається від $A. Як і в команді **ADD**, може виникнути переповнення. Треба зазначити, що через невизначений розмір байта одна й та сама операція може викликати переповнення для однієї машини й не викликати для іншої.

• **MUL** (англ. multiply – *множити*), C = 3, F = поле.

Обчислюється добуток ***V** × $A, зображений числом у 10 байтів, що розміщується в регістрах $A (старші розряди) і $X (молодші розряди). Знаки $A та $X установлюються рівними алгебричному знаку результату.

• **DIV** (англ. divide – *ділити*), C = 4, F = поле.

Значення $A та $X розглядаються разом як одне десятибайтне число зі знаком, рівним знаку $A, що ділиться на значення ***V**. Якщо ***V** = 0 або частка за абсолютним значенням більше 5 байтів (це еквівалентно умові |$A|≥|***V**|), то в регістрах $A та $X буде невизначена інформація і тригер переповнення встановиться в 1. У протилежному ви-

падку частка записується в $A, а залишок – у $X. Після виконання операції знак $A дорівнює знаку частки, а знак $X – знаку діленого, тобто знаку $A до виконання операції.

Команди пересилання адрес. У наступних командах **M** використовується як число зі знаком, а не як адреса комірки пам'яті.

• **ENTA** (англ. enter A – *занести в* A), C = 48, F = 2.

Величина **M** заноситься в $A. Якщо **M** = 0, то завантажується знак команди.

Наприклад, команда **ENTA** 0 записує в $A нуль; **ENTA** 0,1 установлює $A рівним поточному вмісту індексного регістра **R**1.

• **ENTX** (англ. enter X – занести в X), C = 55, F = 2, **ENTi** (англ. enter **R**i – занести в **R**i), 1≤i≤6, C = 48 + i, F = 3.

Як і в команді **ENTA**, виконується завантаження відповідного регістра.

ENNA (англ. enter negative $A – *занести в* $A *з протилежним знаком*), C = 48, F = 3, **ENNX** (англ. enter negative $X, C = 48, F = 3), **ENNi** (англ. enter negative **R**i), 1≤i≤6, C = 48 + i, F = 3.

Виконуються так само, як і команди **ENTA**, **ENTX**, **ENT**i. Відмінність полягає в тому, що при завантаженні знак змінюється на протилежний.

Наприклад, команда **ENN3** 0,3 змінює знак **R**3 на протилежний.

• **INCA** (англ. increase $A – *збільшити* $A), C = 48, F = 0.

Величина **M** додається до $A. Дія команди INCA еквівалентна додаванню до $A слова за командою **ADD**, яке містить величину **M**. Переповнення, що може виникнути при виконанні цієї команди, обробляється так само, як і в команді **ADD**.

Наприклад, команда **INCA** 1 збільшує значення $A на 1.

INCX (англ. increase $X), C = 55, F = 0, **INCi** (англ. increase **R**i), 1≤i≤6, C = 48 + i, F = 0.

Виконуються так само, як і команда **INCA**. Переповнення **R**i не дозволяється. Якщо отриманий результат займає більше двох байтів, то результат операції в цьому випадку невизначений.

• **DECA** (англ. decrease $A – *зменшити* $A), C = 48, F = 1.

Виконується як і команда **INCA**, тільки величина **M** віднімається від $A. Дія команди **DECA** еквівалентна відніманню від $A слова за командою **SUB**, яке містить величину **M**. Переповнення, що може виникнути при виконанні цієї команди, обробляється так само, як і в команді **SUB**.

• **DECX** (англ. decrease $X), C = 55, F = 1, **DECi** (англ. decrease **R**i), 1≤i≤6, C = 48 + i, F = 1.

Виконуються як і команда **DECA**, тільки над іншими регістрами.

Зазначимо, що коди операцій у командах **ENTA**, **ENNA**, **INCA**, **DECA** однакові. Щоб відрізнити ці операції, використовується поле F.

Команди порівнянь. Усі команди порівняння порівнюють значення, що містяться в регістрі, зі значенням, яке міститься в пам'яті. У результаті індикатор порівняння **B** переходить у стан L (менше), E (дорівнює) або G (більше) залежно від того, чи буде значення регістра меншим, рівним або більшим, ніж значення в комірці пам'яті. Нуль зі знаком мінус дорівнює нулю зі знаком плюс.

• **CMPA** (англ. compare $A – *порівняти* $A), C = 56, F = поле.

Значення вказаного поля регістра A порівнюється з тим самим полем у слові **M**. Якщо знакова позиція не входить у поле, то обидва поля розглядаються як додатні; у протилежному випадку при порівнянні до уваги береться знак. (Якщо F задано як (0:0), то завжди буде зафіксована рівність, оскільки нуль зі знаком плюс дорівнює нулю зі знаком мінус.)

• **CMPX** (англ. compare $X), C = 63, F = поле, **CMPI** (англ. compare **R**i), C = 56 + i, F = поле, 1≤i≤6.

Операції аналогічні **CMPA**. При порівнянні в індексних регістрах байти 1–3 вважаються нульовими.

Команди переходу. Зазвичай команди виконуються в послідовному порядку, тобто після виконання чергової команди лічильник команд **PC** автоматично збільшується на 1. Цю послідовність можуть порушувати команди переходу. Існує більше 20 варіантів таких команд. Ми розглянемо лише кілька.

JMP (англ. jump – *перестрибнути*), C = 39, F = 0.

Безумовний перехід – у лічильник **PC** записується **M** – адреса наступної команди. Паралельно (і завжди, коли має місце той чи інший перехід, за винятком команди **JSJ**) у регістр **J** автоматично заноситься адреса наступної команди (тієї, яка б виконувалась за відсутності переходу).

• **JSJ** (англ. jump – *перестрибнути, не змінюючи* **J**), C = 39, F = 1.

Виконується аналогічно **JMP**, але вміст **J** не змінюється.

• **JOV** (англ. jump on overflow – *перестрибнути при переповненні*), C = 39, F = 2.

Якщо тригер переповнення встановлено в 1, то він змінюється на 0 і виконується безумовний перехід **JMP**. У протилежному випадку нічого не відбувається.

• **JANZ** (англ. jump on nonzero $A – *перестрибнути при ненулевому* $A), C = 40, F = 4.

Інші команди:

• **MOVE** (англ. – *переслати*), C = 7, F = число.

Група слів за адресою **M** (кількість їх задана в полі F) пересилається за адресою, що міститься в їхньому індексному регістрі **R**1. Пересилання виконується послідовно по одному слову з одночасним збільшенням на 1 значення регістра **R**1. Таким чином, після виконання операції **R**1 збільшиться на величину F. Якщо F = 0, то нічого не виконується. Особливу увагу треба приділити випадку, коли має місце накладання груп комірок, з якими виконується операція.

Приклад 2.16. Нехай F = 3 та M = 1000. Якщо *R1 = 999, то пересилається значення *1000 за адресою 999, значення *1001 – за адресою 1000, значення *1002 – за адресою 1001, і це природно. Однак якщо б *R1 = 1001, то значення *1000 пересилалося б за адресою1001, потім значення *1001 – за адресою 1002 та *1002 – за адресою 1003, тобто одне й те саме слово (*1000) пересилалося б у три місця ■

• **SLA, SRA, SLAX, SRAX, SLC, SRC** (англ. shift left A, shift right A, shift left AX, shift right AX, shift left AX circularly, shift right AX circularly – *зсунути* A *ліворуч*, *зсунути* A *праворуч, зсунути* AX *ліворуч, зсунути* AX *праворуч, зсунути* AX *ліворуч циклічно*, *зсунути* AX *праворуч циклічно*), C = 6, F = 0, 1, 2, 3, 4, 5, відповідно.

Це команди зсуву. Вони жодним чином не впливають на знаки регістрів $A та $X. Величина **M** указує кількість байтів, на яку зсувається інформація ліворуч або праворуч, і повинна бути невід'ємною. Команди **SLA** та **SRA** не впливають на $X. Усі інші операції зсуву впливають на обидва регістри так, ніби ці регістри є одним десятибайтним регістром. При операціях **SLA**, **SRA**, **SLAX** та **SRAX** у регістр з одного кінця заносяться нулі, а виштовхані з іншого кінця байти зникають. Команди **SLC** та **SRC** викликають циклічний зсув, при якому байти, що залишають один кінець, заходять з іншого. У циклічному зсуві беруть участь обидва регістри, $A та $X.

NOP (англ. no operation – *немає операцій*), C = 0.

Команда не виконує жодної дії. Величини F та M ігноруються.

HLT (англ. halt – *зупинитися*), C = 5, F = 2.

Машина зупиняється. Після того, як оператор знову запустить машину, дія цієї команди буде еквівалентною NOP.

Команди керування пристроями введення-виведення. НМ має два пристрої введення-виведення: пристрій читання перфокарт і екран. Кожному пристрою поставлено у відповідність номер:

Номер пристрою	Тип пристрою	Розмір блока
16	Пристрій читання карт	16 слів
19	Екран	16 слів

Операції введення-виведення використовують коди літер: в одному байті міститься код однієї літери. Таким чином, у кожному слові НМ зберігається п'ять літер. Співвідношення між літерами та їхніми кодами в НМ задає **кодова таблиця НМ**:

	0	1	2	3	4	5	6	7	8	9
0		A	B	C	D	E	F	G	H	I
1	θ	J	K	L	M	N	O	P	Q	R
2	∅	π	S	T	U	V	W	X	Y	Z
3	0	1	2	3	4	5	6	7	8	9
4	.	,	(	)	+	-	*	/	=	$
5	<	>	@	;	:	'	ВК	ПС		

• **IN** (введення), C = 36, F = номер пристрою.

Здійснює пересилання інформації із заданого пристрою в поле **M**. Пересилається один блок інформації. Номер поточного блока зберігається в регістрі $X.

• **OUT** (виведення), C = 37, F = номер пристрою.

Запис на зовнішній пристрій. Інформація, що зберігається в полі за адресою **M**, передається на пристрій.

Команди перетворення. Призначені для перетворення числової інформації в літерну (літерні коди) і навпаки. Використовуються в програмах разом із командами введення-виведення.

NUM (англ. convert to numeric – *перетворити на число*), C = 5, F = 0.

Команда перетворює літерний код на число. Значення **M** ігнорується. Літерний код зберігається в $AX, у $A записується його значення. Байти 00, 10, 20, ..., 90 трактуються як цифра 0; байти 01, 11, 21, ..., 91 – як 1 і т. д. Величина $X і знак A не змінюються. Можливе переповнення. У цьому випадку зберігається залишок за модулем розміру слова (старша цифра зникає).

CHAR (англ. convert to char – *перетворити на літери*), C = 5, F = 1.

Перетворення числа на літерний код призначене для виведення його на перфокарту або екран. Число з $A перетворюється на десятибайтне число в $AX у літерній формі. Знаки A та X не змінюються. Значення **M** ігнорується.

Приклад 2.17. Написати програму, яка обчислює й виводить на екран значення $(a^2 + b - 500)/3$, де a та b – числові значення комі-

рок 2000 та 2001. При переповненні друкує символ '*'. Програму розмістити за адресою 0050.

Щоб за командою **OUT** вивести результат, необхідно попередньо сформувати його літерну форму в певному вихідному полі. Мінімальна довжина такого поля 16 слів (1 блок). Нехай 2004 буде адресою вихідного поля. Спочатку вихідне поле обнулюється (нагадаємо, що в НМ нуль є кодом символу-пробілу). Потім у словах 2004 та 2005 програма формує літерне подання результату.

	Команда	**Коментар**	
50	STZ 2004	Обнулення комірки 2004	
51	ST1 2005	Підготовка $**R**1 для команди MOVE	
52	MOVE 2004(15)	Обнулення вихідного поля	
53	LDA 2000	$A ← a	$(*2000)^2$
54	MUL 2000	$AX ← a×a	
55	JANZ 0070	Переповнення при обчисленні a×a	
56	STX 2002	Вивантажити a×a у слово 2002	
57	LDA 2001	$A ← b	
58	SUM 2002	$A← a×a + b	
59	JOV 0070	Переповнення при обчисленні a×a + b	
60	DECA 500	$A← a×a + b – 500	
61	SRAX 5	Зсув $A у $X	
62	ENT1 3	$**R**1← 3	
63	ST1 2002	Записати 3 у комірку 2002	
64	DIV 2002	$A← a×a + b – 500/3	
65	JOF 0070	Переповнення при діленні	
66	CHAR (1)	$AX ← літерну форму $A	
67	STA 2004	Запис $A у слово 2004	
68	STX 2005	Запис $X у слово 2005	
69	JUMP 0073	Перехід на виведення результату	
70	ENTA 46	Заносить у п'ятий байт $A число 46 (код '*')	
71	SLA 4	Зсуває '*' у перший байт $A	
72	STA 2004	Заносить '*' у вихідне поле	
73	OUT 2004 (16)		
74	HLT		

■

Приклад 2.18. У комірці 0200 міститься слово. Отримати його дзеркальне відображення.

Розв'язок 1. Результат отримаємо в регістрі $A:

Адреса	Команди
1000	LDA 0200(1;1)
1001	SRAX 1
1002	LDA 0200(2;2)
1003	SRAX 1
1004	LDA 0200(3;3)
1005	SRAX 1
1006	LDA 0200(4;4)
1007	SRAX 1
1008	LDA 0200(5;5)
1009	SRAX 1
1010	SLAX 5

Розв'язок 2.

	Команди	Коментар
1000	ENT1 4(2)	
1001	JIN 1007	
1002	LDA 0200	Тіло циклу
1003	SRA 0,1	
1004	SRAX 1	
1005	DEC 1	
1006	JMP 1001	
1007	SLAX 5	

■

Приклад 2.19. Написати вірусну програму, що в усі 4000 слів пам'яті запише команду HLT (C = 5, F = 2).

Розв'язок 1. Вірусна програма складається з десяти команд і розташовується в останніх комірках ОП.

	Команди	Коментар
3990	ENT1 0	Підготовка $**R**1 для команди MOVE
3991	ENT2 999	
3992	J2Z 3999	
3993	MOVE 3996(4)	Пересилання чотирьох слів із командою HLT за адресою **R**1
3994	DEC2 1	Зменшення параметра циклу
3995	JMP 3992	Перехід на початок циклу
3996	HLT	
3997	HLT	
3998	HLT	
3999	HLT	

Нагадаємо, що за допомогою команди MOVE можна одноразово переслати до 63 слів. Ми пересилаємо одночасно по чотири команди HLT. Після повторення 998 разів тіла циклу будуть заповнені комірки від 0 до 3991, а при останньому – 999-му – виконанні тіла – комірки 3992–3995 із самим циклом.

Розв'язок 2. Вірусна програма складається з п'яти команд і теж розташовується в останніх комірках ОП.

	Команди
3995	ENT1 0
3996	MOVE 3998(2)
3997	JMP 3986
3998	HLT
3999	HLT

■

***Література для СР:** обчислювальні системи [68, 110, 115, 117, 122]; машинна арифметика – [19, 60]; навчальна машина – [58, 59].

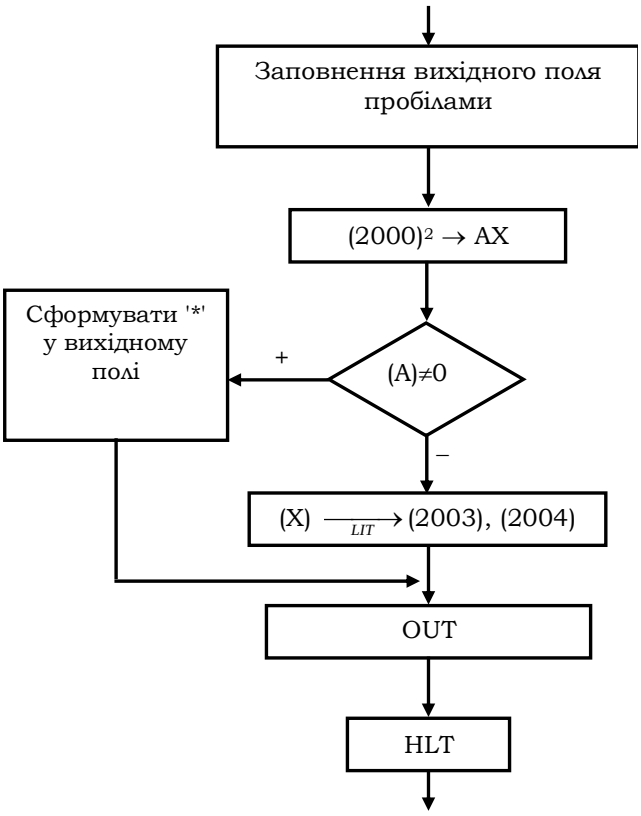
Контрольні запитання та вправи

1. Охарактеризувати загальну структуру комп'ютера.
2. Яка роль лічильника команд у архітектурі процесора?
3. Які функції ОС?
4. Що таке драйвер?
5. Що таке командний процесор?
6. Що таке утиліта?
7. Перерахуйте основні функції системи програмування.
8. Що таке текстовий редактор?
9. Яка різниця між компілятором та інтерпретатором?
10. Яка функція налагоджувача?
11. Що таке об'єктний модуль?
12. Яка роль редактора зв'язків і завантажника в системах програмування?
13. Що таке числа з фіксованою й рухомою точками?
14. Що таке мантиса й порядок РТ-числа?
15. Що таке нормалізоване подання РТ-числа?
16. Знайти двійкові й шістнадцяткові записи десяткових чисел: 15, 255, 256, 32767, 32768, 65537 та 65538.
17. Знайти десяткові й шістнадцяткові записи двійкових чисел: 1100, 1111, 11111111, 1 000 0000, 100 0000 0000.
18. Знайти двійкові й десяткові записи шістнадцяткових чисел: FF2, ABCD, 77F, 1000.

19. Знайти обернений і додатковий двобайтний двійковий код від'ємних чисел: а) –25; б) –102; в) -1334; г) -34000.
20. У системі з основою $p=36$ цифрами є 0, 1,...,9 і великі латинські літери A, B,...., Z. Знайти десяткове значення чисел а) 10_p; б) $ZZZZ_p$; в) $ABC001_p$.
21. Знайти двійкові подання десяткових дробів (перші 12 цифр): а) 0.9999; б) 0.12; в) 3.14.
22. Скласти за допомогою калькулятора таблицю множення 15×15 для шістнадцяткових чисел. За допомогою цієї таблиці знайти добутки 25×125 та $AAAA\times F1$.
23. Що таке переповнення? Навести приклади.
24. Що таке зникнення порядку? Навести приклади.
25. Розв'язати систему лінійних рівнянь у машинній арифметиці із чотирма цифрами в мантисі; те саме – за допомогою калькулятора. Результати порівняти:

а) $\begin{cases}0.2x+3y=11.25\\5x-0.001y=100\end{cases}$; б) $\begin{cases}5x+3.1y=0.333\\x-0.1y=1001\end{cases}$.

26. Що таке кодова таблиця символів?
27. Що таке ASCII-таблиця та її національні варіанти?
28. Яка структура НМ?
29. Яка загальна структура машинної команди НМ?
30. Пояснити дію арифметичних команд НМ. Навести приклади.
31. Пояснити дію команд порівняння. Навести приклади.
32. Пояснити дію команд збереження регістрів. Навести приклади.
33. Пояснити дію команд завантаження регістрів. Навести приклади.
34. Пояснити дію команд пересилання адрес. Навести приклади.
35. Пояснити дію команд керування пристроями введення-виведення. Навести відповідні приклади.
36. У чому полягає смисл команд перетворення?
37. Пояснити дію команд перетворення. Навести приклади.
38. У чому полягає смисл команд переходу?
39. Пояснити дію команд переходу. Навести приклади.
40. Обчислити й вивести значення a^3, де a зберігається в комірці з адресою 1000. У випадку переповнення вивести '*'. *Вказівка.* Скористатися СБС, яка подає схему розв'язку задачі для значення a^2:



41. Написати вірусну програму з кількістю команд менше п'яти.
42. Відстань Хаммінга між бітовими векторами визначається кількістю позицій, в яких вони відрізняються. Знайти цю відстань: а) між регістрами $A $X; б) між довільними полями, адреси яких містяться в регістрах $**R**2 та $**R**3.
43. Реалізувати операцію **MUL** за допомогою алгоритму швидкого множення за допомогою команди **ADD**.
44. Реалізувати операції **DIV** і взяття залишку **MOD** за допомогою команди **SUB**.
45. Реалізувати операцію **POW** швидкого піднесення до степеня за допомогою команди **MUL**.

2.4. Мови програмування

- Дескриптивні системи
- Мови програмування
- Класифікація мов програмування

Ключові слова: *дескриптивна система, синтаксис, семантика та прагматика ДС, символ, ідеограма, вербальна мова, лексема, словник мови, речення, метамова, денотат, термін, предметна константа, дескриптивний термін, кодова таблиця, кодові таблиці ASCII, Unicode, UCS-2 та UCS-4, широка літера, денотаційна й операційна семантика, базові, стандартні й функціональні типи, об'єднана Ω-система типів, операційний об'єкт, зв'язування, дескриптор, оператори опису й декларації, вирази, первинний оператор, прагматика мови програмування, практичні й теоретичні, універсальні та спеціалізовані мови програмування, класифікація мов програмування за предметною орієнтацією, декларативні та процедурно-орієнтовані мови, мови високого рівня, машинно-залежні мови, парадигми декларативного, процедурного, функціонального, логічного, об'єктно-орієнтованого й паралельного програмування, інкапсуляція, спадкування, поліморфізм, мови із сильною та слабкою типізацією.*

Як нам уже відомо, для подання запитів у інформаційних системах і внутрішніх процедур застосовують мови програмування. Однак для такого опису сьогодні, окрім мов програмування, використовують також інші, у тому числі й не вербальні, засоби, в яких усе чіткіше спостерігається тенденція до візуалізації інформації з використанням різних типів граф-схем, діаграм, гештальтів, інтелект-карт[19] тощо. Тому є сенс розпочати не з мов програмування, а із загальних ДС.

2.4.1. ДЕСКРИПТИВНІ СИСТЕМИ

Ми розглядатимемо ДС лише у зв'язку з описом інформаційних систем. *Дескриптивні системи* – це різновид знакових систем, призначених для опису елементів інформаційних систем і роботи з ними. Кожна ДС визначає певну фіксовану сукупність об'єктів, які мають своє ім'я, значення та смисл. Іменами є *знаки* – матеріальні елементи, що сприймаються на чуттєвому або фізичному рівні й зображують у комунікативних процесах значення й певні властивості та відношення між інформаційними об'єктами.

[19] Інтелект-карта зображує слова, ідеї, задачі або інші поняття як асоціативно пов'язані з низкою інших понять, що відходять від певного центрального поняття або ідеї (принцип радіантного мислення).

Семіотика[20] виділяє як визначальні три складові знакових систем: *синтаксис* (описує будову знаків), *семантику* (описує значення об'єктів і самі об'єкти) і *прагматику* (описує смисл об'єктів). Розглянемо ці складові окремо.

Синтаксис. Кожна ДС використовує певну фіксовану сукупність знаків. У семіотиці виділяють кілька різновидів знаків, серед яких особлива роль належить *знакам-символам*, які фізично не пов'язані зі своїми значеннями. Значення та смисл символів установлюються за узгодженням.

Ми вже мали справу зі знаками-символами, коли розглядали константи, змінні й терми у ПЧП. Знакові системи зі знаками-символами називаються *символьними.* Прикладами символьних систем є системи сигналів і знаків дорожнього руху, абетка Морзе, нотна грамота, електричні схеми, природні мови, мова жестів глухонімих тощо. Дескриптологічні знакові системи теж належать до символьних.

Свобода у виборі того чи іншого символу для даного значення надає суттєву перевагу символам перед іншими видами знаків. Завдяки вибору імені й асоціативних зв'язків можна змістовніше подати об'єкт, закріпити за ним певне смислове навантаження (*мнемонічність імені*). Цю ситуацію дуже добре знають програмісти. У хорошій програмі імена даних і функцій обов'язково мнемонічні, і це є однією з ознак хорошого стилю програмування.

Іншою перевагою символів є те, що знак-символ може фізично не прив'язуватись до одного певного значення, а по черзі позначати кілька значень (символи-змінні). Завдяки цьому відбувається економія імен у інформаційних процесах.

Серед усіх видів символів найуживанішими є графічні (письмові) або, як їх ще називають, *ідеограми.* Ідеограма зазвичай має певну внутрішню будову. Вона складається з певних базових графічних елементів. Наприклад, запис натурального числа в десятковій системі числення складається з десяткових літер-цифр, структурна блок-схема алгоритму – із прямокутників, ромбів тощо. Складнішим прикладом ідеограм є інтелект-карти, рисунки тощо.

Серед ідеограм найбільш простими та зручними на практиці виявилися скінченні послідовності літер, тобто слова в певному алфавіті. ДС із символами-словами називаються *словесними*, або *вербальними* (від лат. verbalis – словесний).

Кожна словесна ДС має свій фіксований алфавіт літер $\Sigma = \{a_1, ..., a_n\}$, який є обов'язково лінійно впорядкованим. Серед усіх

[20] Наука про знакові системи.

слів над алфавітом Σ зазвичай тільки частина є правильними й використовуються як символи. Решта – беззмістовні. Усі слова мови поділяються на слова-лексеми й слова-речення. Лексеми об'єднуються в *речення* за допомогою спеціальних літер-роздільників – деякі з них позначають кінець речення. Самі роздільники теж можуть належати до лексем (символи операцій +, - тощо). Таким чином, *лексема* речення – це або префікс, що передує першому в ньому роздільнику, або суфікс, розташований між роздільниками, який не містить усередині інших роздільників. Кожна мова має свій набір літер-роздільників (пробіл, кома, крапка, тире, дужки тощо) і лексем (*словник* мови). За допомогою спеціальних синтаксичних правил із лексем будуються правильні речення мови. Речення може трактуватися також як певне слово у збільшеному алфавіті – алфавіті лексем, тобто це слово, побудоване зі знаків вищого рівня. Саме така ситуація має місце в мовах програмування – транслятор працює вже на рівні лексем.

Словник, разом із правилами утворення лексем, утворює *лексику* мови. Синтаксичному аналізу програми завжди передує фаза *лексичного аналізу*, коли спочатку будуються всі лексеми програми, а потім уже з'ясовується її синтаксична й семантична структура.

Приклад 2.20. Зробимо лексичний аналіз речення:

Маша їла кашу, а м'яч поцілив у ворота.

Лексемами в ньому є слова: <*Маша*>, <*їла*>, <*кашу*>, <*а*>, <*м'яч*>, <*поцілив*>, <*у*>, <*ворота*>. Роздільниками є пробіл (7 входжень), кома (1 входження) і крапка (1 входження) ■

Семантика. Семантичні правила надають деяким символьним іменам їхні значення (*денотати*). Імена зі значеннями називаються *термінами*. Таким чином, термін – це символ, з яким пов'язане конкретне значення або сукупність значень.

Серед термінів виділяють:

а) *терми* – терміни, що подають реальний чи абстрактний об'єкт (з математичними термами ми вже зустрічалися в підрозд. 1.1.1);

б) *предикатні вирази* – терміни, що позначають (виділяють) об'єкти серед інших об'єктів за допомогою їхніх особливих властивостей;

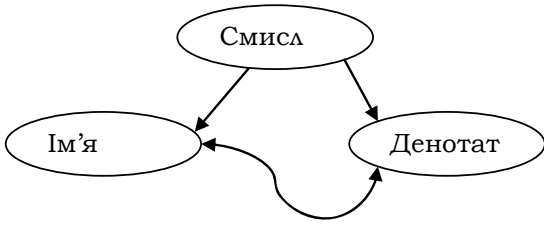
в) *предметно-функціональні вирази* – терміни, що позначають дії, процеси, функції, операції тощо.

У прикл. 2.20 термами є іменники <*Маша*>, <*кашу*>, <*м'яч*> та <*ворота*>, прикметник <*футбольний*> – предикатний вираз, а дієслова <*їла*> та <*поцілив*> – предметно-функціональні вирази.

Сам процес іменування може здійснюватись у два способи: 1) шляхом прямого зіставлення й закріплення за іменем його постійного конкретного денотата; 2) опосередковано – шляхом опису дено-

тата як носія певних ознак і властивостей у вигляді *дескрипцій*. Терміни, що отримали значення в перший спосіб, називаються *предметними константами*. Це назви конкретних предметів, явищ, дій тощо. Решта термінів називаються *дескриптивними*.[21] В описі їхніх значень обов'язково беруть участь інші терміни, наприклад: "$2+7$" ("2" та "7" – предметні константи, "$+$" – символ операції), "перша ЕОМ у континентальній Європі", "число, яке задовольняє рівняння $x^2-2x+2=0$", "студент факультету кібернетики". Остання дескрипція вказує не на конкретну особу, а на сукупність осіб, які є студентами факультету кібернетики, а передостання показує, що термін може мати смисл, але сукупність його можливих значень порожня.

Прагматика. Є визначальним елементом ДС, оскільки саме вона формує мету інформаційних процесів. Прагматика задає як загальний смисл об'єктів у співвідношенні з іншими об'єктами, так і їхній смисл у кожному конкретному знаковому процесі. Смисл засвоюється в процесі розуміння й застосування об'єкта, роботи з ним. Він також може брати участь у визначенні значень об'єктів. Першим звернув на це увагу німецький математик Г. Фреге. У сучасній літературі співвідношення між іменем, його денотатом і смислом часто зображують у вигляді схеми, що дістала назву "трикутник віднесення":



Вершини трикутника зображують три аспекти об'єкта: ім'я, денотат і смисл, взаємовідносини між якими забезпечуються процесом спілкування. При цьому смисл є посередником між іменем і денотатом, способом, яким ім'я позначає денотат. Прямі стрілки вказують на реальні відношення між елементами трикутника, а хвиляста (між іменем і денотатом) – на те, що це відношення виникло опосередковано завдяки смислу, як у наведених вище прикладах: "перша ЕОМ у континентальній Європі", "число, яке задовольняє рівняння $x^2-2x+1=0$" або в описі змінної "int *i*". Значенням першої дескрипції є ЕОМ МЕСМ. Друга дескрипція визначає об'єкт, який є розв'язком відповідного квадратного рівняння. Остання дескрипція задає числовий об'єкт з іменем i, зна-

[21] Звідси й назва "дескриптивні системи".

чення якого зберігається в певному полі ОП і стає доступним за допомогою адреси цього поля. Як бачимо, трикутники віднесення використовуються й у мовах програмування[22].

2.4.2. МОВИ ПРОГРАМУВАННЯ

Розглянемо загальну структуру мов програмування, більшість із яких є вербальними. Ми вже зазначали, що великий вплив на створення мов програмування мало ПЧП. Одне із центральних понять мов програмування – вираз (арифметичний, логічний) – визначається у вигляді математичного терму, конструкції, прямо запозиченої у ПЧП[23].

При вивченні мов семіотика розрізняє *мови-об'єкти* й *метамови*. Останні – це мови, за допомогою яких описуються й досліджуються об'єктні мови. Наприклад, при вивченні англійської користуються рідною мовою, математичного аналізу – знов-таки рідною мовою, а також елементами ПЧП. Тут рідна мова і ПЧП є метамовами відносно мов-об'єктів – англійської й мови математичного аналізу.

Для опису конструкцій мов програмування в інформатиці застосовується цілий арсенал метазасобів: від природних мов до штучних, таких, як ПЧП, мов специфікації функцій і алгоритмів, структурних блок-схем і схем програм, БНФ, КС-граматик, синтаксичних діаграм, різних класів автоматів тощо.

Синтаксис мов програмування. Як вербальна ДС, кожна мова програмування має свій алфавіт літер, що задається *кодовою таблицею*[24]. У такій таблиці літери зображуються разом з їхніми числовими кодами.

Останнім часом у зв'язку з розвитком глобальних мереж докладається багато зусиль для стандартизації алфавітів мов програмування. Необхідно узгодити різні не тільки формально, а й за походженням і природою численні алфавіти національних мов. Один із підходів для розв'язання цієї проблеми запропоновано розробниками Універсального символьного набору багатооктетних кодів (Universal Multiple-Octet Code Character Set (UCS-4)) у стандарті ISO/IES 1064. Цього набору чотирибайтних (чотириоктетних) літерних кодів теоретично має вистачати, щоб подати (із запасом) усі існуючі на сьогодні алфавіти.

---

[22] Ця ситуація добре знайома програмістам, особливо початківцям, які часто спочатку вчаться використовувати ті чи інші програмні конструкції, і тільки за гострої необхідності (програма відмовляється працювати) звертаються до їхньої семантики.

[23] Семантика виразів мов програмування дещо відрізняється від математичної. Це пов'язано з динамічним характером процесів іменування даних.

[24] У літературі інколи невдало, на наш погляд, ототожнюють терміни "символ" і "літера". Літера стає символом, тільки коли їй надається певне значення або смисл.

Існує двобайтний варіант цього набору – UCS-2, що містить усі літери набору UCS-4 (включаючи 20000 ієрогліфів Китаю, Японії та Кореї), у яких два старших байти нульові. Первісно відомий набір символів Unicode, який запровадила організація Unicode Consortium (www.unicode.org), був двобайтним і узгодженим з набором UCS-2. Третя версія цього стандарту – Unicode-3.0 – повністю відповідає стандарту ISO/IES 1064. Усі стандарти – UCS-4, UCS-2 та Unicode – узгоджені зі стандартом ASCII. Це означає, що 16-бітні коди, старші байти яких дорівнюють 0, є 9-бітним розширенням відповідних літер набору ASCII.

Літери, коди яких за розміром більші від восьмибітних, називаються *широкими*. У сучасних версіях мов програмування передбачається використання в символьних і літеральних константах та ідентифікаторах широких літер із наборів UCS-2 і UCS-4.

Як приклад кодової таблиці наведемо верхню частину таблиці кодів Windows-1251, що містить літери українського, російського, білоруського й деяких інших національних алфавітів. Числа під літерами подають 16-ковий код символу в коді UCS-2. Нижня частина цієї таблиці (з кодами від 0 до 127) відповідає стандартному семибітному коду ASCII.

Кодова таблиця Windows-1251 (верхня частина)

	.0	**.1**	**.2**	**.3**	**.4**	**.5**	**.6**	**.7**	**.8**	**.9**	**.A**	**.B**	**.C**	**.D**	**.E**	**.F**
8.	Ђ 402	Ѓ 403	‚ 201A	ѓ 453	„ 201E	… 2026	† 2020	‡ 2021	€ 20AC	‰ 2030	Љ 409	‹ 2039	Њ 40A	Ќ 40C	Ћ 40B	Џ 40F
9.	ђ 452	‘ 2018	’ 2019	" 201C	" 201D	• 2022	– 2013	— 2014		™ 2122	љ 459	› 203A	њ 45A	ќ 45C	ћ 45B	џ 45F
A.	A0	Ў 40E	ў 45E	J 408	¤ A4	Ґ 490	¦ A6	§ A7	Ё 401	© A9	Є 404	« AB	¬ AC	AD	® AE	Ї 407
B.	° B0	± B1	I 406	і 456	ґ 491	µ B5	¶ B6	· B7	ё 451	№ 2116	є 454	» BB	j 458	S 405	s 455	ї 457
C.	А 410	Б 411	В 412	Г 413	Д 414	Е 415	Ж 416	З 417	И 418	Й 419	К 41A	Л 41B	М 41C	Н 41D	О 41E	П 41F
D.	Р 420	С 421	Т 422	У 423	Ф 424	Х 425	Ц 426	Ч 427	Ш 428	Щ 429	Ъ 42A	Ы 42B	Ь 42C	Э 42D	Ю 42E	Я 42F
E.	а 430	б 431	в 432	г 433	д 434	е 435	ж 436	з 437	и 438	й 439	к 43A	л 43B	м 43C	н 43D	о 43E	п 43F
F.	р 440	с 441	т 442	у 443	ф 444	х 445	ц 446	ч 447	ш 448	щ 449	ъ 44A	ы 44B	ь 44C	э 44D	ю 44E	я 44F

З літер будуються лексеми мови, з яких, у свою чергу, утворюються всі інші синтаксичні конструкції, у тому числі й головні – програми. Лексика мов програмування виділяє п'ять класів лексем: ідентифікатори, ключові слова, константи, операції й роздільники. Вони детально розглядаються в підрозд. 3.1.2 на прикладі мови С.

Основні синтаксичні конструкції мов програмування – дані, конструктори типів, вирази, функції та програми. У кожній мові вони мають свої специфічні ідеограми, які визначаються конкретним синтаксисом мови, а от семантика в них спільна.

Семантика мов програмування. Кожна мова програмування описує певну фіксовану сукупність даних і програм. За принципом функціональності кожній програмі відповідає певний $X-Y$-оператор, що пов'язує вхідні дані й результати. Цей оператор є денотатом програми (*денотаційна семантика* програми). Програма має й *операційну семантику* – алгоритм (машинну програму), що описує пошук результатів запиту на машинному рівні.

За принципом типізації всі дані мов програмування згруповуються в типи, які ієрархічно впорядковані й утворюють вежу типів. В основі вежі містяться базові типи, а над ними – похідні, що утворюються з базових за допомогою конструкторів типів. *Базові типи* – це типи даних, що відповідають машинним константам – числам, літерам і адресам. Наступними за базовими у вежі типів є *стандартні* дані – змінні та іменовані константи, значення яких належать певному базовому типу. Особливу роль відіграють *функціональні* типи, які складаються з адрес підпрограм, що реалізують функції певного фіксованого типу.

Базові та стандартні типи, а також їхні операції та інтерфейсні функції є твірними в багатосортній Ω-*системі типів* мови програмування. Носій її складається з універсума всіх типів даних. Сигнатуру Ω утворюють символи операцій, інтерфейсних і стандартних функцій і регулярних композицій, що входять до складу типів мови програмування.

Розглянемо тепер операційний (реалізаційний) аспект даних. Усім даним програми в процесі трансляції ставляться у відповідність *операційні об'єкти* певного типу – поля ОП із певними адресами, довжиною та структурою, в яких зберігаються їхні значення. Ця операція називається *зв'язуванням* даних. Щоб отримати доступ до значення даних, необхідно мати всю інформацію про їхні об'єкти, а для доступу до окремого елемента значення – додатково про його місцезнаходження в структурі об'єкта (зсув відносно адреси поля). Уся ця інформація міститься в *дескрипторах* – кортежах, що характеризують об'єкти в ОП. Кожна змінна в програмі має свій дескриптор, структура якого залежить від типу і для кожного типу різна. Проте обов'язково дескриптор містить інформацію про тип змінної та її адресу.

Імена даних разом із дескрипторами зберігаються в спеціальній таблиці ідентифікаторів, що створюється компілятором на етапі синтаксичного аналізу програми на підставі інформації, яку надають *оператори-описи* й *оператори-декларації.* Оператори опису й декларації на-

зивають *визначенням* даних. Семантика обох операторів визначається їхньою участю у створенні дескриптора даного. Різниця між ними полягає в тому, що оператор-декларація інформує тільки про ім'я й тип даного, тоді як оператор-опис надає повну інформацію про об'єкт, а також може додатково його ініціалізувати. Таким чином, оператори-описи формують початковий стан інформаційного поля програми, який потім у процесі виконання програми буде змінюватися за допомогою операторів введення, присвоювання та інших конструкцій.

Як випливає з принципу композиційності, кожна мова програмування має свою ІЛП, що дозволяє описати внутрішню композиційну будову її програм. За цією логікою семантичні структури мови повністю визначає об'єднана Ω-система базових і стандартних типів: арифметичні та інші операції разом зі стандартними функціями дозволяють отримувати нові значення змінних, а інтерфейсні функції разом із конструкторами типів – формувати всі можливі сукупності даних і отримувати доступ до них. На цьому рівні формується поняття *виразу* мови. Вирази є Ω-термами об'єднаної системи типів і використовуються для обчислення в програмі нових значень змінних. У ІЛП вони відіграють роль аксіом, за якими будуються насамперед оператори присвоювання. З останніх і з інших первинних операторів будуються за допомогою композицій решта операторів мови.

ІЛП = Вирази + Первинні оператори + Композиції

Основу композицій становлять композиції регулярної алгебри. Ураховуючи, що сукупність функцій мов програмування теж типізована (базовими конструкціями її є аксіоми ІЛП, а конструкторами – композиції), можна стверджувати: базові типи, разом із конструкторами та ІЛП, повністю визначають семантику мови програмування.

Вежа типів = Базові типи + Конструктори + ІЛП

Прагматика мов програмування. Програми створюються для того, щоб подати запити суб'єкта-ініціатора та внутрішні алгоритми вихідних систем. Прагматика програм пов'язана зі смислом і якістю тих інформаційних процесів, що реалізують запити.

2.4.3. КЛАСИФІКАЦІЯ МОВ ПРОГРАМУВАННЯ

На сьогодні відомо кілька десятків тисяч мов програмування та їхніх діалектів (кожний новий компілятор мови вводить новий її діалект), і ця кількість невпинно зростає. Щоб якось орієнтуватись у цьому розмаїтті мов, їх об'єднують у класи за певними ознаками. Кла-

си при цьому можуть перетинатися. Розглянемо основні класифікації мов програмування. Програми довільної мови програмування L домовимось називати *L-програмами*.

Як уже зазначалось у вступі, мови програмування розподіляються на *практичні* й *теоретичні*. Останні використовуються в теоретичних моделях інформаційних систем та їхніх компонентів. Це стосується в основному алгоритмічних мов.

Функціональна сила. Ця ознака притаманна теоретичним моделям мов, в яких немає обмежень на потужність типів даних і кількість даних у програмах, тобто потенційно ці величини вважаються зліченними. Якщо це тип цілих чисел, то він збігається з усіма цілими числами тощо. За даною ознакою мови програмування розподіляються на:

- *універсальні*;
- *спеціалізовані*.

Як випливає з принципу функціональності, будь-яка програма є функцією, що пов'язує вхідні дані з результатами, тобто кожній мові програмування L відповідає певна конкретна сукупність подібних функцій. Серед них можна виділити підмножину F базових елементарних функцій, з яких за допомогою певних композицій будуються всі L-програми. У теорії алгоритмів існує поняття класу $U(F)$ усіх алгоритмічно обчислюваних функцій над даним базисом F. Якщо для кожної функції класу $U(F)$ існує L-програма, що обчислює її відносно певних функцій кодування й декодування, то кажуть, що мова L є *універсальною*. У зворотному випадку мову називають *спеціалізованою*. Такі відомі мови програмування, як АЛГОЛ-60, ФОРТРАН, Lisp, ПАСКАЛЬ, C, C^{++}, Java, PHP, Perl, Python та інші є універсальними[25]. Спеціалізованими є, наприклад, мови для підготовки програмної документації (мова РПГ тощо).

Зауважимо, що універсальність мови програмування ще не означає її зручність для розв'язання тієї чи іншої конкретної задачі. Вона забезпечує тільки теоретичну можливість такого розв'язання. Тому на практиці при виборі мови програмування враховують специфіку як задачі, так і конкретних мов та їхніх систем програмування (типів даних, стандартних бібліотек, операційного середовища тощо). Звідси випливає важливість нижченаведеної класифікації.

Предметна орієнтація. Зазвичай поштовхом для розробки й реалізації нової мови програмування або нової версії вже існуючої мови є бажання забезпечити ефективніше програмування запитів у певній

[25] Перші мови програмування так і називали – універсальними, наприклад, Універсальна мова програмування ПЛ/1.

предметній області, ніж це дозволяють існуючі мови та їхні системи програмування. Даний клас запитів визначає *предметну орієнтацію* мови. Предметна орієнтація мови знаходить своє відображення насамперед у її типах даних, а також у структурах програм.

• Мови АЛГОЛ-60, ФОРТРАН-IV, АНАЛІТИК тощо орієнтовані на *числову* обробку даних (задачі числового аналізу, лінійної алгебри, математичної фізики й аналітичних перетворень та ін.). Для подібних задач характерними є порівняно прості структури даних (не виходять за межі багатовимірних масивів) і потужна ІЛП.

• Мови типу ПЛ-1, КОБОЛ, АЛГОЛ-68, ПАСКАЛЬ, С, С++, Delphi, Oberon, Java орієнтовані на обробку *складноструктурованої* інформації. Вони мають розвинену систему типів даних зі структурами, покажчиками тощо й потужну ІЛП.

• Мови типу Modula, Ada – це мови, зорієнтовані на обробку даних *у реальному часі.*

• Мови типу СНОБОЛ, Lisp, РЕФАЛ, ФОЛІ зорієнтовані на *символьну* обробку, що є характерною для задач штучного інтелекту тощо.

• Мови типу Oracle, SQL зорієнтовані на розробку *баз даних.*

• Мови типу HTML, PHP, Perl зорієнтовані на задачі, пов'язані з *WEB-технологіями.*

Рівень абстракції. За рівнем абстракції мови програмування розподіляють на:

• *декларативні, або проблемно-орієнтовані;*

• *процедурно-орієнтовані.*

Перші призначені для опису функціональної семантики програм, тобто того, що має робити програма, але не вказують як. У подібних мовах програма – це формальний опис результатів аналізу задачі. Зазвичай цей опис задає функціональну семантику програми неявно за допомогою різних систем рівнянь, у тому числі й функціональних, записаних у тому чи іншому стилізованому фрагменті ПЧП. Прикладами подібних проблемно-орієнтованих мов є мови УТОПІСТ, НОРЕ, значною мірою ПРОЛОГ тощо. Проблемно-орієнтовані мови використовують зазвичай на ранніх стадіях проектування для швидкого отримання проміжних моделей, макетів і прототипів систем. При цьому прагматичні вимоги можуть тимчасово залишатися на другому плані.

У процедурно-орієнтованих мовах програми специфікуються як функції-процедури. На сьогодні це основний інструментарій у існуючих технологіях програмування. Вони поділяються на два класи:

• *мови високого рівня* призначені для формування запитів: мови типу АЛГОЛ-60, ПАСКАЛЬ, С, С++ тощо;

• *машинно-залежні мови* – це внутрішні алгоритмічні мови ЕОМ або наближені до них: мова НМ, асамблерні й макроасамблерні мови, автокоди, бінарні коди програм.

Мови високого рівня характеризуються зручним для людини синтаксисом, структури даних і програм мають абстрактну математичну семантику, інваріантну відносно архітектури конкретних ЕОМ. Машинно-залежні значною мірою зберігають структуру конкретних машинних мов. Проміжну позицію між мовами високого рівня й машинно-залежними займають високорівневі внутрішні мови, наприклад Адресна, ФОЛІ тощо.

Парадигми. Мови програмування можуть класифікуватися за парадигмами програмування, які вони підтримують (повною мірою або частково). Мова може підтримувати кілька парадигм. Кожна з парадигм орієнтується на певні моделі програм і даних, відповідні засоби їхнього опису й методи реалізації. Найвідомішими парадигмами програмування є:

• *Декларативне програмування.* Має справу з програмами у вигляді формального опису співвідношень між вхідними й вихідними даними задачі. Зазвичай цей опис задає функціональну семантику програми неявно за допомогою різних систем рівнянь.

• *Процедурне програмування.* Має справу з програмами, що містять процедури розв'язання задач. Підтримується більшістю мов програмування.

• *Функціональне програмування.* Програми специфікуються у вигляді функцій-процедур із широким застосуванням індуктивних визначень. Мови Lisp, РЕФАЛ, ML, Haskell, Pyhton.

• *Програмування з абстрактними типами даних* (*АТД*). Для специфікації програм використовуються АТД з об'єктами, класами та інкапсуляцією. Мови SIMULA-67, Modula, Ada, C++.

• *Логічне програмування.* Програми специфікуються у вигляді спеціальних формул ПЧП. Найбільшого поширення набула мова Prolog.

• *Об'єктно-орієнтоване програмування* (*ООП*). ООП – це методологія програмування, що базується на зображенні програми у вигляді сукупності об'єктів, кожен з яких є екземпляром певного *класу*, а класи утворюють ієрархію спадкування. Ураховуючи значну популярність цієї парадигми, зупинимось на ній детальніше.

Об'єктно-орієнтовані можливості класів забезпечують три найважливіші властивості ООП:

1) *Інкапсуляція* – це механізм, який об'єднує дані й методи, що маніпулюють даними, у класи й захищає їх від зовнішнього впливу чи неправильного використання.

2) *Спадкування* – можливість породжувати один клас на основі іншого зі збереженням усіх атрибутів і методів батьківського класу й додавати за необхідності нові атрибути й методи. Набір класів, пов'язаних відношенням спадкування, називають *ієрархією*.

3) *Поліморфізм* – явище, при якому один і той самий програмний код виконується по-різному, залежно від того, об'єкт якого класу використовується при виклику даного коду. Поліморфізм забезпечується тим, що в класі-нащадку змінюють реалізацію методу батьківського класу з обов'язковим збереженням сигнатури методу. Це забезпечує збереження незмінним інтерфейсу батьківського класу й дозволяє здійснювати динамічне зв'язування імені методу в коді з різними класами – з об'єкта якого класу здійснюється виклик, з того самого класу й береться метод із заданим іменем. Мови об'єктного програмування поділяються на *об'єктні*, в яких існують класи й об'єкти (SIMULA-67), і *об'єктно-орієнтовані*, в яких програміст може не лише користуватися визначеними класами, але й задавати свої. Відомими об'єктно-орієнтованими мовами є Smalltalk, Oberon, C++, C#, Java, JavaScript тощо.

• *Паралельне програмування.* Для специфікації програм використовуються засоби розпаралелювання процесів. Мови Ada, Java, High-Performance FORTRAN.

• *Агентне програмування.* У центрі уваги – агент, самодостатня програма, здатна керувати своїми діями в розподіленому інформаційному середовищі, структура якого може динамічно змінюватися. Підтримують мови Java, частково – Zonnon.

Типізація. Розрізняють мови зі *статичною* (*сильною*) і *динамічною* (*слабкою*) типізаціями. У першому випадку змінні й параметри функцій (методів) зв'язуються з типами в момент опису або декларування й не можуть бути змінені пізніше. Це, окрім спрощення трансляції й керування пам'яттю, дає можливість здійснювати часткову перевірку семантичної коректності використання змінних уже на етапі компіляції, зокрема перевірку узгодженості їхніх типів із типом операцій. У другому випадку змінні й параметри функцій (методів) зв'язуються з типами в момент присвоювання значення або передавання параметра у функцію. У цьому випадку одна й та сама змінна в різні моменти виконання програми може буде зв'язана зі значеннями різних типів.

• *Статична типізація.* Мови ФОРТРАН, АЛГОЛ-60, ПАСКАЛЬ, C, C++, Modula.

• *Динамічна типізація.* Мови Lisp, ML, Pyhton, Perl, Smalltalk.

Наведені класифікації мов програмування склалися історично й відображають переважно прагматичні властивості мов і особливості їх-

ньої трансляції. Сьогодні існує проблема створення наукового підходу до класифікації й порівняльного аналізу мов програмування за тими чи іншими ознаками їхньої внутрішньої структури. Виходячи з проголошених нами загальних і спеціальних принципів програмування, можна стверджувати, що будь-який науковий підхід до класифікації мов програмування має спиратись на попереднє уточнення вежі типів даних, яка визначає всю логіко-функціональну семантику мови. Легкість моделювання базової Ω-системи типів мови, а також конструкторів для побудови похідних типів засобами іншої мови, характеризує подібність двох мов або реальну логіко-функціональну різницю між ними.

***Література для СР:** знакові системи – [56, 63, 81]; мови програмування – [103, 114]; класифікація мов програмування – [114]; парадигми й стилі програмування – [10, 52, 90, 91]; інтелект-карти – [117].

Контрольні запитання та вправи

1. Що таке ДС?
2. Що таке синтаксис, семантика та прагматика ДС?
3. Що таке знак-символ?
4. Що таке ідеограма? Навести приклади.
5. Що так денотат?
6. Що таке термін?
7. Що таке трикутник віднесення?
8. Які бувають терміни?
9. Що таке дескрипція?
10. Що таке метамова?
11. Що таке кодова таблиця?
12. Які коди відповідають символам кирилиці (укр.) у кодових таблицях: а) ASCII; б) Unicode); в) UCS-2; г) UCS-4?
13. Що таке широка літера?
14. Що таке лексема й лексика мови?
15. Що таке денотаційна й операційна семантики програми?
16. Що таке операційний об'єкт?
17. Що таке зв'язування змінної?
18. Що таке дескриптор змінної?
19. Яка роль операторів-описів і операторів-декларацій?
20. Що таке вираз мови програмування?
21. Що таке первинний оператор?
22. У чому полягає прагматика програми?
23. Який програміський сенс трикутників віднесення?

24. Які основні аспекти програм?
25. Що таке вежа типів мов програмування?
26. Що таке об'єднана Ω-система типів мови програмування?
27. Охарактеризувати основні класифікації мов програмування.
28. У чому полягає класифікація мов програмування за предметною орієнтацією?
29. У чому полягає класифікація мов програмування за рівнем абстракції?
30. Що таке мови високого рівня й машинно-залежні?
31. Сформулювати ознаки парадигм: а) декларативного; б) процедурного; в) функціонального; г) логічного; д) об'єктно-орієнтованого; е) паралельного; є) агентного програмування.
32. Яка різниця між статичною й динамічною типізаціями?

2.5. Структурні програми й рекурентні функції

- Структурні схеми програм і структурні алгоритми
- Системи рекурентних послідовностей
- Рекурентні функції
- Техніка побудови стандартних програм

Ключові слова: *структурна схема програм, інтерпретація ССП, структурний алгоритм, абстрактна структурна програма, система рекурентних послідовностей, конструктор рекурентної послідовності, система т-рекурентних послідовностей, вхідна, опорна й проміжна послідовності, вимір системи рекурентних послідовностей,* μ*-оператор, рекурентна й т-рекурентна функція, параметризована рекурентна функція.*

Означені у підрозд. 2.1.5 структурні алгоритми становлять основу циклічних конструкцій процедурних мов програмування. Необхідно лише доповнити їх інтерфейсними функціями й відповідними операціями присвоювання для роботи з іменованими даними. У даному підрозділі розглядається універсальний метод побудови таких алгоритмів і програм. Він спирається на поняття системи рекурентних послідовностей і рекурентної функції.

Однак спочатку розглянемо лінійну форму структурних алгоритмів.

2.5.1. СТРУКТУРНІ СХЕМИ ПРОГРАМ І СТРУКТУРНІ АЛГОРИТМИ

Структурні схеми програм (*ССП*) є словесними формами структурних блок-схем. Вони теж визначаються алгебрично як елементи замикання, породженого базовими схемами Box, $Con1$, $Con2$ за допомогою ключових слів while, do, case, detcase, if, else і допоміжних символів {, }, (,), :. Схеми $Con1$ та $Con2$ подають довільний і заключний предикати[26]. Нехай $P_1,...,P_n$ – довільні ССП. Нові ССП будуються таким чином:

1) *складена ССП*: $\{P_1...P_n\}$;

2) *розгалуження*: if$(Con1)$ P_1 else P_2;

3) *обхід*: if$(Con1)P$;

4) *недетермінований вибір*: case $\{Con1:P_1,...,Con1:P_n\}$;

5) *детермінований вибір*: detcase $\{Con:P_1,...,Con:P_n\}$, де Con – це $Con1$ або $Con2$;

6) *ітерація*: while $(Con1)P$;

7) *повторення*: do P while $(Con1)$.

Отже, ССП є або базовими, або мають вигляд 1)–7). Фактично ССП є термами регулярної алгоритмічної алгебри, записаними в стилі, наближеному до мов програмування (див. підрозд. 1.4.3). При цьому в ССП, на відміну від регулярних термів, логічні вирази не структуровані.

Інтерпретуються ССП так само, як і структурні блок-схеми. Входження елементів *Box* та *Con1, Con2* замінюються елементарними перетвореннями станів і предикатами певного обчислювального простору $\Pi = <\Gamma, S, \Delta>$.

Інтерпретацією ССП P називається трійка $I=(S,\Delta,\delta)$, де S – множина станів, $\Delta=\{\varphi_1,...,\varphi_k\}\cup\{\gamma_1,...,\gamma_l\}$ – сукупність елементарних перетворень станів і предикатів на станах, $\varphi_i: S\to S$, $i=\overline{1,k}$, $\gamma_j: S\to \text{Bool}$, $j=\overline{1,l}$, δ – функція, яка кожному входженню базового елемента схеми ставить у відповідність певний елемент з інтерпретації схеми – елементарне перетворення, якщо базовим елементом є *Box*; елементарний предикат, якщо ним є $Con1$ або $Con2$. Предикати, що відповідають входженням $Con2$, вважаються *заключними*. Інтерпретовані ССП, як і інтерпретовані СБС, називаються *структурними алгоритмами* й позначаються P^I.

[26] Заключні предикати використовуються в детермінованому виборі (див. підрозд. 1.3.3).

Опишемо семантику структурних алгоритмів. Вона визначається структурою схем їхніх програм і відповідає семантиці регулярних композицій функцій. Нехай $P_0, P_1, ..., P_n$ – довільні ССП; $I = (S, \Delta, \delta)$ – деяка їхня спільна інтерпретація; $\varphi_i : S \to S$ – функція, що є значенням ССП P_i при інтерпретації I, тобто $P_i^I = \varphi_i, i = \overline{1,n}$; $Con_1, ..., Con_n$ та $Con1_1, ..., Con1_n$ – послідовності базових схем $Con1$ та $Con2$ і відповідно схеми $Con1$; $\rho_i, \theta_i \in \Delta, i = \overline{1,n}$ – елементарні предикати – значення Con_i та $Con1_i$ при інтерпретації I.

Для $P = Box$ покладемо $P^I = Box^I$.

Увага! Щоб зробити ССП і алгоритми зрозумілішими, будемо вставляти в їхній текст коментарі-пояснення вигляду /*...*/, які не впливають на дію програм і можуть бути без наслідків вилучені з них ►

Для $P = \{P_1 ... P_n\}$ $P^I = \varphi_1 ... \varphi_n$. /*Множення*/

Для $P = \text{if}(Con1_1) P_1 \text{ else } P_2$ $P^I = (\theta_1 \to \varphi_1 \mid \varphi_2)$. /*Розгалуження*/

Для $P = \text{if}(Con1) P_1$ $P^I = (\theta_1 \to \varphi_1)$. /*Обхід*/

Для $P = \text{case}\{Con1_1 : P_1; ...; Con1_n : P_n\}$ $P^I = (\theta_1 \to \varphi_1 \mid\mid ... \mid\mid \theta_n \to \varphi_n)$. /*Вибір*/

Для $P = \text{detcase}\{Con_1 : P_1; ...; Con_n : P_n\}$ $P^I = (\rho_1 \to \varphi_1 \mid ... \mid \rho_n \to \varphi_n)$. /*Детермінований вибір*/

Для $P = \text{while}(Con1_1) P_1$ $P^I = \{\theta_1 \to \varphi_1\}$. /*Ітерація*/

Для $P = \text{do } P_1 \text{ while}(Con1_1)$ $P^I = [\theta_1 \to \varphi_1]$. /*Повторення*/

Повернемося до структурних алгоритмів із прикл. 2.4.

Приклад 2.21. Структурні алгоритми для обчислення функцій $\text{sign}(x)$, $\pm\sqrt{x}$ і найбільшого спільного дільника двох чисел:

$\text{case}\{x < 0 : -1; x = 0 : 0; x > 0 : 1\}$ /*обчислює функцію $\text{sign}(x)$*/;

$\text{case}\{x \ge 0 : \sqrt{x}; x \ge 0 : -\sqrt{x}\}$ /*обчислює $\pm\sqrt{x}$ */;

$\{\text{while}(q) \text{ do if}(p)\alpha \text{ else } \beta \quad pr_1\}$ /*обчислює НСД (x, y), де елементарні операції та предикати ті самі, що й у прикл. 1.8*/■

Важливу роль відіграють *абстрактні структурні програми.* Це структурні алгоритми зі спеціально структурованими станами й операторами присвоювання в якості елементарних перетворень станів. Праві частини цих операторів задаються термами певної алгебричної

системи $\boldsymbol{A} = (U;\Omega_F;\Omega_P)$ (будемо називати її *базовою*), а ліві є іменами, що належать деякій сукупності імен $V = \{x, x_1, ...y, y_1, ...\}$. При цьому терми (у тому числі й предикатні) трактуються як еквітонні X-арні операції, імена змінних у яких розглядаються як операції читання (див. прикл. 2.2). Предикатні терми називаються *умовами*. Стани таких програм – це інформаційні поля об'єктів з іменами з V і значеннями з універсума U. Абстрактні програми можуть мати параметри – фрейми X та Y, що визначають відповідно вхідні й вихідні змінні. Графіками таких програм є певні $X-Y$-оператори O. Кажуть, що абстрактна програма *обчислює* векторний аналог $\bar{O}$ свого $X-Y$-оператора. Якщо абстрактна програма не містить схем детермінованого вибору, то її називають *стандартною*. Базова алгебрична система $\boldsymbol{A}$ може бути багатосортною Ω-системою.

Абстрактні структурні програми над багатосортними базовими Ω-системами є формальними моделями програм мов програмування із простими структурами даних.

Як приклад стандартної програми розглянемо ще один варіант алгоритму Евкліда для обчислення НСД(n,m).

Приклад 2.22. Обчислення НСД(n,m) через залишки від ділення чисел:

```
/*Вх: x,y                    Вих: x*/

while (x≠y) do if (x<y)  y←y%x; else   x←x%y;
```

Параметризований векторний аналог даної програми зі вхідними змінними x,y і вихідною x збігається з НОД(n,m). Станами програми є інформаційні поля з двома змінними: $\{(x,n),(y,m)\}$, де $n,m \in N$, які змінюються в процесі обчислення операторами присвоювання $y \leftarrow y\%x$; та $x \leftarrow x\%y$;. У результуючому стані значення змінних однакові й дорівнюють НОД(n,m) ■

2.5.2. СИСТЕМИ РЕКУРЕНТНИХ ПОСЛІДОВНОСТЕЙ

Нехай T – довільний універсум елементів, $f,g,...$ – певні часткові операції на T, які будемо називати *конструкторами* рекурентних послідовностей. Система послідовностей

$$\begin{cases} a_0, a_1, \ldots \in T \\ b_0, b_1, \ldots \in T \\ \ldots \end{cases}$$

називається *рекурентною* відносно конструкторів $f, g, \ldots$, якщо всі її члени, за винятком, можливо, перших, пов'язані співвідношеннями

(*) $a_i = f(a_{i-1}, b_{i-1}, \ldots)$, $b_i = g(a_{i-1}, b_{i-1}, \ldots)$, ..., $i = 1, 2, \ldots$.

Кількість послідовностей у системі називається її *виміром*, а самі послідовності – *рекурентними.* Ми розглядатимемо тільки рекурентні системи певного фіксованого скінченного виміру. Щоб повністю визначити таку систему, достатньо задати нульові члени $a_0, b_0, \ldots$ і конструктори. Решту членів знаходять за допомогою конструкторів. Дійсно, $a_1 = f(a_0, b_0, \ldots)$, $b_1 = g(a_0, b_0, \ldots)$ тощо.

Іноді зручно нумерувати члени рекурентних послідовностей, розпочинаючи з 1. Члени рекурентних послідовностей можуть залежати від членів не всіх сусідніх послідовностей, а тільки деяких, тому на практиці конструктори можуть мати (і зазвичай мають!) різну арність. У деяких випадках у системі співвідношень (*) наступні члени послідовностей можуть обчислюватись не тільки за попередніми, а й за поточними членами сусідніх послідовностей, а також за своїм індексом. В останньому випадку це означає, що в системі неявно присутній лічильник (див. прикл. 2.23).

Розглянемо кілька відомих прикладів рекурентних послідовностей.

Приклад 2.23. Рекурентні послідовності:

1) *Послідовність-константа*: (*) $a_1 = a_2 = \ldots = c$.

Конструктор – тотожна операція $f(x) = x$.

2) *Арифметична прогресія з різницею d*:

(*) $a_1 = a$, $a_1 = a_{i-1} + d$, $i > 1$.

Конструктор – операція $f(x) = x + d$.

3) *Лічильник* i: арифметична прогресія з початковим членом 0 або 1 і різницею $d = 1$. Значення лічильника на k-му кроці збігається з k.

4) *Геометрична прогресія зі знаменником* q:

(*) $b_1 = b$, $b_i = b_{i-1} \times q$, $i > 1$.

Конструктор – операція $f(x) = x \times q$.

5) *Гармонічні числа*:

$H_i = \sum_{k=1}^{i} 1/i$, $i \ge 1$, задаються системою рекурентних співвідношень

$$(*)\begin{cases} H_1 = 1, H_i = H_{i-1} + 1/((i-1)+1); \\ i - \text{лічильник}, i_1 = 1. \end{cases}$$

Конструктором першої послідовності є операція $f(x,y) = x + 1/(y+1)$. За означенням $H_2 = H_1 + 1/(i_1 + 1) = 1 + 1/2 = 3/2$, $i_2 = 2$ тощо ■

2.5.3. РЕКУРЕНТНІ ФУНКЦІЇ

З кожною рекурентною системою послідовностей (*) виміру n пов'язана певна функція F типу $T^n \times N \to T^n$, яка вектору $\alpha = (a_0, b_0, \ldots)$ початкових членів і числу $i \ge 0$ ставить у відповідність вектор з i-х членів даних послідовностей:

$$F(\alpha, i) \overset{def}{=} \alpha_i, \text{ де } \alpha_i = (a_i, b_i, \ldots).$$

Щоб знайти значення функції $F(\alpha, m)$ для довільного $m \ge 0$, достатньо послідовно побудувати всі попередні значення $F(\alpha, i)$, $i = \overline{1, m-1}$.

Зазвичай для фіксованого початкового вектора α інтерес становлять не всі значення функції F – члени послідовності α_i, $i \ge 0$, а лише виділені, і найчастіше – лише одне. Щоб відокремити потрібне значення, або прямо задають його конкретний номер, або його виділяють неявно за допомогою умови-фільтра Q типу $T^n \to \text{Bool}$. Щоб такий вибір був однозначним, обмежуються, наприклад, першим з α_i, що задовольняє (порушує) фільтр. Якщо всі α_i порушують (задовольняють) Q, то результат вибору конкретного значення на даному вході α буде невизначений. Далі вважається, що вибирається перше з α_i, на якому порушується умова. Позначимо нову функцію $\mu_Q^{(*)}$ і назвемо її *рекурентною*, породженою системою (*) і фільтром Q. Вона є частковою й має тип $T^n \to T^n$. Композицію, визначену на конструкторах системи (*) і фільтрі Q, результатом якої є функція $\mu_Q^{(*)}$, назвемо μ-*оператором.*

На практиці можуть варіюватися початкові значення тільки окремих із n аргументів рекурентної функції, а решта відіграють роль параметрів – вони або просто фіксуються, або відразу обчислюються

за допомогою конструкторів. Перші називаються *вхідними параметрами*, або *входами*, рекурентної функції. Цікавими з погляду результатів можуть бути не всі компоненти в значенні рекурентної функції, а тільки окремі з них – *вихідні*. Послідовності, що відповідають вихідним компонентам, називаються *опорними* в рекурентній системі. Послідовності, що не належать ні до вхідних, ні до опорних, називаються *проміжними*, або *робочими*. Нехай m та r – кількість відповідно вхідних і опорних послідовностей. Рекурентні функції $\mu_Q^{(*)} : T^m \to T^r$ із виділеними сукупностями вхідних і опорних послідовностей називаються *параметризованими.*

Наведемо кілька прикладів параметризованих рекурентних функцій.

Приклад 2.24. n-й член арифметичної прогресії.

Нехай $a_1, a_2, \ldots$ – арифметична прогресія з певною фіксованою різницею d і початковим членом $a_1 = a$. Позначимо a_n n-й член арифметичної прогресії. Розглянемо систему (*) із трьох рекурентних послідовностей: арифметичної прогресії a_k, послідовності-константи n і лічильника i з початковим значенням 1. Візьмемо за фільтр умову $Q(i,n) = i < n$, за вхідні послідовності – n і прогресію a_k, за опорну – прогресію a_k. Тоді a_n збігається зі значенням $\mu_Q^{(*)}(a,n)$. Коректність такого подання a_n випливає з того, що лічильник із початковим значенням 1 стає перший раз хибним саме на n-му кроці, тобто при $k = n$ ■

Приклад 2.25. Гармонічна функція.

Гармонічна функція задає n-те гармонічне число H_n і є рекурентною функцією $\mu_Q^{(*)}$, породженою фільтром $Q(i,n) = i < n$ і системою рекурентних співвідношень (*):

$$\begin{cases} H_1 = 1, H_i = H_{i-1} + 1/i; \\ i - \text{лічильник}, \ i_1 = 1; \\ n - \text{константа}. \end{cases}$$

Вхідною послідовністю є n, а вихідною – H_i ■

Нехай m – певне фіксоване натуральне число. Послідовності

$$\begin{cases} a_1, a_2, \ldots \in T \\ b_1, b_2, \ldots \in T \\ \ldots \end{cases}$$

називаються *m-рекурентними* відносно конструкторів $f, g, \ldots$, якщо всі їхні члени для $i \ge m+1$ пов'язані співвідношеннями

$$(**)\ a_i = f\left(a_{i-1}, a_{i-2}, \ldots, a_{i-m}, b_{i-1}, b_{i-2}, \ldots, b_{i-m}, \ldots\right),$$
$$b_i = g\left(a_{i-1}, a_{i-2}, \ldots, a_{i-m}, b_{i-1}, b_{i-2}, \ldots, b_{i-m}, \ldots\right),$$
$$\ldots .$$

Щоб повністю задати таку систему послідовностей, достатньо задати перші m членів кожної послідовності $a_1, a_2, \ldots, a_m, b_1, b_2, \ldots b_m, \ldots$. Решту знаходять за допомогою конструкторів. Наприклад, для 2-рекурентної послідовності $a_3 = f_1\left(a_1, a_2, b_1, b_2, \ldots\right)$, $b_3 = f_2\left(a_1, a_2, b_1, b_2, \ldots\right)$ тощо m-рекурентні послідовності можуть індексуватися і з 0. Відомий приклад 2-рекурентної послідовності – числа Фібоначчі ■

Приклад 2.26. Числа Фібоначчі.

За означенням $f_0 = f_1 = 1, f_k = f_{k-2} + f_{k-1}$, $k \ge 2$, тоді $f_2 = f_0 + f_1 = 1 + 1 = 2$, $f_3 = f_1 + f_2 = 1 + 2 = 3$, $f_4 = f_2 + f_3 = 2 + 3 = 5$ тощо ■

Для m-рекурентних послідовностей теж вводиться поняття m-рекурентної функції $\mu_Q^{(*)}$ та μ-оператора. Конструкція аналогічна вищенаведеній, m-рекурентна функція $\mu_Q^{(*)}$ має тип $\left(T^m\right)^n \to T^n$.

Теорема 2.5 (про обчислення m-рекурентних функцій). Параметризовані m-рекурентні функції, і тільки вони, обчислюються детермінованими стандартними програмами.

Доведення. Обмежимося доведенням тільки для випадку непараметризованих функцій і програм. Доведення в загальному випадку суттєво не зміниться (див. вправу 12). Нехай на універсумі T задано довільну непараметризовану m-рекурентну систему (*) виміру n і фільтр $Q: T^n \to \mathrm{Bool}$. Покажемо, як за допомогою стандартної програми обчислити m-рекурентну функцію $\mu_Q^{(*)}$.

Розглянемо спочатку для простоти випадок $m = 1$. Для кожної з n рекурентних послідовностей введемо змінну типу T та її дублікат. Нехай $V =$ **{x,xx,…,z,zz}** – сукупність усіх $2n$ таких змінних, де **xx,…,zz** – дублікати відповідних змінних. Тоді стандартна програма Prog має вигляд:

/*Вх: x,…,z; **Вих: x,…,z*/**

{x:=a_1 **;… z:=**b_1 **;**

```
while (Q(x,...,z)) do
{ /*обчислення нових значень змінних*/
    xx:=f(x,...,z);
      …
    zz:=g (x,..,z);
  /*оновлення змінних*/
     x:=xx;
     …
     z:=zz;
   }
}
```

Присвоювання перед циклом початкових значень змінним називається їх *ініціалізацією.* Чергові нові члени рекурентних послідовностей спочатку зберігаються в дублікатах **xx,…,zz,** і тільки після цього пересилаються в змінні **x,…,z**. Якщо відразу їх записати в змінні, то будуть втрачені попередні значення, від яких може залежати обчислення нових значень у сусідніх послідовностях. Позначимо $\overline{\text{Prog}}$ векторний аналог програми Prog. У нашому випадку тип $\overline{\text{Prog}}$ збігається з $T^n \to T^n$. Покажемо, що $\overline{\text{Prog}}\,(\alpha) = \mu_Q^{(*)}(\alpha)$ для будь-якого набору початкових значень $\alpha = (a_1, b_1, \ldots)$ із T^n. Нехай $\overline{\text{Prog}}\,(\alpha) = (a, b, \ldots)$. Це означає, що існують таке $k \geq 1$ і такі послідовності значень $a_1'(= a_1), a_2', \ldots, a_k'(= a)$ змінної **x** тощо $b_1'(= b_1), b_2', \ldots, b_k'(= b)$ змінної **z**, що

(i) $\forall i = 1, \ldots, k-1 \;\; Q(a_i', \ldots, b_i') = 1$; (ii) $Q(a_k', \ldots, b_k') = 0$.

За побудовою для всіх $1 \leq i \leq k$ значення вектора $\alpha_i' = (a_i', b_i', \ldots)$ збігається зі значенням функції $F(\alpha, i)$, породженої даною системою рекурентних співвідношень. Однак з умов (i) та (ii) випливає: k є найменшим індексом таким, що $Q(a_k', \ldots, b_k') = 0$, тобто $\mu_Q^{(*)}(\alpha) = \alpha_k =$ $= \overline{\text{Prog}}\,(\alpha)$, що й треба було довести.

Аналогічно будується стандартна програма Prog для m-рекурентних функцій. Тільки в цьому разі необхідно зберігати m останніх членів кожної з n m-рекурентних послідовностей. Для цього необхідно ввести $n \times m$ змінних, а також, як і вище, ще n змінних для обчислення нових членів послідовностей. У тілі циклу після визначення нових членів необхідно поновити значення всіх $n \times m$ змінних.

Нехай тепер маємо певну стандартну непараметризовану детерміновану програму Prog.

Покажемо: якщо її складові обчислюють рекурентні функції, то й вона робить те саме. Знову припустимо, що $m = 1$. Для $m > 1$ доведення буде аналогічним. Нехай складовими програми Prog є програми $\text{Prog}_1,\ldots,\ \text{Prog}_n$ з однаковими сукупностями змінних такі, що $\overline{\text{Prog}}_k = \mu_{Q_k}^{(*_k)}, k = \overline{1,n}$ для певних рекурентних систем

$(*_k)\quad a_i = f^{(k)}(a_{i-1}, b_{i-1},\ldots),\ b_i = g^{(k)}(a_{i-1}, b_{i-1},\ldots),\ \ldots,\ i \geq 2$

і фільтрів Q_k. Потрібно за ними побудувати рекурентну систему (*) $a_i = f(a_{i-1}, b_{i-1},\ldots),\ b_i = g(a_{i-1}, b_{i-1},\ldots),\ \ldots,\ i \geq 2$, і визначити фільтр S так, щоб $\overline{\text{Prog}} = \mu_S^{(*)}$.

1) Нехай програма Prog складена: $\{\text{Prog}_1 \text{Prog}_2\}$. Визначимо додаткову рекурентну послідовність: $r_1 = Q_1(a_1, b_1,\ldots)$, $r_i = (r_{i-1} = 0 \to r_{i-1} \mid Q_1(a_i, b_i,\ldots),\ i \geq 2$, для контролю за перебуванням обчислення в просторі систем $(*_1)$ та $(*_2)$, а саме: $r_i = 1$ означає, що виконується Prog_1, у протилежному випадку – Prog_2. Система (*) складається з послідовності r_i і послідовностей систем $(*_1)$ та $(*_2)$, об'єднаних в одну рекурентну систему такими умовними конструкторами:

$$a_i = (r_{i-1} = 1 \to f^{(1)}(a_{i-1}, b_{i-1},\ldots) \mid f^{(2)}(a_{i-1}, b_{i-1},\ldots)), i \geq 2$$

$$b_i = (r_{i-1} = 1 \to g^{(1)}(a_{i-1}, b_{i-1},\ldots) \mid g^{(2)}(a_{i-1}, b_{i-1},\ldots)), i \geq 2$$

$$\ldots$$

.

Візьмемо за фільтр $S = (r = 1 \vee Q_2)$. За побудовою система (*) і фільтр S є шуканими.

2) Нехай програма Prog є розгалуженням $\text{if}(C)\ \text{Prog}_1\ \text{else}\ \text{Prog}_2$. Введемо додаткову контролюючу послідовність $r_1 = \bar{C}(a_1, b_1,\ldots)$, де $\bar{C}$ – векторний аналог умови *C*, $r_i = r_{i-1},\ i \geq 2$, смисл якої той самий, що й у п. 1). Відмінність полягає в тому, що тут r_i є константою. Рекурентною системою (*) для програми Prog теж буде об'єднання відповідних систем $(*_1)$ та $(*_2)$ із доданою послідовністю r_i. Конструктори при цьому отримують охорону – $r = 1$ у системі $(*_1)$ та $r = 0$ у системі $(*_2)$ – і стають умовними, а фільтром S буде формула $(r = 1 \,\&\, Q_1 \vee r = 0 \,\&\, Q_2)$.

3) Аналогічно будується рекурентна система послідовностей для випадку, коли програма Prog є обходом (частковий випадок розгалуження) і вибором. Зазначимо, що охорони у виборі попарно не перетинаються (умова детермінованості Prog). Тоді, як і у 2), для кожної з альтернатив у систему (*) вводиться своя контролююча послідовність $r_i^{(k)}, k = \overline{1,n}$, конструктори мають відповідну охорону, а фільтр S – це диз'юнкція всіх формул вигляду $(r^{(k)} = 1 \,\&\, Q_k)$, $k = \overline{1,n}$, що відповідають контролюючим послідовностям і фільтрам складових програми Prog.

4) Нехай тепер програма Prog є ітерацією while (C) do Prog_1. Система (*) буде збігатися з такою для випадку обходу, а фільтр S – це формула $Q_1 \vee \neg Q_1 \,\&\, \bar{C}$, де $\bar{C}$ – векторний аналог умови C.

5) Випадок, коли програма Prog є повторенням вигляду do Prog_1 while (Q), зводиться до вже розглянутих, оскільки така програма збігається з програмою {Prog_1 while (Q) do Prog_1} ■

Увага! Рекурентні співвідношення (*) та (**) в означенні рекурентних і m-рекурентних послідовностей задають тільки загальний вигляд взаємозв'язку між такими послідовностями. У багатьох випадках цей зв'язок виявляється досить простим. Зокрема, члени рекурентних послідовностей зазвичай залежать не від усіх, а тільки від окремих сусідів. Це дозволяє в програмах для рекурентних функцій шляхом підбору порядку оновлення змінних уникнути дублювання нових членів (усіх чи частини), а отже, і введення змінних-дублікатів (див. прикл. 2.27-2.32). Якщо ж член рекурентної послідовності не залежить від своїх попередників, то немає потреби задавати початковий член для даної послідовності ►

2.5.4. ТЕХНІКА ПОБУДОВИ СТАНДАРТНИХ ПРОГРАМ

Теорема 2.1 відкриває універсальний шлях до побудови стандартної програми, яка обчислює задану залежність між величинами, а саме: спочатку необхідно специфікувати дану залежність як m-рекурентну за допомогою відповідного μ-оператора, а потім за нею побудувати стандартну програму. При цьому теорему можна узагальнити й використовувати як конструктори будь-які похідні операції та предикати базової алгебри. На практиці найчастіше це – складені й умовні операції.

Проілюструємо дану техніку на прикладах із різних предметних областей.

Приклад 2.27. Для даних $x \in R$, $n \in N$ обчислити суму $\sin x + \sin x^2 + ... + \sin x^n$.

Аналіз.

Вхід: $x \in R$, $n \in N$.

Вихід: $s \in R$.

Залежність: $s = \sum_{i=1}^{n} \sin x^i$.

Вимоги: побудувати стандартну програму (Prog9.7) для обчислення s. Базовою Ω-системою є дійсна й натуральна арифметики з операціями та предикатами $+$, $-$, $*$, $/$, %, $=$, $<$, до яких додано функцію sin.

Проектування. Побудувати стандартну програму для обчислення s означає побудувати таку програму, параметризований векторний аналог якої буде повертати як результат s. Побудуємо 1-рекурентну функцію $\mu_Q^{(*)}$ для обчислення s, конструктори якої є похідними операціями базової Ω-системи, і застосуємо до неї теорему 2.5. Ураховуючи, що сума s залежить від входів x та n, додамо до системи (*) вхідні послідовності-константи n та x і опорну послідовність $s_1, s_2, ..., s_n, s_{n+1}, ...$. Спробуємо побудувати систему так, щоб $s_n = s$. Оскільки $s_n = \sum_{i=1}^{n} \sin x^i$, то покладемо $s_k = \sum_{i=1}^{k} \sin x^i$, $k \geq 1$, тобто за члени s_k виберемо часткові суми перших k доданків суми s. Для рекурентності послідовності s_k необхідно знайти залежність суми s_k від попередньої суми s_{k-1}. Ця залежність має вигляд $s_k = s_{k-1} + \sin(x^k)$. Оскільки операція додавання й функція sin є припустимими, то псує дану рекурентну залежність тільки степінь x^k. Уведемо в систему допоміжну рекурентну послідовність $a_k = x^k$, $a_1 = x$, $a_k = a_{k-1} * x$. Тоді $s_k = s_{k-1} + \sin(a_k)$ і конструктор праворуч уже є припустимим. Щоб вибрати саме n-й член опорної послідовності, додамо до системи лічильник $k, k = 1$. Тоді за фільтр Q можна взяти умову $Q(k,n) = k < n$. Дійсно, для $1 \leq k \leq n-1$ $Q(k,n) = 1$ та $Q(n,n) = 0$. Таким чином, побудована система (*) має вигляд

$$\begin{cases} a_1 = x, a_k = a_{k-1} * x; \\ s_1 = \sin(x), s_k = s_{k-1} + a_k; \\ k - \text{лічильник}, k_1 = 1; \\ n - \text{константа}; \\ x - \text{константа}. \end{cases}$$

За побудовою $s = \mu_Q^*(n, x)$.

Кодування. Нехай ініціалізація вигляду **v:=!** означає присвоювання змінній **v** довільного припустимого значення. Для послідовностей системи виберемо змінні з найменуваннями відповідно **a,s,k,n,x**. Серед них **n, x** – вхідні, **s** – вихідна, а **a, k** – робочі змінні. Програма Prog9.7:

Лістинг.

```
/*Вх: x,n          Вих: s */

{x:=!; n:=!;s:=sin(x); k:=1; a:=x;
while (i<n) {a:=a*x;
  s:=s+sin(a);
  k:=k+1;
 }
}
```

Структура програми Prog9.7 вимагає пояснення, оскільки відрізняється від стандартної, застосованої в доведенні теореми 2.5. Як бачимо, член s_k залежить від s_{k-1} та $a_k (a_k = a_{k-1} * x)$ і не залежить від лічильника k, а член a_k і лічильник k залежать тільки від своїх попередників a_k та $k-1$ і не залежать від інших послідовностей. Вибраний порядок зміни значень змінних забезпечує коректне їхнє оновлення й дозволяє не вводити в програмі Prog9.7 відповідні дублікати-змінні ■

Приклад 2.28. Знайти n-те число Фібоначчі, $n \geq 0$ (див. прикл. 2.26).

Аналіз.

Вхід: $n \in N$.

Вихід: $f \in N$.

Залежність: $f = f_n$, де $f_0 = f_1 = 1$, $f_i = f_{i-1} + f_{i=2}$, $i > 1$.

Вимоги: побудувати стандартну програму Prog9.8 для обчислення f. Базовою Ω-системою є натуральна арифметика зі стандартними операціями та предикатами: $+$, $-$, *, /, %, $=$, $<$.

Проектування. Маємо послідовності: опорну f_i та $g_i = f_{i-1}$ і послідовність-константу n. Щоб вибрати саме n-й член опорної послідов-

ності, введемо до системи лічильник i з початковим значенням 2. Тоді за фільтр Q можна взяти умову $i < n$. Таким чином, побудована система (*) має вигляд:

$$\begin{cases} f_0 = f_1 = 1, f_i = f_{i-1} + g_{i-1}, i \geq 2; \\ g_1 = 1, g_i = f_{i-1}, i \geq 2; \\ i_1 = 2, i = (i-1)+1; \\ n - \text{константа}. \end{cases}$$

За побудовою $f_n = \mu_Q^{(*)}(n)$.

Кодування. Виберемо змінні програми Prog9.8: **f,ff,g,i** та **n**. Серед них: **n** – вхідна, **f** – вихідна, а **ff, g** та **i** – робочі змінні. Програма Prog9.8 будується стандартно:

```
/*Вх: n∈N                    Вих: f∈N */

{f:=1; g:=1;
i:=2; n:=!;
    if (n≥2)
 while (i<n)
 {i:=i+1;
  ff:=f+g;
  g:=f;
  f:=ff;
 }
}
```

У програмі використано змінну-дублікат **ff** для змінної **f**. Якщо цього не зробити, то після присвоювання **ff:=f+g** зникне значення f_{i-1}, необхідне для оновлення **g** ■

Приклад 2.29. Для даних $x \in R$, $n \in N$ обчислити суму $x + x^4 + x^9 + \ldots + x^{n^2}$, уникнувши вкладених циклів.

Аналіз.

Вхід: $x \in R$, $n \in N$.

Вихід: $s \in R$.

Залежність: $s = \sum_{i=1}^{n} x^{i^2}$.

Вимоги: побудувати стандартну програму Prog9.9 для обчислення s. Базовою Ω-системою є дійсна й натуральна арифметики зі стандарт-

ними операціями та предикатами. Програма не має містити вкладені цикли, тобто цикли, які є в тілі іншого циклу.

Проектування. Як і у прикл. 2.27, шукана рекурентна система (*) включає три послідовності: вхідні послідовності – константи n та x – і послідовність часткових сум $s_1, s_2, \ldots, s_n, s_{n+1}, \ldots$ таку, що $s_n = s$. Для останньої маємо співвідношення $s_k = s_{k-1} + x^{k^2}$. Уведемо в систему допоміжну послідовність $a_k = x^{k^2}$. Тоді $s_k = s_{k-1} + a_k$ і $a_1 = x$ та $a_k = a_{k-1} * x^{2k-1}$, $k > 1$. Дійсно, якщо поділити a_k на a_{k-1}, то отримаємо саме x^{2k-1}. Знову конструктор містить піднесення до степеня та є неприпустимим у базовій алгебрі. Тому необхідно ввести ще одну додаткову послідовність $b_k = x^{2k-1}$, $k \geq 1$. Маємо $a_k = a_{k-1} * b_k$, $b_k = b_{k-1} \times x^2, b_1 = x.$ Як бачимо, остання послідовність рекурентна з конструкторами – похідними операціями базової алгебри. Отже, побудована система (*) має вигляд:

$$\begin{cases} a_1 = x, a_k = a_{k-1} * b_k; \\ b_1 = x, b_k = b_{k-1} * x^2; \\ s_1 = x, s_k = s_{k-1} + a_k; \\ k - \text{лічильник}, k_1 = 1; \\ n - \text{константа}; \\ x - \text{константа}. \end{cases}$$

За побудовою $s = \mu^*_Q(n, x)$.

Кодування. Змінні програми Prog9.9: **a,b,s,k,n** та **x** – без дублікатів.

```
{n:=!; x:=!;k:=1;s:=a:=b:=x;
while (i<n) {B:=B*X*X;
A:=A*B;
S:=S+A;
K:=K+1;
}
} ■
```

Приклад 2.30. Наближено обчислити суму $\sum_{k=1}^{\infty} (-1)^{[\lg k]} / k$. До кінцевої часткової суми додаються тільки члени, які за модулем строго більші заданого $\varepsilon > 0$. При цьому слід уникати вкладених циклів.

Аналіз.

Вхід: $\varepsilon \in R$.

Вихід: $s \in R$.

Залежність: $s = \sum_{k=1}^{n} (-1)^{[\lg k]} / k$, де n задовольняє умову $\forall 1 \le k \le n \,|\, (-1)^{[\lg k]} / k \,| > \varepsilon \& \,|\, (-1)^{[\lg(n+1)]} / (n+1) \,| \le \varepsilon$.

Вимоги: побудувати стандартну програму Prog9.10 для обчислення s. Інші вимоги – ті самі, що й у прикл. 2.29.

Проектування. Початок такий, як і для обчислення сум. Знову візьмемо три послідовності: вхідну послідовність-константу ε, лічильник k і опорну – із часткових сум $s_1, s_2, \ldots, s_n, s_{n+1}, \ldots$ таку, що $s_n = s$. Маємо $s_1 = 1$ та $s_k = s_{k-1} + (-1)^{[\lg k]} / k$, $k > 1$. Щоб рекурентно подати степінь $(-1)^{[\lg k]}$, розглянемо дві допоміжні послідовності: $a_1 = 0, a_k = [\lg k]$, $k > 1$ та $b_1 = 10, b_k = 10^{(a_k+1)}$, $k > 1$. Тоді

$s_k = s_{k-1} + (a_k \% 2 = 0 \to 1 \,|\, -1) / k$,

$a_k = (k < b_{k-1} \to a_{k-1} \,|\, a_{k-1} + 1)$,

$b_k = (k < b_{k-1} \to b_{k-1} \,|\, b_{k-1} \times 10)$.

Таким чином, маємо потрібну рекурентну систему. Лишилося знайти фільтр Q. Щоб припинити ітерацію, достатньо перевірити на наступному кроці, чи буде модуль нового доданка $(= 1/(k+1))$ більшим ніж ε. Якщо так, то продовжити ітерацію, якщо ні – вийти з неї. Отже, фільтром $Q(k, \varepsilon)$ може бути умова $1/(k+1) > \varepsilon$. Остаточно система (*) має вигляд:

$$\begin{cases} a_1 = 0, a_k = (k < b_{k-1} \to a_{k-1} \,|\, a_{k-1} + 1); \\ b_1 = 10, b_k = (k < b_{k-1} \to b_{k-1} \,|\, b_{k-1} * 10); \\ s_1 = 1, s_k = s_{k-1} + (a_k \% 2 = 0 \to 1 \,|\, -1) / k; \\ k_1 = 1, k = (k-1) + 1; \\ \varepsilon - \text{константа}. \end{cases}$$

За побудовою $s = \mu_Q^*(\varepsilon)$.

Кодування. Змінні програми Prog9.10: **a,b,s,k** та ε – без дублікатів. Сама програма Prog9.10 має вигляд:

/*Вх: $x \in R$, $n \in N$ **Вих**: $s \in R$ ***/**

```
{ε:=!; k:=1;s:=1;a:=0;b:=10;
while (1/(k+1)>ε) {k:=k+1;
a:=(k<b→a |a+1);
b:=(k<b→b | b*10);
s:=s+(a%2=0→1 | -1)/k;
 }
}
```

Тут, як і в усіх попередніх програмах, удалося уникнути введення дублікатів змінних за рахунок спеціального порядку їхнього оновлення: спочатку змінився лічильник, потім – змінна **a** (це принципово, тому що її оновлення залежить від поточного, а не нового значення **b**), а потім, у довільному порядку, – змінні **b** та **s** ■

Приклад 2.31. Швидке піднесення до степеня. Дано $n \in N$. Реалізувати піднесення до степеня в півгрупі $P = (A, \circ)$ з кількістю множень в обчисленні $c(n) = \mathrm{O}(\log_2 n)$.

Аналіз

Вхід: $x \in A$, $n \in N$.

Вихід: $p \in R$.

Залежність: $p = x^n$.

Вимоги: побудувати стандартну програму Prog9.11 для обчислення p із кількістю множень $c(n) = \mathrm{O}(\log_2 n)$. Базовою Ω-системою є півгрупа $P = (A, \circ)$ і натуральна арифметика зі стандартними операціями та предикатами.

Проектування. Якщо вибрати прямий підхід до обчислення степеня й покласти $p_1 = x$, $p_k = p_{k-1} \circ x$, то отримаємо кількість множень $c(n) = n - 1$.

Щоб підвищити ефективність алгоритму, подамо показник степеня n у двійковій системі. Нехай $n = b_0 + b_1 2^1 + b_2 2^2 + \ldots + b_k 2^k$ для певних $b_i \in \{0, 1\}$, де $k = \lfloor \log_2 n \rfloor$. Тоді з урахуванням асоціативності півгрупового множення степінь x^n можна записати як добуток $x^{b_0} \times x^{b_1 \times 2} \times x^{b_2 \times 2^2} \times \ldots \times x^{b_k \times 2^k}$, в якому міститься вже тільки k множень. Розглянемо три послідовності: вхідні послідовності-константи x та n і опорну – із часткових добутків $p_0, p_1, \ldots, p_k$, тобто $p_i \overset{def}{=} \prod_{j=0}^{i} x^{b_j 2^j}$. Тоді

$p_0 = x^{b_0}$, $p_k = p$, $p_i = p_{i-1} \circ x^{b_i 2^i} = p_{i-1} \circ (b_i = 1 \to x^{2^i} \mid 1)$, $i \geq 1$. Нам знадобляться ще три додаткові рекурентні послідовності: $a_i \overset{def}{=} x^{2^i}$, $q_i = q_{i-1} / 2$, $b_i = q_i \% 2$. Дві останні необхідні для відшукання двійкових цифр натурального числа n. При цьому $q_0 = n$, $b_0 = n\%2$, $a_0 = x$ та $a_i = a_{i-1} \circ a_{i-1}$. Дійсно, якщо розділити a_i на a_{i-1}, то отримаємо в результаті a_{i-1}. Залишилось визначитись із фільтром обчислення. Фільтр із лічильником не зовсім влаштовує, оскільки він потребує додаткового обчислювання значення k. Менш очевидним, але набагато простішим, є фільтр $Q(q_i) = (q_i > 0)$. З ним отримуємо систему (*):

$$(*_5)\begin{cases} a_0 = x, a_i = a_{i-1} \circ a_{i-1}; \\ p_0 = (n\%2 = 1 \to x \mid 1), p_i = p_{i-1} \circ (b_i = 1 \to a_i \mid 1); \\ q_0 = n, q_i = q_i / 2; \\ b_0 = n\%2, b_i = q_i\%2; \\ x - \text{константа}; \\ n - \text{константа}. \end{cases}$$

За побудовою $p = \mu^*_Q(x,n)$.

Кодування. Змінні програми Prog9.11 **a,b,x,n,p,q** – без дублікатів. Програма Prog9.11:

```
/*Вх: x∈A, n∈N                    Вих: p∈R */

{x:=!;n:=!;a:=x;q:=n;b:=n%2;
 p:=(n%2=1→x|1);
while (q>0) {a:=a∘a;
q:=q/2;
b:=q%2;
p:=p∘(b=1→a|1);
}
}
```

Її можна дещо удосконалити, усунувши змінну **b**. Ця змінна відіграє роль логічного прапорця, який можна замінити його поточним значенням **q%2**. Новий варіант програми Prog9.11 може бути таким:

```
/*Вх: x∈A, n∈N.                   Вих: p∈R */
```

```
{x:=!;n:=!;a:=x;q:=n;
 p:=(n%2=1→x|1);
while (q>0) do {a:=a∘a;
q:=q/2;
p:=P∘(q%2=1→a|1);
}
}■
```

Нескладно перевірити, що кількість множень $c(n)$ у обчисленнях за програмою перебуває в межах $c(n) \le 2[\log_2 n]$, що задовольняє умову задачі.

Приклад 2.32. Дзеркальне обернення кортежів.

Аналіз.

Вхід: $f \in T^*$, $k \ge 0$.

Вихід: $g \in T^*$.

Залежність: $g = f^R$, де f^R – дзеркальне обернення кортежу f.

Вимоги: побудувати стандартну програму Prog9.12 для обернення кортежів. Нехай $f = [f_1,...,f_n]$. Базовими операціями є: взяття голови $\text{head}(f) = f_1$, взяття хвоста $\text{tail}(f) = [f_2...f_n]$ та pre - додавання компоненти в початок кортежу. Базовим предикатом є рівність $=$.

Проектування. Нехай $f = [f_1,...,f_n]$ і $u_1,...,u_i,...,u_k$ – опорна послідовність така, що $u_k = g$. Обернення кортежів має важливу властивість: $[f_1, f_2,..., f_n]^R = \text{app}\left([f_2,..., f_n]^R, f_1\right)$. Покладемо $u_1 = \text{head}(f)$ і введемо допоміжну послідовність $v_1 = f$, $v_i \overset{def}{=} [f_i,..., f_n]$, $i > 1$. Тоді визначимо кортеж u_i рекурентно як $u_i \overset{def}{=} \text{pre}\left(\text{head}(v_i), u_{i-1}\right)$. Для v_i справедливе співвідношення $v_i = \text{tail}(v_{i-1})$. За фільтр можна взяти предикат $Q(v_i) = []$, де [] – порожній кортеж. Отже, побудована рекурентна система (*) має вигляд:

$$\begin{cases} v_1 = f, v_i = \text{tail}(v_{i-1}); \\ u_1 = \text{head}(f), u_i = \text{pre}\left(\text{head}(v_i), u_{i-1}\right); \\ f - \text{константа}. \end{cases}$$

За побудовою $g = \mu_Q^*(f)$.

Кодування. Змінними в програмі є **v,u,f** без дублікатів. Програма Prog9.12 будується стандартно за рекурентною системою (*):

```
/*Вх: f∈T* .          Вих: u∈T* */

{f:=!;v:=f;u:=head(f);
 while (v≠[]) {v:=tail(v);
 u:=pre(head(v),u));
}
}
```
■

Увага! Коректність побудованих у прикл. 2.27–2.32 програм нескладно довести математично. Для цього достатньо перевірити коректість рекурентних функцій, породжених відповідними системами (*), і потім послатися на теорему 2.5 ►

Повернемося до прикл. 2.27. Небхідно довести, що $s = \mu_Q^*(n,x)$. Для цього достатньо показати, що правильно вибрані конструктори для опорноюї й решти послідовностей, а також фільтр.

Спочатку перевіримо коректність послідовностей $a_1 = x$, $a_k = a_{k-1} * x$ та $s_1 = \sin(x)$, $s_k = s_{k-1} + \sin(a_k)$. Необхідно показати істинність твердження $P(x)$: $\forall k \geq 1 \quad (a_k = x^k \;\&\; s_k = \sum_{i=1}^{k} \sin x^i)$. Скористаємося математичною індукцією за k.

База індукції: $a_1 = x$, $s_1 = \sin(x)$ (за означенням).

Індуктивний перехід. Припустимо, що $a_{k-1} = x^{k-1}$ та $s_{k-1} = \sum_{i=1}^{k-1} \sin x^i$ для довільного $k > 1$. Тоді за рекурентними співвідношеннями: $a_k = a_{k-1} * x$, $s_k = s_{k-1} + \sin(a_k)$ і, з урахуванням індуктивного припущення, $a_k = x^k$, $s_k = \sum_{i=1}^{k-1} \sin x^i + \sin x^k = \sum_{i=1}^{k} \sin x^i$. Таким чином, індуктивний перехід має місце і з повноти індукції випливає $P(x)$. Щоб довести коректність вибору фільтра $Q(k,n) = k < n$, необхідно показати коректність твердження $1 \leq \forall k \leq n-1 \; (k < n) \& (\neg n < n)$. Оскільки $(n < n)$ хибне, то $\neg(n < n)$ – істинне, а істинність першого

коньюнктивного члена випливає з означення лінійного порядку на множині натуральних чисел. Отже, ми довели, що $\mu_Q^*(n,x) = s_n = s$ ■

У загальному випадку в подібних доведеннях застосовується метод структурної індукції, що розглядається в підрозд. 1.4.1.

Простота доведення корректності рекурентних функцій не випадкова. Вона обумовлена близкістю природи рекурентних співвідношень і математичної та структурної індукцій.

Наведена техніка побудови стандартних програм універсальна й може бути покладена в основу технології доказового програмування циклічних програм.

***Література для СР:** СБС і алгоритми – [19, 32, 49, 50]; рекурентні співвідношення й функції – [19, 28, 122].

Контрольні запитання та вправи

1. Дати визначення структурних композицій.
2. Що таке ССП?
3. Як інтерпретуються ССП?
4. Що таке структурний алгоритм і абстрактна структурна програма? Яка між ними різниця?
5. Що таке циклічний структурний алгоритм?
6. Що таке рекурентна послідовність?
7. Що таке конструктор рекурентної послідовності?
8. Дати визначення системи m-рекурентних послідовностей.
9. Що таке рекурентна функція?
10. Дати визначення μ-оператора.
11. Як пов'язані рекурентні функції й циклічні блок-схеми?
12. Довести теорему 2.5 для параметризованих функцій і програм.
13. Показати, що теорема 2.5 залишається вірною, якщо у стандартних програмах зафіксувати базову алгебричну систему, а серед конструкторів у μ-операторах припустити будь-які її похідні операції.
14. Показати, що композиція мінімізації (див. вправу 15 із підрозд. 1.3) є частковим випадком μ-оператора.

    Вказівка. У вправах 15-21 необхідно побудувати стандартні програми з доданими функціями $\cos(x)$, $\sin(x)$ та $\sqrt{\ }$. У вправі 20 мається на увазі стандартний арифметичний базис. Усі числа розглядаються в десятковій системі.

15. Дано $n \in N$.
    а) З'ясувати, чи містить десяткове подання числа n входження цифри 7, і підрахувати кількість таких входжень.
    б) Поміняти місцями першу й останню цифри в числі.
    в) Приписати одиницю ліворуч і праворуч до числа n.
16. Дано $a, b, c, d \in N$. Знайти найбільший спільний дільник цих чисел. Використати алгоритм Евкліда для відшукання НСД двох натуральних чисел за допомогою віднімання.
17. Дано $n \in N$. Обчислити:
    а) $\left(1+\frac{1}{1^2}\right)+\left(1+\frac{1}{2^2}\right)+....+\left(1+\frac{1}{n^2}\right)$;
    б) $\underbrace{\sqrt{2+\sqrt{2+...+\sqrt{2}}}}_{n}$;
    в) $\frac{\cos 1}{\sin 1} \times \frac{\cos 1+\cos 2}{\sin 1+\sin 2} \times ... \times \frac{\cos 1+...+\cos n}{\sin 1+...+\sin n}$;
    г) $\prod_{i=1}^{n}\left(1+\frac{\sin(k \cdot x)}{k!}\right)$.
18. Дано $a \in R, n \in N$. Обчислити:
    а) $\sin x+\sin^2 x+...+\sin^n x$;
    б) $\sin x+\sin 2x+...+\sin nx$;
    б) $\underbrace{\sin x+\sin\sin x+...+\sin...\sin x}_{n}$.
19. Дано $n \in N$. Обчислити:
    а) $n!! = \begin{cases} 1 \times 3 \times ... \times n, \text{ якщо } n - \text{непарне} \\ 2 \times 4 \times ... \times n, \text{ якщо } n - \text{парне} \end{cases}$;
    б) добуток перших n співмножників $\frac{2}{1} \times \frac{2}{3} \times \frac{4}{3} \times \frac{4}{5} \times \frac{6}{5} \times \frac{6}{7} \times ...$
20. Дано $x \in R, n \in N$. Обчислити, уникнувши вкладених циклів:
    а) $\sum_{i=1}^{n} \frac{(-1)^{i(i+1)/2}}{i!}$; б)* $\sum_{k=1}^{n} \frac{(-1)^{[\sqrt{k}]}}{k} x^k$; в)* $\sum_{k=1}^{n} \frac{c(k)}{k^2}$, $c(k)$ – кількість десяткових цифр у числі k; г)* $\sum_{i=1}^{n} x^{i^3}$; д)* $\sum_{i=1}^{n} x^{i^4}$.
21. Дано $x \in R, n \in N$. Реалізувати швидке множення $n \times m$ із кількістю додавань $O([\log_2 n])$.
22. Довести коректність стандартних програм із прикл. 2.28-2.32.

2.6. Індуктивні означення

- Індуктивні означення множин та їхніх систем
- Індуктивні означення функцій
- Рекурсивні специфікації

Ключові слова: *індуктивне означення множини й систем множин, конструктор, повнота, повнота слабка, фільтр, правило виведення, висновок, гіпотеза, аксіома, теорема, індуктивне означення функції й систем функцій, індуктивні функції, правила обчислення значень індуктивних функцій, індуктивне означення функцій* (*систем функцій*) *типу згортки* (*розгортки*), *рекурсивні специфікації, елімінація рекурсії.*

Індуктивні означення множин і функцій відіграють надзвичайно важливу роль у математичній логіці, теорії алгоритмів і програмуванні. Вони лежать в основі рекурсивних методів програмування.

2.6.1. ІНДУКТИВНІ ОЗНАЧЕННЯ МНОЖИН ТА ЇХНІХ СИСТЕМ

Індуктивні означення базуються на понятті алгебричного замикання (див. підрозд. 1.4.1). Тому повернемося до нього й розглянемо детальніше.

Нехай B^* – замикання довільної підмножини $B \subseteq A$ в алгебрі $\boldsymbol{A} = (A;\Omega_F)$.

Лема 2.1. Для замикання B^* має місце рівність $B^* = \bigcup_{i=0}^{\infty} B_i$ (*), де $B_0 = B$, $B_{i+1} = B_i \cup \tilde{B}_i$, а сукупність $\tilde{B}_i$ складають усі можливі значення операцій алгебри $\boldsymbol{A}$ на аргументах із B_i, $i \geq 0$.

Доведення. За побудовою $B_i \subseteq B_{i+1}$, $i \geq 0$. Позначимо $\tilde{B} = \bigcup_{i=0}^{\infty} B_i$. Нескладно перевірити, що $\tilde{B}$ є замкненою надмножиною B і для всіх $i \geq 0$ $B_i \subseteq B^*$ (математична індукція по i), отже, $B^* = \tilde{B}_i$ ■

Як випливає з рівності (*), кожний елемент b системи B^* належить усім множинам B_i, $i \geq k$, для деякого $k \in N$. Тоді за означенням або

$b \in B_{k-1}$, або b побудований з елементів сукупності B_{k-1} за допомогою певної операції алгебри. Це означає, що він або твірний, тобто належить B, або може бути побудованим із твірних за допомогою скінченної кількості застосувань операцій алгебри. Іншими словами, кожний елемент замикання можна ототожнити з його твірними й послідовністю відповідних операцій. Цей факт сформулюємо у вигляді леми.

Лема 2.2. Кожний з елементів замикання B^* є значенням певного Ω-терму.

Доведення. Зафіксуємо певну сигнатуру констант Ω_c для елементів множини $B = B_0$ і сигнатуру функціональних символів $\Omega_f = \left\{ f_1^{n_1}, f_2^{n_2}, \ldots \right\}$ – для операцій алгебри. Тоді за (*) елементам із B_1 відповідають константи з Ω_c і терми вигляду $f_i^{n_i}\left(c'_1, \ldots, c'_{n_i}\right)$, побудовані з термів-констант, елементам із B_2 – терми елементів із B_1 і терми вигляду $f_i^{n_i}\left(t_1, \ldots, t_{n_i}\right)$, де $t_1, \ldots, t_{n_i}$ – терми елементів із B_1; узагалі, елементам із B_i, $i \in N$, відповідають терми елементів із B_{i-1} і побудовані з них терми вигляду $f_i^{n_i}\left(t_1, \ldots, t_{n_i}\right)$ ■

Нехай U – певний універсум елементів. Структура індуктивного означення (ІО) довільної підмножини M елементів універсума має такий вигляд:

(**Б**) **База індукції.** *Фіксується певна підмножина $M_0 \subset U$ апріорі виділених базових об'єктів.*

(**І**) **Індуктивний перехід.** *Вибирається певна сукупність $\{F_i\}$ конструкторів – операцій на універсумі U. Конструктори за базовими й уже побудованими елементами універсума дозволяють отримувати його нові елементи.*

(**ПС**) **Повнота слабка.** *Вибирається певний предикат-фільтр $P(x)$, визначений на універсумі U, який здійснює відбір елементів замикання U_0^* до означуваної множини M: $M \overset{def}{=} P(U_0^*)$.*

Фільтри, подібні $P(x)$, називаються *термінаторами*, а елементи, які вони пропускають – *термінальними.* Наприклад, добре відома роль термінатора $P(x) = "x \in \Sigma^*"$ у КВ-граматиках, який з усіх слів, що виводяться з аксіоми, відбирає слова в основному алфавіті.

Отже, індуктивно означену множину складають термінальні елементи алгебричного замикання.

Увага! Якщо всі базові й усі нові побудовані за ними елементи належать означуваній множині, то $M = U_0^*$ і фільтр стає непотрібним. У цьому випадку (**ПС**) замінюється на повноту (**П**). Якщо в означенні взагалі відсутня згадка про повноту, то це означатиме, що йдеться про повноту (**П**) ►

Індуктивно означені множини будемо називати також просто *індуктивними* й позначати $M = M\left(M_0, \{F_i\}, P\right)$. У математичній логіці конструктори F_i називають *правилами виведення.* Рівність $x = c\left(x_1,...,x_n\right)$, $c \in \{F_i\}$, позначають $x_1,...,x_n \underset{c}{\Rightarrow} x$. При цьому елементи $x_1,...,x_n$ називають *гіпотезами*, x – *висновком* правила c, а базові елементи з M_0 – *аксіомами. Виведенням* елемента a з аксіом називається послідовність елементів $\left\langle a_1,...,a_m = a\right\rangle$ така, що для всіх $0 \le i \le m$ a_i є або аксіомою, або висновком певного правила виведення з гіпотез, що належать $a_1,...,a_{i-1}$. Індуктивно означену множину $M\left(M_0, \{F_i\}, P\right)$ складають *теореми* – ті елементи, що можуть бути виведені з аксіом і є термінальними.

Наведемо кілька важливих прикладів індуктивних множин.

Увага! На практиці при введенні індуктивних означень фіксують певну сукупність базових операцій і за конструктори вибирають ті чи інші похідні операції цієї сукупності. Найчастіше це – складені операції й розгалуження ►

Приклад 2.33. Натуральні числа (IO_1).

(**Б**$_1$) 0 – натуральне число.

(**І**$_1$) Конструктор $s : N \to N$ породжує наступне за порядком число, тобто $s(n) = n + 1$.

Дійсно, $0 \Rightarrow s(0)(=1) \Rightarrow s\left(s(0)\right)(=2)$ тощо ■

Приклад 2.34. Слова (IO_2).

Нехай $\Sigma = \{a_1, a_2,..., a_n\}$ – довільний алфавіт.

(**Б**$_2$) $\varepsilon \in \Sigma^*$ – порожнє слово й літери алфавіту Σ є базовими.

(**І**$_2$) Конструктор $\text{pre} : \Sigma \times \Sigma^* \to \Sigma^*$ визначимо так:

$$\text{pre}(a, b_1 b_2 ... b_m) = a b_1 b_2 ... b_m .$$

(**ПС**$_2$) Фільтр-термінатор $P(x) \Leftrightarrow x \in \Sigma^*$.

Як бачимо, не всі базові елементи термінальні. Односимвольні слова за даним означенням мають вигляд $\mathrm{pre}(a,\varepsilon), a \in \Sigma$. Побудуємо виведення слова $abba$ в означенні ІО$_2$:

$$\langle a,b,\varepsilon,\mathrm{pre}(a,\varepsilon),\mathrm{pre}\big(b,\mathrm{pre}(a,\varepsilon)\big)(=ba),\mathrm{pre}\big(b,\mathrm{pre}\big(b,\mathrm{pre}(a,\varepsilon)\big)\big)(=bba),$$
$$\mathrm{pre}\big(a,\mathrm{pre}\big(b,\mathrm{pre}\big(b,\mathrm{pre}(a,\varepsilon)\big)\big)\big)=(abba)\rangle.$$

Наведемо інше індуктивне означення (ІО$_{2'}$) слів над Σ:

(**Б**$_{2'}$) $\varepsilon \in \Sigma^*$ – порожнє слово й літери алфавіту Σ є базовими.

(**І**$_{2'}$) Визначимо конструктор $app : \Sigma^* \times \Sigma \to \Sigma^*$:

$$\mathrm{app}\big(b_1b_2\ldots b_m,a\big) = b_1b_2\ldots b_ma.$$

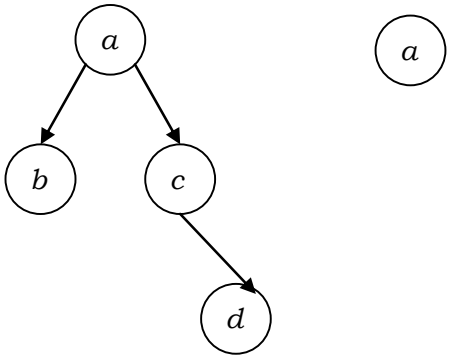
(**ПС**$_2$) Фільтр-термінатор $P(x) \Leftrightarrow x \in \Sigma^*$ ■

Приклад 2.35. Бінарні дерева у префіксному поданні (Б-дерева) (ІО$_3$). Нехай $A = \{a_1,\ldots,a_n\}$ – сукупність вершин дерев.

(**Б**$_3$) $\varnothing$ – порожнє Б-дерево.

(**І**$_3$) Для вершини $a \in A$ і довільних Б-дерев d_1, d_2 вираз $a(d_1,d_2)$ є Б-деревом.

Бінарним деревам на рисунку за означенням ІО$_3$ відповідають Б-дерева $a\big(b(\varnothing,\varnothing),c\big(\varnothing,d(\varnothing,\varnothing)\big)\big)$ та $a(\varnothing,\varnothing)$.



■

Звернемося тепер до індуктивних означень систем множин (ІОС). Нехай $P_1(x_1),\ldots,P_k(x_k)$, $k \geq 1$, – фільтри на універсумі U, що виділяють у ньому певні підмножини – *сорти* елементів $U_1,\ldots,U_k$. При цьому

області істинності сортових фільтрів можуть перетинатися. Індуктивне означення системи множин $\langle M_1,...,M_k\rangle$ має вигляд:

(**БС**) Фіксується певна підмножина $M_0 \subset U$ апріорі виділених базових об'єктів.

(**ІС**) Вибирається певна сукупність $\{F_i\}$ конструкторів – операцій на універсумі U. Конструктори за базовими й уже побудованими елементами дозволяють отримувати нові елементи універсума.

(**ПС**) Вибираються фільтри $P_1(x),...,P_k(x)$, $k \geq 1$, що виділяють в універсумі U сорти термінальних елементів $T_1,...,T_k$ і здійснюють відбір елементів до означуваної системи множини: $M_i \overset{def}{=} P_i(U_0^*)\ 1 \leq \forall i \leq k$.

Приклад 2.36. КВ-мови.

Покажемо, як за допомогою ІОС можна задавати КВ-мови в даному алфавіті $T = \{a_1, a_2, ..., a_n\}$. Нехай $N = \{A_1, A_2, ..., A_m\}$ – певний алфавіт нетерміналів. Універсум U складають натуральні числа й сукупність $(T \cup N)^*$ усіх слів у алфавіті основних і допоміжних символів. Предикат $Q_i(x)$, $i = \overline{1,m}$, буде виділяти сорт слів у об'єднаному алфавіті, що можуть бути виведені з аксіоми A_i. Розглянемо систему мов $\langle L_1,...,L_m\rangle$, $L_i \subseteq \Sigma^*$, що визначається таким ІОС:

(**БС**$_4$) Усі натуральні числа є базовими. Кожному нетерміналу $A_i \in V$, $i = \overline{1,m}$, поставимо у відповідність скінченну непорожню підмножину слів $R_i = \{u_1,...,u_{r_i}\} \subset (\Sigma \cup V)^*$. Слова з R_i, $i = \overline{1,m}$, – базові. За означенням $Q_i(u_j) = 1$ для всіх $j = \overline{1,r_i}$.

(**ІС**$_4$) Єдиний конструктор $\mathrm{Sub}: \Sigma^* \times N \to \Sigma^*$ кожному числу $k \geq 0$ і слову вигляду $u = vA_iw$, де $v \in T^*$ – префікс u довжиною k, A_i – допоміжний символ, ставить у відповідність слова вигляду vu_jw, $j = \overline{1,r_i}$. Конструктор залишає слово u без змін у разі відсутності в ньому термінального префікса довжиною k і наступного за ним допоміжного символу. За означенням конструктор не змінює сорт слова, тобто слово u передає у спадок новому слову vu_jw свій сорт.

(**ПС**$_4$) Фільтрами є предикати $P_i(x) \Leftrightarrow Q_i(x)\ \&\ x \in \Sigma^*$, $i = \overline{1,m}$.

Нескладно перевірити, що мови L_i, $i=1,m$, збігаються із сукупністю $L(A_i)$ усіх слів у алфавіті Σ, що можуть бути лівосторонньо виведені у КВ-граматиці з правилами виведення $A_i \to u_j$, $i=\overline{1,m}$, $j=\overline{1,r_i}$, із нетермінала A_i як аксіоми ■

2.6.2. ІНДУКТИВНІ ОЗНАЧЕННЯ ФУНКЦІЙ

Індуктивно означені функції й операції (ІОФ) визначаються на індуктивних скінченно означених областях. Необхідно, щоб кожний елемент області визначення можна було розкласти на близькі до нього один або кілька елементів того самого типу. Наприклад, число $n+1$ розкладається на числа n та 1. ІОФ теж має базу індукції (**БФ**), індуктивний перехід (**ІФ**) і повноту (**ПФ**). Розглянемо їх спочатку на прикладі основних операцій на словах.

Операція $\circ : \Sigma^* \times \Sigma^* \to \Sigma^*$ – конкатенація слів:

$$(\mathbf{БФ}_5)\ \varepsilon \circ w_2 = w_2 \circ \varepsilon = w_2 .$$

$(\mathbf{ІФ}_5)\ \mathrm{pre}(a, w_1) \circ w_2 = \mathrm{pre}(a, w_1 \circ w_2)$ для кожного $a \in \Sigma$, $w_1, w_2 \in \Sigma^*$.

Означення коректне. Воно спирається на той факт, що кожне слово складається з його першої літери та хвоста (слова без першої літери). Дійсно, знайдемо, наприклад, значення

$$cb \circ dd = \mathrm{pre}(c,b) \circ dd = \mathrm{pre}(c, b \circ dd) = \mathrm{pre}(c, \mathrm{pre}(b,\varepsilon) \circ dd) =$$

$$= \mathrm{pre}(c, \mathrm{pre}(b, \varepsilon \circ dd)) = \mathrm{pre}(c, \mathrm{pre}(b, dd)) = \mathrm{pre}(c, bdd) = cbdd .$$

Операція $\mathrm{head} : \Sigma^* \to \Sigma$ повертає перший символ (голову) слова, операція $\mathrm{bottom} : \Sigma^* \to \Sigma$ – останній символ слова:

$$(\mathbf{БФ}_6)\ \mathrm{head}(\varepsilon) = \mathrm{bottom}(\varepsilon) = \# .$$

$(\mathbf{ІФ}_6)\ \mathrm{head}(aw) = a$, $\mathrm{bottom}(a\varepsilon) = a$, $\mathrm{bottom}(aw) = \mathrm{bottom}(w)$.

Операція $\mathrm{tail} : \Sigma^* \to \Sigma^*$ повертає хвіст слова, тобто слово без його голови:

$$(\mathbf{БФ}_7)\ \mathrm{tail}(\varepsilon) = \varepsilon\ ,\ (\mathbf{ІФ}_7)\ \mathrm{tail}(aw) = w .$$

Операція $\mathrm{lead} : \Sigma^* \to \Sigma^*$ повертає слово без останнього символу:

$$(\mathbf{БФ}_8)\ \mathrm{lead}(\varepsilon) = \varepsilon\ \ (\mathbf{ІФ}_8),\ \mathrm{lead}(a\varepsilon) = \varepsilon\ ,\ \mathrm{lead}(aw) = \big(w = \varepsilon \to \varepsilon \mid a\,\mathrm{lead}(w)\big) .$$

Функція $|\ | : \Sigma^* \to N$ обчислює довжину слова:

$$(\mathbf{БФ}_9)\ |\varepsilon| = 0\ ,\ (\mathbf{ІФ}_9)\ |aw| = 1 + |w| .$$

Предикат $\prec : \Sigma^* \times \Sigma^* \to \text{Bool}$ визначає лінійний лексикографічний порядок (нестрогий) на словах над алфавітом $\Sigma = \{a_1, a_2, ..., a_n\}$. Нагадаємо, що символи алфавіту Σ лінійно впорядковані: $a_i \le a_j \Leftrightarrow i \le j$ для $1 \le i, j \le n$. Тоді для довільних $v, u \in \Sigma^+$:

$$(\mathbf{БФ}_{10})\ (\varepsilon \prec \varepsilon) = (\varepsilon \prec v) = 1, (u \prec \varepsilon) = 0,$$

$$(\mathbf{ІФ}_{10})\ u \prec v =$$

$$= \left(\text{head}(u) = \text{head}(v) \to \text{tail}(u) \prec \text{tail}(v) \mid \text{head}(u) \le \text{head}(v)\right).$$

Наприклад, для довільних символів a, b таких, що $a < b$:

$$(\varepsilon \prec a) = 1, (b \prec \varepsilon) = 0,\ (ab \prec ab) = (b \prec b) = (\varepsilon \prec \varepsilon) = 1,$$

$$(ab \prec aba) = (b \prec ba) = (\varepsilon \prec a) = 1,\ (abba \prec aaa) = (bba \prec aa) = (b \prec a) = 0.$$

Нехай $M = M(M_0, \{F_i\}, P)$ – певна індуктивна множина зі скінченними множинами твірних і конструкторів.

Загальне ІОФ функції $f : M^m \to A$, визначеної на M, має вигляд:

(**БФ**) **База індукції.** $\forall x \in M_0 : f(x_1, ..., x_{m-1}, y) = g(x_1, ..., x_{m-1}, y)$, *де* g *– відома функція, визначена на підмножині* $M^{m-1} \times M_0$.

(**ІФ**) **Індуктивний перехід.** *Для кожного конструктора* $c : M^n \to M$ *із* $\{F_i\}$ *існує конструктор значень функції* $h_c : M^{n+m} \times A^n \to A$ *такий, що* $f(x_1, ..., x_{m-1}, c(y_1, ..., y_n)) =$ $= h_c(x_1, ..., x_{m-1}, y_0, ..., y_n, f(x_1, ..., x_{m-1}, y_1), ..., f(x_1, ..., x_{m-1}, y_n))$ *для будь-яких* $x_1, ..., x_{m-1}, y_0, ..., y_n \in M, y_0 = c(y_1 ..., y_n)$.

(**ПФС**) **Повнота слабка.** *Вибирається певний фільтр* $P(x_1, ..., x_{m-1}, y)$, *визначений на універсумі* $M^m \times A$, *який здійснює відбір результатів означуваної функції.*

Увага! Деякі аргументи в конструкторі h_c можуть бути фіктивними, а за конструктор h_c можна брати й саму означувану функцію f. Найчастіше конструктор є умовним, а взагалі він може мати довільну регулярну структуру ►

За лемою 2.1 $M = \bigcup\limits_{i=0}^{\infty} M_i$, де $M_{i+1} = M_i \cup \tilde{M}_i$, а сукупність $\tilde{M}_i$ складають усі можливі значення конструкторів $\{F_i\}$ на аргументах із M_i, $i \geq 0$. ІОФ визначає функцію $f : M^m \to A$, графік якої $\Gamma(f)$ задається рівностями:

1) $\Gamma(f) \overset{def}{=} P(H)$;

2) $H = \bigcup\limits_{i=0}^{\infty} H_i$, де $H_0 = \Gamma(g)$, $H_{i+1} = H_i \cup \tilde{H}_i$, $i \geq 0$,

$$\tilde{H}_i = \{(x_1,...,x_{m-1}, c(y_1,...,y_n),\ h_c(x_1,...,x_{m-1}, y_0,...,y_n, z_1,...,z_n)) :$$
$$x_1,...,x_{m-1} \in M, y_1,...,y_n \in M_i,\ y_0 = c(y_1,...,y_n), c \in \{F_i\},$$
$$(x_1,...,x_{m-1}, y_1, z_1),...,(x_1,...,x_{m-1}, y_n, z_n) \in H_i)\}.$$

Такі ІОФ задають операцію множення натуральних чисел через додавання:

$$(\textbf{БФ})\ (x \times 0) = 0,$$
$$(\textbf{ІФ})\ (x \times s(y)) = (x \times y) + x$$

і відому композицію примітивної рекурсії для n-арної натуральної функції f:

(**БФ**) $f(x_1,...x_{n-1}, 0) = g(x_1,...x_{n-1})$ для деякої функції g;

(**ІФ**) $f(x_1,...x_{n-1}, s(y)) = h(x_1,...x_{n-1}, y,\ f(x_1,...x_{n-1}, y))$ для деякого конструктора h.

Приклад 2.37. Задача про Ханойські вежі.

Є три упорядкованих кілочки, на першому з яких розташована вежа з n дисків, яка має вигляд конуса. За один хід можна взяти верхній диск із будь-якого кілочка й покласти як верхній на один із решти кілочків за умови, що на останньому не буде порушено умову конусності його вежі. Необхідно перенести задану вежу на другий або третій кілочок. Нехай, наприклад, це буде третій кілочок.

Покладемо $T = \{1,2,3\}$ – номери кілочків.

Нехай $Hanoi : T \times T \times N \to (T \times T)^*$ – функція, що задає послідовність дій із переміщення вежі висотою n з одного кілочка на інший. Для подання однієї такої дії достатньо вказати пару чисел вигляду <звідки, куди>, що належать $T \times T$. Нехай "$\circ$" – операція конкатенації двох послідовностей. $ІОФ_{11}$ функції Hanoi основане на індуктивній природі піраміди: в якому б місці її не поділити навпіл – обидві час-

тини будуть пірамідами, зокрема, якщо взяти її основу й ту частину, що зверху неї. Щоб перенести піраміду з кілочка x на кілочок y, достатньо: 1) перенести верхню її частину висотою $n-1$ на третій кілочок із номером $6-x-y$; 2) основу перенести з x на y; 3) перенести всю піраміду висотою $n-1$ із кілочка $6-x-y$ на y. Це можна записати так:

$$(\mathbf{БФ}_{11})\ \mathrm{Hanoi}(x,y,1)=(x,y).$$

$$(\mathbf{ІФ}_{11})\ \mathrm{Hanoi}(x,y,s(n))=\mathrm{Hanoi}(x,6-x-y,n)\circ$$
$$\circ\left[(x,y)\right]\circ\mathrm{Hanoi}(6-x-y,y,n).$$

Коректність ІОФ_{11} просто довести структурною індукцією за n (див. вправу 15) ■

Індуктивно означені функції будемо називати просто *індуктивними.*

Зупинимось тепер на правилах обчислення значень індуктивних функцій. Ураховуючи, що замкнення M скінченно породжене і його конструкторів скінченна кількість, для обчислення значення індуктивної функції $f(x_1,\ldots,x_{m-1},a)$ можна було б послідовно будувати всі множини H_i, починаючи з H_0, доти, доки не буде побудовано пару $(a,f(a))\in H_n$ для деякого n. Однак така процедура в більшості випадків буде вкрай неефективна.

Раціональніше було б не генерувати виведення всіх можливих елементів замкнення, а знайти одне з виведень саме елемента a в замкненні M і побудувати за ним значення $b=f(x_1,\ldots,x_{m-1},a)$ за допомогою відповідних конструкторів. Цю побудову, у свою чергу, можна подати у вигляді певного виведення на множині термів $<t_0=f(x_1,\ldots,x_{m-1},a),t_1,t_2,\ldots,t_n=b>$. Правило виведення (отримання терму t_{i+1} із терму t_i) може бути таким: усі виклики функції f у t_i одночасно розкриваються відповідно до ІОФ, тобто виклики $f(x_1,\ldots,x_{m-1},x),x\in M_0$, замінюються на терм $g(x_1,\ldots,x_{m-1},x)$, а виклики $f(x_1,\ldots,x_{m-1},c(y_1,\ldots,y_n))$ – на терм $h_c(x_1,\ldots,x_{m-1},y_0,\ldots,y_n,f(y_1),\ldots,f(y_n))$, після чого вже у t_{i+1} розкриваються всі виклики базових функцій $g(x_1,\ldots,x_{m-1},x)$ і предикатів (в умовних виразах) – вони замінюються на їхні значення, а умовні підтерми з булевими константами на місці умов – на альтернативи. Виведення або закінчується термом t_n, який не містить входжень f і збігається (за побудовою) з b, або воно є нескінчен-

ним, що означає невизначеність функції f на аргументі $(x_1,...,x_{m-1},a)$. Таке правило виведення називається *повною підстановкою.* Обчислимо за його допомогою, наприклад, значення $x\times 3$ через додавання: $t_0=(x\times 3)$, $t_1=((x\times 2)+x)$, $t_2=(((x\times 1)+x)+x)$, $t_3=((((x\times 0)+x)+x)+x)$, $t_4=(((0+x)+x)+x)=3x$.

Використовують також інші правила виведення для обчислення значень індуктивної функції. Найвідоміші з них: а) правило найбільш лівої-найбільш внутрішньої заміни – виклик за значенням (замінюється тільки один найбільш лівий найбільш внутрішній виклик функції f); б) правило найбільш лівої заміни – виклик за іменем (замінюється тільки один найбільш лівий виклик функції f) тощо. Слід мати на увазі: якщо конструктор h умовний, то результати застосування наведених вище правил виведення можуть не збігатися (див. вправу 10).

На практиці, щоб скоротити пошук виведення елемента a, намагаються максимально спростити будову замкнення M. Для цього насамперед обмежуються *однозначними* замиканнями. Індуктивні функції в загальному випадку недетерміновані. Причина в тому, що елемент індуктивної множини $M=M\left(M_0,\{F_i\},P\right)$ може мати кілька суттєво різних виведень з аксіом. Два виведення в M називаються *еквівалентними*, якщо одне з них може бути отримане з іншого перестановкою його членів. Індуктивна множина $M=M\left(M_0,\{F_i\},P\right)$ називається *однозначною*, якщо в ній відсутні елементи, що мають нееквівалентні виведення з аксіом. Однозначність індуктивної множини гарантують такі три умови (але вони не є необхідними):

1) області значень будь-яких двох конструкторів c_1 та c_2 із $\{F_i\}$ не перетинаються;

2) для кожного $x\in M$ існує тільки одна сукупність гіпотез $x_1,...,x_n$ така, що $x=c\left(x_1,...,x_n\right)$ для конструктора c;

3) M_0 – найменша та єдина множина твірних у M.

Умова 2) виконується для однозначних конструкторів. Якщо звернутися до наведених вище індуктивних означень IO_1–IO_3 натуральних чисел, слів і Б-дерев, то всі вони задовольняють умови 1)–3) і тому є однозначними. Наступне ж означення натуральних чисел:

(**Б**) 0 та 1 – натуральні числа,

(**І**) конструктор $\text{s}\left(n\right)=n+1$

– неоднозначне. Дійсно, кожне число $n > 0$ за ним має два нееквівалентні виведення, що розпочинаються з 0 та 1. Наприклад, для 1 – це виведення $\langle 1 \rangle$ та $\langle 0, \mathrm{succ}(0) \rangle$, для 2 – $\langle 1, \mathrm{succ}(1) \rangle$ та $\langle 0, \mathrm{succ}(0), \mathrm{succ}(\mathrm{succ}(0)) \rangle$ тощо.

Приклад 2.38. ІОФ_{12} спирається на індуктивне означення слів ІО_2 і задає функцію $|\,|: \Sigma^* \to N$, що обчислює довжину слова:

(**БФ**$_{12}$) $|\varepsilon| = 0$, (**ІФ**$_{12}$) $|\,|\mathrm{pre}(a, w)| = |w| + 1$ ■

Приклад 2.39. Обчислення ваги зваженого Б-дерева.

Нехай БД_A – сукупність усіх зважених Б-дерев із вершинами з A. Індуктивна функція $\mathrm{Wage}: \text{БД}_A \to A$ задає вагу Б-дерева:

(**БФ**$_{13}$) $\mathrm{Wage}(\varnothing) = 0$,

(**ІФ**$_{13}$) $\mathrm{Wage}(a(d_1, d_2)) = a + \mathrm{Wage}(d_1) + \mathrm{Wage}(d_2)$ ■

За означенням індуктивна функція f здійснює індуктивний перехід тільки за одним, не обов'язково останнім, аргументом, а решта можуть розглядатися як параметри. Аналогічний вигляд мають індуктивні означення функцій, в яких індуктивний перехід здійснюється одночасно за кількома аргументами, а також систем взаємно індуктивних функцій. Останні особливо важливі в застосуваннях. Обмежимося для простоти випадком бінарних функцій. У загальному випадку означення будуть аналогічними.

Нехай $M = M(M_0, \{F_i\}, P)$ – довільна індуктивна множина. Домовимося послідовність вигляду $a_{11}, a_{12}, \ldots, a_{1m}, \ldots, a_{n1}, a_{n2}, \ldots, a_{nm}$ записувати скорочено: $\left[a_{ij}\right]_{i,j=1}^{n,m}$. Вираз $\left[a_{ij}\right]_{i,j=0}^{n,m}$ означає аналогічну послідовність з індексами, що розпочинаються не з 1, а з 0, але без елемента a_{00}. Нехай $\bar{n} = n + 1, \bar{m} = m + 1$.

Загальне індуктивне означення бінарної функції $f: M \times M \to A$ має вигляд:

(**БФ**) $\forall \langle x, y \rangle \in D_0 : f(x, y) = g(x, y)$, де $g(x, y)$ – *відома функція, задана на множині* $D_0 \subset M \times M$, *яка є надмножиною множини* $M_0 \times M \cup M \times M_0$.

(**ІФ**) *Для кожної пари конструкторів* $c: M^n \to M$, $d: M^m \to M$ *із* $\{F_i\}$ *існує конструктор* $h_{c,d}: M^{\bar{n}} \times M^{\bar{m}} \times A^{\bar{n}\times\bar{m}-1} \to A$ *значень функції такий, що для будь-яких* $x_1,...,x_n, y_1,...,y_n \in M$ *та* $x_0 = c(x_1,...,x_n) \in M$, $y_0 = d(y_1,...,y_m) \in M$:

$$f\left(c(x_1,...,x_n), d(y_1,...,y_m)\right) = h_{c,d}\left(x_0,...,x_n, y_0,...,y_m, \left[f(x_i,y_j)\right]_{i=0,j=0}^{n,m}\right).$$

(**ПФС**) *Вибирається певний фільтр* $P(x,y,z)$, *визначений на універсумі* $M \times M \times A$, *що здійснює відбір результатів означуваної функції.*

Шляхом фільтрації множини $M = M\left(M_0, \{F_i\}, P\right)$ можна за необхідності типізувати означувану функцію f . Зауважимо, що на практиці індуктивна функція може мати інші аргументи, за якими не здійснюється індукція, конструктори $h_{c,d}$ досить прості й більшість їхніх аргументів – фіктивні.

Приклад 2.40. ІОФ_{14} задає ще один варіант визначення множення натуральних чисел через додавання з індуктивним переходом за обома аргументами:

(**БФ**$_{14}$) $(x\times 0) = (0\times y) = 0$, (**ІФ**$_{14}$) $\left(\mathrm{s}(x)\times \mathrm{s}(y)\right) = x\times y + x + y + 1$.

Єдиним конструктором $h(x_1,x_2,x_3)$ тут є операція $x_1 + x_2 + x_3 + 1$` ■

Увага! Кількість індуктивних переходів при обчисленні добутку $x\times y$ в ІОФ_{14} збігається з величиною $\min\{x,y\}$, тоді як ця величина у випадку ІОФ з індуктивним переходом тільки за y збігається з y . З урахуванням комутативності множення цю різницю (яка в багатьох випадках досить суттєва) можна нівелювати, помінявши попередньо місцями аргументи. Однак у загальному випадку ІОФ за кількома аргументами може виявитись набагато ефективнішим, ніж еквівалентне визначення за одним аргументом ►

Як і вище, за конструктор $h_{c,d}$ можна брати й саму функцію f . Це значно розширює можливості ІОФ. Наведемо один із таких прикладів.

Приклад 2.41. Функції Аккермана $\mathrm{B}(x,y)$.

Вони відомі тим, що надзвичайно швидко зростають.

(**БФ**$_{15}$) $\mathrm{B}\left(\mathrm{s}(x),0\right) = \mathrm{sg}(x)$, $\mathrm{B}(0,y) = 2 + y$.

(**ІФ**$_{15}$) $\mathrm{B}\left(\mathrm{s}(x),\mathrm{s}(y)\right) = \mathrm{B}\left(x,\mathrm{B}\left(\mathrm{s}(x),y\right)\right)$.

В індуктивному переході (**ІФ**$_{15}$) перший аргумент $(x_0 = \mathrm{s}(x))$ функції B ліворуч входить і в праву частину. Проте важливо, що при цьому другий аргумент у внутрішньому входженні B праворуч спростився: був $\mathrm{s}(y)$, а став y. Аналогічно в зовнішньому входженні B праворуч спростився перший аргумент. Це скеровує обчислення значення функції до бази визначення (**БФ**$_{15}$). Можна перевірити, наприклад, що $\mathrm{B}(3,1) = 2$, $\mathrm{B}(3,2) = 2^2$, $\mathrm{B}(3,3) = 2^{2^2}$ тощо ■

Індуктивні означення систем функцій (ІОФС) розглянемо на прикладі бінарних функцій типу $M \times M \to A$. ІОФС системи бінарних функцій $[f_1,\ldots f_l], l > 0$, має структуру:

(**БФС**) *Для* $x,y \in D_0 : f_k(x,y) = g_k(x,y)$, *де* $g_k(x,y)$ – *відома функція, задана на множині* $D_0 \subset M \times M$, *яка є надмножиною множини* $M_0 \times M \cup M \times M_0$, $k = 1,l$.

(**ІФС**) *Для кожної функції* f_k, $k = \overline{1,l}$, *і пари конструкторів* $c : M_1{}^n \to M_1$ *та* $d : M_2{}^m \to M_2$ *існує конструктор* $h_{c,d,k} : M_1{}^n \times M_2^m \times A^{n\times m\times k} \to A$ *значень функції такий, що для будь-яких* $x_1,\ldots,x_n \in M_1$, $y_1,\ldots,y_m \in M_2$ *та* $x_0 = c(x_1,\ldots,x_n) \in M_1$, $y_0 = d(y_1,\ldots,y_m) \in M_2$:

$$f_k\left(c(x_1,\ldots,x_n), d(y_1,\ldots,y_m)\right) =$$

$$= h_{c,d,k}\left(x_0,\ldots,x_n,y_0,\ldots,y_m,\left[f_1(x_i,y_j)\right]_{i,j=0}^{n,m},\ldots,\left[f_k(x_i,y_j)\right]_{i,j=0}^{n,m}\right).$$

(**ПФС**) *Вибираються фільтри* $P_k(x,y,z)$, $k = \overline{1,l}$, *визначені на універсумі* $M_1 \times M_2 \times A$, *що здійснюють відбір результатів означуваних функцій.*

Приклад 2.42. Обчислення кількості комутативних розбиттів натурального числа.

Позначимо цю кількість $f(n)$. Тоді $1+1+1 = 1+2 = 3$ та $f(3) = 3$ (розбиття $1+2$ та $2+1$ не розрізняються). Розглянемо допоміжну функцію $q : N \times N \to N$. Функція $q(n,m)$ – це кількість комутативних роз-

биттів числа n із доданками, що не перевищують m. Нескладно перевірити, що наступне ІОФС задає пару функцій $[f,q]$:

(**БФС**$_{18}$) $q(1,m)=q(n,1)=1$, $f(1)=1$.

(**ІФС**$_{18}$) $q(n,m)=\begin{bmatrix}(m\ge n\to 1+q(n,n-1)|\\ q(n-m,m)+q(n,m-1))\end{bmatrix}$, $f(n)=q(n,n)$ ■

Усі наведені ІОФ мають спільну рису – значення індуктивної функції на своїх аргументах при індуктивному переході визначаються за допомогою її ж значень на простіших аргументах. Таку індукцію будемо називати індукцією *типу згортки*. Іншим, дуальним, варіантом індуктивних означень є означення функцій типу *розгортки*, коли значення функції на аргументах при індуктивному переході визначаються за допомогою її ж значень на заданих або нових аргументах, отриманих за допомогою конструкторів області визначення функції. У цьому випадку база індукції задає умову завершення індуктивних переходів по зовнішньому фронту області визначення функції.

Проілюструємо цей тип індукції на прикладі 91-шої функції Мак-Картні, яка задається таким ІОФ (див. підрозд. 1.3.3):

(**БФ**) $f_{91}(x)=x-10$, для $x>100$;

(**ІФ**) $f_{91}(x)=f_{91}(f_{91}(x+11))$ для $x\le 100$.

Тут конструктор h збігається з функцією f_{91} і застосовується до значення означуваної функції на новому аргументі $x+11$. Як і у випадку індукції типу згортки, для обчислення значення такої функції f_{91} можна застосувати правило повної підстановки. Обчислимо, наприклад, значення $f_{91}(99)$: $t_0=f_{91}(99)$, $t_1=f_{91}(f_{91}(110))$, $t_2=f_{91}(f_{91}(110))=f_{91}(100)=f_{91}(f_{91}(111))$, $t_3=f_{91}(101)=91$.

Загальне індуктивне означення функцій *типу розгортки* (ІОФР) розглянемо на прикладі бінарної функції $f:M\times M\to A$. Воно має вигляд:

(**БФР**) *Для* $x,y\in D_0: f(x,y)=g(x,y)$, *де* $g(x,y)$ – *відома функція, задана на деякій підмножині пар* $D_0\subseteq M\times M$.

(**ІФР**) *Є один або кілька конструкторів значень функції вигляду* $h:M^n\times A^{n\times m}\to A$ *таких, що для будь-яких* $(x,y)\in M\setminus D_0: f(x,y)=h\left(z_1,\ldots,z_n,\left[f(z_i,z_j)\right]_{i=1,j=1}^{n,m}\right)$, *де* $z_1,\ldots,z_k$ *або збі-*

гаються з одним із x, y, *або є теоремами, виведеними в* M *із базових елементів і гіпотез* x, y.

(**ПФР**) *Вибирається певний фільтр* $P(x, y, z)$, *визначений на універсумі* $M \times M \times A$, *що здійснює відбір результатів означуваної функції.*

Як ми вже бачили, конструктор h може збігатися з функцією f, а більшість його параметрів на практиці є фіктивними. Як і раніше, функція f може мати інші аргументи, за якими не здійснюється індукція.

Аналогічний вигляд мають індуктивні означення *систем функцій типу розгортки* (ІОФСР). Розглянемо два приклади побудови таких індуктивних означень.

Приклад 2.43. Розкладання числа на прості множники.

Знайдемо систему функцій $[f, g]$, $f : N \to N^*, g : N \times N \to N^*$, в якій f продукує розкладання натурального числа на прості множники, а g є допоміжною функцією, що знаходить усі дільники даного числа, які не менші ніж другий аргумент. Наприклад, $f(24) = [2,2,2,3]$.

(**БФСР**$_{16}$) $f(1) = g(1, x) = \varepsilon$ для всіх $x \in N$, $g(x, y) = \mathrm{pre}(x, \varepsilon)$ для всіх $1 < x < y^2$ і в x немає дільників менших ніж y.

(**ІФСР**$_{16}$) Для будь-яких $x, y > 1$:

$$f(x) = \left(\left(x\%2 = 0 \to \mathrm{pre}(2, f(x/2))\right) \mid g(x, 3)\right),$$

$$g(x, y) = \begin{bmatrix} (x = 1 \to \varepsilon \mid \\ \left(x < y^2 \to [x]\right) \mid \\ \left(x\%y = 0 \to \mathrm{pre}(y, g(x/y, y)) \mid g(x, y+2)\right)\right) \end{bmatrix}.$$

Означення ІОФСР$_{16}$ є дійсно індуктивним типу розгортки, оскільки праворуч є виклики g, в яких x не змінюється, а y збільшується на 2 ■

Приклад 2.44. Задача про n ферзів (правила гри в шахи див. у підрозд. 2.4.4).

Необхідно знайти всі позиції n ферзів на шаховій дошці розміром $n \times n$, в яких вони не загрожують один одному Нехай $Z = \{0,1,\ldots,n\}$, $W = \{\overline{0}, \overline{1}, \ldots, \overline{n-1}\}$, $\mathrm{Bool} = \{\overline{0}, \overline{1}\}$. Функція $\mathrm{Queen} : N \to \left(W^*\right)^*$ генерує всі розв'язки задачі. Інші функції в системі:

$\mathrm{BackTrack}: W^* \times Z \times \left(W^*\right)^*$, $\quad \mathrm{Q}: Z \times W^* \times W \to \mathrm{Bool}$, $\quad \mathrm{P}: W^* \times W \to \mathrm{Bool}$, $\exists u(\prec u): W^* \times W \to \mathrm{Bool}$, $\mathrm{Ju}(\prec u): W^* \times W \to W$, $\mathrm{h}: W^* \to W^*$, $\mathrm{L}: W^* \to Z$, $[\ ]: W^* \times W \to Z$ є допоміжними. Конфігурації ферзів подаються словами $\bar{a}_1\bar{a}_2...\bar{a}_r$ в алфавіті W. Слово $\bar{5}\bar{3}\bar{0}$ подає конфігурацію трьох ферзів. Вони розташовані в перших трьох вертикалях дошки: на першій вертикалі ферзь стоїть на шостій клітині (нумерація клітин вертикалі починається з 0), на другій – на четвертій позиції, на третій вертикалі – на першій позиції. Функція BackTrack реалізує бектрекінг і генерує найближчу (у лексикографічному порядку) до x вірну конфігурацію ферзів, розташованих у перших кількох поспіль вертикалях. Якщо довжина чергової вірної конфігурації n, то вона додається до списку. Відповідне ІОФС має вигляд:

(**БФС**$_{17}$) $\mathrm{BackTrack}(\varepsilon,k,y)=y$, $\quad \mathrm{h}(\varepsilon)=\varepsilon$, $\quad x \underset{k}{\prec} u=\bar{0}$ для $k=n$, $\mathrm{Q}\left(j,x,\bar{k}\right)=\bar{1}$ для $j>|x|$, $\bar{a}x[1]=a$ для довільного x, $|\varepsilon|=0$.

(**ІФС**$_{17}$) Для будь-яких $n>3$, $x \in W^*$, $y \in \left(W^*\right)^*$, $0 \le k \le n-1$:

$$\mathrm{Queen}(n)=\mathrm{BackTrack}\left(\bar{1},3,\varepsilon\right),$$

$$\mathrm{BackTrack}(x,k,y)=$$

$$=\begin{pmatrix} |x|=n \to \mathrm{BackTrack}(\mathrm{h}(x),\mathrm{L}(x)+1,\mathrm{pre}(x,y)) \mid \\ |x|<n \to (\exists u(x \underset{k}{\prec} u) \to \mathrm{BackTrack}(\mathrm{Ju}(x \underset{k}{\prec} u),0,y) \mid \mathrm{BackTrack}(\mathrm{h}(x),\mathrm{L}(x)+1,y)) \end{pmatrix}$$

$$|\bar{a}x|=1+|x|,\ \mathrm{h}\left(\mathrm{app}\left(z,\bar{a}\right)\right)=z,\ \mathrm{L}\left(\mathrm{app}\left(z,\bar{a}\right)\right)=a,\ \mathrm{P}\left(x,\bar{k}\right)=\mathrm{Q}\left(1,x,\bar{k}\right),$$

$$\bar{a}x[k]=x[k-1],\exists u\left(x \underset{k}{\prec} u\right)=\left(P\left(x,\bar{k}\right) \to \bar{1} \mid \exists u\left(x \underset{k+1}{\prec} u\right)\right),$$

$$\mathrm{Ju}\left(x \underset{k}{\prec} u\right)=\left(\mathrm{P}\left(x,\bar{k}\right) \to x \circ \bar{k} \mid \mathrm{Ju}\left(x \underset{k+1}{\prec} u\right)\right),$$

$$\mathrm{Q}\left(j,x,\bar{k}\right)=\left(\left(x[j]=k \vee \left(|x[j]-k|=|x|+1-j\right)\right) \to \bar{0} \mid Q\left(j+1,x,\bar{k}\right)\right).$$

Щодо коректності означень ІОФСР$_{16}$ та ІОФСР$_{17}$ див. вправу 15 ■

Увага! Індуктивно означені функції як відображення мають також денотаційну семантику, основану на теоремі про нерухому точку неперервних функцій (див. підрозд. 1.2.5). Уперше звернули на це увагу Д. Скот і К. Стречі (див. літературу для **СР**) ►

2.6.3. РЕКУРСИВНІ СПЕЦИФІКАЦІЇ

Сучасні мови високого рівня й мови специфікацій або прямо підтримують індуктивні означення таких важливих структур даних, як натуральні числа, послідовності однотипних елементів (числові, символьні), Б-дерева, списки (у тому числі й списки списків) тощо (мови Lisp, Meta IV, Z, RSL, HOPE, ML, Python та ін.), або дозволяють ефективно їх моделювати за допомогою покажчиків. Специфікація функції називається *рекурсивною*, якщо її тіло містить власні виклики. Наявність у мові програмування рекурсивних функцій забезпечує можливість реалізації індуктивно означених функцій та їхніх систем. При цьому індуктивні означення або просто кодуються відповідними мовними засобами, або спочатку програмується відповідна алгебра (моделюються об'єкти й конструктори), а вже потім на цій основі здійснюється трансформація індуктивних означень функцій у програми.

Важливим питанням рекурсивних специфікацій функцій є складність алгоритмів їхнього обчислення (див. підрозд. 3.6.3). Це стосується, насамперед, просторової складності $P_A(n)$. Справа в тім, що при черговому рекурсивному виклику функції відбувається переривання й консервація в системному стеку поточного виклику. Максимальна кількість таких перерваних і незавершених викликів називається *глибиною* рекурсії для даного початкового виклику. Зазвичай ця кількість прямо залежить від вхідних параметрів виклику. Наприклад, для функцій, що обчислюють довжину слова (див. прикл. 2.38) і вагу Б-дерева (прикл. 2.39), глибина рекурсії збігається відповідно з довжиною вхідного слова й висотою дерева.

Увага! Ураховуючи, що розмір системного стеку стандартний (у багатьох ОС він не перевищує одного сегмента), глибина рекурсії може виявитися занадто великою для даного операційного середовища. Як наслідок, виконання програми може перериватися з повідомленням про нестачу місця на системному стеку ►

У деяких складних випадках на завершальних етапах розробки рекурсивні програми замінюють їхніми більш надійними ітеративними варіантами. Цей процес називається *елімінацією* рекурсії. При цьому застосовують як загальні, так і спеціальні прийоми елімінації. Загальні теж використовують стек, але не системний, а програмний. Спеціальні враховують специфіку конкретних схем рекурсії.

***Література для СР:** Індуктивні означення множин і функцій – [121, 140, 146]; теорія нерухомої точки для рекурсивних функцій – [82, 118, 119, 148]; рекурсивні специфікації – [2, 139]; елімінація рекурсії – [1, 125, 139].

Контрольні запитання та вправи

1. Дати індуктивне означення: а) множини; б) систем множин. Навести приклади.
2. Яка роль конструкторів і фільтра в ІО?
3. Що таке повнота й повнота слабка в ІО?
4. Що таке виведення елемента за ІО?
5. Що таке гіпотеза й висновок в ІО?
6. Що таке аксіома й теорема в ІО?
7. Дати індуктивне означення функції й систем функцій.
8. Що таке індуктивна функція?
9. Сформулювати правило повної підстановки для обчислення значень індуктивних функцій.
10. Навести приклади ІОФ, для яких правила повної підстановки й виклик за значенням приводять до різних значень [82].
11. У чому полягає різниця між ІОФ типу згортки й розгортки?
12. Що таке рекурсивна специфікація функцій?
13. Побудувати ІО для мов: 1) L_2 із прикл. 1.24; 2) Діка D_{2n}.
14. Показати, що строгість індуктивного означення області визначення індуктивної функції не є необхідною умовою її детермінованості. Навести відповідні приклади.
15*. Обґрунтувати за допомогою структурної індукції коректність індуктивних означень функцій із прикл. 2.37–2.44.
16. Нехай u та v – довільні слова в алфавіті Σ. Дати індуктивні означення для наведених словарних функцій. Припустимими операціями в конструкторах є стандартні арифметичні операції й операції на словах – взяття голови та хвоста слова, конкатенація слів і порівняння символів з Σ:
    а) $\mathrm{ne}(u,v)$ перевіряє нерівність $u \neq v$;
    б) $\mathrm{eq}(u,v)$ перевіряє рівність $u = v$;
    в) $\mathrm{gt}(u,v)$ перевіряє нерівність $u > v$;
    г) $\mathrm{addsym}(u,c,k)$ вставляє на k-ту позицію в u символ c;
    д) $\mathrm{delsym}(u,k)$ усуває в u k-й символ.

17. Дати індуктивні означення арифметичних функцій, $n > 0$:
    а) $\mathrm{count}(n)$, що обчислює кількість цифр у числі n;
    б) $f(n)$, що міняє старшу й молодшу цифри в числі n;
    в)* $\mathrm{prime}(n)$, що перевіряє число n на простоту;
    г)* $\mathrm{perfect}(n)$, що перевіряє число n на досконалість[27].
18. Індуктивне означення натуральної функції $f(x_1,\ldots x_n)$ називається *примітивно-рекурсивним,* якщо воно має вигляд:
    (**БФ**) $f(x_1,\ldots x_{n-1},0) = g(x_1,\ldots x_{n-1})$ для деякої функції g;
    (**ІФ**) $f(x_1,\ldots x_{n-1},s(y)) = h(x_1,\ldots x_{n-1},y,\ f(x_1,\ldots x_{n-1},y))$ для деякого конструктора h.
    Базовими назвемо функції $s(x), O(x) = 0$ і функції проекції $I_m^n(x_1,\ldots x_n) = x_m, m = \overline{1,n}$. Функція називається примітивно-рекурсивною, якщо вона отримана з базових скінченною кількістю застосувань операцій суперпозиції та примітивно-рекурсивних означень. Показати, що функції примітивно-рекурсивні:
    а) $x + y$; б) $x \times y$; в) x / y; г) $x\%y$; д) x^y; е) $\mathrm{sg}(x) = (x = 0 \to 0 \mid 1)$;
    є) $x \dot{-} y = (x \geq y \to x - y \mid 0)$; ж) $\mathrm{prime}(n)$ із вправи 17; з) $\mathrm{perfect}(n)$ із вправи 17; и) $\pi(n)$ – кількість простих чисел, не більших ніж n.
19. Побудувати індуктивне означення функції, що обраховує кількість листків бінарного дерева.
20. Побудувати індуктивне означення функції, що знаходить висоту бінарного дерева.
21. Побудувати індуктивне означення функції, що усуває у зваженому дереві всі вузли із заданою вагою.

[27] Натуральне число називається *досконалим,* якщо воно збігається із сумою всіх своїх власних дільників.

2.7. Елементи технології програмування

- Стиль
- Специфікації системи
- Мова UML
- Рефакторинг
- Налагодження
- Тестування

Ключові слова: *стиль програмування, мова специфікацій, мова UML, сутності й відношення UML, діаграми використання* (*прецедентів*), *класів, поведінки, взаємодії й реалізації, прикордонні й керівні класи, класи-сутності, абстрактні й конкретні класи, відношення між класами: узагальнення, залежності, асоціації, агрегації, композиції, діаграми станів і діяльності, діаграми компонентів і розгортання, рефакторинг, налагоджувальні оператори, твердження, стопори помилок, тестування граничних умов, тестування білої й чорної скриньок, модульне та інтеграційне тестування.*

У підрозд. 2.2 були розглянуті поняття життєвого циклу інформаційних систем і його реалізації. Продовжимо розгляд цих та інших питань, але з практичнішого погляду. Матеріал не орієнтований на вивчення конкретних середовищ розробки систем, тому ми наводимо лише огляд деяких важливих можливостей сучасних ТхП.

Увага! Для ілюстрації елементів ТхП використовується мова C/C++ (див. розд. 3) ►

2.7.1. СТИЛЬ

Написати програму – це більше, ніж правильно її оформити та змусити коректно виконуватись. Програми згодом неминуче модифікуються, тому велике значення має якість їхньої структури, що безпосередньо залежить від простоти сприйняття та зрозумілості. Інколи добре написану чужу програму набагато простіше зрозуміти, ніж власну, але написану погано. Культура написання коду сприяє зменшенню помилок у програмах і полегшує їхню модифікацію.

Під *стилем* програмування зазвичай розуміють набір прийомів і методів, що застосовуються з метою одержання правильних і ефективних програм, зручних для сприйняття, застосування й модифікації.

Вибір імені. Розпочнемо з простого на перший погляд питання вибору імені. Імена мають самі програми, дані, типи, класи, функції, методи. Коли їх у програмі з десяток чи два, то вибір не є дуже принциповим питанням. Однак коли їхня кількість у системі сягає десятків, сотень, а то й тисяч, то, щоб не втратити контроль над системою, при виборі імен необхідно дотримуватись певної строгої дисципліни й порядку. Дамо кілька загальних порад.

Доцільно використовувати осмислені імена для глобальних змінних і, за можливості, короткі – для локальних. Глобальні змінні, функції, класи та структури за означенням можуть з'являтись у довільному місці програми, тому найважливішою для них є інформативність, а не довжина. Краще використовувати глобальні імена так, щоб вони явно вказували на смисл понять, що ними подаються, тобто мали відповідну *мнемоніку*. Корисно опис кожної глобальної змінної, функції, типу даних, константи супроводжувати коротким коментарем.

Приклад 2.45. Стиль найменувань.

```
/*структура, що задає дату*/
struct date {
    int day; /*день*/
int month; /*місяць*/
unsigned year; /*рік*/
};

/*Друкування дати*/
void printDate(struct date *pd);

int iCountElements=0; /*поточна кількість елементів*/
■
```

Однак коментарі не мають містити очевидної інформації на зразок того, що оператор **i++** збільшує змінну **i** тощо.

Для локальних змінних, навпаки, краще підходять короткі імена. Для використання всередині функції цілком підійдуть імена **i, j, n, npoints** тощо. При цьому для лічильників циклу зазвичай використовуються імена **i**, **j**, для покажчиків – **p**, **q**, для рядків – **s**, **t**. Якщо це можливо, використовують англомовні назви змінних, типів, класів, функцій, констант, які відповідають їхній ролі в програмі.

Існує багато корпоративних домовленостей і традицій іменування. Традиційно для функцій використовуються імена, побудовані з дієслів та іменників, причому перші літери слів великі (**LoadData()**), а для змінних – іменники.

Оператори й вирази. Ці основні конструкції програм слід писати так, щоб їхній зміст був максимально зрозумілим. Найпростіший спосіб досягти цього – форматування коду за допомогою відступів, табуляцій, порожніх рядків. Розглянемо два тексти однієї й тієї самої програми. Очевидно, що другий текст більш читабельний саме завдяки використанню його якісного форматування.

Приклад 2.46. Форматування коду.

Лістинг 1.

```
/*бінарний пошук без форматування тексту*/
int BinarySearch (char *items, int count, char key)
{int low,high,mid;
low=0; high=count-1;
while(low<=high){
mid=(low+high)/2;
if(key<items[mid])high=mid-1;
else if(key>items[mid]) low=mid+1;
else return mid;}
return -1;}
```

Лістинг 2.

```
/*бінарний пошук із форматуванням тексту*/
int BinarySearch (char *items, int count, char key)
{
int low=0, high=count-1, mid;
while(low<=high) {
     mid=(low+high)/2;
     if(key<items[mid])
       high=mid-1;
     else
       if(key>items[mid])
          low=mid+1;
       else
          return mid; /*ключ знайдено*/
}
return -1;
}■
```

Іноді можуть поліпшити сприйняття виразу додаткові дужки, що підкреслюють структуру коду. Їх застосовують, навіть якщо синтаксично в цьому немає потреби. Зрозумілість і стисліть коду – не одне й те саме. Головним критерієм вибору синтаксису для коду має бути простота його сприйняття.

Відступи, табуляції, переходи на новий рядок допомагають краще структурувати код. Однак якому стилю тут віддати перевагу? Наприклад, чи варто розташовувати відкриваючу фігурну дужку в тому самому рядку, що й `if` або `while`, чи в наступному? Єдиного рецепта немає. Важлива не стільки конкретика вибраного стилю, скільки логічність і послідовність його застосування.

Вважається, що одна функція (метод) не має займати більш ніж один екран консолі. Якщо це не так, то їх краще розбити на дрібніші. Якщо певний фрагмент коду вимагає коментування, то його теж краще виділити в окрему функцію (метод).

2.7.2. СПЕЦИФІКАЦІЇ СИСТЕМИ

Сучасні програмні проекти реалізуються колективами різних фахівців – від аналітиків, проектувальників, кодувальників, тестувальників до менеджерів проектів, кожний з яких виконує свою роботу на різних етапах ЖЦ. Для фіксації й обміну результатами роботи використовують спеціальні ДС, що отримали назву *мов специфікації*. Наприклад, як ми вже знаємо з підрозд. 2.2.1, результатом роботи аналітиків є зовнішня специфікація системи, а кодувальників – програмний код.

Специфікації описують ту чи іншу модель вхідної системи або її частини, тобто моделюють систему й можуть бути використані замість неї в деяких контекстах. За властивостями специфікацій системи можна робити висновки щодо властивостей самої системи.

При побудові специфікації має бути чітко поставлена мета: для чого дана модель створюється, які завдання вона повинна розв'язати тощо. Зазвичай мета задає набір властивостей, що мають бути відображені в моделі, і контекст, в якому систему можна замінити моделлю. Наприклад, кінцевою метою етапу проектування є отримання специфікації системи у вигляді, за яким легко писати код програми.

Основними вимогами до специфікацій є їхня повнота, точність і зрозумілість.

Специфікації можуть використовуватися:

- для уточнення вимог, узгоджень їх із замовником і побудови прототипів;
- при проектуванні – для контролю за правильністю проекту;
- при реалізації – для формулювання завдань розробникам і створення документації;
- при тестуванні – для перевірки виконання вимог;
- при супроводженні – для уточнення змін, підтримки узгодженості документації із системою тощо.

Однією з важливих переваг використання специфікацій є поглиблення розуміння системи, що уточнюється. У процесі створення специфікації розробники мають більше можливостей для виявлення недоліків, непослідовностей, неоднозначностей і неповноти проекту. Специфікація є засобом зв'язку між замовником і проектувальником, проектувальником і розробником, а також між розробником і тестувальником. Вона часто використовується як супутня документація до програмного коду системи, але має формальніший рівень опису.

Модель у специфікації може описуватися в термінах стану системи, вхідних і вихідних даних, кінцевих станів, потоків даних і керування тощо. Для відображення різних аспектів системи застосовуються різні набори понять. Для опису характеристик системи можна скористатися кількома моделями в межах певних формалізмів. Одні нотації й мови орієнтовані на доступність і прозорість опису, тому є менш формальними, інші – більш формальні та орієнтовані на аналіз ефективності системи чи трансляцію.

Розглянемо як приклад кілька специфікацій функції $n!$. Скористаємося для цього засобами, розглянутими в попередніх підрозділах.

Приклад 2.47. Одинадцять специфікацій функції $n!$.

1) $n!$ – добуток перших n натуральних чисел, розпочинаючи з 1;

2) $n! \overset{def}{=} 1 \times 2 \times \ldots \times n$;

3) $n! \overset{def}{=} \prod_{i=1}^{n} i$;

4) (БФ) $0! = 1$, (ІФ) $(n+1)! = (n+1) \times n!$;

5) (БФ) $n! = \mathrm{g}(1,n)$, $\mathrm{g}(n,n) = 1$;

(ІФ) $\mathrm{g}(i,n) = i \times \mathrm{g}(\mathrm{s}(i),n)$;

6) найменший розв'язок функціонального рівняння

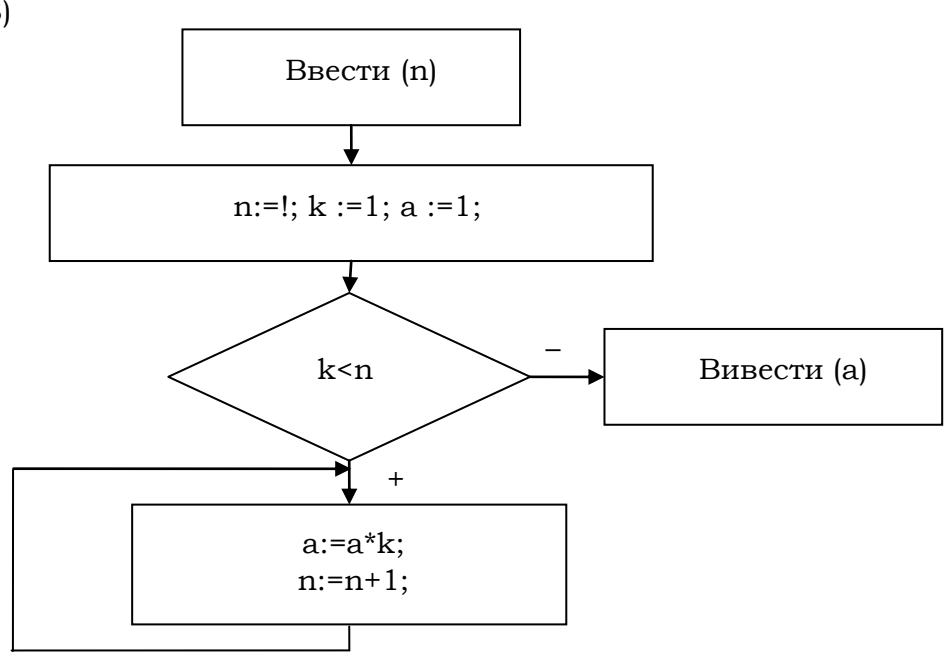
$\mathrm{f}(n) = (n = 0 \to 1 \mid n \times \mathrm{f}(n-1))$;

7) $n! = \mu_Q^*(n)$ – функція, породжена рекурентною системою (*)

$$(*)\begin{cases} a_1 = 1, a_k = a_{k-1} * k, \\ k - \text{лічильник}, \ k_1 = 1, \\ n - \text{константа} \end{cases}$$

і фільтром $\mathrm{Q}(k,n) = k < n$;

8)



9)

```
{n:=!; k:=1; a:=1; i:=1;
while (k<n) do {k:=k+1;
                a:=a*k;
  }
}
```

10)

```
long f(int n)
{int k=1;
  long a=1;
   while (k<n) do {k++;
                   a*=k;
                  }
   return a;
```

11)

```
long f(int n)
   {
      return (n<2) ? 1:n*f(n-1);
   }
```

Перші три специфікації відомі зі школи. Третій варіант використовує поняття добутку індексованої сукупності чисел (див. підрозд. 1.2.3); четвертий і п'ятий – два різновиди індуктивних означень (підрозд. 2.6.2). У них застосовано індукції відповідно типу згортки й розгортки. Шоста специфікація – функціональне рівняння (див. підрозд. 1.3.3); сьома – рекурентне означення (див. підрозд. 2.5.3); восьма – стандартна програма для обчислення $n!$; дев'ята – її лінійна форма; десята й одинадцята – ітеративна й рекурсивна функції мови С.

Усі наведені специфікації є повними й точними, але різними за метою та зрозумілістю. Перші вісім орієнтовані на людину, решта – на машинну обробку. Деякі з них відрізняються лише синтаксисом, наприклад 1–3, 8 та 9, 4 та 11.

Найпростішими специфікаціями є 1 та 2, що показують, як обчислити значення $n!$. Щоб скористатися ними, треба лише вміти перемножити два числа й розуміти українську мову. Наступним за прозорістю є, на наш погляд, індуктивний варіант типу розгортки 4. Найскладніші ж ті, що потребують володіння спеціальними поняттями й технічними навичками – 7 (методи розв'язку функціональних рівнянь), 8 (методи побудови рекурентних співвідношень і стандартних програм), 10–11 (знання мови програмування С) ■

Серед відомих *формальних* мов специфікацій можна виділити такі, як VDM, Z, B, CCS, LOTOS тощо. Подібні мови призначені для розробки систем із підвищеними вимогами до безпеки.

Як уже зазначалось, написання формальних зовнішніх специфікацій системи дозволяє виявити помилки й невідповідності в неформалізованих чи слабоформалізованих вимогах до системи. Ця можливість виявлення помилок на ранніх стадіях розробки – вагомий аргумент на користь формальних специфікацій. Помилки у вимогах, виявлені, наприклад, під час атестації, зазвичай вимагають великих витрат на виправлення.

Звернемось до практики програмування й порівняємо витрати на розробку систем без використання формальних специфікацій і з таким на різних стадіях. За окремими даними [120] у першому випадку вартість атестації системи становить біля 50 % усієї вартості продукту, а вартість проектування й реалізації вдвічі більше вартості етапу аналізу системи. У другому випадку вартості проектування й реалізації, з одного боку, і аналізу – з іншого, зіставні, проте вартість атестації значно зменшується. У цілому загальна вартість системи в обох випадках якщо й не зіставна, то різниця у вартості може нівелювати-

ся більшою надійністю системи, розробленою за допомогою формальних специфікацій.

Іншим важливим аспектом формальних специфікацій систем є те, що вони створюють основу для технологій доказового програмування (див. підрозд. 2.6) і оцінки просторової й часової складності систем.

Немає сумніву, що з підвищенням вимог до програмного забезпечення й культури програмування, удосконаленням засобів формальної специфікації їхня роль у ЖЦ буде зростати.

2.7.3. МОВА UML

Серед практичних методів проектування систем важливе місце посідає об'єктно-орієнтоване проектування й програмування систем (див. підрозд. 2.3). В ООП для опису архітектурних моделей найчастіше використовується мова специфікацій UML (Unified Modeling Language). Стисло розглянемо її основні ідеї й засоби.

В UML знайшли відображення методи об'єктно-орієнтованого моделювання, розроблені Бучем, Рамбо (метод Object Modeling Technique – OMT) і Якобсоном (метод Object-Oriented Software Engineering – OOSE). Метод Буча був орієнтований на проектування систем, OMT – на аналіз, а OOSE забезпечував можливості для специфікації бізнес-проектів і аналізу вимог за допомогою сценаріїв використання. Усі вказані методи відомі як засоби візуального моделювання, що добре зарекомендували себе на практиці.

Зокрема, UML дозволяє описати визначальні типи об'єктних моделей:

- структурні (статичні) – описується структура сутностей системи, включаючи класи, інтерфейси, відношення, атрибути;
- моделей поведінки (динамічних) – описується поведінка (функціонування) об'єктів системи, включаючи методи, взаємодію, процес зміни станів окремих компонент чи всієї системи.

UML забезпечує моделювання різних зображень архітектури:

- структури,
- (динамічної) поведінки,
- керування моделями.

Концептуальна модель UML. Словник UML містить сутності, відношення й діаграми. Перші є первинними елементами моделі, другі пов'язують різні сутності, а треті групують сукупності сутностей.

В UML існує низка семантичних правил, що дозволяють коректно й однозначно визначати:

- імена, які можна давати сутностям, відношенням і діаграмам;

- область дії імен (контекст, в якому ім'я має деяке значення);
- видимість (коли імена видимі й можуть використовуватися іншими елементами);
- цілісність (узгоджене співвідношення елементів системи);
- виконання моделі.

UML має механізми:

- специфікації;
- доповнення;
- розподілу;
- розширення.

Сутності мови UML. В UML є чотири типи сутностей:

- структурні;
- поведінкові;
- групувальні;
- анотаційні.

Структурні сутності зазвичай є статичними елементами системи, що відповідають концептуальним чи фізичним її елементам. Виділяють такі типи структурних сутностей: клас, інтерфейс, кооперація, прецедент, активний клас, компонент, вузол. Основні сутності: актор, сигнал, утиліта (утиліти є видами класів); процес і нитка (нитки є видами активних класів). До решти належать: застосування, документ, файл, бібліотека, сторінка й таблиця (таблиці є видами компонентів).

Поведінкові сутності є динамічними складовими моделі UML. Виділяють такі типи поведінкових сутностей: взаємодія й автомат.

Групувальні сутності є організуючими частинами моделі UML. На даний момент реалізована лише одна групувальна сутність – пакет.

Анотаційні сутності – роз'яснювальні частини моделі UML, що зображуються примітками.

Відношення мови UML. В UML існує чотири типи відношень:

- залежність;
- асоціація;
- узагальнення;
- реалізація.

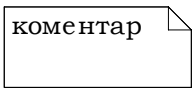
Є також їхні варіації, наприклад уточнення, трасування, включення й розширення (для залежностей), агрегація й композиція.

Діаграми мови UML. Це основні об'єкти UML. Існує кілька типів діаграм, що подають різні аспекти систем. Набори діаграм дозволяють отримати достатньо повне уявлення про всю проектовану систему та її окремі компоненти. *Діаграма* в UML – це графічне

подання набору елементів, зображуване зазвичай у вигляді зв'язного графа з вершинами (сутностями) і ребрами (відношеннями).

В UML визначено вісім видів діаграм, кожна з яких може містити елементи певного типу. Типи можливих елементів і відношень між ними залежать від виду діаграми. Види діаграм:

- діаграма використання;
- діаграма класів;
- діаграми поведінки, до яких належать діаграми станів, діяльності, взаємодії; у свою чергу, до діаграм взаємодії належать діаграми послідовності й кооперації;
- діаграми реалізації, до яких належать діаграми компонентів і розгортання.

На усіх діаграмах можуть бути присутні коментарі: 

.

Коментар приєднується таким відношенням: .

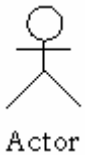
Діаграма використання (прецедентів) (Use case diagram). Така діаграма задає концептуальну модель системи: визначаються загальні кордони й контекст систем, уточнюється їхня зовнішня функціональна поведінка. Саме тут з'являється первісна документація, що може використовуватись для предметного обговорення майбутньої системи розробниками, замовниками, користувачами та іншими зацікавленими сторонами.

Діаграми прецедентів є основою для подальшої деталізації системи у формі логічних і фізичних моделей.

Окрема діаграма прецедентів складається з:

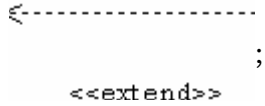
прецедентів – 

;

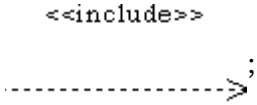
акторів – 

;

і відношень:

а) розширення – 

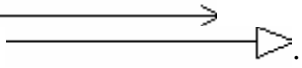
;

б) включення – 

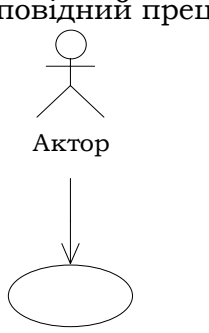
;

в) асоціації – 

;

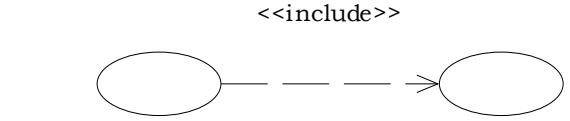
г) узагальнення –  .

Між актором і прецедентом можливий зв'язок *асоціації*, який указує на те, що актор ініціює відповідний прецедент:



Відношення *розширення* (**extend**) є відношенням залежності між базовим та іншим прецедентом, функціональна поведінка якого використовується базовим не завжди, а лише при виконанні деяких умов. Стрілка напрямлена до базового прецеденту (у всіх інших відношеннях – від прецеденту, що викликає, до того, який викликається):

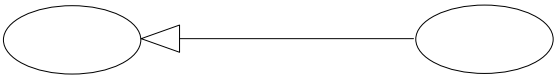
Відношення *включення* (**include**) у мові UML є відношенням залежності між прецедентами й указує на те, що задана поведінка для одного прецеденту включає як обов'язкову складову частину поведінку іншого:



Видача готівки в банку Перевірка стану рахунку

Один прецедент може включатися в кілька інших або, у свою чергу, включати інші прецеденти. Прецедент, який включається, не залежить від базового в тому розумінні, що він надає останньому інкапсульовану поведінку, деталі реалізації якої приховані. Таким чином, базовий прецедент залежить лише від результатів виконання прецеденту, який у нього включається, а не від його структури.

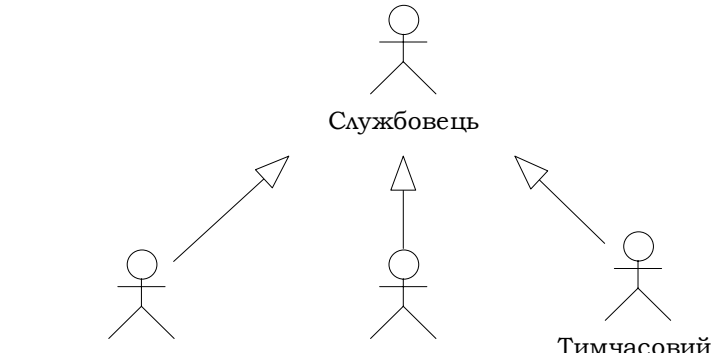
Відношення *узагальнення* може бути як між двома акторами, так і між двома варіантами використання. Для прецедентів воно вказує, що один із них (нащадок) є спеціалізацією іншого (батьківського). Наприклад:



Надання кредиту Надання кредиту корпоративним клієнтам

Нащадок успадковує всі властивості поведінки батьківського прецеденту (актора) і може мати додаткові.

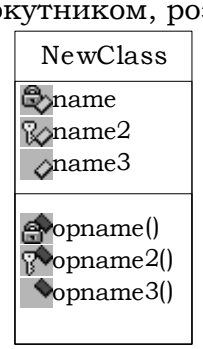
Приклад узагальнення між акторами:



Діаграма класів (Class diagram). Описує статичну структуру класів. Дозволяється на концептуальному рівні формувати словник предметної області й на рівнях специфікацій і реалізацій визначати структуру класів у програмній реалізації системи.

Діаграми класів можуть використовуватись для генерації каркасного програмного коду (у реальній мові програмування).

Клас зображується прямокутником, розділеним на три секції:



.

У верхній секції розміщено ім'я, у середній – атрибути, у нижній – методи класу.

Атрибути й методи класу описуються із зазначенням специфікаторів доступу. В UML вони позначаються +, # та – і означають:

+ ( ) – публічні (**public**) атрибути;

( ) – захищені (**protected**) атрибути;

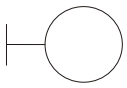
- ( ) – приватні (**private**) атрибути;

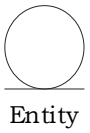
+ ( ) – публічні (**public**) методи;

( ) – захищені (**protected**) методи;

( ) – приватні (**private**) методи.

Виділяють такі типи класів (класифікація за призначенням):

• прикордонні (**boundary**), або інтерфейсні: 

• класи-сутності (**entity**): 

• керівні (**control**) (класи-менеджери): 

.

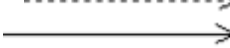
Інтерфейси позначаються так: 

Класи можуть бути абстрактними чи конкретними, що зазначається при їхньому описі.

Типи відношень між класами:

узагальнення –  ;

залежність –  ;

асоціація –  ;

агрегація –  ;

композиція –  .

Відношення *узагальнення* відображає ієрархічну будову класів і успадкування властивостей і поведінки.

Як обмеження можуть бути використані такі ключові слова UML:

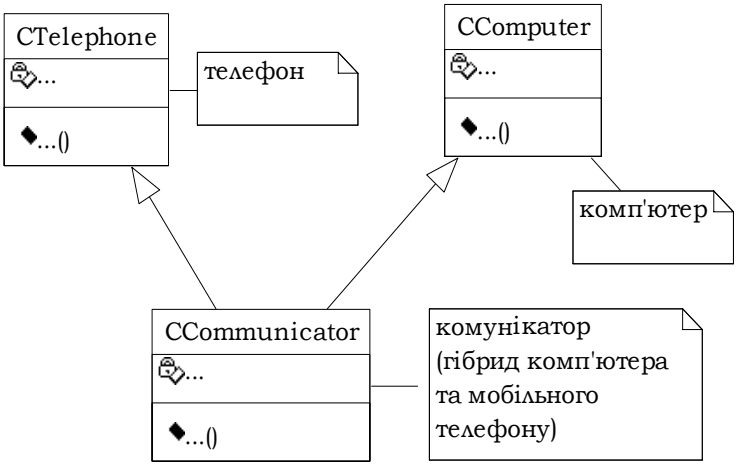
{complete} – у даному відношенні узагальнення специфіковані всі класи-нащадки – інших нащадків у даного класу бути не може;

{incomplete} – на цій діаграмі зображені не всі нащадки, додавання нових нащадків не вплине на побудовану діаграму;

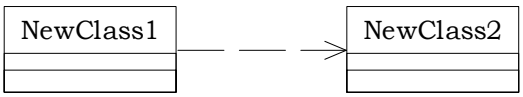
{disjoint} – класи-нащадки не можуть містити об'єктів, що одночасно є екземплярами двох чи більше класів;

{overlapping} – окремі екземпляри класів-нащадків можуть належати одночасно кільком класам.

Приклад успадкування:

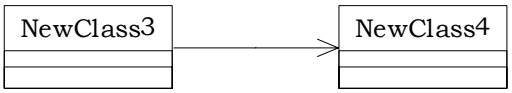


Відношення *залежності* відображає зв'язок між класами, коли залежний клас залежить від визначень, зроблених іншим класом. Залежність – однонапрямлений зв'язок від залежного класу до класу, від якого він залежить:



При генерації коду для залежних класів до них не додаються нові атрибути, але створюються необхідні команди (у код вставляється директива препроцесору **`#include`**).

Відношення *асоціації* між класами – це семантичний зв'язок між ними:

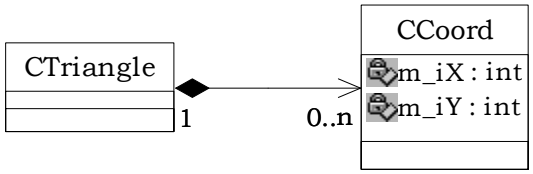


Асоціація може бути однонапрямленою й двонапрямленою. Напрямок асоціації можна визначити за діаграмами послідовності й кооперації.

Відношення *агрегації* – спеціальна форма асоціації, яка використовується для зображення відношення "частина-ціле" між агрегатом "ціле" і його складовою "частина":

При генерації коду відношення агрегації додає відповідний атрибут у код класу "ціле".

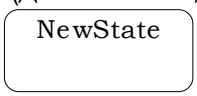
Сильніший різновид агрегації – *композиція*, коли об'єкт – частина – може належати лише єдиному цілому, а при видаленні цілого знищується й частина:



Діаграми поведінки (Behavior diagrams). Поділяються на діаграми станів і діаграми діяльності.

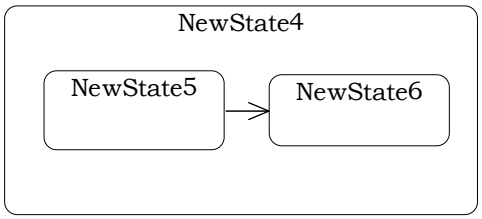
Діаграма станів (Statechart diagram). Визначає всі можливі стани, в яких може перебувати конкретний об'єкт, а також процес зміни станів об'єкта внаслідок деяких подій.

Діаграма станів складається з таких елементів:

- стан (дозволяє відобразити стан чи ситуацію протягом життя об'єкта): 

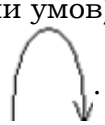
;
- початковий стан:  ;
- фінальний стан:  .

Зі станом асоціюється деяка діяльність, що позначається міткою й зображується як **`<виконати>/<дія>`**. Стани можуть утворювати групи, як у наведеному прикладі діаграми станів:



Між станами можливі такі відношення:

• перехід (дозволяє здійснити перехід з одного стану в інший у випадку настання певної події чи зміни умов) – ⟶;

• перехід на себе (рефлексивний) –  .

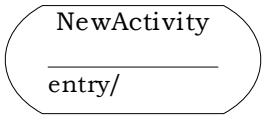
Перехід може мати мітку, синтаксис якої складається із трьох необов'язкових частин: **<Подія>[<Умова переходу>]/<Дія>**. Якщо мітка переходу не містить жодної *події*, то це означає, що перехід відбудеться, як тільки завершиться діяльність, асоційована з даним станом. *Дія* – це набір простих операцій, які можуть відбуватися:

а) при вході в стан – на схемі позначається написом **entry/<дія>;**

б) при виході зі стану – на схемі позначається написом **exit/<дія>;**

в) при переході – дія, яка відбувається при здійсненні переходу (тобто після дії при виході з вихідного стану перед дією при вході у вхідний стан); на схемі позначається міткою переходу **/<дія>**.

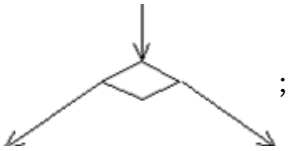
Діаграми діяльності (Activity diagram). Відображають динаміку системи, подаючи схеми потоків керування (від однієї діяльності до іншої). Можуть зображувати паралельні потоки, альтернативні потоки (розгалуження). Діяльність (активність) на діаграмі діяльності зображується так:

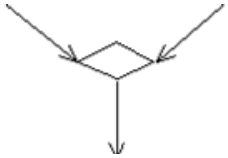


.

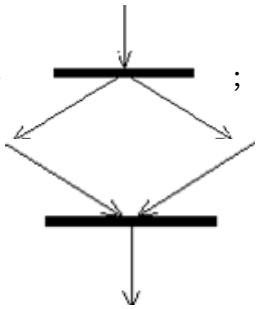
Між діяльностями можливі відношення переходу ⟶.

Умовна поведінка зображується таким чином:

• розгалуження (початок умовної поведінки) –  ;

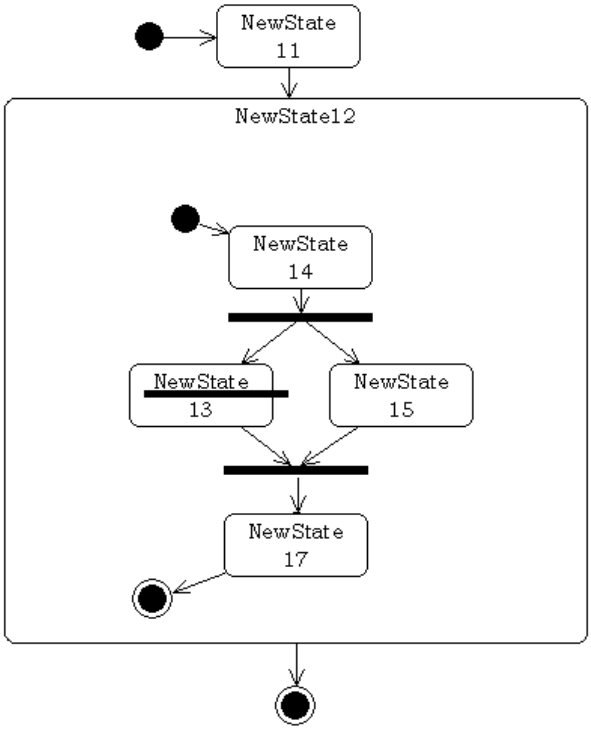
• з'єднання (завершення умовної поведінки) –  .

Паралельна поведінка (синхронізація паралельної поведінки) зображується так:

• розділення (початок паралельного виконання) –  ;

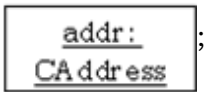
• злиття (завершення паралельного виконання) –

Діяльність може складатися з піддіяльностей.
Приклад діаграми діяльності:



Діаграми взаємодії (Interaction diagrams) поділяються на діаграми послідовності **(Sequence diagram)** і кооперації **(Collaboration diagram).** Кожен сценарій реалізується сукупністю об'єктів, що взаємодіють між собою. Така сукупність називається кооперацією.

Діаграма *послідовності* дозволяє зобразити сценарій прецеденту графічно, шляхом відображення послідовності повідомлень, якими обмінюються об'єкти. Складається з таких елементів:

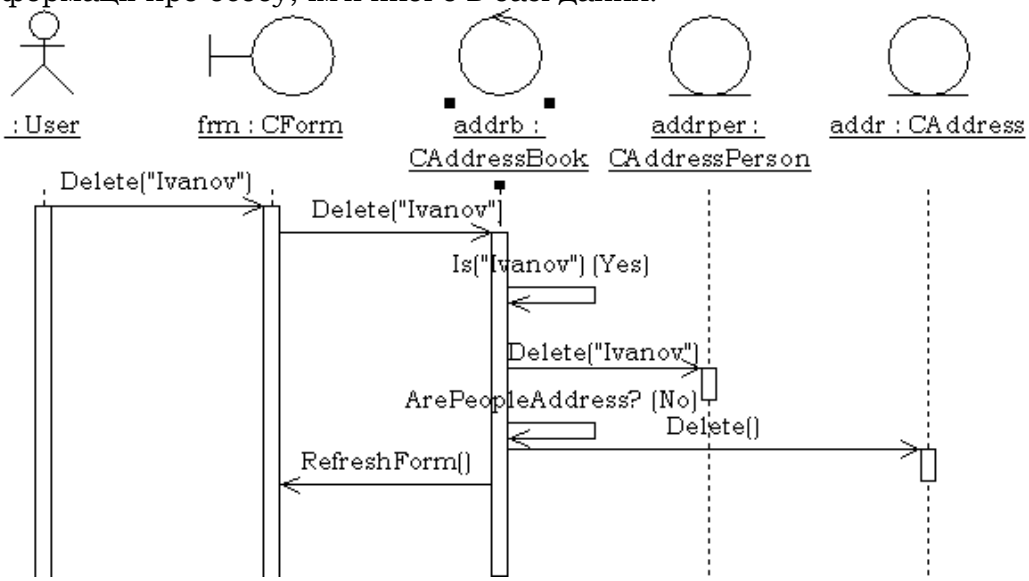
- об'єкт – 

;
- повідомлення (передається від одного об'єкта іншому) –  ;
- рефлексивне повідомлення –  .

Кожне повідомлення містить мітку (обов'язково зазначається ім'я повідомлення, до того ж можна вказати аргументи й деяку керуючу інформацію).

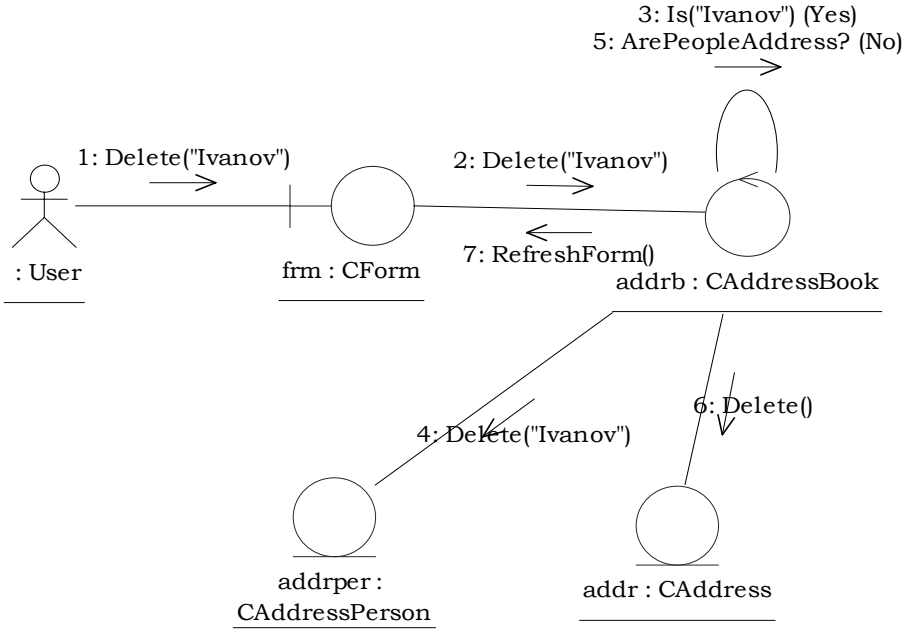
Вертикальною лінією під позначенням об'єкта на діаграмі послідовності відображена його лінія життя – ЖЦ об'єкта в процесі взаємодії. Для відображення періоду активності об'єкта використовується прямокутник активності.

Діаграма послідовності для адресної книги: операція видалення інформації про особу, ім'я якої є в базі даних:



На діаграмі *кооперації* екземпляри об'єктів і повідомлення між ними відображаються так само, як і на діаграмі послідовності. Проте їхня часова послідовність указується за допомогою нумерації повідомлень. Ці діаграми відображають конкретний сценарій варіанта використання.

Діаграма кооперації для адресної книги: операція видалення інформації про особу, ім'я якої є в базі даних:

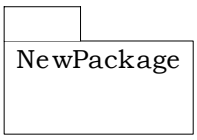


Діаграми реалізації (Implementation diagrams). Поділяються на діаграми компонентів і розгортання.

Діаграми компонентів (Component diagram). Зображують компоненти системи та зв'язки між ними. Застосовуються тими учасниками проекту, які відповідають за компіляцію системи. Із цих діаграм видно, в якому порядку потрібно компілювати компоненти, а також які виконувані компоненти будуть при цьому створені.

Діаграми компонентів складаються з таких елементів.

Складні пакети:

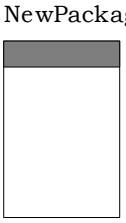


– групи логічно пов'язаних компонентів, дозволяють структурувати модель системи.

Компонента 

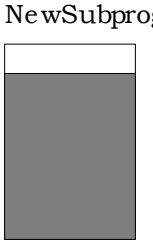
– модуль програмної системи (виконуваний, DLL, вихідний код тощо) з можливим зазначенням інтерфейсу.

Залежно від типу програмного модуля виділяють такі види компонентів:

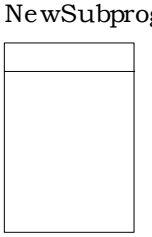
головна програма (main program): 

;

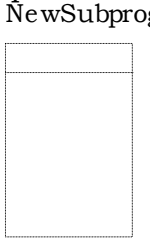
підпрограма (subprogram) – використовується для підпрограм, а також для необ'єктно-орієнтованих компонент;

тіло підпрограми (subprogram body): 

;

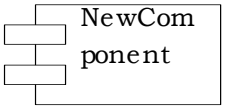
специфікація підпрограми (subprogram specification): 

;

родова специфікація підпрограми (generic subprogram specification): 

.

Для подання .exe- та .dll-файлів використовуються компонент типу EXE й компоненти типу DLL:



Для подання баз даних використовують компоненту

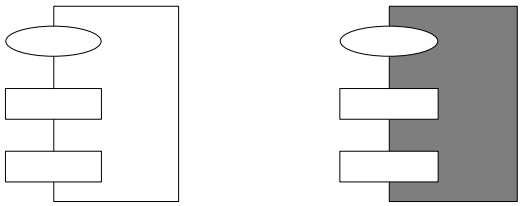
NewComponent2

Компоненти Task specification і Task body для специфікації відповідно задач і тіла задач дозволяють відобразити незалежні потоки в багатопотоковій системі:

NewTaskSpec1 NewTaskBody1

Компоненти Package specification і Package body для специфікації відповідно пакетів і тіл пакетів:

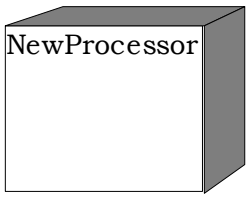
NewPackageSpec1 NewPackageBody



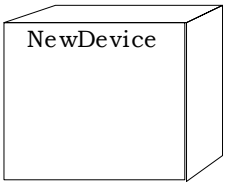
У мовах C, C++ дані компоненти використовуються таким чином: Package specification – для .h-файлів, а Package body – для файлів із розширеннями .c та .cpp.

Діаграма розгортання (Deployment diagram). Використовується менеджером проекту, користувачами, архітектором системи для розуміння фізичного розміщення системи та її окремих підсистем.

Processor – компонента апаратних засобів ЕОМ, що можуть виконувати програми:



Device – компонента апаратних засобів ЕОМ, що не можуть виконувати програми (модем, принтер, монітор тощо):



Відношення між сутностями: ——————.

Спрощена стратегія використання мови UML. При проектуванні програмної системи доцільно спочатку розробити діаграми прецедентів як концептуальну модель системи. Ці діаграми уточнюють зовнішню функціональну поведінку, можуть використовуватися як документація й обговорюються всіма зацікавленими сторонами при розробці програмної системи. Створені діаграми прецедентів є основою для подальшої роботи над проектом (розробка інших діаграм, тестування тощо).

Наступним етапом є розробка діаграм класів і взаємодії. Діаграми класів можуть використовуватися для автоматичної генерації каркасного програмного коду. Для визначення поведінки класів зі складною динамікою реагування на події можуть бути задіяні діаграми станів і діяльності.

Розміщення об'єктів за програмними модулями фіксується компонентними діаграмами, а розташування програмних модулів стосовно вузлів (комп'ютерів) мережі – діаграмами розгортання.

Приклади застосування наведених засобів ООП див. у підрозд. 3.11 та 4.6.

2.7.4. РЕФАКТОРИНГ

Останнім часом спостерігається тенденція до збільшення тривалості ЖЦ програмних систем, зростає обсяг успадкованого коду. Тому надзвичайно важливими є задачі, пов'язані з полегшенням модифікації й розвитком існуючого програмного коду.

При внесенні в систему змін часто втрачається її структурованість, розібратися в ній стає важче і, як наслідок, її програмний код стає погано придатним для супроводження. Крім того, недбало спроектований код зазвичай займає занадто багато місця, у ньому часто трапляються повтори. Для розв'язання цих проблем застосовують рефакторинг.

Рефакторинг – це процес змінювання програмної системи, при якому не змінюється зовнішня поведінка програми, але поліпшуються її внутрішня структура та стиль уже написаного коду.

Рефакторинг здійснює зміни, пов'язані з методами, переміщеннями функцій між об'єктами, організацією даних, спрощенням умовних виразів і викликів методів, задачами узагальнення (виділення інтерфейсу, підкласу, батьківського класу тощо).

Перелічимо коротко деякі з елементів рефакторингу.

Зміна сигнатури методу полягає в додаванні, зміні або видаленні параметра методу. Змінивши сигнатуру методу, необхідно скорегувати звернення до нього в коді всіх клієнтів. Ця зміна може торкнутися зовнішнього інтерфейсу програми. Якщо програмісту не доступні всі клієнти інтерфейсу, то може знадобитися та чи інша форма реєстрації змін інтерфейсу.

Інкапсуляція поля – якщо в класі є відкрите поле, то необхідно зробити його закритим і забезпечити методи доступу.

Виділення класу – іноді клас реалізує функцію, яку слід розподілити між кількома класами. Тоді рефакторинг дозволяє створити новий клас, перемістити відповідні атрибути й методи зі старого класу в новий.

Виділення підкласу – якщо клас виконує різні види робіт, то доцільно створити підкласи, які йому спадкують і виконують кожен свій вид роботи.

Виділення надкласу – якщо маємо кілька класів зі схожими методами й полями, то можна перемістити їх у надклас (батьківський для всіх даних підкласів).

Виділення інтерфейсу – якщо кілька клієнтів використовують одну підмножину інтерфейсу класу або в кількох класах певна частина інтерфейсу спільна, то рефакторинг забезпечує виділення цієї підмножини в окремий інтерфейс.

Виділення локальної змінної – якщо певний вираз обчислюється в тілі методу в кількох місцях, то його значення слід обчислити один раз і запам'ятати в новій локальній змінній. Наприклад, код

```
func(a+b, (a+b) / 2, a+b-c);
```

слід рефакторизувати в

```
t=a+b;
func(t, t/2, t-c);
```

попередньо оголосивши `t` локальною змінною відповідного типу.

Вбудовування змінної – підстановка єдиний раз присвоєного значення існуючої локальної змінної замість неї самої. Можна використовувати для економії (стекової) пам'яті, що корисно в рекурсивних функціях. Наприклад, код

```
double y=f(x);
if (y<0) {…}
```

слід рефакторизувати в

```
if (f(x)<0) {…}
```

Виділення методу полягає у виділенні з довгого чи важкого для сприйняття коду окремих його фрагментів і перетворенні їх в окремі методи (функції).

Вбудовування методу протилежне виділенню методу. Коли тіло методу (функції) так само зрозуміле, як і його (її) назва, тоді виклик методу замінюють на код. Крім того, такий рефакторинг застосовують при невдалому розбитті на методи – тоді кілька методів об'єднують, а потім знову виділяють окремі методи.

Підйом поля (*методу*) – якщо у двох підкласах є однакове поле чи метод, то їх слід підняти в батьківський клас.

Спуск поля (*методу*) – якщо в батьківському класі є поля чи методи, що належать лише до деяких його підкласів, то їх потрібно перемістити в ці підкласи.

Заміна умовного оператора поліморфізмом – умовний оператор із кількома розгалуженнями замінюється викликом поліморфного методу деякого базового класу, що має підкласи для кожної гілки вихідного оператора. Вибір гілки здійснюється неявно.

Переміщення методу застосовується для методу, що частіше звертається до іншого класу, ніж до того, у якому він розташований. Наприклад:

```
class library {
    book[1000] books;
}
class book {
    int id;
    int InLibrary(library Lib) {
          for(int i=0; i<1000; i++)
          if (id==Lib.books[i].id) return 1;
          return 0;
    }
}
```

При застосуванні останнього рефакторингу цей код перетвориться на такий:

```
class library {
    book[1000] books;
    int InLibrary(book Book) {
          for(int i=0; i<1000; i++)
          if (Book.id==books[i].id) return 1;
          return 0;
    }
}
class book {
    int id;
}
```

Отже, підсумуємо переваги рефакторингу.

Рефакторинг поліпшує декомпозицію й розуміння програмної системи. Таке розуміння істотно прискорює процес програмування, допомагає знайти помилки та, як наслідок, дозволяє швидше отримати працюючий код системи, а також полегшує наступне супроводження програмної системи. У той самий час процес рефакторингу надзвичайно трудомісткий, потребує уважного перегляду всього коду. Не випадково існують численні засоби його автоматизації.

2.7.5. НАЛАГОДЖЕННЯ

Як ми вже знаємо з підрозд. 2.2.1, налагодження є важливою частиною етапу обґрунтування правильності системи. Воно полягає в організації й проведенні комплексного тестування програми та усуненні виявлених при цьому помилок. Такий складний процес може перебігати тривалий час, тому важливою практичною проблемою є скорочення часу. Технічні прийоми, що сприяють зменшенню витрат на налагодження, включають гарний дизайн і стиль, перевірку граничних умов і правильності вихідних тверджень, добре розроблені інтерфейси між підсистемами, обмежене використання глобальних даних тощо.

Зазвичай середовище розробки IDE містить спеціальні засоби для налагодження програм – *налагоджувачі.* Налагоджувач має інтерфейс користувача для покрокового виконання програми оператор за оператором або функція за функцією із зупинками в контрольних точках програми (англ. – breakpoint) чи при досягненні якоїсь умови та для динамічного відображення значень змінних (*трасування* змінних). Існують також налагоджувачі з простим консольним інтерфейсом на зразок нала-

годжувача Numega SoftIce [3]. За допомогою останнього можна налагоджувати системні драйвери й навіть ядро операційної системи.

Налагоджувач може використовуватися безпосередньо або запускатись автоматично, якщо під час виконання програми щось відбувається не так як треба. Є можливість виявити місце аварійного переривання програми. Можна розглянути послідовність викликів функцій, що виконувалися в момент переривання (це називається переглядом стеку викликів), відобразити значення локальних і глобальних змінних. Цієї інформації зазвичай буває достатньо для виявлення помилки. В іншому випадку можна повторно запустити програму в покроковому режимі, щоб виявити, де саме розпочинається невірна поведінка. Проте навіть найкращий налагоджувач може лише допомогти локалізувати помилку, а виправити її повинен програміст.

Однак деякі програми не дуже добре піддаються налагодженню: багатопроцесорні програми, операційні й розподілені системи найчастіше мають налагоджуватися більш низькорівневими засобами. Локалізації помилок у таких ситуаціях може допомогти вставлення в програму спеціальних *налагоджувальних* операторів. Такі оператори неважко вилучити з програми після виявлення й виправлення помилок. Наведемо лише кілька типів налагоджувальних операторів:

1) оператори видачі повідомлень на екран (виведення вхідних даних, проміжних результатів, даних після розрахунків);

2) лічильники, що дозволяють дізнатися про кількість виконань циклів;

3) обчислення значення певного виразу, перед- і постумов (див. підрозд. 2.7.6).

Іншим можливим шляхом локалізації помилок є скорочення вхідних даних до мінімальних розмірів, за яких програма все ще відмовляється працювати. Ще один можливий крок – видалення фрагментів коду, що гарантовано не пов'язані з проблемою.

Слід мати на увазі, що в процесі налагодження системи виправлення одних помилок може призвести до появи інших. Тому система після відповідних виправлень потребує додаткової перевірки.

2.7.6. ТЕСТУВАННЯ

Тестування й налагодження часто згадуються разом, проте це різні етапи у ЖЦ програми (див. підрозд. 2.2). Спрощено можна казати, що *налагодженням* називається процес із виявлення й виправлення помилок, коли відомо, що програма не працює.

Тестування – це послідовні спроби "зламати програму", домогтися помилки від програми, яка вважається працюючою. Воно є одним із найважливіших етапів перевірки якості розробленої системи – знаходження й виправлення помилки на стадії розробки й тестування обходиться набагато дешевше, ніж після того, як готовий продукт потрапить до користувача. Не випадково тестування великих систем стали доручати окремим вузькоспеціалізованим тестувальним бригадам і фірмам.

Дуже важливо тестувати програму на кожному етапі. При цьому потрібно чітко уявляти, що саме тестується та які результати очікуються. Тестування має проводитися послідовно, щоб нічого не пропустити: поточні результати тестування необхідно обов'язково записувати, щоб розуміти, що вже зроблено, а що залишилося зробити.

Серед усіх прийомів тестування виділяють перевірку правильності для стандартних вхідних даних, перевірку спроможності обробити такі вхідні дані, які для своєї обробки вимагають великої кількості ресурсів машини, і перевірку поведінки при некоректних вхідних даних.

Одним із найважливіших видів тестування є перевірка *граничних умов*. Щоразу, написавши невеликий фрагмент коду, наприклад цикл або умовний вираз, перевіряємо, чи тіло циклу повториться потрібну кількість разів, а умовний вираз правильно виконується. Основна ідея полягає в тому, що більшість помилок виникає саме при екстремальних значеннях. Якщо якийсь фрагмент коду містить помилку, то, швидше за все, ця помилка міститься на границі.

Ще один спосіб запобігти виникненню проблем – перевірити, що дані задовольняють певні *перед-* і *постумови*. Останні, зокрема, включають перевірку значень на вході й виході на належність припустимому діапазону значень. У мовах C та C++ існує можливість використання спеціального механізму тверджень, що дозволяє включати у програму перевірку перед- і постумов (бібліотека **<assert.h>**). Оскільки невиконане твердження перериває роботу програми, то використовують його зазвичай у ситуаціях, коли збій не очікується, але теоретично може виникнути й завадити нормальному продовженню роботи.

Одним із прийомів захисту є перевірка значення функцій (у тому числі й бібліотечних). Наприклад, потрібно перевіряти, чи коректно відбулося відкривання чи закривання файла. Для цього достатньо перевірити значення функції **fclose()**, яке дорівнює 0 у випадку помилки.

Виділяють тестування білої й чорної скриньок. Відмінність між ними полягає в тому, чи має розроблювач тестів доступ до вихідного коду

системи, чи тестування виконується через інтерфейс користувача або прикладний програмний інтерфейс, наданий модулем, що тестується.

При *тестування білої скриньки* розроблювач тесту має доступ до вихідного коду й може змінювати його. Це типово для тестування, при якому тестуються окремі частини системи.

При *тестуванні чорної скриньки* тестувальник має доступ до системи тільки через ті інтерфейси, що й замовник або користувач, або через зовнішні інтерфейси, що дозволяють іншому комп'ютеру чи іншому процесу підключитися до системи для тестування. Наприклад, модуль, який тестує, може імітувати натискання клавіші або кнопки миші в програмі за допомогою механізму взаємодії процесів, щоб викликати ту саму відповідь, що й реальні натискання клавіш і кнопок миші.

Створювані сучасні програмні системи є достатньо складними, а їхнє ручне тестування – дуже трудомістким. При нормальному підході необхідно обробити безліч тестів і порівняти їхні результати. Недоліком ручного тестування є й те, що результати виконання тестів не зберігаються та їх важко повторити. Таким чином, при тестуванні складних програмних систем необхідно застосовувати засоби автоматизації. Автоматичні тести дозволяють спростити процес ручного тестування, зробити його зручнішим і точнішим.

Для великих програмних комплексів доводиться розробляти тести різного призначення: модулів, інтеграційні, системні.

Модульне тестування – це тестування програми на рівні окремих модулів, функцій або класів. Мета модульного тестування полягає у виявленні локалізованих у модулі помилок у реалізації алгоритмів, а також визначенні ступеня готовності системи до переходу на наступний рівень розробки й тестування. Модульне тестування проводиться за принципом білої скриньки.

Проте, на жаль, перевірка коректності всіх модулів не гарантує коректності функціонування їхньої системи. Далі використовується *інтеграційне* тестування, коли система будується поетапно, групи модулів додаються поступово. Основне завдання інтеграційного тестування – пошук дефектів, пов'язаних із помилками в реалізації та інтерпретації інтерфейсної взаємодії між модулями.

Повністю реалізований програмний продукт піддається *системному тестуванню*. На даному етапі перевіряється не коректність реалізації окремих процедур і методів, а правильність усієї системи в цілому, як її бачить користувач. Системне тестування проводиться зазвичай за принципом чорної скриньки. Основою для

тестів є загальні вимоги до системи, включаючи не тільки коректність реалізації функцій, але й продуктивність, час відгуку, стійкість до збоїв, атак і помилок користувача тощо.

Як уже зазначалося вище, тестування має проводитися на всіх етапах розробки системи. Тому різні роботи в процесі побудови програм мають узгоджуватися з роботами з тестування. Відповідно інструменти тестування повинні бути добре інтегровані з іншими інструментами розробки систем.

Для автоматизації тестування існує велика кількість спеціальних програм, найпопулярнішими серед яких є Mercury LoadRunner, Segue SilkPerformer та Rational TestStudio. Їхнє використання допомагає на етапах модульного, інтеграційного й системного тестування.

На етапі модульного тестування засоби автоматизації реалізують механізм перевірочних тверджень (assertion). За допомогою *тверджень* можна сформулювати вимоги до вхідних і вихідних даних функцій (методів) класів у формі логічних умов. Аналогічно можна задавати інваріантні вимоги до об'єктів.

На етапі інтеграційного тестування засоби автоматизації забезпечують тестування систем, побудованих за однією з компонентних технологій (напр., UML). Передбачається набір шаблонів для створення різних компонентів тестової програми, зокрема тестів для модулів, сценаріїв.

На етапі системного тестування засоби автоматизації дозволяють створити й реалізувати його сценарій. Відповідна програма запише всю вхідну інформацію, що надходила від користувача (натискання клавіш на клавіатурі, рухи миші тощо) і згенерує відповідний скрипт[28]. Отриманий скрипт можна багаторазово запускати, вносячи в нього за необхідності невеликі зміни. У записі скрипта можна передбачати зупинки, щоб зазначати, які відповіді системи в конкретній ситуації треба розглядати як правильні, які варіації вхідних даних користувача можливі тощо. За наявності таких варіацій при черговому відтворенні тесту програма самостійно буде обирати одну з певних альтернатив. При розбіжності відповіді системи з очікуваною відповіддю буде фіксуватися помилка.

Однак автоматичні тести не можуть повністю замінити ручне тестування. Автоматизація всіх випробувань – дуже дорогий процес, тому автоматичне тестування є лише частиною ручного.

[28] Фрагмент коду.

***Література для СР:** стиль програмування – [53, 54, 90, 127]; проектування й реалізація – [12, 16, 68, 96, 120]; UML – [12, 96]; рефакторинг – [112]; налагодження й тестування – [49, 53, 76, 127, 128]; автоматизація тестування – [18, 30, 35].

Контрольні запитання та вправи

1. Що розуміють під стилем програмування?
2. Що таке модель предметної області?
3. Що таке специфікація ПС і для чого вона застосовується?
4. Що таке візуальне моделювання?
5. Що таке сутності й відношення UML?
6. Перелічити й охарактеризувати основні види діаграм UML.
7. Що таке прикордонні й керівні класи, класи-сутності, абстрактні й конкретні класи?
8. Охарактеризувати відношення узагальнення й залежності між класами.
9. Охарактеризувати відношення асоціації, агрегації й композиції між класами.
10. Що таке діаграми станів і діяльності та яке їхнє призначення?
11. Що таке діаграми компонентів і розгортання та яке їхнє призначення?
12. Опишіть спрощену стратегію використання UML-діаграм при моделюванні ПС.
13. Що таке діаграми прецедентів і для чого вони використовуються?
14. Які відношення між акторами й прецедентами ви знаєте?
15. Які відношення залежності між прецедентами ви знаєте?
16. Що включають етапи кодування, налагодження й тестування програми?
17. Що таке рефакторинг і яке його призначення?
18. Що таке тестування білої й чорної скриньок?
19. Для чого застосовуються модульне, інтеграційне й системне тестування?

20*. Розробити стратегію й провести модульне та інтеграційне тестування аналізатора з прикл. 4.33. Спочатку провести тестування окремих функцій, а потім – усієї програми.

Розділ III

МОВИ ПРОГРАМУВАННЯ С ТА С++

Розділ присвячено опису синтаксису й семантики мов С та С++.

3.1. Лексична й синтаксична структури С-програм

- Стандарти мови С
- Лексика мови С
- Структура програм

Ключові слова**: стандарти С89 та С99, структура С-програми, функція* **`main`, заголовний h-файл, лексема, ідентифікатори, ключові слова, константи, вхідний символ, керівна послідовність, операції й роздільники, роздільна компіляція, оператори опису й декларації, клас пам'яті, зовнішні, автоматичні, регістрові та статичні дані, кваліфікатори типу* **`const`** *та* **`volatile`***, прототип і опис функції, модифіковане тіло функції, область існування даного, ініціалізація даного, глобальні й локальні дані, функція з побічним ефектом.*

За розглянутими нами класифікаціями мова С є універсальною мовою високого рівня. Первісно вона була створена Д. Річі у 1972 р. як робочий інструмент при написанні ОС UNX для машини PDP-11 і реалізована в межах цієї ОС. По суті й сама ОС, і С-компілятор, і майже всі компоненти системи були написані мовою С. Завдяки тому, що С є певним гібридом мови програмування високого рівня з елементами мов внутрішнього рівня, вона в багатьох випадках виявилася зручнішою, гнучкішою й ефективнішою, ніж більш потужні мови. Сьогодні мова С реалізована практично на всіх існуючих ЕОМ. Її популярності сприяє широке застосування ОС UNIX у сучасних системах обробки даних, у тому числі в розподілених системах і мережах.

3.1.1. СТАНДАРТИ МОВИ С

Перший класичний опис мови С побачив світ у 1978 р. [55]. Після цього мова продовжувала вдосконалюватися, з'являлися різні її версії. З метою стандартизації Американський національний інститут стандартів (ANSI) у 1989 р. затвердив стандарт мови С та її бібліотек, відомий як ANSI C. Потім шляхом деяких поправок цей стандарт був перетворений на Міжнародний стандарт ISO/IES 9899:1990, що отримав назву Standart C (1989), або C89. Стандарти ISO/IES Міжнародної організації стандартів періодично переглядаються й оновлюються. Ми будемо дотримуватися стандарту ISO/IES, затвердженого у 1999 р. під назвою Standart C (1999), або C99. Стандарт C99 вніс значні зміни до стандарту C89 як у саму мову, так і в стандартну бібліотеку, але не змінив фундаментальних основ мови, з якими ми ознайомимося в наступних підрозділах. При описі мови С спиратимемося на обидва стандарти – C89 та C99.

3.1.2. ЛЕКСИКА МОВИ С

Як нам уже відомо з підрозд. 2.3.2, програми складаються з однієї чи кількох одиниць трансляції, що вводяться й зберігаються в символьних файлах у вигляді послідовностей кодів літер алфавіту мови. Стандарти мов програмування зазвичай не прив'язуються до конкретних алфавітів, а тільки фіксують їхні обов'язкові частини (вхідні набори) і загальні вимоги до складу.

Вхідні літери. Стандартний вхідний набір літер алфавіту мови С містить такі літери з коду ASCII:

1) 53 літери, що використовуються для утворення ключових слів та ідентифікаторів. До цієї групи належать великі й малі латинські літери, а також символ підкреслювання: A, B, ... , Z, a, b, ..., z, _;

2) 10 цифр: 0, 1, 2, ..., 9;

3) пробіл і три керівних символи: горизонтальної (HT) і вертикальної (VT) табуляції та нова_сторінка (FF);

4) знаки операцій й пунктуатори.

Символи операцій і пунктуатори:

! Знак оклику	+ Плюс	" Лапки	# Знак номера
= Рівність	{ Відкрита фігурна дужка	% Відсоток	~ Тільда
} Закрита фігурна дужка	^ Диакритичний знак	[Відкрита квадратна дужка	, Кома

& Амперсанд	] Закрита квадратна дужка	. Крапка	* Зірочка
' Апостроф	< Менше	(Відкрита дужка	| Вертикальна риска
> Більше	_ Знак підкреслювання	\ Бек-слеш	/ Слеш
) Закрита дужка	; Крапка з комою	? Знак питання	– Мінус
: Двокрапка			

Компілятори можуть розширювати (модифікувати) вхідний набір і включати в нього інші літери алфавіту мови (кодової таблиці), наприклад літери національного алфавіту тощо. Будь-яка літера кодової таблиці може бути подана в програмах за допомогою спеціальних *керівних послідовностей* (*ескейп-послідовностей*), складених із вхідних символів.

Серед літер алфавіту виділяють групу службових порожніх символів, призначених для форматування інформації й виділення лексем, а також два керівні. До порожніх належать чотири вхідні символи: пробіл ' ', символи табуляції: **HT** – горизонтальної, **VT** – вертикальної та **FF** – нова сторінка. Керівними є **LF** – новий рядок і **CR** – повернення каретки. Їхні ескейп-послідовності відповідно мають вигляд **\t**, **\v**, **\f**, **\n** та **\r**. При введенні вхідні файли розбиваються на рядки. Про закінчення рядка свідчить символ (група символів) о*знаки кінця рядка.* Такими ознаками (окрім символу **LF**) є символи **VT**, **CR** та **FF**. Наприклад, для закінчення рядка при консольному введенні використовується символ повернення каретки **CR** (на клавіатурі йому відповідає клавіша Enter). Щоб закінчити рядок, у вхідному потоці достатньо її натиснути. У результаті у вхідний потік буде вставлено символ **LF**, а на екрані консолі курсор опиниться в першій позиції наступного рядка.

У С99 передбачена нотація для використання в лексемах широких літер із наборів UCS-2 та UCS-4. Синтаксис цієї нотації:

```
<універсальний-код-широкої-літери>::=\(u|U) nnnn[nnnn]
n::=<шістнадцяткова-цифра>
```

Записи **\unnnn** та **\Unnnn** еквівалентні. У мові С заборонені універсальні коди, менші ніж 00A0 (=160).

Препроцесорна обробка. Одиниця трансляції проходить фазу препроцесорної обробки до початку синтаксичного аналізу. Препроцесор отримує вхідний файл, виділяє в тексті програм лексеми, здійснює певні їхні перетворення, розпізнає й виконує директиви препроцесора (див. підрозд. 3.9). Результатом роботи препроцесора є

програма у вигляді послідовності лексем, яка надходить на вхід компілятора мови С.

У мові С розрізнюють фізичні й логічні рядки літер. Фізичні – це ті, що відповідають символам форматування у вхідному файлі (напр. **LF**), тобто це підпослідовності сусідніх літер, розділені входженнями таких символів. Препроцесування розпочинається з перетворення фізичних рядків на логічні. Таке перетворення зачіпає тільки фізичні рядки, які закінчуються літерою бек-слеш **** (решта збігаються з логічними). Кілька поспіль фізичних рядків і наступний за останнім із них препроцесор об'єднує в один логічний рядок. При цьому всі входження літери ****, що відповідають за об'єднання, і всі проміжні символи форматування не входять до заключного логічного рядка. Гранична довжина логічного рядка залежить від компілятора. С99 установлює її рівною 4095.

Приклад 3.1. Два вхідні рядки з'єднуються в один логічний:

```
printf ("AAAA\
BBBB\n "); /*закінчення фізичного рядка*/
printf("AAAABBBB\n"); /*еквівалентний логічний рядок*/
```

У другому рядку символ **** є частиною ескейп-послідовності **\n**, а не перенесенням, до того ж він не останній у рядку ■

Нагадаємо, що сукупність лексем довільного речення мови складають: префікс, що передує першому роздільнику лексем, і всі найбільші суфікси, розташовані між роздільниками. Специфікою мов програмування і, зокрема, мови С є те, що всі роздільники лексем, за винятком порожніх символів, самі належать до лексем. Роздільниками лексем є: а) усі порожні символи; б) операції; в) пунктуатори; г) коментарі.

Мова С має п'ять класів лексем: ідентифікатори, ключові слова, константи, операції й роздільники:

Лексеми = ідентифікатори + ключові слова + константи +операції + + роздільники.

Ідентифікатори. Це слова, що розпочинаються з латинської літери або символу '_' і складаються з латинських літер, десяткових цифр і спеціальних символів. У С99 спеціальний символ один – символ підкреслювання '_'. Довжина ідентифікаторів обмежена 31 літерою.

Ключові слова. Деякі ідентифікатори мають спеціальне призначення в мові. Їх називають *ключовими*, або *службовими, словами*. Ключові слова не можуть використовуватись як ідентифікатори. Усі ключові слова наведено в табл. 3.1.

Таблиця 3.1. **Ключові слова мови С (С99)**

auto	double	int	struct	break	else
register	typedef	char	extern	return	void
unsigned	default	for	signed	union	do
volatile	continue	enum	short	while	static
_Bool	_Imaginary	restrict	_Complex	inline	long
const	switch	float	sizeof	goto	case
if					

Константи. У С виділяють чотири типи констант: цілі, константи з рухомою точкою, символьні й літерали.

Увага! Розпочинаючи з визначення символьних констант і далі, для опису синтаксичних конструкцій будемо користуватися розширеними БНФ (див. підрозд. 1.4.4). Для перенесення продовження конструкції на наступний рядок будемо використовувати символ бек-слеш **** ►

Символьні константи формуються з літер алфавіту й записуються за допомогою апострофа:

```
<символьна-константа>::='<вхідний-символ-кодової-таблиці>' |
                        '<керівна-послідовність>' |
                       L'<широка-символьна-константа>'
```

Керівні послідовності наведено в табл. 3.2. Вони подають деякі керівні символи, а також дозволяють задавати всі літери кодової таблиці за допомогою їхніх числових кодів (вісімкових і шістнадцяткових). При цьому довжина вісімкової послідовності обмежена трьома літерами, а шістнадцяткової – довільна. С99 забороняє всі коди, що виходять за межі беззнакового символьного типу (**unsigned char**).

Таким чином, щоб отримати символьну константу, необхідно взяти в лапки потрібну вхідну літеру або керівну послідовність.

Префікс **L** перед символьною константою свідчить про запис широкої константи. Широка символьна константа зображується або за допомогою універсального коду певної широкої літери, або послідовністю символів, серед яких можуть бути й керівні, які формують один багатобайтний символ. Спосіб такого формування залежить від реалізації мови.

Таблиця 3.2. **Керівні послідовності**

Керівна послідовність	Значення	Десяткова	Шістнадцяткова	Символьна
\a	Дзвоник	7	0x07	BEL
\b	Крок_назад	8	0x08	BS
\t	Горизонтальна_табуляція	9	0x09	HT

Закінчення табл. 3.2.

Керівна послідовність	Значення	Десяткова	Шістнадцяткова	Символьна
`\n`	Новий_рядок	10	0x0a	`LF`
`\v`	Вертикальна_табуляція	11	0x0b	`VT`
`\r`	Повернення_каретки	13	0x0d	`CR`
`\f`	Нова_сторінка	12	0x0c	`FF`
`\"`	Подвійна_лапка	34	0x22	`"`
`\'`	Апостроф	39	0x27	‘
`\0`	Нуль-символ	0	0x00	
`\\`	Бек-слеш	92	0x5c	`\`
`\?`	Знак_питання	63	0x3f	`?`
`\ццц`$_8$	Символ із вісімковим кодом ццц$_{(8)}$	ццц$_{(8)}$	(ццц$_{(8)}$)$_{(16)}$	Ідеограма символу
`\хц...ц`$_{16}$	Символ із шістнадцятковим кодом ц…ц$_{(16)}$	ц…ц$_{(16)}$	ц…ц$_{16}$	Ідеограма символу

Приклад 3.2. Символьні константи:

1) символи в апострофах: `'a', 'A' , '/', '9'`;

2) вісімкові й шістнадцяткові коди: `'\007', '\x7'` – символ Дзвоник;

3) у вхідному рядку `\0007` препроцесор розпізнає два символи: нуль-символ і `'7'`, а в рядку `abc\025\xFF 125` – дев'ять символів – `'a', 'b', 'c', '\025', '\xFF',' ','1', '2', '5'` ■

Літерали – це послідовність літер, розміщених у подвійних лапках: "…" (не плутати символ подвійних лапок із двома апострофами). Підкреслюємо, що до складу літерала входять саме літери, а не символьні константи в апострофах. Наприклад, "факультет кібернетики". Лапки не входять до складу літерала. Для включення в літерал подвійних лапок і літер `\` та `LF` використовуються керівні послідовності – відповідно `\",\\` та `\n`. Наприклад, `"\"Гайдамаки\"\n"` – правильний літерал.

Зберігаються літерали у статичній пам'яті. Якщо n – довжина літерала, то для нього буде виділено $n+1$ байтів пам'яті – по одному для коду кожної літери й додаткового символу `'\0'` із кодом `0`. Останній додається обов'язково – як ознака закінчення поля літерала.

Приклад 3.3. Літерали в пам'яті

Літерал	Подання в пам'яті										
`""`	0										
`"UEFA-2012\n"`	85	69	70	65	45	50	48	49	50	10	0
`"25/*int*/"`	50	53	47	42	105	110	116	42	47	0	
`"\"Дніпро\""`	34	E4	ED	B3	EF	F0	EE	34	0		

У першому рядку подано порожній літерал. Довжина його нульова, а довжина поля в пам'яті становить 1. У другому рядку літерал містить керівний символ **LF** – новий_рядок із кодом **10**, у четвертому – усередині лапки " ■

Операції й роздільники. Лексеми операцій і пунктуатори наведені в табл. 3.3.

***Таблиця 3.3.* Лексеми операцій і пунктуатори**

Підклас лексем	Лексеми
Прості операції	`!|%|^|&|*|-|+|=|~|||.|,|<|>|/|?|,`
Складені присвоювання	`+=|-=|*=|/=|%=|<<=|>>=|&=|^=||=`
Складені операції	`->|++|--|<<|>>|<=|>=|==|!=|&&`
Пунктуатори	`(|)|[|]|{|}|;|:|#|'|"|\`

Коментар – це довільна послідовність символів, розташована між парами символів **/*** та ***/**, що не містить усередині іншої пари ***/**. У стандарті С99 коментарями є також підрядки, що розпочинаються двома символами слеш **//** і закінчуються символом **LF**. Коментарі не розпізнаються як лексеми в символьних константах і літералах (див. прикл. 3.3), не задають жодних дій і замінюються препроцесором на один пробіл. Їхнє основне призначення – пояснити в тексті програми окремі її конструкції (див. прикл. 3.5).

Увага! У програмах будь-які сусідні константи, ідентифікатори та службові слова мають розділятися одним або кількома порожніми символами. Пробіли між сусідніми лексемами необов'язкові, якщо однією з них є роздільник або символ операції ►

3.1.3. СТРУКТУРА ПРОГРАМ

Основними елементами структури С-програм є функції й директиви препроцесора. Функції здійснюють обробку даних у С-програмах, а препроцесор готує програму до компіляції, зокрема здійснює синтаксичні перетворення тексту програми згідно з директивами препроцесора.

С-програма – це послідовність функцій, директив препроцесора й операторів опису й декларації даних, яка обов'язково містить функцію з іменем `main`.

Підключення файлів. Кожна директива розпочинається літерою #. Щоб не писати в програмах кожний раз деякі стандартні оголошення, їх можна розмістити один раз в окремих *заголовних файлах* і підключати (вставляти) за необхідності у С-файли під час препроце-

сування. Це, окрім інших переваг, сприяє скороченню текстів програм. Заголовні файли можуть самі використовувати інші заголовні файли. Подібні підключення здійснює директива **#include:**

<директива_підключення>::= #include \<file1\>| #include "file2" [23]

Файли в обох варіантах директиви відрізняються лише місцем їхнього розташування. Перший файл **file1** препроцесор буде шукати тільки у стандартних бібліотеках, наприклад у папці INCLUDE, а файл **file2** – там само, але після того, як його не буде знайдено в поточному каталозі. Зазвичай це файли, підготовлені самим програмістом. Дія директиви полягає у вилученні самої директиви з тексту програми і вставленні (підключенні) на її місце всього вмісту текстових файлів **file1** або **file2**.

Препроцесорні константи. Препроцесор дозволяє вводити в текст програм власні (препроцесорні) константи:

```
#define <ІМ'Я-КОНСТАНТИ> <слово>
```

Ім'я та слово обов'язково мають бути розділені в директиві принаймні одним пробілом. У слові праворуч немає якоїсь спеціальної ознаки кінця. Щоб виділити імена препроцесорних констант у тексті програм, їх краще писати великими літерами. Директива наказує препроцесору замінити всі наступні входження в програмі ідентифікатора константи на відповідне слово.

Приклад 3.4. Наведені директиви підключають до програми заголовний файл **<stdio.h>**, що підтримує стандартні функції консольного введення-виведення, і визначають препроцесорні константи **N** та **ALPHA**:

```
# include <stdio.h> /*необхідний для забезпечення в/в*/
#define N 10000
#define N=10000 /*невірна директива: між N та 10000 відсутній
 пробіл*/
#define ALPHA "abcdefgheijklmnopqrstuvwxyz" ■
```

Роздільна компіляція С-програми. Складові елементи С-програми розміщуються в текстових файлах, що належать певній одиниці компіляції. Окремий файл може містити кілька елементів із набору, а вся програма – складатися з кількох одиниць компіляції (*роздільна компіляція*).

[23] У формулі праворуч літери ‘<’ ,‘>’ є термінальними, а ліворуч – метасимволами.

Кожна з одиниць компілюється окремо й об'єднується в один образ задачі вже на етапі компонування програми.

Файли С-програм бувають двох типів: *заголовні,* або *h-файли* (мають розширення **.***h*), і *С-файли* (мають розширення **.**c). Імена С++-файлів мають розширення **.**cpp, **.**cxx або **.**cc.

Щоб отримати уявлення про С-програму, розглянемо приклад простої програми з кількома одиницями компіляції, а потім повернемося до загальної структури програм.

Приклад 3.5. Роздільна компіляція. Нехай необхідно створити програму **prog**, що знаходить куб суми й суму кубів двох введених із клавіатури чисел і виводить їх на екран.

Програма складається з кількох файлів. У першому – **funcs.c** – розміщуються описи функцій для обчислення куба суми й суми кубів двох чисел, у другому – **start.c** – забезпечується введення двох чисел, знаходження від них куба суми й суми кубів і виведення їх на екран.

Зауважимо, що у С-програмах діє правило: першій появі виклику будь-якої функції (у тому числі й стандартної) має передувати її прототип – заголовок із символом ';' у кінці. Прототипи стандартних функцій розташовані в заголовних файлах, прототипи решти – безпосередньо в текстах С-програм. Щоб прототип потрібної стандартної функції потрапив у текст С-програми, необхідно до неї підключити відповідний заголовний файл. Таке підключення здійснюється директивою препроцесора **#include** і зводиться до текстової заміни директиви на текст заголовного файла, що підключається.

Лістинг **funcs.c**:

```
#include <math.h> /*підключення заголовного файла math.h, що міс-
тить прототип стандартної функції pow для обчислення степеня*/[24]
/*опис функції обчислення куба суми x та y*/
int cube_sum(int x, int y)
{
return pow(x+y,3);
}
/*опис функції обчислення суми кубів x та y*/
int sum_cube(int x, int y)
{
return pow(x,3)+pow(y,3);
}
```

[24] **pow(x,y)** = x^y .

Файл **start.c** містить основну частину програми, в якій вводяться з клавіатури два дійсні числа й за допомогою функцій **cube_sum** і **sum_cube** знаходяться потрібні числа, які, у свою чергу, виводяться на екран. Для того, щоб програмно об'єднати файли **funcs.c** та **start.c**, в останній вводять прототипи функцій **cube_sum** та **sum_cube** з інформацією про те, що їхні описи містяться в іншому файлі (кваліфікатор **extern**).

Лістинг **start.c**:

```
  #include <stdio.h> /*підключення файла stdio.h із прототипами
стандартних функцій введення scanf і виведення printf*/

  /*прототипи зовнішніх функцій cube_sum та sum_cube*/
  extern int cube_sum(int x, int y);
  extern int sum_cube(int x, int y);

  /*maim - головна функція*/
  int main()
  {int x,y;

   /*читання з клавіатури значень і присвоювання їх змінним x,y*/
  scanf("%d%d",  &x,&y);    /*про функцію scanf див. підрозд.
3.8.2*/

  /*виведення на екран значень функцій cube_sum(x,y) та
sum_cube(x,y)*/
  printf("\n       cube_sum=%d\nsum_cube=%d",       cube_sum(x,y),
sum_cube(x,y)); /*про функцію printf  див. підрозд. 3.8.2*/
   }
```

Програма складається з двох одиниць компіляції. Перша – файл **funcs.c**, друга – файл **start.c**. Можна було б вибрати простішу структуру програми й одразу описати функції у файлі **start.c** та отримати програму у вигляді однієї одиниці компіляції. Однак запропонована структура гнучкіша. Вона дозволяє мінімальними зусиллями модифікувати програму, не змінюючи її загальної структури, а за необхідності додати нові функції для обробки чисел або замінити існуючі на інші.

У системі UNIX для препроцесування, компілювання, компонування й виконання програми **prog** потрібно видати дві команди мовного процесора:

```
% cc -o prog func.c start.c
% prog
```

Перша команда компілює й компонує два вхідні файли в образ задачі з іменем **prog**, друга – завантажує програму **prog** і виконує її. Якщо ввести числа 2.0 та 3.0, то на екрані отримаємо результат:

```
2.0 3.0
cube_sum=125
sum_cube=35
```

У сучасних ОС замість явного застосування інструкцій командного процесора використовують відповідні кнопки інтегрованого середовища розробки IDE ■

Заголовні файли. Прикл. 3.5 показує, як заголовні файли використовуються для організації взаємодії різних частин програми. Основна задача *h*-файлів – описати або попередньо задекларувати спільні для кількох одиниць трансляції (або програм) дані й функції з метою їх використання в програмі ще до повного опису.

Задають такі визначення *оператори опису* й *декларації*. Скорочено будемо називати їх *описом* і *декларацією* даних. Декларації функцій мають спеціальну назву – *прототипи*.

Декларації даних (функцій) надають інформацію, достатню, щоб ними можна було оперувати в програмі. Ця інформація обов'язково містить ім'я даних (не стосується параметрів у заголовках функцій), клас пам'яті й тип значень. Інформація про тип, зокрема, визначає можливості доступу до даних і коло операцій над їхніми значеннями. При цьому може залишатися відкритим питання про розмір об'єктів типу, а у випадку функцій – про конкретну дію функції тощо.

Увага! Оператори декларації не пов'язують з іменем конкретний об'єкт у ОП ►

Загальний вигляд оператора декларації:

```
<оператор-декларації>::=[<клас_пам'яті>]   [<кваліфікатор_типу>]
<тип>\
<специфікатор-імені>{,<специфікатор-імені>};
<клас_пам'яті>::=extern| static|auto|register
<кваліфікатор_типу>::=const | volatile | restrict
<тип>::=<специфікатор_типу>
```

Специфікатори імені подають імена змінних (функцій) і додаткову інформацію про їхній тип:

```
<специфікатор-імені>::=<прямий-специфікатор> |
<специфікатор-покажчик>
```

```
<прямий-специфікатор>::=<простий-специфікатор>|
(<специфікатор-імені>) |
<неповний-специфікатор-масиву>|
<специфікатор-функції>
```

Прості специфікатори використовують для декларації тих змінних, специфікатори типу яких повністю визначають тип: це стосується арифметичних і зліченних типів, структур і об'єднань, типу **void**. Розглянемо приклад:

```
int n;
struct S {int counter; float a} x;
```

В обох випадках специфікатори цілого типу **int** і типу структури **struct S {int counter; float a}** повністю визначають тип змінних, тому для декларації відповідних змінних достатньо простих описувачів – ідентифікаторів **n** та **x**. Конкретна будова специфікаторів змінних залежить від типу змінних і буде розглядатися окремо для кожного з відповідних типів.

Специфікатори типу надають інформацію про тип змінної. Як уже зазначалося, ця інформація може доповнюватися специфікатором змінної:

```
<специфікатор_типу>::=<специфікатор-зліченного-типу>
<специфікатор_цілого_типу>
<специфікатор_дійсного_типу>
<специфікатор_типу_структур>
<специфікатор_типу_об'єднання>
<специфікатор_імені_typedef>
<специфікатор_типу_void>
```

Специфікатори типу розглядатимемо в підрозділах, присвячених конкретним типам.

Мова С є мовою із сильною типізацією – це означає, що перед використанням змінна в програмі має бути принаймні задекларована.

Увага! Діє правило:

Кожна константа, змінна чи функція програми:

1) *мають бути описані оператором опису;*

2) *якщо вони не описані перед своїм першим застосуванням у тексті програми, то обов'язково мають бути перед цим задекларовані* ►

Перший пункт обумовлений тим, що тільки при описі даного чи функції під них фактично виділяється пам'ять, тобто вони зв'язуються з реальним об'єктом в ОП і розпочинають діяти інтерфейсні функції доступу до них.

Клас пам'яті й час існування даних. Компілятор, аналізуючи інформацію про програмні дані, розподіляє їх за класами пам'яті. Стандарт С містить чотири класи пам'яті:

- зовнішня (**extern**);
- статична (**static**);
- автоматична (**auto**);
- регістрова (**register**).

Зовнішні дані розміщуються в глобальній пам'яті й доступні в усіх функціях програми протягом усього часу її виконання. Це означає також, що до них буде застосовано зовнішнє зв'язування, тобто на етапі компонування в різних об'єктних модулях однойменні дані будуть зв'язані з одним об'єктом. Це стосується насамперед функцій, які за умовчанням мають клас пам'яті **extern**.

З поняттям даного пов'язане таке поняття, як час існування в процесі виконання програми. *Час існування* даного – це час існування в ОП об'єкта, що зберігає значення даного, тобто відрізки часу з моменту створення об'єкта в ОП до його руйнації.

Автоматичні дані розміщуються на стеку під час виклику функції й доступні тільки всередині неї, де вони описані. Після завершення виклику доступ до них припиняється. Тому час існування автоматичних даних – виклики функцій.

Регістрові змінні розміщуються на регістрах.

Статичні дані розміщуються в купі один раз і не змінюють свою адресу до кінця виконання програми. Якщо вони описані в тілі функції, то доступні тільки всередині функції та зберігають свою адресу й значення між викликами функції.

Час існування зовнішніх і статичних даних – весь час виконання програми. У С дозволяється також створювати й знищувати об'єкти в процесі виконання програми програмними засобами. Час існування таких об'єктів визначається в ручному режимі й регулюється програмістом. Подібні об'єкти зберігаються в купі, іменуються за допомогою покажчиків і називаються *динамічними*.

У програмах можуть бути також присутні об'єкти, що існують тільки в процесі їхньої компіляції. До них належать типи, що визначені специфікатором типу **typedef**, теги типів і порядкові типи.

В оголошенні може бути задано не більше одного специфікатора класу пам'яті. Якщо в оголошенні відсутній специфікатор класу пам'яті, то спрацьовує правило умовчання (табл. 3.4), яке враховує контекст декларації.

Таблиця 3.4. **Правило умовчання для класів пам'яті**

Розташування	Вид	Клас пам'яті за умовчанням
Верхній рівень	Усі	**Extern**
Заголовок функції	Усі	Відсутній
Усередині блока	Функції	**Extern**
Усередині блока	Решта	**Auto**

Кваліфікатори типу. Впливають на засоби доступу до змінної. Стандарт С89 визначає два кваліфікатори: **const** та **volatile**. У С99 з'являється третій – **restrict**. Класифікатор типу передує специфікатору типу у визначенні даного.

Змінні з кваліфікатором **const** доступні тільки для читання. Класифікатор **const** часто використовується, наприклад, для того, щоб запобігти модифікації функцією об'єкта, з яким зв'язується параметр функції в процесі її виклику. Без нього функція може змінити цей об'єкт.

Приклад 3.6. Опис константи й декларації параметра з класифікатором **const**:

```
const int a=10; /*ціла константа зі значенням 10*/
char *strcat (char *s1, const char *s2); /*символьний параметр
– константа*/
```

Дія функції **strcat** полягає в додаванні до рядка символів першого аргументу символів рядка другого аргументу. Зрозуміло, що перший рядок мусить змінитися, а от кваліфікатор **const** перед другим гарантує збереження його значення на виході з функції ■

Значення змінної з кваліфікатором **const** не може змінюватися в програмі, але може змінюватися внаслідок зовнішнього впливу. Наприклад, адресу глобальної змінної-константи можна передати у функцію ОС, що стежить за часом, і тоді змінна буде відображати системний час. У цьому випадку її значення буде змінюватися примусово без участі операторів програми.

Такі подробиці важливі для компіляторів, тому що більшість із них автоматично оптимізує вирази, які не змінюють свої значення. Якщо у виразі присутня змінна, що не змінюється явно операціями присвоювання, проте змінюється поза програмою, то така оптимізація коду може виявитись некоректною. Кваліфікатор **volatile** запобігає

подібній оптимізації, указуючи компілятору, що значення змінної може змінитися незалежно від програми.

Приклад 3.7. Опис змінної з кваліфікатором **volatile**:

```
volatile int contDay=7; ■
```

Кваліфікатор **restrict** пов'язаний із покажчиками й буде розглянутий у підрозд. 3.5.1.

С-файли. Складаються з прототипів і описів функцій, декларацій і описів зовнішніх змінних, а також директив препроцесора. Оператор *опису функції* має таку структуру:

```
<опис_функції>::=<заголовок_функції><тіло_функції>
<заголовок_функції>::=[<клас_пам'яті>]<скалярний-тип>\
<специфікатор-функції>
<скалярний_тип>::=<базовий-тип>|<тип-покажчиків>|<тип-void>
<тіло_функції>::=<блок>
<блок>::={<оператор>{,<оператор>}}
```

Опис функції складається із заголовка й тіла функції. Останнє описує дію функції. *Заголовок* розпочинається з класу пам'яті функції як об'єкта (**extern** або **static**), потім описується тип значення, яке повертає функція. Це може бути один зі скалярних типів, наприклад цілий – **int** або тип покажчиків на ціле – **int***. Якщо цей тип є типом **void**, то кажуть, що функція не повертає значення, а діє як певний оператор.

Потім іде специфікатор функції. У найпростішому варіанті – це просто певний ідентифікатор – ім'я функції, за яким у круглих дужках іде список декларацій параметрів.

Зазначимо, що в прототипах функції при декларуванні параметра можна обмежитися тільки його типом. Наприклад, прототипи функції **strlen: long strlen (const char *s)** та **long strlen (const char *)** еквівалентні. Це єдине, що синтаксично відрізняє заголовки функцій у прототипах від заголовків функцій у їхніх описах.

Повний синтаксис специфікатора функцій наведено в підрозд. 3.6.1.

Тіло складається з послідовності операторів (можливо порожньої), розміщеної у фігурних дужках. Така конструкція називається блоком (див. підрозд. 3.3.2). Серед операторів тіла можуть бути й оператори опису даних, що визначають власні дані функції.

З погляду семантики функція як програмна одиниця задає алгоритм для обчислення певної X-арної U-функції або $X-Y$-оператора, де специфікатор скалярного типу визначає множину U значень функції, а склад фреймів X та Y визначається структурою програми.

Дія алгоритму полягає у виконанні модифікованого тіла функції. Як саме відбувається модифікація тіла функції перед його виконанням і як обчислюється результат функції, буде сказано нижче. Наразі поговоримо про структуру і склад *X*-фрейму, який обробляється тілом функції. Його складають дані й об'єкти, описані у функції й поза нею.

Опис змінних та інших об'єктів. Дані, з якими має справу програма, розподіляються на проблемні та службові. *Проблемні* – це дані, описані в програмі, над якими безпосередньо проводяться обчислення, а *службові* – це об'єкти, що використовуються для опису проблемних даних і організації обчислень. До службових даних належать: типи, теги типів, мітки операторів, макроси препроцесора, регістр стану процесора PS, лічильник команд PC, параметри ОС тощо.

Системні дані, доступні в програмі й на які вона може впливати, залежать від компілятора й не описуються у стандарті мови. Усі решта мають опис.

Змінна визначена в підрозд. 2.1.2 як інформаційний об'єкт, що може під час інформаційного процесу набувати різних значень. У мовах програмування поняття змінної набуває подальшої конкретизації. По-перше, воно типізується (див. підрозд. 2.2.2), по-друге, з ним пов'язується певний операційний (машинний) аспект. Нагадаємо, що кожній змінній програми на машинному рівні відповідає об'єкт ОП, в якому зберігається його значення. Інтерфейсні функції за іменем даного здійснюють доступ до цього об'єкта (його частини). Розрізняють п'ять рівнів доступу в програмах:

0 – відсутність доступу;
1 – доступ для читання (для констант);
2 – доступ для запису (для змінних);
3 – повний доступ (для читання й запису);
4 – повний доступ до копії даного.

В останньому випадку доступ до самого даного блокується.

Увага! У мові С поняття змінної й константи об'єднуються в одне – змінної, а константами вважаються змінні з доступом тільки для читання (змінні з кваліфікатором типу **`const`**) ►

Фрагмент програми, в якому змінна має ненульовий рівень доступу, називається її *областю існування* в програмі.

Як зазначалося вище, кожна змінна має бути описана в програмі. Відмінність між декларацією й описом змінної полягає в тому, що останній повністю визначає її дескриптор і зв'язує зі змінною об'єкт у пам'яті.

Оператор *опису змінних* має вигляд:

```
<опис-змінної>::=[<клас_пам'яті>][<кваліфікатор_типу>]<тип>\
<специфікатор-імені>[<ініціатор>]\
{,<специфікатор-імені>[<ініціатор>]};
```

Якщо не враховувати, що опис змінної може включати ініціатор, то синтаксичні відмінності між деклараціями й описами змінних мають місце тільки в специфікаторах імені та стосуються тільки масивів і похідних від них типів:

```
<специфікатор-імені>::=<прямий-специфікатор>|
<специфікатор-покажчик>
<прямий-специфікатор>::=<простий-специфікатор>|
(<специфікатор-імені>)|
<специфікатор-функції>|
<специфікатор-масиву>
```

Якщо специфікатор масиву в декларації може не уточнювати його розмір (неповний специфікатор), то специфікатор в описі масиву робить це обов'язково.

Приклад 3.8. Декларації й опис цілого масиву:

```
int array[]; /*декларація масиву array із цілими компонентами*/
int array[5]; /*опис масиву array з п'ятьма цілими компонентами*/
```

У декларації специфікатор **array[]** тільки засвідчив, що змінна **array** є цілим масивом, у той час як **array[5]** довершив визначення масиву **array** як цілого з п'ятьма компонентами ■

Детальніше про специфікатори масиву див. у підрозд. 3.4.1.

Оператор опису змінної за допомогою *ініціатора* може надати початкове значення її об'єкту в момент зв'язування.

Приклад 3.9. Ініціалізація змінних:

```
{
char ch='#';
int n=256;
int array[5]={10,20,30,40,50};
…
}
```

Після виконання операторів опису змінна **ch** набуде значення 35, змінна **n** – значення 256, а компоненти масиву **array** – відповідно значень 10, 20, 30, 40 та 50 ■

Про специфікатори імен та ініціатори змінних будемо говорити детальніше при розгляді відповідних типів.

Описи змінних і функцій можуть мати досить складну внутрішню будову. Наприклад, специфікатори імен можуть містити інші (вкладені) специфікатори імен тощо. Щоб спростити подібні описи й додати виразності програмам, у мові С існує оператор **typedef**, який дозволяє вводити синоніми для будь-яких типів. Синтаксично він збігається з описом змінної, якщо в ньому елемент **<клас-пам'яті>** замінити на службове слово **typedef**. Тоді ім'я у специфікаторі імені буде не іменем змінної, а новим іменем типу, який описується в даному операторі. Нові імена повністю замінюють типи у відповідних контекстах.

Приклад 3.10. Оператор **typedef**:

```
typedef unsigned char ascii; /*ascii – символьний тип для
подання ASCII-кодів*/
typedef const char *arg; /*arg – тип константних символьних
покажчиків для аргументів функцій*/
typedef double (matr[80])[2];/*matr – тип двовимірних
масивів для подання прямокутних
матриць розміром 80×2*/
ascii ch; /*ch : беззнакова символьна змінна*/
matr aa; /*aa : двовимірний дійсний масив*/
long strlen (arg s); /*s – константний символьний
покажчик*/
double maltyply (matr x,matr y);*/x,y – двовимірні дійсні
масиви*/ ■
```

У зв'язку з деклараціями й описами зовнішніх імен виникає проблема їхнього узгодження в програмах, що розміщуються в кількох файлах. Наприклад, що відбудеться, якщо два описи однієї й тієї самої змінної будуть у різних файлах ініціалізувати змінну по-різному? Щоб уникнути неоднозначності в подібних ситуаціях, виділяють один опис змінної як *визначальний*. Усі решта її описів і декларацій мають статус вторинних, підпорядкованих визначальним. Компілятори порізному здійснюють цей розподіл. Для більшості компіляторів буде прийнятним таке правило:

1) *Кожну зовнішню змінну в програмі описувати тільки в одному файлі. У визначальному описі не вказувати специфікатор* **extern** *та ініціалізувати змінну: наприклад*, **char c2='0'.**

2) *У всіх інших файлах описана зовнішня змінна обов'язково декларується зі специфікатором* **extern**: *наприклад*, **extern char c2.**

3) *Якщо в зовнішньої змінної немає ініціатора, то компілятор ініціалізує її за умовчанням зі значенням* 0.

Приклад 3.11. Функції та змінні в різних файлах програми.

Перший файл:	Другий файл:

`int i=1, j=1;`
`char c='a';`
`void int f1(void);`
`void int f2(void);`
 	
`int main(void)`
`{`
`f1();`
`f2();`
`}`
	`extern int i,j;`
	`static char c='b';`
	
	`void int f1(void)`
	`{i=j+2;}`
	
	`void int f2(void)`
	`{j=j+c;}`

Зовнішні імена функцій **f1** та **f2** декларуються в першому файлі й описуються в другому. У другому файлі специфікатор **extern** декларує змінні **i** та **j** як зовнішні, а визначальним є опис їх у першому файлі. На етапі компонування об'єктних модулів після роздільної компіляції файлів до функцій і змінних буде застосоване зовнішнє зв'язування – вони будуть зв'язані з об'єктами своїх тезок, описаними у відповідних файлах. Специфікатор **static** оголошує символьну змінну **c** у другому файлі як статичну й обмежує її область існування тільки цим файлом. Це означає, що зовнішня змінна **c** у першому файлі й дана статична змінна **c** різні й до них буде застосоване внутрішнє зв'язування ■

Глобальні й локальні дані. *X*-фрейм даних, що обробляється функцією, складають: 1) параметри функції; 2) змінні й константи, описані в тілі функції; 3) глобальні дані (тобто описані за межами функції), які в тілі функції мають непорожню область існування.

Дані 1)–2) називаються *локальними.* Локальні змінні є *автоматичними*, тобто мають клас пам'яті **auto** і зберігаються в автоматичній пам'яті (на стеку) або в купі. Їхня область існування – блок, в якому вони декларовані або описані. В іншому блоці може визначатися нова змінна з таким самим іменем. Зазвичай локальні змінні оголошуються на початку блока, одразу після фігурної дужки, але за C99 їх можна оголошувати й усередині блока [25].

[25] У стандарті C89 усі локальні змінні мають бути описані обов'язково на початку блока.

Між даними блока існує певний ієрархічний порядок, пов'язаний із тим, що блок може містити інші оператори-блоки. Тоді про дані самого верхнього блока можна говорити як про дані першого рівня, про дані вкладеного в нього блока – як про дані другого рівня, що є родичами даних першого рівня, про дані двічі вкладеного блока – як про дані третього рівня, що є родичами відповідних двох верхніх блоків тощо. Якщо в блоці визначено кілька сусідніх блоків, то в ньому можуть існувати паралельні однойменні дані – не родичі, які мають однаковий рівень.

Приклад 3.12. Дані різних рівнів:

```
{int n,m; /*змінні першого рівня*/
{char ch; /*змінна другого рівня – родич змінних n,m*/
int n; /*змінна другого рівня – родич змінних n,m*/
}
{int k; /*змінна другого рівня – родич n,m, але не родич
змінної ch*/
}
}
```

■

Усі дані, визначені в блоках меншого рангу відносно даного блока, називаються *глобальними* відносно нього. Якщо імена глобальної змінної та змінної, визначеної у внутрішньому блоці, збігаються, то змінна блока приховує (затіняє) у ньому глобальну змінну. У внутрішньому блоці глобальна змінна має нульовий рівень доступу там, де діє опис однойменної змінної, і ненульовий – поза межами його дії.

Зовнішні змінні (з класом пам'яті **extern**) є глобальними відносно всіх функцій, що містяться в їхній області існування. Вони зберігають своє значення протягом усього часу виконання програми. Щоб створити глобальну змінну, достатньо її визначити за межами функції. Зовнішній клас пам'яті призначається кожній такій змінній за умовчанням, тобто без кваліфікатора **extern**. Область існування такої змінної закінчується кінцем файла. Щоб розповсюдити її на інші файли, необхідно повторити в них декларації цієї змінної та явно вказати клас пам'яті **extern**.

Усі імена функцій (за умовчанням) мають статус зовнішніх і кваліфікуються як функціональні константи. Їхніми значеннями є адреси полів із підпрограмами, що обчислюють ці функції.

Приклад 3.13. Область існування змінних у програмі.

Літери 'i' , 'c' та 'f', що розпочинають ідентифікатори, указують на тип змінної (відповідно **int**, **char**, **float**).

Лістинг **variable1.c**:

```
1:  #include <stdio.h>
2:
3:  int Var=100;
4:
5:  void Func1(float fX, int iX)
6:  {
7:        int i=1, j=10;    /*локальні змінні*/
8:
9:        printf("\ni=%d", i);
10:       printf("\nj*Var=%d\niX+fX=%f", j*Var, iX+fX);
11: }
12:
13: void Func2(int iX)            /*iX – формальний параметр*/
14: {
15:       int i;
16:
17:       printf("\nInput i:");
18:       scanf("%d", &i);        /*введення i*/
19:
20:       if(i==1)
21:       {
22:             char cS='a';
23:             int i=10;
24:             printf("\ncS=%c\ni=%d", cS, i);
25:       }
26:
27:       printf("\ni=%d\niX*i=%d", i, iX*i);
28: }
29: int main(void)
30: {
31:       int iX=10;
32:
33:       Func1(2.5, iX);
34:       Func2(Var);
35:
36:       return 0;
37: }
```

У рядку 3 описано й проініціалізовано цілочислову глобальну змінну **Var** з областю існування від рядка 3 до рядка 37.

У рядку 5 описана функція **Func1**, що містить формальні параметри **fX** та **iX**. Формальні параметри існують у межах функції (від рядка 5 до 11).

У 7-му рядку описані цілочислові локальні зміні **i** та **j** функції **Func1**, що існують у межах тіла функції (від рядка 6 до 11). Змінній **j** присвоєне початкове значення 10.

У рядку 10 окрім локальних змінних використовується й глобальна змінна **Var**.

Рядок 13 містить опис функції **Func2** із формальним параметром **iX** і областю існування – рядками 13-28. Зверніть увагу, що параметр **iX** присутній в обох функціях **Func1** та **Func2**, але це різні змінні.

У 15-му рядку описана локальна змінна **i**. Область її існування – тіло функції **Func2**. Змінна **i** також присутня у функції **Func1**, але це різні змінні.

У рядку 22 описується локальна змінна **cS**, яка демонструє можливість опису змінних усередині блока. Область існування цієї змінної – блок (рядки 22-25).

У рядку 23 описується локальна змінна **i** з областю існування – рядками 23-25. Глобальна змінна з таким іменем уже існує у функції **Func2**, тому з її області існування (рядки 15-28) вилучаються рядки 23-25, що складають область існування однойменної змінної, яка її перекриває ■

Глобальна змінна може бути використана в будь-якому блоці програми. Визначати глобальні змінні рекомендується у верхній частині програми. При цьому слід ураховувати, що програма може складатися з кількох одиниць трансляції, а статичні зовнішні дані (з класом пам'яті **static**) доступні тільки в межах файла, де вони описані. Щоб зберегти значення локальної змінної між викликами функції, її потрібно описати з класом пам'яті **static.** *Статична* змінна після зв'язування з полем в ОП не змінює своєї адреси до кінця роботи програми.

Виклик функції. Повернемося до процесу обчислення значення функції. Нехай

```
T func (T1 x1,…,Tn xn) {O1… Om}
```

– опис певної функції з іменем **func**, де **T** – тип її значення, який не збігається з типом **void**; **x1,…,xn** – параметри з типами **T1,…,Tn**; **O1… Om** – оператори тіла функції, серед яких є обов'язково оператор(и) повернення значення **return e;**. Нехай **Y={y1,…,yк}** та **Z={z1,…,zl}** – сукупності всіх глобальних і локальних змінних функції.

Фрейм даних, на станах якого визначена функція **func**, складається з її параметрів і глобальних змінних, тобто збігається із сукупністю **X={x1,…,xn, y1,…,yк}.** Сформувати конкретний початковий стан *X* -фрейму й обчислити на ньому значення функції дозволяє виклик функції – вираз вигляду

```
func (e1,…,en),
```

де **e1,…,en** – вирази типів, узгоджених із типами **T1,…,Tn**, які називаються *фактичними параметрами* функції [26]. Про узгодженість типів див. у підрозд. 3.2.4.

Нехай $a1,\ldots,an$ – значення виразів $e1,\ldots,en$, а $b1,\ldots,bk$ – поточні значення глобальних змінних. Тоді початковим станом X-фрейму для даного виклику функції буде стан {(**x1**,$a1$),…, (**xn**,an), (**y1**,$b1$),…, (**yk**,bk)}. Розпочинається виконання виклику функції з *ініціалізації* параметрів і локальних даних, під час якої параметри й локальні дані зв'язуються з певними об'єктами в динамічній пам'яті, після чого тіло функції стає *модифікованим* і готовим до виконання.

Результатом виклику функції є значення виразу **e** в операторі **return e;**, який здійснює вихід із функції в процесі виконання модифікованого тіла.

Якщо типом значення функції **T** є **void**, то її модифіковане тіло завершує роботу як звичайний складений оператор-блок і результатом його роботи буде оновлення значень глобальних змінних функції. Таким чином, у цьому випадку функція діє як $X-Y$-оператор. Якщо звернутись до функцій **f1** та **f2** із прикл. 3.11, то перша діє як **{j}-{i}**-оператор, а друга – як **{j,c}-{j}**-оператор.

Однак слід мати на увазі, що функція може діяти як $X-Y$-оператор і у випадку, коли повертає значення. Вона називається *функцією з побічним ефектом*[27], якщо після виходу з неї оновлюється значення принаймні однієї глобальної змінної.

Приклад 3.14. Функція **red_func** має побічний ефект:

```
int k=10; /*зовнішня глобальна змінна k*/
int red_func()
{…
  k=k+1; /* збільшення глобальної змінної k на 1*/
   …
  return 0;
}
```

■

Детальніше про виклики функцій йтиметься в підрозд. 3.3.2, про синтаксис і семантику функцій – у підрозд. 3.6.

Функція main(). Як уже зазначалось, серед функцій С-програми обов'язково має бути спеціальна функція зі стандартним іменем **main()**. Вона називається *точкою входження* у С-програму. Саме з неї

[26] У літературі вираз *виклик функції* іноді називають покажчиком функції. Ми не будемо застосовувати цю практику.

[27] В Алгол-60 такі функції називалися *червоними*.

розпочинається й нею закінчується виконання програм. Найменша С-програма не є винятком – вона також містить функцію **main**:

```
/*Найменша С-програма*/
int main()
{}
```

Дія програми зводиться до виконання порожнього тіла (оператора) функції.

Прототип функції **main** без параметрів має вигляд

```
int main([void]);
```

Функція **main()** повертає ціле число процесу, який ініціював її виклик. Зазвичай таким процесом є операційна система. Якщо функція **main()** не повертає значення явно (тобто оператором **return**), то процес отримує формально невизначене значення. Більшість компіляторів С у цьому випадку передбачають повернення 0.

Функція **main** може мати також параметри (див. підрозд. 3.6.5).

***Література для СР:** стандарти С89 – [55]; С99 – [133, 136]; структура С-програм – [55, 133, 136].

Контрольні запитання та вправи

1. Які символи називають вхідними та як вони класифікуються?
2. Що таке широкі символи?
3. Дати визначення основних класів лексем мови С.
4. Провести лексичний аналіз рядків:
   а) **"string ""FFF""" \r**
   б) **"\"F\xFF\?""" \n**
   в) **"word 07ff\125 \?"\n**
   г) **abs \x25\n\XFF**
   д) **\t\t C 1982 Denis Ritchi \n**
   е) **x+=a+b;printf("\vlength=%d\n", sqr(x));\n**
   є) **_int\t\t\tx,y;**
   ж) **_int_main(void){}**
   з) **x==(a+++b*2)**
   и) **q=a->b->p--;**
5. Провести лексичний аналіз опису функцій: а) **cube_sum, sum_cube** та **main** (див. прикл. 3.5); б) **Func1** та **Func2** (див. прикл. 3.13).
6. Який рядок поверне препроцесор після обробки такого фрагмента С-програми:

```
/**/***/*"/****//*,
///**/*/***///*//**/?
```

7. Яким буде подання в пам'яті літералів і якою буде довжина цих подань:
   `"\t\077"`;
   `"abc'1234'/*sos!*/\v"`;
   `" \x2514 3"`;
   `"\\\"\nLF"`?
8. Яка структура С-програм?
9. Яка роль функції **main** у С-програмі?
10. Що таке препроцесор?
11. Що таке підключення файлів?
12. Що таке препроцесорні константи?
13. Яка роль заголовних файлів?
14. Що таке роздільна компіляція?
15. У чому полягає різниця між деклараціями й описом даних?
16. Які є класи пам'яті змінних і функцій?
17. Що таке час існування даного?
18. Що таке визначальний опис даного?
19. Чим відрізняється прототип від опису функції?
20. Що таке виклик функції?
21. Навести склад *X*-фрейму даних, що обробляє модифіковане тіло функції.
22. Що таке модифіковане тіло функції?
23. Що таке ініціалізація даних?
24. Що таке функція з побічним ефектом? Навести свої приклади таких функцій.
25. Що таке локальна змінна?
26. Що таке глобальна змінна?
27. Що таке формальні параметри? Яка область їхньої дії?
28. Яке значення має змінна **i** у виділених точках програми:

```
int i=0;
main()
{ auto int i=1; /* 1 */
{int i=2;   /* 2 */
{ i+=1; /* 3 */
}
/* 4 */
}
/* 5 */
}  ?
```

29. Чи може дане у С-програмі бути одночасно і глобальним, і локальним?
30. Що виведуть такі фрагменти програм:

```
а) int f(int x, int y)
    { static int count=0;
      count++;
      printf("%d,",count);
      return (x+y)/2;
    }
    main()
  { printf("%d\n",f(5,7));
  printf("%d\n",f(25,13));
  printf("%d\n",f(1024,-40));
   } ,
```

```
б) int f(int x, int y)
    {static int count=0;
      count++;
      printf("%d,",count);
      return
(x>0?f(x/2,y/2):1);
    }
   main()
    {  printf("%d\n",f(5,7));
    } ?
```

3.2. Вирази

- Структура виразів
- Базові типи
- Стандартні типи
- Порядкові типи
- Сумісність, зведення й узгодженість типів

Ключові слова: *вираз, структура виразів, значення виразу, ліва й права асоціативність, l-вирази, r-вирази та f-вирази, вирази-присвоювання, таблиця операцій, пріоритет операцій, унарні й бінарні арифметичні операції, унарні й бінарні логічні операції, базові й похідні типи мови С, стандартні змінні, арифметичні, символьні, булеві й порядкові змінні, умовні й послідовні вирази, тип void, сумісність і узгодженість типів, порядок на числових типах за діапазоном їхніх значень, зведення цілих і дійсних типів, зведення типів у операціях присвоювання, унарні й бінарні зведення, зведення фактичних параметрів функцій, операція примусового зведення типів.*

Основою будь-якої мови є терми – конструкції, що подають реальний чи абстрактний об'єкт. У мовах програмування терми називаються *виразами*. Вони є певним спеціалізованим варіантом Ω-термів. Як випливає з принципу типізації, мову програмування можна подати у вигляді вежі типів зі структурою, яка описується об'єднаною Ω-системою її типів даних. Носій системи є універсумом усіх даних вежі типів, а сигнатуру Ω утворюють символи операцій та інтерфейсних і стандартних функцій, визначених на типах.

Виразами мови С називаються Ω-терми її системи типів.

Щоб з'ясувати, що таке вираз мови С (і будь-якої іншої), необхідно й достатньо описати всю її вежу типів разом із сигнатурою. Усі типи вежі поділяються на *базові* й *похідні* (рис. 3.1). Базові містяться безпосередньо в машинних командах. Для процесора ці дані є цілісними (неділимими) і твірними усієї системи даних. З іншого боку, базові типи є наближеними комп'ютерними моделями цілої, дійсної й комплексних арифметик. Це визначає їхню надзвичайно важливу роль у запитах вхідних систем.

У даному підрозділі ми розглянемо базові й стандартні типи (окрім типу покажчиків і комплексних чисел). Комплексні типи (а їх шість різновидів) передбачені лише в С99. Комплексним змінним відповідають об'єкти зі значеннями – двокомпонентними полями одного із трьох дійсних типів. Їхній опис і арифметику можна знайти в [133]. Решту похідних типів вивчатимемо в наступних підрозділах.

Спочатку конкретизуємо й розглянемо детальніше структуру виразів, а потім перейдемо до типів.

3.2.1. СТРУКТУРА ВИРАЗІВ

Вирази будуються з первинних виразів і викликів функцій за допомогою операцій. Усі операції разом з їхньою сигнатурою й пріоритетами наведено в табл. 3.5.

Як і у ПЧП, операції мають пріоритет і підлягають правилу асоціативності (для бінарних операцій). Якщо порядок виконання операцій не визначається дужками, то операнди додаються до знаків операцій із вищим пріоритетом. Найвищим пріоритетом є 1.

Якщо дві операції мають однаковий пріоритет, то операнди приєднуються до них залежно від напрямку асоціативності. Розрізняють ліву й праву асоціативність. Якщо вираз $a \bullet b \circ c$ означає $(a \bullet b) \circ c$, то говорять про *ліву* асоціативність, а якщо $a \bullet (b \circ c)$ – то про *праву*. Правило асоціативності можна узагальнити:

Усі бінарні С-операції, за винятком присвоювань, мають ліву асоціативність.

Для унарних операцій діє правило:

Постфіксні унарні С-операції мають вищий пріоритет, ніж префіксні унарні. Послідовності унарних операцій без дужок обчислюються справа наліво.

Усі вирази мови С розподіляються на три види: *l*-, *r*- та *f* -вирази.

l - та *r* -*вирази* – це конструкції, що за синтаксисом можуть бути відповідно лівим і правим операндами в операції присвоювання (див. підрозд. 2.1.2). *f* -*виразами* називаються конструкції, які використовуються як імена функцій (англ. – function designator). Вони розглядаються в підрозд. 3.6.1.

С-вирази $= l$ -**вирази** $+ r$ -**вирази** $+ f$ -**вирази**

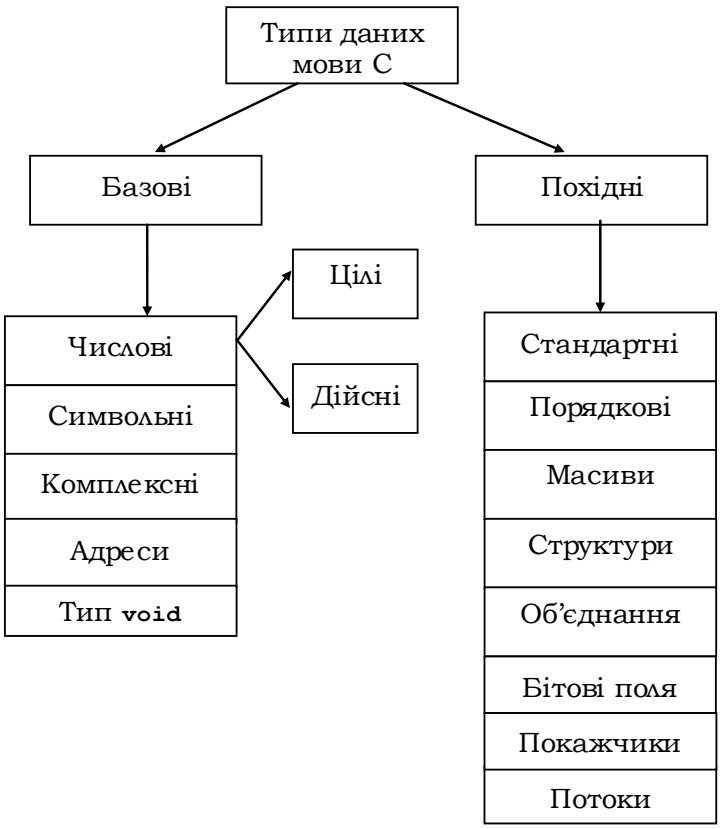


Рис. 3.1

Таблиця 3.5. **Операції мови С**

Пріоритет	Знак операції	Тип операції	Асоціативність
1	() виклик функції		Ліва
	[] індексація	Бінарна	⇒
	. взяття поля	Бінарна	
	-> розіменування поля	Бінарна	

Закінчення табл. 3.5.

Пріоритет	Знак операції	Тип операції	Асоціативність
2	~ побітове заперечення ! заперечення * розіменування & взяття адреси + + збільшення -- зменшення +, - (тип) зведення типу sizeof розмір типу	Унарні - - - - - - - -	Права ⇐
3	* множення / ділення % залишок	Мультиплікативні	Ліва ⇒
4	+, -	Адитивні	
5	<< зсув ліворуч >> зсув праворуч	Побітові	
6	<, >, <=, >=	Відношення	
7	== рівність != нерівність	Відношення	
8	& кон'юнкція	Побітова	
9	^ додавання за mod 2	Побітова	
10	| диз'юнкція	Побітова	
11	&& кон'юнкція	Логічна	
12	|| диз'юнкція	Логічна	
13	?: умовна		
14	=, *=, /=, %=, +=, -=, &=, |=, >>=, <<=, ^=	Присвоювання	Права ⇐
15	, послідовне обчислення		Ліва ⇒

Приклад 3.15. Пріоритет і асоціативність операцій.

Вираз	Еквівалентний вираз	Коментар
`a+b/c`	`a+(b/c)`	Мультиплікативна
`a*=b=c`	`a*=(b=c)`	Права асоціативність
`a-77+b`	`(a-77)+b`	Ліва асоціативність
`sizeof(double)*4`	`(sizeof(double))*4`	Пріоритет **sizeof** вищий
`*p->q`	`*(p->q)`	Пріоритет **->** вищий
`*x++`	`*(x++)`	Тут ++ – постфіксна й унарна
`**x**y`	`(*(*x))*(*y)`	Пріоритет розіменування вищий
`**x++++*y`	`(*(*((x++)++)))*y`	Тут ++ – постфіксна й має найвищий пріоритет, а унарна * вища, ніж бінарна *

■

l-вирази:

```
<l-вираз>::=<змінна-стандартного-типу>|<індексований-вираз>|
<вибір-компонентів>|<змінна-покажчик>|
<розіменування-покажчика>|(<l-вираз>)
```

Значеннями l-виразів є адреси об'єктів певного типу з рівнем доступу до запису або до запису й читання. Однак потрібно взяти до уваги, що в деяких контекстах ці об'єкти не допускають запису. Не є l-виразами змінні з кваліфікатором const, імена масивів, функцій тощо. У табл. 3.6 перелічені всі l-вирази мови С.

Таблиця 3.6. **l-вирази мови С**

l-вираз	**Значення**	**Коментар**
e	Адреса змінної e	e – стандартного типу
(e)	Значення виразу e	e – l-вираз
$e.v$	Адреса поля v	e – структура
$e \to v$	Адреса поля v	e – покажчик на структуру
$^{*}e$	Значення розіменованої змінної	e – покажчик
$e[i]$	Значення виразу $^{*}(e+i)$	e – покажчик

До операцій, що вимагають операндів – l-виразів, належать: взяття адреси &, збільшення $++$, зменшення – і присвоювання.

Змінні стандартного типу розглядаються в підрозд. 3.2.3, індексовані вирази й вибір компонентів – у підрозд. 3.4, 3.7, присвячених масивам і структурам, змінні-покажчики й розіменування покажчиків – у підрозд. 3.5.

***r*-вирази** – найчисленніший за кількістю операцій і найскладніший вид виразів. Нагадаємо (див. підрозд. 2.1.2), що r-вирази належать до X-арних U-функцій, які виробляють значення змінних. Оскільки всі змінні типізовані, то й значення кожного r-виразу типізовані, тобто належать одному з типів. Цей тип називається *типом r-виразу* (див. підрозд. 3.2.4).

```
<r-вирази>::=<первинний-вираз>|<вираз-присвоювання>|
 <постфіксний-вираз>|<унарний-вираз>
 |<бінарний-вираз>|<логічний-вираз>|<умовний-вираз>|
 |<послідовний-вираз>|<константний-вираз>
```

Розглянемо деякі види r-виразів. Унарні, бінарні й логічні вирази описуються в підрозд. 3.2.3.

З *первинних* виразів за допомогою операцій будуються всі решта:

```
<первинний-вираз>::=<l-вираз>|<константа>|(<вираз>)
```

Структуру *l*-виразів розглянуто вище. *l*-вирази як первинні відіграють роль функцій читання, тому семантика їх залежить від контексту. Діє правило: якщо наступною за *l*-виразом праворуч є операція присвоювання, то *l*-вираз подає себе (тобто адресу), а якщо ні – то функцію читання. З операційного погляду функція читання повертає значення об'єкта, адресу якого задає *l*-вираз.

Приклад 3.16. Нехай `int i` – ціла змінна. Тоді в операції присвоювання `i=i+2` ліворуч ім'я `i` подає *l*-вираз, а праворуч – функцію читання. Якщо значенням змінної `i` в полі даних α є 5, то обчислення цього виразу на полі α приведе до значення 7 і заміни в ньому значення змінної `i` на 7 ■

Вирази-константи – це лексеми-константи зі значеннями відповідного типу. Літерали при цьому мають тип символьного покажчика.

Тип виразу в дужках збігається з типом внутрішнього виразу. За допомогою дужок групують внутрішні підвирази для зміни стандартного порядку операцій або поліпшення читабельності виразів.

Приклад 3.17. Константи й вирази в дужках.

Нехай `double a=1.0, b=-2.5;` – опис дійсних змінних з ініціалізацією.

Вираз	Тип	Значення
`33`	`unsigned int`	33
`0777`	`unsigned int`	511_{10}
`'a'`	`unsigned char`	97
`0L`	`long`	0
`3.1428`	`double`	3.1428
`"3.1428"`	`*char`	адреса символу
`(a+b/2)`	`double`	-0.25
`((a+b)/2)`	`double`	-0.75

■

Вирази *присвоювання* оновлюють значення змінних:

```
<присвоювання>::=<l-вираз> <операція-присвоювання> <r-вираз>
<операція-присвоювання>::= =|*=|/=| %= |+=|-=|&=|
 =|>>=|<<=|^=.
```

Вираз простого присвоювання $e1 = e2$ – це підстановки в операцію присвоювання на місце першого й другого операндів *l*- та *r*-виразів $e1$ та $e2$, типи яких збігаються або узгоджені. Узгодженими є всі арифметичні типи. Узгодженість решти типів розглядається у відповідних під-

розділах. Семантику операції присвоювання описано в підрозд. 2.1.2, а семантика виразу-присвоювання – це семантика складеної функції:

1) спочатку обчислюються значення виразів $e1$ та $e2$;

2) за несумісності типів $e1$ та $e2$ значення $e2$ зводиться до типу $e1$;

3) отримані значення підставляються в операцію присвоювання на місце відповідних аргументів.

Про сумісність і зведення типів йдеться в підрозд. 3.2.4. Семантика складеного присвоювання ‘O=', де O – арифметична операція, зводиться до випадку простого присвоювання за формулою $e1 = e1Oe2$.

Постфіксні вирази:

```
<постфіксний-вираз>::=<вираз-збільшення>|<вираз-зменшення>
<вираз-збільшення>::=<l-вираз>++
<вираз-зменшення>::=<l-вираз>--
```

Типи й значення виразів $e++$ та $e--$ збігаються й дорівнюють значенню виразу e.

Суть цих операцій у побічному ефекті – вони додають (віднімають) 1 до (від) значення e виразу після операції. У випадку покажчиків це означає їхнє пересування до наступного (попереднього) об'єкта в ОП даного типу.

Виклики функцій розглядаються в підрозд. 3.3.2, 3.6.2.

Умовні вирази є варіантом умовних термів і мають вигляд $e?e1:e2$, де e – цілий вираз. Значення виразу задається умовним термом $(e \neq 0 \to e1 \mid e2)$.

Послідовні вирази мають вигляд $e1,e2$. Семантика виразу полягає в послідовному обчисленні зліва направо виразів $e1,e2$. Значення $e1$ відкидається, а значення $e2$ є значенням усього виразу. Операція асоціативна, тому можна послідовно об'єднувати в один єдиний довільну кількість виразів.

Константні вирази – це вирази, які в процесі обробки компілятором замінюються на значення-константи. До них належать препроцесорні константні вирази в директивах #*if* та #el*if*, цілі константні вирази, що використовуються при описі границь масивів, значень зліченних типів тощо та ініціалізаційні константні вирази в операторах опису змінних.

3.2.2. БАЗОВІ ТИПИ

Мова С містить кілька варіантів кожного з базових типів, але операціями вони не відрізняються. Базові *цілі типи* зображені в мові С цілими константами й арифметичними операціями. Останні наведені в табл. 3.5 і розглядаються в підрозд. 3.2.3. Визначено кілька цілих

типів різних розмірів. Константи цілого типу можуть записуватись у десятковій, вісімковій і шістнадцятковій системах із суфіксами **u (U)** для беззнакових цілих, **l(L), ll(LL)** – для довгих цілих:

```
<ціла-константа>::=<десяткова-константа> <суфікс-цілого>|
 |<вісімкова-константа> <суфікс-цілого>|
 |<шістнадцяткова-константа> <суфікс-цілого>
<десяткова-константа>::=<ненулева-цифра>{<цифра10>}
<вісімкова-константа>::=0{<цифра8>}
<шістнадцяткова-константа>::=(0x|0X)<цифра16>{<цифра16>}
<суфікс-цілого>::=<суфікс-беззнакового>[<суфікс-довгого>] |
 |<суфікс-довгого>[<суфікс-беззнакового>]
<суфікс-беззнакового>::=u | U
<суфікс-довгого>::=l | L| ll| LL
```

Для визначення основи системи числення цілої константи необхідно послідовно виконати п. 1-3 такого правила:

1) *Якщо константа має один із префіксів* **0x, 0X**, *то вона шістнадцяткова.*
2) *Якщо константа має префікс* **0**, *то вона вісімкова.*
3) *Це десяткова константа*?

Зазначимо, що для отримання від'ємного цілого необхідно застосувати до нього операцію унарний мінус. Базовим цілим відповідають об'єкти ОП, в яких зберігаються на машинному рівні цілі числа даного діапазону. Зазвичай вони зберігаються у форматі двійкових обернених кодів (див. підрозд. 2.3.3). Фактичний діапазон типу цілих констант може залежати від їхнього розміру, системи числення, суфіксів і внутрішнього подання, вибраного компілятором (табл. 3.7).

Приклад 3.18. Цілі константи: **10U** – десяткове беззнакове ціле; **25uL** – десяткове беззнакове довге; **1LL** – десяткове довге-довге; **0L** – вісімковий довгий нуль 0_8; **037** – вісімкове; **-0777** – від'ємне вісімкове -777_8; **-0x25** – від'ємне шістнадцяткове -25_{16} ■

У С99 *дійсні константи* записуються або за допомогою мантиси з точкою (не обов'язково нормалізованою), або цілого з порядком, або мантиси з порядком:

```
<дійсна-константа>::=<десяткова-дійсна-константа> |
<шістнадцяткова-дійсна-константа>
<десяткова-дійсна-константа>::=
```

```
<десяткова-константа-з-0> <експонента> [<суфікс-дійсного>] |
 <дійсне-з-фіксованою-точкою>[<експонента>][<суфікс-дійсного>]
<десяткова-константа-з-0>::=<десяткова-константа> | 0
<дійсне-з-фіксованою-точкою >::=[<десяткова-константа-з-0>]. \
[<десяткова-константа-з-0>]
<експонента>::=(e|E)[<знак>]<десяткова-константа>
<знак>::=+|-
<суфікс-дійсного>::=f|F|l|L
```

У числах із фіксованою точкою не можуть бути одночасно відсутні й ціла, і дробова частини. Суфікси виділяють константи типів **float** та **long double**. Відсутність суфікса означає, що тип константи – **double**.

Для отримання від'ємної дійсної константи необхідно, як і у випадку цілих, застосувати операцію унарний мінус.

Три дійсні типи відрізняються тільки точністю й розміром порядку. Якщо точність чисел не принципова й необхідно економити пам'ять (напр., у таблицях), то використовують тип **float**.

Приклад 3.19. Десяткові дійсні константи `0.f(=0.0)`, `1.0f(=1.0)`, `.25 f(=0.25)` належать до типу **float**, константа `1e-2L (=0.01)` – до типу **long double**, константа `1.0e+25 (=10`^{25}`.0)` – до типу **double** ■

Якщо компілятор не може подати дану дійсну константу точно, то він вибирає найближче до неї значення, яке може подати. Якщо дане дійсне більше найбільшого чи менше найменшого значення в дійсному типі, то результат буде непередбачуваний.

Як і у випадку базових цілих, базовим дійсним відповідають об'єкти ОП, в яких зберігаються на машинному рівні дійсні числа даного діапазону (див. підрозд. 2.3.3). Фактичний їхній діапазон, щільність, кількість цифр у мантисі й порядку залежать від компілятора.

У С99 передбачено заголовний файл **float.h**, в якому розміщуються граничні значення для дійсних констант у даному компіляторі. У табл. 3.7 наведено деякі з них.

На відміну від стандарту С89, стандарт С99 дозволяє використовувати також шістнадцяткові дійсні константи. Формат схожий на формат десяткових дійсних. Таким константам передує шістнадцятковий префікс **0x, 0X**. Експонентою є двійкова експонента вигляду $p \pm n_{16}$ зі значенням $2^{\bar{n}}$, де $\bar{n} = \bar{n}_{16}$.

Приклад 3.20. Шістнадцяткові дійсні константи:

`0X1.0(=1.0), 0x19.(=25.0), 0XAp-2(10*2`$^{-2}$`=2.5)` ■

Символьний базовий тип складають звичайні й широкі символьні константи. Коди широких символів складають тип **wchar_t**, який визначено у заголовному файлі **stddef.h**. Прототипи функцій для виділення різних груп широких символів та їхні відображення розташовані в заголовному файлі **wctype.h**, прототипи решти символьних функцій і функцій для обробки рядків широких символів – у файлі **wchar.h**.

Тип **void** має порожню множину значень. Він використовується: 1) у поданні типу функції, якщо вона не повертає значень; 2) для формування безтипового покажчика **void***; 3) для опису порожнього списку параметрів функції. Наприклад, головна функція **void main (void){}** не повертає значення й не має параметрів.

3.2.3. СТАНДАРТНІ ТИПИ

До стандартних типів даних належать сукупності значень числових змінних різних розмірів. Такі змінні мають рівень доступу 3, тобто повний. Стандартні типи розподіляються на цілі, дійсні, символьні, комплексні й булевий. Вважається, що всі вони містять беззмістовне значення **#**, яке не має своєї ідеограми в мові С і якого набуває вираз у позаштатних (аварійних) ситуаціях [28].

Цілі типи. Діапазон конкретного цілого типу визначається специфікатором типу й реалізацією і для різних процесорів і компіляторів може бути різним. При моделюванні цілої арифметики цілими машинними типами слід ураховувати обмеженість (скінченність) моделей. Це зумовлює такі їхні особливості, як переповнення при виконанні операцій і беззмістовність у моделі тих цілих чисел, що розташовані за межами машинного типу:

```
<специфікатор-цілого-типу>>::=[<модифікатор-знака>][<розмір>]\
 \[<тип>]|_Bool
<тип>::=char| int
<розмір>::=short |long |long long
<модифікатор-знака>::=signed | unsigned
```

Основою всіх цілих типів є тип **int.** Розмір об'єкта типу **int** зазвичай збігається з розміром машинного слова в конкретному середовищі програмування. У більшості випадків у 16-розрядному середовищі (DOS чи Windows 4.1) **int** займає 16 бітів, а у 32-розрядному (Windows 95/98/NT/2000) – 32 біти. Зазначимо, що стандарт С обу-

[28] Тут # є метасимволом.

мовлює тільки мінімальний діапазон значень кожного типу даних, але не розмір у байтах.

Розмір типу **char** – 1 байт. Даний тип використовується для подання кодів символів набору ASCII.

Обидва типи можуть бути модифіковані за знаком і розміром. За знаком вони розподіляються на знакові (з модифікатором **signed**) і беззнакові (з модифікатором **unsigned**). У беззнаковому варіанті старший знаковий біт використовується як числовий. Тому діапазон беззнакового типу вдвічі ширший, ніж такий самий знаковий. Якщо модифікатор знака відсутній, то змінна – знакова. За розміром цілі поділяються на короткі (**short**), довгі (**long**) і довгі-довгі (**long long**). Розміри типів залежать від компілятора, їх можна знайти за допомогою операції **sizeof**. Наприклад, **sizeof(long)=4**, **sizeof(short)=2** тощо. Дана операція дозволяє також отримати розмір об'єкта, заданого виразом **sizeof(15l)=4**, **sizeof('b')=1** тощо. Стандартні розміри типів наведені в табл. 3.7.

У заголовному файлі **<limits.h>** визначаються розміри граничних констант для цілочислових типів компілятора. У табл. 3.8 наведені деякі з них.

***Таблиця 3.7.* Розмір цілих типів**

Тип	Типовий розмір у бітах	Мінімальний діапазон значень
char	8	Від -127 до 127
unsigned char	8	Від 0 до 255
signed char	8	Від -127 до 127
int	16 чи 32	Від -32767 до 32767
unsigned int	16 чи 32	Від 0 до 65535
signed int	16 чи 32	Те саме, що **int**
short int	16	Від -32767 до 32767
unsigned short int	16	Від 0 до 65535
signed short int	16	Те саме, що **short int**
long int	32	Від -2 147 483 647 до 2 147 483 647
long long int	64	Від $-(2^{63}-1)$ до $(2^{63}-1)$, доданий стандартом С99
signed long int	32	Те саме, що **long int**
unsigned long int	32	Від 0 до 4 294 967 295
unsigned long long int	64	Від 0 до $(2^{64}-1)$, доданий у С99
float	32	Від 1E-37 до 1E+37, з точністю не менше 6 значущих десяткових цифр
double	64	Від 1E-37 до 1E+37, з точністю не менше 10 значущих десяткових цифр

Таблиця 3.8. **Граничні цілі константи з файла** `limits.h`

Константа	Значення (у бітах)	Визначення
CHAR_BIT	8	Розмір поля для **char**
UCHAR_MAX	255	Максимальне значення в **char**
INT_MAX	+32767	Максимальне значення в **int**
INT_MIN	-32768	Мінімальне значення в **int**
LONG_MAX	+2147483647	Максимальне значення в **long**
UINT_MAX	65535	Максимальне значення в **unsigned int**

У С99 передбачені додаткові розширені цілі типи (**extended integer type**), доступ до яких здійснюється через заголовні файли **stdint.h** та **inttypes.h.** На ці типи поширюється вся ціла арифметика й правила перетворення типів.

У С99 введено також цілий булевий тип **_Bool** зі значеннями 0 та 1. При перетвореннях усіх цілих типів до **_Bool** ненульові значення перетворюються до 1, а нульові – до 0.

Операції на цілих типах – це звуження на них арифметичних операцій над цілими числами. Якщо ж результат певної операції виходить за межі типу, то виникає ситуація *переповнення.* Переповнення порушує основні арифметичні закони (асоціативність, дистрибутивність). В обчисленнях, коли переповнення відсутнє, машинна арифметика є точною моделлю звичайної цілої арифметики.

Розпочнемо з *унарних цілих префіксних операцій*:

```
<унарна-ціла-префіксна-операція>::=<унарний-мінус>|
                  |<унарний-плюс>|<заперечення>|
                  |<побітове-заперечення>
|<префіксне-збільшення>|
|<префіксне-зменшення>
```

Операція *унарний мінус* (–) змінює знак виразу на протилежний. Це скорочений варіант віднімання: $-e = 0 - e$. Операція *унарний плюс* (+) тотожна: $+e = 0 + e$. Операція *заперечення* ! для довільного цілого виразу e дає результат $!e = (e = 0 \to 1 \mid 0)$. Операція *побітового заперечення* ~ інвертує біти в цілому об'єкті. При цьому знаковий біт не зачіпається. Операція *префіксного збільшення* ++ (*зменшення* --) додає (віднімає) 1 до (від) поточного значення цілого l-виразу. Ці операції, як і постфіксні їхні варіанти, мають побічний ефект – нові значення записуються у l-вираз.

Приклад 3.21. Нехай `int n=5; unsigned char c='0';`

Вираз	Значення	Коментар
`!n`	`0`	
`!-5`	`!(-5)=0`	
`~c`	`~0011 0000=1100 1111=207`	`'0'=48`
`++n`	`6`	
`--n`	*4*	

■

Приклад 3.22. Надрукувати значення масиву `int s[100]` зі 100 цілих чисел можна таким чином:

```
for(int i=0; i<100; i++)  printf("\ns[%d]=%d",i,s[i]); ■
```

Бінарні цілі операції – це операції з цілими операндами, що відповідають рядкам 3-15 табл. 3.5.

Вираз **x % y** дає *залишок* від ділення **x** на **y**. Якщо **x** ділиться на **y** націло, то результат – 0. Наприклад, рік є високосним, якщо він ділиться на 4, але не ділиться на 100, або якщо ділиться на 400. Це можна записати так:

```
if(((year % 4==0) && (year % 100 !=0)) || (year % 400==0))
          printf("Рік %d високосний", year);
else      printf("Рік %d невисокосний", year);
```

Якщо обидва операнди % невід'ємні, то й результат невід'ємний і менший за дільник. Яким буде знак результату операції % із від'ємними операндами, залежить від процесора, при цьому гарантується лише, що абсолютне значення залишку менше, ніж абсолютне значення дільника.

Операції зсуву – **<<** (ліворуч) та **>>** (праворуч). Перший операнд визначає число, що зсувається, другий – кількість бітів, на яку відбувається зсув. Тип результату – це тип першого операнда. При зсуві ліворуч зайві старші біти відкидаються, а нові молодші заповнюються 0. При зсуві праворуч молодші біти відкидаються, значення звільнених старших бітів залежить від типу першого операнда й реалізації. Однак якщо даний тип беззнаковий або значення додатне, то ці біти будуть 0. Якщо значення другого операнда від'ємне або більше ніж довжина в бітах першого операнда, то результат буде беззмістовним.

Зазначимо, що зсув ліворуч відповідає множенню першого операнда на степінь числа 2, що дорівнює другому операнду, а зсув праворуч – діленню першого операнда на 2 у степені, що дорівнює другому операнду.

Побітові операції: **&** (*кон'юнкція*), **^** (*додавання за модулем* 2) та **|** (*диз'юнкція*). Усі побітові операції комутативні й асоціативні.

Під час виконання операції **&** порівнюється кожен біт першого операнда з відповідним бітом другого. Якщо обидва біти – одиниці, то відповідний біт результату буде 1, інакше – 0.

Під час виконання операції | порівнюється кожен біт першого операнда з відповідним бітом другого. Якщо один із бітів – 1, то відповідний біт результату буде 1, інакше – 0.

Під час виконання операції ^ порівнюється кожен біт першого операнда з відповідними бітами другого. Якщо один із порівнюваних бітів дорівнює 0, а другий – 1, то відповідний біт результату буде 1, інакше – 0.

Приклад 3.23.

Вираз	Значення

`int i=3;`
`i<<2`
`i>>2`
`i& ~07`
	
	00001100_2
	00000000_2
	`00000011 & ~00000111==00000011 & 111111000 ==00000000`

`int i=0x1234, j, k;`
`k=i<<4;`
`j=i<<8;`
`i=j>>8;`
	
	`k=0x2340`
	`j=0x3400`
	`i=0x0034`

`int i=0x45FF,`
` j=0x00FF,`
` r;`
 	
`r=i^j;`
`r=i|j;`
`r=i&j;`
	`i=0100 0101 1111 1111`
	`j=0000 0000 1111 1111`
	
	`r=0x4500=0100 0101 0000 0000`
	`r=0x45FF=0100 0101 0000 0000`
	`r=0x00FF=0000 0000 1111 1111`

■

Логічні цілі операції:

<логічна-ціла-операція>::=<відношення> | <булева-операція>
<відношення>::=<|<=|>|>=|==| !=
<булева-операція>::=! |&&| ||

Результат відношення $e1 \otimes e2$ буде 1, якщо відношення виконується ($e1$ менше, не більше, більше, не менше, дорівнює, не дорівнює $e2$), інакше – 0.

Булеві операції **&&** та **||** трактуються як тризначні й набувають одного із трьох значень: 1, 0 або **#** (див. табл. 1.1). В обох операціях першим обчислюється лівий операнд. Якщо він істинний (у випадку диз'юнкції) чи хибний (у випадку кон'юнкції), то другий операнд не обчислюється. Операцію заперечення **!** розглянуто вище.

Дійсні змінні. Діапазон цілих змінних залежить від специфікаторів їхніх типів і реалізації та для різних процесорів і компіляторів може бути різним. При моделюванні дійсної арифметики дійсними ти-

пами теж слід ураховувати скінченність моделей. Це зумовлює такі їхні особливості, як нерівномірну щільність, переповнення при виконанні операцій і беззмістовність у моделі дійсних поза межами діапазону типу (див. підрозд. 2.3.3):

<специфікатор-дійсного-типу>::=float |double|long double

Стандарт С99 вимагає, щоб характеристики дійсних типів були прописані в заголовному файлі **float.h** (табл. 3.9).

До операндів дійсного типу можна застосовувати більшість арифметичних і логічних операцій. Виняток становлять побітові операції та операція %.

Стандартні функції для роботи з дійсними типами містяться в бібліотеці **<math.h>** (табл. 3.10). Усі функції повертають значення типу **double**. Кути в тригонометричних функціях задаються в радіанах.

***Таблиця 3.9.* Деякі граничні дійсні константи з файла `float.h`**

Константа	Значення	Визначення
FLT_DIG	**6**	Кількість десяткових цифр у мантисі для типу **float**
FLT_EPSILON	**1E-5**	Машинний нуль у типі **float**
FLT_MAX	**1E +37**	Максимальне значення в типі **float**
FLT_MIN	**1E-37**	Мінімальне значення в типі **float**
FLT_MIN_EXP		Найменше n таке, що 10^n належить типу **float**
DBL_DIG	**10**	Кількість десяткових цифр у мантисі в типі **double**
DBL_EPSILON	**1E-9**	Машинний нуль $\min\{x : 1.0 + x \neq 1.0\}$ у типі **double**
DBL_MAX	**1E +37**	Максимальне значення в типі **double**
DBL_MIN	**1E-37**	Мінімальне значення в типі **double**
DBL_MIN_EXP		Найменше n таке, що 10^n належить типу **double**

***Таблиця 3.10.* Деякі з функцій бібліотеки `<math.h>`.**

```
double x,y; int n;
```

Функція	Визначення
sin(x)	Синус x
cos(x)	Косинус x
tan(x)	Тангенс x
asin(x)	Арксинус x, $x \in [-1,1]$
atan(x)	Арктангенс x
exp(x)	Експонента e^x
log(x)	$\ln(x), x > 0$
log10(x)	$\log_{10}(x), x > 0$

Закінчення табл. 3.10.

Функція	Визначення
`pow(x,y)`	Степінь x^y
`sqrt(x)`	$\sqrt{x}$, $x \geq 0$
`ceil(x)`	Найближче до x ціле в типі **double**, не менше за x
`floor(x)`	Найближче до x ціле в типі **double**, не більше за x
`fabs(x)`	$\lvert x \rvert$
`frexp(x,int *exp)`	Повертає нормалізовану мантису m у діапазоні $[1/2,1)$ і двійковий порядок 2^p, який зберігається за адресою **exp**. Таким чином, $x = m * 2^p$. Для $x = 0$ обидві частини результату теж дорівнюють 0
`modf(x, double *ip)`	Виділяє цілу й дробову частини, що мають знак, однаковий із x. Ціла частина зберігається за адресою ip, дробова – повертається як результат

Символьні змінні:

```
<специфікатор-символьного-типу>::=char|signed char|unsigned char
```

Тип **char** – цілий. Незважаючи на те, що зазвичай він використовується для збереження символів, його доцільно використовувати і для збереження маленьких (до 255) цілих значень. Семантично тип **char** належить до цілочислових типів розміром 1 байт. На ньому визначена вся цілочислова арифметика. Символьні дані можуть зустрічатися в арифметичних виразах. Значеннями їх є числові коди символів. Наприклад, `'0'+2=48+2=50` тощо. Тип **char** залежно від реалізації може бути беззнаковим або знаковим. Ураховуючи, що функції консольного введення-виведення по досягненні кінця файла повертають від'ємне значення, задане макросом **EOF**, при читанні символів безпечніше використовувати знакову змінну типу **int**, а не **char**, яка може виявитися беззнаковою.

Булеві змінні. Беззнаковий цілий тип **_Bool** з'явився у С99 і складається із двох значень: 0 (хиба), 1 (істина). При перетвореннях скалярних виразів[29] до булевого типу ненульові значення перетворюються на 1, а решта на – 0. У заголовному файлі **stdbool.h** визначено макрос **bool** як синонім для типу **_Bool**, а також **false** та **true** – для 0 та 1. На булевому типі визначено всі логічні операції.

[29] До скалярних належать усі неструктуровані типи – числові, символьні й покажчики.

3.2.4. ПОРЯДКОВІ ТИПИ

У програмах можна створювати свої типи, що складаються із цілих констант із вибраними (мнемонічними) найменуваннями. Кожен такий тип має свій конструктор із ключовим словом **enum** і списком поіменованих констант:

```
<порядковий-тип>::=enum [<ім'я-тегу>]\
\{<ім'я>[=<константний-вираз>],\
{<ім'я>[=<константний-вираз>]}\}
```

Тегом називається конструкція у фігурних дужках зі списком поіменованих констант. Тег може мати своє ім'я, що записується відразу після ключового слова **enum**. Усі константи в описі мають цілий тип.

Приклад 3.24.

```
enum {saturday=6, sunday=7} freeday;
enum weekend {saturday=6, sunday=7} freeday;
enum weekend freeday;
```

В останньому рядку ім'я тегу **weekend** дозволяє використовувати тег типу повторно без перерахування його значень ■

Порядковим константам значення призначаються:

1) за допомогою константного виразу в описі типу:

```
enum weekend {saturday=6, sunday=saturday+1};
```

2) якщо немає явного присвоювання значення, то перша константа отримує значення 0;

3) поточна константа в разі відсутності явного присвоювання отримує значення на 1 більше, ніж значення попередньої константи.

Приклад 3.25. В описі типу

```
enum wage {small, medium=50, pretty_med, large=90};
```

константи **small**, **medium**, **pretty_med** та **large** мають значення відповідно 0, 50, 51 та 90 ■

На порядкових типах діє ціла арифметика.

3.2.5. СУМІСНІСТЬ, ЗВЕДЕННЯ Й УЗГОДЖЕНІСТЬ ТИПІВ

Два типи даних є *сумісними*, якщо вони збігаються або є настільки близькими, щоб їх вважати однаковими в більшості контекстів. Сумісність типів значною мірою залежить від реалізації мови.

Стандартні числові типи сумісні тільки тоді, коли вони ідентичні, тобто коли їм відповідає одна сукупність об'єктів (за розміром і структурою полів, а також рівнем доступу). При описі типу можуть опускатися деякі специфікатори й навіть назви типів, але всі ці його форми є ідентичними й задають один тип. Наприклад, типи **short** та **short int** ідентичні, а отже, сумісні, а типи **unsigned**, **int** та **short** неідентичні й несумісні. Тип **const int** не сумісний із жодним варіантом типу **int** (різний рівень доступу).

Кожне визначення *порядкового типу* породжує новий цілий тип, який за С99 є сумісним із базовим цілим типом його констант. Жодні два різних порядкових типи, визначені у вхідному файлі, не є сумісними. Однак зліченні типи сумісні, якщо в одному з них визначається тег з іменем, а в другому використовується ім'я цього тегу. Наприклад, типи змінних **x** та **y** в операторах

```
enum s {p, q} x;
enum s y;
```

є сумісними.

В арифметичних виразах мови С дозволяється *змішувати* типи числових операндів, що є природним у застосуваннях. Наприклад, коректним є вираз $\mathrm{sqrt}(b*b-4*a*c)$, який обчислює дискримінант квадратного рівняння з дійсними коефіцієнтами a,b,c. Це ускладнює реалізацію виразів, оскільки ті самі ціла й дійсна арифметики суттєво відрізняються (як поданням об'єктів у ОП, так і реалізацією операцій). Щоб урегулювати за необхідності розбіжність типів операндів, на числових типах установлено лінійний порядок, який базується на діапазонах їхніх значень:

```
char <unsigned char <short <unsigned short <int <unsigned int
<long <unsigned long <long long <float <double <long double
```

Згідно із цим порядком кожний числовий вираз має свій тип, яким є старший із типів його операндів. Це не стосується виразів присвоювання, тип яких збігається з типом l-виразу в лівій частині.

У процесі обчислення значення виразу всі операнди автоматично зводяться до його типу. Зведення до типу може вимагатися й у таких ситуаціях:

- в операціях примусового зведення до типу;
- при ініціалізації параметрів функції (значення фактичного параметра зводиться до типу формального параметра);
- при виході з функції значення, що повертається, зводиться до типу її результату.

У процесі зведення операнда до іншого типу може змінюватися як розмір зв'язаного з ним об'єкта, так і внутрішня його структура.

Тип даних $T1$ називається *узгодженим* із типом $T2$, якщо визначене правило зведення констант типу $T1$ до типу $T2$.

Зведення цілих типів. Здійснюється, коли вираз має один із цілих типів. Якщо тип зведення дозволяє точно подати значення виразу, то останнє перетворюється до нього, інакше – до беззнакового типу **`unsigned int`**, **`unsigned long`** або **`unsigned long long`**. При перетворенні беззнакових цілих до знакових того самого розміру виникає ситуація переповнення, якщо старший біт беззнакового цілого є 1. Перетворення від'ємного цілого до беззнакового невизначене. При перетворенні беззнакових цілих до коротших беззнакових відкидаються зайві старші розряди.

Зведення цілих і дійсних типів. При перетворенні значень із дійсного типу на цілочисловий дробова частина відкидається; якщо отримане ціле міститься за межами даного цілого типу, то результат невизначений. Зокрема, дана ситуація має місце при перетворенні від'ємних дійсних до беззнакових цілих. Якщо ціле значення перетворюється до дійсного й міститься в допустимому діапазоні, але подається неточно, то результатом буде найближче до даного цілого дійсне.

Зведення дійсних типів. При перетворенні меншого типу до більшого значення не змінюється. Навпаки, якщо перетворюється значення від старшого типу до молодшого, яке міститься в діапазоні меншого типу, то результатом буде найближче до нього значення в меншому типі (округлення). Якщо результат виходить за межі меншого типу, то він невизначений.

Зведення типів у операціях присвоювання. При обчисленні значення виразу операнд $e2$ перетворюється до типу лівого операнда за вказаними вище правилами.

Унарні зведення. Призначені для автоматичної стандартизації типів операндів операцій і аргументів функцій. Наприклад, у C89 усі дійсні аргументи функцій бібліотеки **`<math.h>`** автоматично перетворюються до типу **`double`**. Унарні перетворення наведені в табл. 3.11.

Таблиця 3.11. **Унарні перетворення**

Тип операнда	С99: тип результату	Традиційна С: тип результату
`float`	Без перетворення	`double`
Масив типу `T`	Покажчик типу `T`	Так само
Функція зі значенням типу `T`	Покажчик на функцію зі значенням типу `T`	Так само
Цілий не нижчий ніж `int`	Без перетворення	Так само
Цілий нижчий ніж `int`	`Int`	Так само
Беззнаковий нижчий ніж `int`, але зі значенням у `int`	`int`	`unsigned int`
Беззнаковий нижчий ніж `int`, але зі значенням поза `int`	`unsigned int`	Так само

Бінарні зведення. У процесі бінарних перетворень над кожним з операндів спочатку здійснюються унарні перетворення, у результаті яких збільшуються розміри коротких значень, а імена масивів і функцій перетворюються на покажчики. Потім здійснюються необхідні зведення операндів до типу виразу.

Зведення фактичних параметрів функцій. Фактичні параметри зводяться до типу формальних параметрів згідно з прототипом функції. Жодні інші правила не діють. У С99 передбачено окремий набір стандартних функцій для кожного з типів `float`, `double` та `long double`.

Приклад 3.26. Зведення арифметичних типів:

```
1:   int i;
2:   char ch;
3:   float f;
4:   void func(void)
5:   {
6:    ch=i; /*int--->char*/
7:   i=f; /*float--->int*/
8:    f=ch; /*char--->float*/
9:    f=i; /*int--->float*/
10:  }
```

У шостому рядку цього прикладу старші двійкові розряди цілої змінної `i` відкидаються, а у `ch` заносяться молодші 8 бітів. Якщо значення `i` лежить в інтервалі від 0 до 255, то значення `ch` та `i` будуть однаковими, при цьому інформація не втрачається. Інакше у `ch` будуть занесені тільки молодші розряди змінної `i`. У сьомому рядку в `i` буде записана ціла частина числа `f`; у восьмому – відбудеться перетворення цілого числа, що зберігається у `ch`, на число в плаваючому форматі; у дев'ятому – те саме, тільки з 16-розрядним цілим ■

Примусове зведення типів. Здійснює операція (**T**)*e*, яка значення виразу *e* перетворює до типу **T**. Наприклад, унарна операція (**float**) як старша щодо ділення перетворює нульове значення цілого виразу **1/3** до значення (**float**) **1/3=0.333333.**

Приклад 3.27. У наведеній програмі ця ідея використана для збереження точності при виведенні результатів обчислення:

```
#include <stdio.h>
int main(void) /*друкування i*i/3 із дробовою частиною*/
{
 int i;

 for(i=1;i<20;++i)
 printf("%d/3=%f\n",i,(float)i/3);

 return 0;
}
```

Без операції примусового зведення (**float**) виконувалося б цілочислове ділення ■

***Література для СР:** вирази – [55, 102, 132, 133, 136]; особливості арифметичних операцій стандартних типів [19, 60]; перетворення типів – [55, 133].

Контрольні запитання та вправи

1. Що таке вираз?
2. Що таке ліва й права асоціативність?
3. Що таке *l*-вираз?
4. Що таке *r*-вираз?
5. Які бувають *r*-вирази?
6. Що таке первинний *r*-вираз?
7. Як обчислюються значення виразу-присвоювання?
8. Що таке базовий і похідний типи?
9. Який вигляд має ціла константа?
10. Який вигляд має дійсна константа?
11. Який вигляд має символьна константа?
12. Перерахувати й визначити унарні й бінарні арифметичні операції.
13. Перерахувати й визначити унарні й бінарні логічні операції.
14. Який вигляд має опис стандартної змінної?
15. Який вигляд має символьна константа?
16. Що таке модифікатор знака змінної цілого типу?
17. Перерахувати розміри змінних цілих типів.
18. Перерахувати типи дійсних змінних.

19. Що таке умовні й послідовні вирази?
20. Що виведе такий фрагмент програми:

```
1){int x =011, y = 9, z = 4; printf("%d\n",
  x|z<<y);}
2){int x, y, z; x = y = z = 1; ++x || ++y && ++z;
  printf("%d\t%d\t%d",x,y,z);}
3){int x =011, y = 9, z = 5;     printf("%d\n",
  x<=y ? x&~z:z&y);}
4){int x,y,z; x = y = z = 1; x++ && ++y ||
  ++z;printf("%d\t%d\t%d",x,y,z);}
5){int x = 8, y = 3, z = 1;    x <<= y;
  printf("%d\n", x); }
6){int x,y,z; x = y = z = -1;  x++ && ++y && ++z;
  printf("%d\t%d\t%d",x,y,z);}
7){int x =011, y = 9, z = 4; printf("%d\n", x | y
  ^y );}
8){int x, y, z; x = y = z = 1; ++x && ++y || ++z;
  printf("%d\t%d\t%d",x,y,z); }
9){int x = 8, y =3, z =1;printf("%d\n",z += x < y
  ? x ++ : y ++ );      printf("%d\n",z);}
10){int x =8, y =3, z =1; printf("%d\n", z >= y
  >= x ? 1 : 0 ); }
11){int  x  =8,  y  =3,  z  =1;     y  >>=  x;
  printf("%d\n",y); }
12){int x =05, y = 02, z = 01;   printf("%d\n", x
  ^ y & ~z ); }
```

21. Який вигляд має опис порядкової змінної?
22. Що таке тип void?
23. Визначити порядок на числових типах за діапазоном їхніх значень.
24. Як знайти розмір змінної?
25. Що таке сумісні й угоджені типи?
26. Чи сумісні (узгоджені) пари типів:
    int та **signed int;**
    char та **unsigned char;**
    float та **double;**
    short та **int** ?
27. Сформулювати правила зведення: а) двох цілих типів; б) двох дійсних типів; в) цілого й дійсного типів; г) типів у операціях присвоювання.
28. Що таке унарні й бінарні зведення?
29. Як зводяться фактичні параметри функцій?
30. Що таке примусове зведення типів?
31. Нехай a, b, c – імена цілих змінних. Написати арифметичний (не умовний) вираз, значенням якого є: а) більше з двох значень *a* та *b*; б) значення останньої цифри в десятковому по-

данні a (напр., при $a = 125$ – це 5); в) сума значень цифр двозначного a (при $a = 39$ – це 12); г) сума значень цифр тризначного a (при $a = 214$ – це 7).

32. Нехай a, b, c, d, e, a_1, b_1, c_1 та a_2, b_2, c_2 – імена цілих змінних із додатними значеннями. Написати умовний вираз, значенням якого є 1 тоді й тільки тоді, коли: а) a, b, c мають однакові значення; б) a, b, c задають сторони трикутника; в) a, b, c задають сторони прямокутного трикутника; г) a, b, c задають сторони гострокутного трикутника; д) a, b, c задають сторони рівнобедреного трикутника; е) a, b, c задають сторони різнобічного трикутника; є) a, b, c, d задають сторони паралелограма; ж) a_1, b_1, c_1 та a_2, b_2, c_2 задають сторони двох рівних трикутників; з) цеглину $a \times b \times c$ можна просунути в прямокутне вікно $d \times e$ так, щоб її грані були паралельні сторонам вікна.
33. Написати вираз, значенням якого є 1 тоді й тільки тоді, коли дві прямі, задані цілими коефіцієнтами рівнянь вигляду $ax + by + c = 0$: а) паралельні й не збігаються; б) паралельні (можливо, збігаються); в) збігаються; г) перетинаються; д) перпендикулярні.
34. Підлога в кімнаті складається з клітин і має розмір $n \times m$. На двох клітинах поставлено стовпи. Написати вираз, яким задається ознака того, що підлогу можна покрити дошками розміром 2×1.
35. Написати послідовність виразів-присвоювань, що здійснює обмін значеннями двох змінних за умови: а) можна використовувати третю змінну; б) третю змінну не використовувати, але змінні числові.
36. Дано дійсні змінні x, y, z. Написати вираз для обміну їх значеннями згідно з перестановкою: а) $\begin{pmatrix} xyz \\ zxy \end{pmatrix}$; б) $\begin{pmatrix} xyz \\ yzx \end{pmatrix}$; в) $\begin{pmatrix} xyz \\ yzx \end{pmatrix}$, не використовуючи четверту змінну.
37. Нехай x – числова змінна. Використовуючи лише операції множення й присвоювання, обчислити значення степеня з мінімальною кількістю множень. Наприклад, обчислення x^6 можна задати так: $x2 = x * x$, $x4 = x2 * x2$, $x4 * x2$.

а) x^{10}; д) a^5 та a^{19}; з) a^4 та a^{20}; й) a^{28}.
б) x^{12}; е) a^{15}; и) a^5 та a^{13};
в) x^{16}; є) a^{21}; і) a^2, a^5, a^{17};
г) x^{32}; ж) a^{64}; ї) a^4, a^{12}, a^{28};

38. Дано дійсне x. Написати вираз для обчислення значення багаточлена $2x^4 - 3x^3 + 4x^2 - 5x + 6$. Мінімізувати кількість операцій у виразі.
39. Дано дійсні числа x, y, z. Написати арифметичний (не умовний) вираз для отримання: а) $\max(x, y)$; б) $\min(x, y)$; в) $\max(x + y + z, xyz)$; г) $\max^2(x + y + z / 2, xyz) + 1$; д) $\text{sign}(x)$. Допустимою є також функція модуль $abs(x)$.
40. Функція **color(x, y)** визначає колір поля *xy* на шаховій дошці 8×8:

```
//0 - білий колір, 1 - чорний колір
int color(char x,int y)
{return odd(c(x))?odd(y)?1:0:odd(y)?0:1;}
```

Допоміжна функція **int c(char x))** перетворює символи **'a', 'b', 'c', 'd', 'e', 'f','g','h'** відповідно на числа 1, 2, 3, 4, 5, 6, 7, 8, а функція **int odd(int n)** перевіряє число **n** на непарність (1 – непарне, 0 – парне). Як правильно розташувати дужки в умовному виразі оператора **return**? Запрограмувати функції **c** та **odd.**

41. Дано шахову дошку 8×8 і два поля xy та pq, подані в шаховій нотації (див. вправу 11, підрозд. 1.3). Потрібно з'ясувати:
    а) чи ці поля одного кольору;
    б) чи загрожує ферзь на полі xy полю pq;
    в) чи загрожує кінь на полі xy полю pq;
    г) чи може тура з поля xy за один хід потрапити на поле pq; якщо ні, то з'ясувати, як це можна здійснити за два ходи (указати поле, на яке приводить перший хід);
    д) чи може слон із поля xy за один хід потрапити на поле pq; якщо ні, то з'ясувати, як це можна здійснити за два ходи (указати поле, на яке приводить перший хід);
    е) чи може король із поля xy за один хід потрапити на поле pq; якщо ні, то з'ясувати, за скільки ходів це можна здійснити;
    є) чи загрожує слон із поля xy королю на полі pq;
    ж) чи загрожує тура з поля xy коню на полі pq, і навпаки.
    Розв'язок оформити у вигляді функцій **int f(char x,int y,char p, int q),** тіла яких є послідовностями виразів присвоювань і умовних виразів.

3.3. Оператори

- Базові оператори
- Структуровані оператори

Ключові слова: *оператор, базовий оператор, структурований оператор, порожній оператор, оператор переходу, оператор із міткою, присвоювання, оператор виклику функції, ініціалізація параметрів і локальних даних, оператор-вираз, оператори продовження, переривання й виходу з функції, безумовний перехід, продовження, переривання, групове присвоювання, складений оператор, розгалуження, обхід, ітерація, повторення, структурний цикл, перемикач.*

Оператори – це конструкції, що безпосередньо задають обробку даних у програмах (створюють нові дані або змінюють значення вже існуючих). З погляду семантики вони є $X-Y$-операторами оновлення (див. підрозд. 2.1.2). Композиційну будову операторів описують структурні схеми програм (підрозд. 2.5.1).

В операторних інтерпретаціях ССП станами є сукупності даних, а елементарними предикатами й перетвореннями – відповідно арифметичні вирази й базові $X-Y$-оператори оновлення. Загальна синтаксична структура операторів виглядає так:

```
<оператор>::=<оператор-без-мітки>|
                        <мітка>:<оператор-без-мітки>
<оператор-без-мітки>::=<базовий-оператор>|
                        <структурований-оператор>
<мітка>::=<ідентифікатор>
<базовий-оператор>::=<порожній-оператор>|<оператор-вираз>|
|<оператор-переходу>|<оператор-повернення>
```

3.3.1. БАЗОВІ ОПЕРАТОРИ

Усі базові оператори синтаксично закінчуються символом ';'.

Порожній оператор. Не впливає на стан пам'яті. Однак, якщо його наділити міткою, то на нього можна передати керування в програмі за допомогою оператора безумовного переходу:

```
<порожній-оператор>::=;
```

Оператори-вирази. Це вирази, що закінчуються '**;**':

```
<оператор-вираз>::=<вираз>;
```

У процесі виконання оператора обчислюється значення виразу. Якщо у виразі немає побічних ефектів, то результат його як $X-Y$-оператора буде такий самий, як у порожнього оператора. Наприклад:

```
1;              /*оператор, еквівалентний порожньому оператору*/
i++;            /*збільшує значення змінної i на 1*/
--k;            /*зменшує значення змінної i на 1*/
```

Основні $X-Y$-оператори мови С присвоювання й виклику функції є операторами-виразами. Вирази-присвоювання розглядалися в підрозд. 3.2.1, а виклики функцій – у підрозд. 3.1.3, 3.6.2. Груповим присвоюванням називається оператор присвоювання вигляду: x=y=…=e;, де x,y… – змінні, e – вираз. У результаті групового присвоювання всі змінні ліворуч від виразу e отримують його значення.

Фактичних параметрів у списку виклику функції стільки, скільки формальних параметрів у заголовку функції (її прототипі) – по одному для кожного формального параметра. При цьому тип фактичного параметра має бути сумісним із типом відповідного до нього формального.

Виконання оператора виклику функції зводиться до таких дій. Спочатку здійснюється *ініціалізація* параметрів і локальних даних. Параметрам функції, її локальним даним, результату й точці продовження відводяться ділянки автоматичної й динамічної пам'яті. *Точка продовження* – це адреса машинної команди, що йде відразу за командою даного виклику функції (адреса продовження)[30]. Обчислюються значення фактичних параметрів, які зводяться до типу своїх формальних параметрів і присвоюються останнім як початкові значення. У відведене поле записується адреса продовження.

Далі виконується *модифіковане тіло функції* (тіло функції після етапу ініціалізації параметрів і локальних даних).

Останній етап – *вихід із функції*. Вихід із функції, яка повертає результат, дещо відрізняється від виходу з функції, яка не повертає результат. У першому випадку вихід обов'язково здійснює *оператор повернення*:

```
return[<вираз-результат>];
```

у другому це може бути також автоматичний вихід після виконання останнього оператора в модифікованому тілі. У процесі виходу: а) припиняється доступ до автоматичної пам'яті, яка використовува-

[30] У НМ для цього передбачено регістр **J**.

лася при виклику функції; б) у місце виклику повертається результат (якщо вихід здійснюється оператором **return** із виразом-результатом); в) у лічильник команд **PC** завантажується точка продовження.

Оператори переходу. Змінюють лічильник команд процесора (**PC**) і тим самим впливають на хід обчислень:

```
<оператор-переходу>::=<безумовний-перехід>|<продовження>|
<переривання>|<вихід-із-функції>
```

Оператори в тілі функції можуть мати *мітки*, що належать до службових даних і є ідентифікаторами. Щоб описати мітку, достатньо поставити її перед оператором у певному блоці й відокремити від нього символом '**:**'. Наприклад, **{…L1: printf ("Error --1"); …}**. Областю існування мітки як даного є блок, в якому вона описана. За допомогою мітки на її оператор може бути здійснена передача керування та ініційоване продовження виконання блока саме з даного оператора. Таку передачу керування здійснює *оператор безумовного переходу:*

```
goto <мітка>;
```

На операційному рівні оператору **goto L;** відповідає команда безумовного переходу, яка заносить у лічильник команд адресу першої з групи команд, що відповідають у машинному коді оператору з міткою **L**.

Оператори продовження **continue;** і **переривання** **break;** застосовуються в циклах (останній – і в перемикачах). Перший перериває поточний ітеративний крок внутрішнього циклу, у тілі якого перебуває, і примушує розпочати новий. Другий примусово перериває й закінчує виконання внутрішнього циклу або перемикача, у тілі якого він міститься.

Оператор виходу з функції **return[<вираз-результат>]** обчислює значення виразу-результату, перетворює його до типу результату функції, записує в поле результату, пересилає звідти в місце виклику й передає керування програмою за адресою поля продовження.

3.3.2. СТРУКТУРОВАНІ ОПЕРАТОРИ

Загальний вигляд:

```
<структурований-оператор>::=<складений-оператор>|
                  <розгалуження>|
                  <обхід>|
                  <перемикач>|
```

```
<ітерація>|
<повторення>|
<цикл>
```

Складений оператор. Подається складеною регулярною схемою програм:

```
<складений-оператор>::=\{{<оператор>}\}
```

Дія складеного оператора зводиться до послідовного виконання операторів. Серед операторів можливі й оператори опису даних. У цьому випадку складений оператор називається *блоком.* Оператори опису визначають локальні дані блока.

Розгалуження й обхід. Подається регулярними схемами програм розгалуження й обходу. Нульове значення цілого виразу означає хибу, а всі інші – істину:

```
<розгалуження>::=if(<цілий-вираз>)<оператор>
else <оператор>
<обхід>::=if(<цілий-вираз>)<оператор>
```

Допускається використання вкладених операторів розгалуження й обходу.

Приклад 3.28. Наведений фрагмент ілюструє вкладені оператори **if-else:**

```
char znac;
int x,y,z;
…
      if (znac= ='-')x=y-z;
      else if (znac= ='+')x=y+z;
           else if (znac= ='*')x=y*z;
              else if (znac= ='/')x=y/z;
                 else printf("error");
…
```

■

Із цього прикладу видно, що каскадні вкладення операторів розгалуження не завжди зручні. Іншим способом організації вибору з множини різних варіантів дій є використання оператора-перемикача **switch.**

Перемикач. Оператор-перемикач – це випадок регулярного детермінованого вибору detcase:

```
<перемикач>::=switch(<цілий-вираз>) {
case <цілий-вираз-конст1>:<оператор>…<оператор>…
```

```
[default:<оператор>…<оператор>]
…
case <цілий-вираз-констN>:<оператор>…<оператор>
}
```

Усі вирази-константи мають бути попарно різними. Варіант дій із випадком **default** може бути тільки один. У процесі виконання оператора обчислюється значення цілого виразу й порівнюється з кожною **case**-константою. Якщо знаходиться варіант зі збігом значень, то далі послідовно виконуються оператори даного варіанта й усіх наступних. Якщо жодна із **case**-констант не знайдена, то керування передається на **default** за умови, що він є в конструкції. Інакше виконується порожній оператор. У тілі оператора **switch** можна використовувати вкладені оператори-перемикачі.

Щоб у перемикачі забезпечити виконання операторів тільки одного варіанта, у кінці кожного з варіантів розміщують оператор переривання **break;**, який примусово завершує роботу перемикача.

Приклад 3.29. Перепишемо фрагмент коду з прикл. 3.28 із використанням перемикача:

```
char znac;
int x,y,z;

switch (znac)
{
 case'+':x=y+z; break;
 case'-':x=y-z; break;
 case'*':x=y*z; break;
 case'/':x=u/z; break;
 default:printf("error");
}
```

■

Ітерація, повторення і структурний цикл. Перші два цикли подаються регулярними схемами програм ітерації й повторення:

```
<ітерація>::=while(<умова>)<оператор>
<повторення>::=do<оператор> while (<умова>)
```

Умовою повторення оператора є ненульове значення цілого виразу. На відміну від ітерації тіло оператора повторення завжди виконується принаймні один раз.

Структурний цикл має вигляд:

```
<структурний-цикл>::=for([<вираз1>];[<цілий-вираз2>];[<вираз3>])\
                        <оператор>
```

Кожний із виразів (і всі одночасно) може бути відсутнім. Відсутність другого виразу еквівалентна виразу 1.

Семантика задається складеним оператором:

```
{<вираз1>; while (<цілий-вираз2>) {<оператор> <вираз3>;}}
```

Спочатку обчислюється як оператор перший вираз. Зазвичай це послідовність присвоювань, що ініціалізують значення змінних перед їхньою обробкою в циклі. Далі під керуванням другого виразу відбувається ітерація, тіло якої завершується обчисленням як оператора третього виразу, тобто в третьому виразі подаються змінні, що змінюються останніми в циклі.

Приклад 3.30. Обчислення біфакторіала:

```
long bifactorial(unsigned int n)
{       unsigned long k,p;
       p=k=(n%2==0? 2:1);
       while (k<n) {
           k+=2; /*k=k+2; */
          p*=k; /*p=p*k; */
                              }
       return p;
}

long bifactorial2(unsigned int n)
{      unsigned long k, p;
       for (p=k=(n%2==0? 2:1); k<n; k+=2) p*=k;
       return p;
}
```

У циклі **for** змінна **k** буде змінюватися після змінної **p** ■

Приклад 3.31. Швидке піднесення до степеня (див. прикл. 2.31):

```
double qpow(double x, unsigned int n)
{double p, a=x;
    unsigned int q=n;
    if (n%2==0) p=1;
    else p=x;
    while (q>0)
    {
          a=a*a;
          q=q/2;
```

```
            if (q%2==1) p=p*a;
        }
        return p;
}
```

■

***Література для СР:** оператори – [55, 102, 133, 136].

Контрольні запитання та вправи

1. Що таке оператор із семантичного погляду?
2. Описати синтаксис і семантику порожнього оператора.
3. Описати синтаксис і семантику операторів переходу.
4. Описати синтаксис і семантику оператора присвоювання.
5. Що таке оператор групового присвоювання?
6. Які складні оператори присвоювання є в мові С?
7. Описати синтаксис і семантику оператора виклику функції.
8. Що таке ініціалізація параметрів і локальних даних?
9. Описати синтаксис і семантику оператора-виразу.
10. Описати синтаксис і семантику операторів продовження й переривання.
11. Описати синтаксис і семантику оператора виходу з функції.
12. Описати синтаксис і семантику операторів розгалуження й обходу.
13. Описати синтаксис і семантику оператора-перемикача.
14. Описати синтаксис і семантику операторів ітерації, повторення та структурного циклу.
15. Робота з текстом програм. Що виведе послідовність операторів:

```
int i=1, x=1, y=2;
while(i<=10)
{++i;
x*=i; y*=2;
printf ("\n%d\t%d\t%d", i, x, y);
}
```

16. Робота з текстом програм. Що виведе програма:

а)
```
#include <stdio.h>
int main()
{int counter;
printf("while\tcount\n");
counter=1;
while(counter<=10)
{printf("%d\n", counter);
counter++;
}
  return 0;
```

```
}
```

б)
```
#include <stdio.h>
int main()
{ int counter;
printf("while count\t");
counter=10;
while(counter>=1)
{ printf("\n%d", counter);
counter--;
}
return 0;
}
```

в)
```
#include <stdio.h>
int main()
{ char c;
printf("while\talphabet\n");
c='A';
while(c<='Z')
{ printf("%c\t", c);
c++;
}
return 0;
}
```

г)
```
#include <stdio.h>
int main()
{ int counter;
printf("do-while\ncount\n");
counter=0;
do
{counter+=2;
printf("%d\n", counter);
} while(counter<20)
return 0;
}
```

д)
```
#include <stdio.h>
int main()
{int counter;
printf("while count\n");
counter=20;
while(counter>=1)
{          printf("\t%d\n", counter);
counter-=2;
}
return 0;
}
```

17. Робота з текстом програм. Перевірити й виправити програму. Що вона виведе?

а)
```
#include <stdio.h)
int main()
{ printf("Print\n\n\n\n)
for(i=0; i<=10, i++)
printf("i=%c", i);
return 0;
}
```

б)
```
(* program test *)
#include "stdio.h>
int main()
(int ivalue;
float fvalue=3.14159;
ivalue=12;
printf("Print\n",ivalue="%d", ivalue)
printf("%f", fvalue);
}
return 0;
)
```

в)
```
{comment}
include <stdio.h>
int main()
{ int counter=1;
printf("while count\n", counter);
while(counter>0)
{ printf("%d\n", counter);
  counter++;
}
return 0;
}
```

г)
```
#include stdio.h
int man()
{
integer i=2;
printf("while\talphabet\n", i);
while(i<=245)
{ printf('%c\t", c);
c--;
]
return 0;
}
```

д)
```
include <stdio.h>
int main()
( int counter=0;
```

```
printf("do-while\ncount\n");
do
( c+=2
printf("%d\n", counter);
) while(counter>0)
return 0;
```

18. За цілим $a > 0$ обчислити найбільше n таке, що $n! < a$.
19. Написати функцію обчислення $n!$, $n \geq 0$ $(0! = 1)$.
20. Дано цілі числа $m, n \neq 0$. Знайти всі їхні загальні дільники.
21. Перевірити, чи є задане натуральне число простим.
22. Натуральне число називається досконалим, якщо воно збігається із сумою всіх своїх власних (тобто менших за нього) дільників. Наприклад, першим таким числом є $6 = 1 + 2 + 3$. Знайти всі досконалі числа не більші ніж n.
23. Обчислити вираз $1 - 1/2 + 1/3 - \ldots + 1/999 - 1/1000$: а) зліва направо; б) зліва направо, але спочатку знайти суму членів із непарними знаменниками, потім – з парними, а потім узяти їхню різницю; в) справа наліво, г) див. б), але обчислення проводяться справа наліво. Порівняти й пояснити можливі розбіжності в результатах машинних обчислень.
24. Знайти, порівняти й пояснити різницю в машинних нулях в околах чисел 1.0 та $1.0E9$. Це найбільші числа вигляду $\delta = \frac{1}{2^n}$ такі, що $1 + \delta = 1$ та $1.0E9 + \delta = 1.0E9$, відповідно.
25. Знайти наближені значення функцій e^x та $\sin(x)$ із заданою точністю $\varepsilon > 0$ за допомогою степеневих рядів відповідно $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$ та $\sin(x) = \sum_{i=1}^{\infty} (-1)^{i-1} \frac{x^{2i-1}}{(2i-1)!}$. Ураховувати тільки ті члени ряду, які за абсолютною величиною більші ε.
26. Знайти наближені значення констант із заданою точністю $\varepsilon > 0$, використовуючи їхні подання у вигляді нескінченних степеневих рядів і добутків. Для сум ураховувати тільки ті члени ряду, які за абсолютною величиною більші ε, а для добутків – тільки ті співмножники, що відрізняються від 1 більш ніж на ε: а) $2 = \sum_{i=0}^{\infty} \frac{1}{2^i}$; б) $2/3 = \sum_{i=0}^{\infty} (-1)^i \frac{1}{2^i}$;

в) $\pi^2/6=\sum_{i=1}^{\infty}\frac{1}{i^2}$; г) $\pi^2/12=\sum_{i=1}^{\infty}(-1)^{i-1}\frac{1}{i^2}$; д) $1/2=\sum_{i=1}^{\infty}\frac{1}{(2i-1)(2i+1)}$;

е) $\pi/2=\prod_{i=1}^{\infty}\frac{4i^2}{4i^2-1}$.

27. Знайти наближені значення функцій із зад. 25 з урахуванням їх повільного спадання для великих аргументів. Використати тотожність $e^x=e^{[x]}\cdot e^{\{x\}}$, $x\ge 1$ і періодичність функції $\sin(x)$. Тут $\{x\}$ – дробова частина числа x. Обчислення першої e^x за допомогою ряду $\sum_{i=0}^{\infty}\frac{x^i}{i!}$ звести тільки для $0<x<1$, а другої – $\sin(x)=\sum_{i=1}^{\infty}(-1)^{i-1}\frac{x^{2i-1}}{(2i-1)!}$ – тільки для $0\le x\le \pi/2$. З'ясувати, як змінюється точність обчислення зі зростанням $x=0.1,\ 1.0,\ 10.0,\ 100.0$ тощо для $\varepsilon=10^{-6}$, порівнюючи відповідні значення зі значенням стандартних функцій $\exp(x)$ та $\sin(x)$.

28. Знайти наближене значення числа π за формулою Мечина: $\pi/4=4\text{arctg}(1/5)\text{-arctg}(1/239)$ [33]. Для обчислення функції arctg скористатися рядом: $\text{arctg}(1/x)=\sum_{i=1}^{\infty}\frac{(-1)^{i+1}}{(2i-1)x^{2i-1}}$.

29. Нехай $a_1=u;\ b_1=v;\ a_k=2b_{k-1}+a_{k-1};\ b_k=2a_{k-1}^2+b_{k-1}$. Дано дійсні u, v, натуральне n. Знайти $\sum_{k=1}^{n}\frac{a_k b_k}{(k+1)!}$.

30. Нехай $x_1=x_2=x_3=1;\ x_i=x_{i-1}+x_{i-3},\ i=4,5,\ldots$ Знайти $\sum_{i=1}^{100}\frac{x_i}{2^i}$.

31. Нехай $a_0=a_1=1;\ a_k=ka_{k-1}+1/a_{k-2}$. Дано натуральне n. Отримати a_n.

32. Нехай $x_1=y_1=1;\ x_i=0.3x_{i-1};\ y_i=x_{i-1}+y_{i-1},\ i=2,3,\ldots$ Дано натуральне n. Знайти $\prod_{i=1}^{n}\frac{x_i}{1+|y_i|}$.

33. Нехай $a_1 = b_1 = 1;\ a_k = 3b_{k-1} + 2a_{k-1};\ b_k = 2a_{k-1} + b_{k-1}$. Дано натуральне n. Знайти $\sum_{k=1}^{n} \frac{2^k}{\left(1 + a_k^2 + b_k^2\right)k!}$.

34. Нехай

$$a_1 = b_1 = 1;\ a_k = \frac{1}{2}\left(\sqrt{b_{k-1}} + \frac{1}{2}\sqrt{a_{k-1}}\right);\ b_k = 2a_{k-1}^2 + b_{k-1},\ k = 2,3,\ldots$$

Дано натуральне n. Знайти $\sum_{k=1}^{n} a_k b_k$.

3.4. Тип масивів

- Специфікатор масивів
- Динамічні масиви
- Масиви як параметри функцій
- Масиви й символьні рядки

Ключові слова: *тип масивів, масив, оператори опису й декларації масиву, гнучкий тип, специфікатор масиву, багатовимірні масиви, змінні з індексами, операція індексування, ініціатор масиву, динамічні масиви, масиви як формальні й фактичні параметри функцій, символьні рядки, ініціалізація символьних рядків.*

Тип масивів – це тип, значеннями якого є вектори фіксованої довжини з однотипними компонентами. Дані цього типу називаються *масивами*. Тип компонентів масивів задають у їхньому описі або декларації. Обмежень на нього немає. Єдине – він не може бути неповним (див. підрозд. 3.4.1) або порожнім типом **void.**

3.4.1. СПЕЦИФІКАТОР МАСИВІВ

Специфікатор змінної-масиву задає її ім'я та довжину. Тип компонент задають у операторі опису масиву:

```
<специфікатор_масиву>::=\(<специфікатор-одновимірного-масиву>\)|
\(<специфікатор-масиву>\) \[<константний_вираз>\]
<специфікатор-одновимірного-масиву>::= <ім'я>\[<константний_вираз>\]
```

Усі неодновимірні масиви називаються *багатовимірними.*

Увага! Круглі дужки в специфікаторі масиву можна опустити. Їх використовують для поліпшення читабельності деяких специфікаторів ►

Приклад 3.32. Опис масивів:

```
char ascii[256];/*ascii – одновимірний символьний масив довжиною 256*/
static int a[10]; /*a – статичний цілий масив довжиною 10*/
int y [2]; /*y – цілий масив довжиною 2*/
int (c[2])[3]; /*c – двовимірний масив довжиною 2 з компонентами – цілими масивами довжиною 3*/
int c[2][3]; /*c – те саме*/
int ((z[2])[3])[4]; /*z – тривимірний масив довжиною 2 з компонентами – двовимірними масивами довжиною 3 з компонентами – цілими масивами довжиною 4*/
int z[2][3][4]; /*те саме, що й у попередньому прикладі*/
T x[n]; /*x – масив типу T довжиною n, n-константа*/
```

■

Далі наведені масиви без спеціального посилання будуть використовуватися для ілюстрації тих чи інших конструкцій, пов'язаних із масивами.

Компоненти масиву розміщують послідовно в одному полі ОП. Наприклад, цілому масиву **y** відповідає поле довжиною 4 байти – по 2 байти на кожний компонент.

На масивах визначена єдина операція **sizeof**, яка повертає розмір пам'яті масиву в байтах. Для описаних вище масивів:

```
sizeof(ascii)=256*sizeof(char)=256,
sizeof(a)=10*sizeof(int)=20.
```

В описі довільного масиву **x** типу **T** ім'я **x** не тільки подає відповідний вектор елементів, а й має певне значення в системі базових типів – це *покажчик-константа*, що адресує перший компонент масиву. Усю обробку масивів забезпечують змінні з індексами. Компоненти масиву **x** індексуються числами від 0 до *n*-1 і автоматично зв'язуються з виразами **x[0]**,…,**x[n-1]**, які називаються *змінними з індексами*. Адреса змінної **x[0]** є значенням **x** як покажчика. Змінні з індексами забезпечують *доступ* до компонентів масиву відповідно до їхнього типу **T**. Наприклад, для масивів із прикл. 3.32 змінні **ascii[0]**,..., **ascii[255]** – символьні, **a[0]**,..., **a[9]** – цілі, **(c[2])[0]**, **(c[2])[1]**, **c[2][0]** та **c[2][1]** – теж цілі, а змінна **c[2]** – цілий масив із трьох компонентів.

Ефективна обробка масивів базується на *індексованих виразах*, що будуються за допомогою операції індексування. Для масиву **x** ці вирази мають вигляд **x[e]**, де **e** – цілий арифметичний вираз. Значенням індексованого виразу **x[e]** є змінна з індексом **x[к]**, де **к** – значення виразу **e**.

Приклад 3.33.

```
int i,j;
ascii[0]='\0'; /*нульовий компонент ascii замінюється на 0*/
a[i]=a[i]+1; /*i-й компонент масиву a збільшується на 1*/
a[i]++; /*те саме*/
c[i][j]=2*c[i][j]; /*j-й компонент масиву c[i] подвоюється*/
z[i][j][n]=i+j+n; /*n-й компонент масиву z[i][j] замінюється на
суму індексів*/
```

Наступні два цикли вводять значення компонентів масиву **a** зі стандартного вхідного потоку й виводять на екран масив **c**:

```
 /*виведення масиву a*/
for (i=0; i<10; i++) scanf ("%d", &a[i]); /*&a[i] – адреса
                                          змінної a[i]*/
/*виведення масиву c*/
for (i=0; i<2; i++)
 {printf ("\n"); /*перехід на новий рядок екрана*/
 for (j=0; j<3;j++ ) printf ("%d", (c[i])[j]);
 }
```

У зовнішньому циклі спочатку вибирається **i**-й компонент **c[i]** масиву **c**, у внутрішньому – безпосередньо виводяться на екран усі його компоненти **c[i][j]** ■

В описі за допомогою ініціатора можна присвоїти початкові значення компонентам масиву:

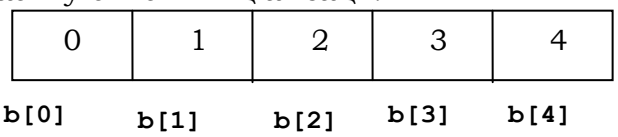
```
<ініціатор-масиву>::={<вираз>{,<вираз>}}
```

Вирази в правій частині мають відповідати типу компонентів масиву. Стандарт С99 допускає неконстантні ініціатори, а також правила для скорочення запису ініціаторів. Зокрема, якщо ініціаторів полів менше ніж потрібно, то використовуються їхні значення за умовчанням або хвіст масиву просто не ініціалізується, якщо більше – це викликає помилку в роботі компілятора. В описах з ініціалізацією масиву можна використовувати специфікатори масиву без довжини. У цьому випадку компілятор сам установлює довжину масиву за кількістю елементів у ініціаторі.

Приклад 3.34. Ініціалізація масивів:

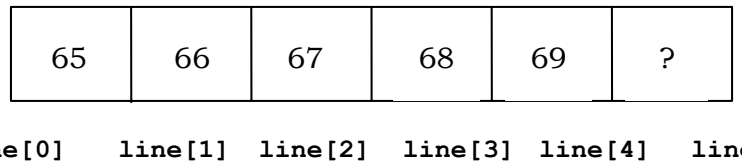
а) `int b[5]={0,1,2,3,4} ; /*ініціалізація цілого масиву b*/`

Стан поля масиву `b` після ініціалізації:

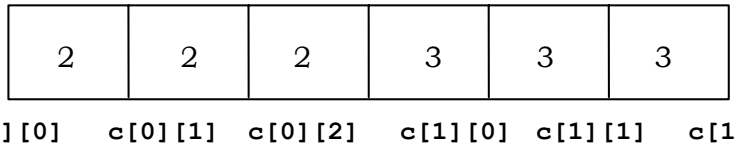


Таким самим буде результат описів: `int n=4;int b[]={0,1,2,3,n};`

б) `char line [6]={'A','B','C','D','E' };`
`/*ініціалізація символьного масиву line: останній компонент не ініціалізується*/`



в) `int c[2][3]={{2,2,2}, {3,3,3}}; /*ініціалізація двовимірного масиву c*/`



Таким самим буде й результат описів з ініціалізаціями:

`int c[2][3]={2,2,2,3,3,3};` та `int c[][3]={{2,2,2}, {3,3,3}};` ■

Неповний специфікатор масиву в операторі декларації має формат

<неповний-специфікатор-масиву>::=<ім'я>[]|
(<неповний-специфікатор-масиву>)[<константний_вираз>]

Кажуть, що задекларований масив має *неповний* тип, оскільки в ньому відсутня інформація про довжину масиву. Такі масиви використовуються як параметри функції, у невизначальних зовнішніх деклараціях і при ініціалізації масивів. Однак у будь-якому разі, якщо типом компонент є масив, то його опис має бути повним.

Приклад 3.35.

```
extern int a[]; /*a – зовнішній статичний цілий масив*/
int f (char y []); /*y – символьний масив-параметр функції*/
extern int (c[0])[2]; /*c – зовнішній двовимірний масив*/
extern int c[][2]; /*c – те саме*/
int z[][3][4];
```

■

3.4.2. ДИНАМІЧНІ МАСИВИ

У С99, на відміну від попередніх версій С, є масиви зі змінною довжиною – *динамічні масиви*, які вперше з'явилися в мові Алгол-60. Їхній розмір з'ясовується не під час компіляції, а в процесі виконання програми. Опис динамічного відрізняється від опису звичайного масиву тим, що константний вираз у специфікаторі масиву замінюється на довільний цілий вираз зі змінними.

У процесі виконання оператора опису динамічного масиву спочатку обчислюється його довжина (має бути строго більше 0), потім надається відповідне поле в пам'яті. Звернення до елементів масиву не повинне виходити за його межі – оскільки автоматична перевірка порушення цієї вимоги не здійснюється, то таке звернення може мати непередбачувані наслідки. Це стосується й звичайних масивів.

Динамічні масиви можуть бути тільки локальними, визначеними в межах певного блока, і не можуть мати класи пам'яті `extern` та `static`. Діє таке правило:

Якщо розмір динамічного масиву в заголовку функції залежить від змінної, то вона має бути або глобальною для функції, або її опис як параметра повинен передувати опису динамічного масиву в заголовку функції.

Після виходу з блока доступ до динамічного масиву припиняється, і при наступному виконанні блока динамічний масив створюється заново.

До *гнучких типів* належать динамічні масиви та інші типи, що містять такі масиви в описах (напр., тип покажчика на динамічні масиви, тип функцій з параметрами – динамічними масивами).

Приклад 3.36. Розглянемо фрагмент програми, в якому два упорядковані динамічні масиви об'єднуються в третій – теж упорядкований динамічний масив. Таке об'єднання масивів називається злиттям:

```
/*merge здійснює злиття двох упорядкованих динамічних масивів
a[n] та b[n] у масив c[2*n]*/
```

```
double merge (int n, double a[n], double b[n], double c[2*n])
{int i, j, k;
 i=j=k=0;
 while (i<n && j<n)
 if (a[i]<b[j]) c[k++]=a[i++];
 else c[k++]=b[j++];
 while (i<n) c[k++]=a[i++];
 while (j<n) c[k++]=b[j++];
}

int main(void)
{
 scanf("%d", &n);
double aa[n];
double bb[n];
double cc[2*n];
…
merge (aa, bb, cc);
…
}
```

■

3.4.3. МАСИВИ ЯК ПАРАМЕТРИ ФУНКЦІЙ

Оскільки перевірка виходу індексів за межі масиву в С не передбачена, то як формальні параметри функцій допускаються масиви повного й неповного типів.

У викликах функції параметру-масиву типу `T` відповідає фактичний параметр – покажчик типу `T`, який задає адресу конкретного масиву, що передається як фактичний параметр. Таким покажчиком може бути ім'я певного масиву. Цієї адреси достатньо, щоб за допомогою операції індексації отримати доступ до кожного компонента масиву, що передається функції для обробки. Таким чином, при передаванні у функцію фактичного параметра масиву передається тільки його адреса, а самі елементи залишаються на місці, не передаються й не копіюються.

Для того, щоб контролювати межі параметра-масиву, до списку параметрів часто додають цілий параметр, що задає його довжину.

Якщо параметром функції є динамічний масив, то в його декларації в прототипі функції розмір можна подати символом ‘`*`'. Будь-який неконстантний розмір `[e]` динамічного масиву в прототипі функції вважають еквівалентним `[*]`.

Приклад 3.37. Масиви як параметри функцій:

```
   /*f1 знаходить середнє арифметичне компонентів дійсного масиву
v довжиною n*/
   double f1 (int n, double v[])
   {
   double s=0;
   for (int i=0;i<n;i++) s+=v[i];
   return s/n;
   }
   extern void f2 (int n, int w[*][5]); /*n – довжина
    динамічного масиву w*/

   int main(void)
   {
   int m;
   int b[10];
   scanf("%d", &m);
    int z[m+1][5]; /*z – динамічний масив w*/
   …
   printf("\n%f", f1(10, b)); /*виклик функції f1 із фактичним па-
раметром – масивом b*/
   f2(m+1, z); /*виклик функції f2 із фактичним параметром – дина-
мічним масивом z*/
   …
   }
```

■

3.4.4. МАСИВИ Й СИМВОЛЬНІ РЯДКИ

Символьний рядок – це послідовність символів, що завершується символом-нулем `'\0'`.

Рядки, довжина яких обмежена певною кількістю n, зручно зображувати символьним масивом довжиною $n+1$. Наприклад, для $n=9$ опис відповідного масиву може бути таким: `char str[10];`. Такі рядки можна ініціалізувати за допомогою літералів:

```
char str[10]="TEST";
```

Поле рядка `str` після ініціалізації має вигляд

84	69	83	84	00					

Як бачимо, у кінець рядка компілятор автоматично додає нульовий символ (який не враховується при обчисленні довжини рядка).

Для роботи з рядками у С99 існує бібліотека стандартних функцій. Прототипи їх містяться в бібліотеці `<string.h>` (деякі описані в підрозд. 3.5.3).

***Література для СР:** Масиви – [55, 102, 132, 133].

Контрольні запитання та вправи

1. Що таке масив?
2. Елементи яких типів можуть утворювати масиви?
3. Який вигляд має оператор опису (декларації) масиву?
4. У чому полягає смисл змінних з індексами й операції індексування?
5. Що таке багатовимірний масив?
6. Як розташовані в пам'яті компоненти масивів **int(c[2])[3]** та **int ((z[2])[3])[4]**?
7. Який вигляд має ініціатор масивів **int(c[2])[3]** та **int ((z[2])[3])[4]**?
8. Як зображується рядок у мові С?
9. Що таке повний і неповний типи масивів?
10. Що таке гнучкий тип? Навести кілька прикладів гнучких типів.
11. Що таке динамічний масив? Навести приклади декларації динамічного масиву.
12. Які з декларацій масивів **int(c[])[3], int(c[2])[2]** та **int(c[2])[]** є коректними й чому?
13. Які особливості передавання одновимірних і багатовимірних масивів у функції?
14. Які з декларацій динамічних масивів:
    а) **double f (int n, double v[m] [n])**,
    б) **double f (int m, double v[*][m])**,
    в) **double f (int n, double v[n])** не є коректними й чому?
15. Навести приклади ініціалізації рядків.
16. Дано послідовність цілих чисел $a_1,..,a_n$ і число *c*. Знайти перше й останнє місцезнаходження (перший індекс ліворуч і праворуч) числа *c* у даній послідовності. Побудувати відповідні рекурентні послідовності для розв'язку задачі.
17. Що виведуть такі фрагменти програм:

```
int a[]={0,1,2,3,4};
int b[3][3]={{1,2,3},{4,5,6},{7,8,9}};
int *pb[3]={b[0],b[1],b[2]};
int *p = b[0];
```

а)

```
void main()
{ int i, *p;
for(p=a,i=0;p+i<=a+4;p++,i++) printf (*(p+i));
}
```

б)

```
void main()
{ int i;
for(i=0; i<3;i++)
printf( "%d\t,%d\t,%d\t",b[i][2-
i],*b[i],*(*(b+i)+i));
}?
```

18. Дано послідовність цілих чисел $a_1,..,a_n$. Знайти найбільший (найменший) її елемент. Побудувати відповідні рекурентні послідовності для розв'язку задачі.
19. Дано послідовність цілих чисел $a_1,..,a_n$. Знайти її медіану, тобто різницю між найбільшим і найменшим елементами. Поміняти місцями в ній найбільші й найменші елементи за умови: а) усі a_i попарно різні; б) у загальному випадку.
20. Дано послідовність цілих чисел $a_1,..,a_n$. Перебудувати її таким чином, щоб: а) спочатку йшли поспіль (у тому ж самому порядку) від'ємні її члени, потім – нульові, а вже потім – додатні; б) те саме, що в а), але нульові члени мають залишатися на своїх місцях.
21. Дано багаточлен $P(x)$ степеня n, заданий масивом $a(n)$ його цілих коефіцієнтів. Знайти коефіцієнти багаточлена: а) $(x-4)P(x)$; б) $P^2(x)$.
22. Дано багаточлен $P(x)$ степеня n із цілими коефіцієнтами. Побудувати рекурентні послідовності й написати функцію **`double polinom(int a[], int n, double x)`** для обчислення за схемою Горнера значення $P(x)$. Знайти: а) $P(1)$; б) $P(1)/P'(1)$; в) $\int_0^1 P(x)dx$; г) усі цілі корені $P(x)$. Функцію **`polinom`** зберігати в окремому файлі **`"p.c"`**. Для включення її до основної програми застосувати директиву **`#include "p.c"`**.
23. Написати функцію для відшукання перших n простих чисел. Надрукувати перші 100 простих чисел.
24. Знайти всі прості числа на інтервалі від 2 до n. Застосувати ґратку Ератосфена: спочатку із сукупності $[2,n]$ відкидаються парні числа, потім з того, що залишилося – кратні 3 тощо.
25. Обчислити надвеликі числа: а) 2^{100}; б) 100!.
26*. Обчислити 100 перших десяткових цифр основи натурального логарифма e за допомогою ряду із вправи 25, підрозд. 3.3 [33].

27. Дано числовий масив довжиною n. Надрукувати всі 2^n його підмасивів: а) з використанням рекурсії; б) без неї.
28. Скорочення перебору. Дано числовий масив довжиною n невід'ємних елементів і число s. Знайти всі його підмасиви із сумою елементів s. У процесі перебирання ігнорувати всі підмасиви із сумою елементів більше s.
29. Надрукувати гістограму частоти символів тексту, що вводиться з клавіатури.
30. Знайти індекси двох однакових елементів матриці.
31. Дано матрицю розміром $n \times m$. Розташувати її рядки: а) так, щоб перший стовпчик став упорядкованим за зростанням; б) за зростанням сум елементів рядків; в) за зростанням найменших елементів рядків.
32. Характеристикою вектора називають середньоарифметичне його компонентів. Написати функцію для перестановки в матриці двох рядків за їхніми номерами. Переставити рядки в матриці відповідно до зростання їхніх характеристик.
33. Визначити, чи впорядкований даний вектор (за зростанням чи спаданням). Знайти номер максимального за характеристикою (див. вправу 32) упорядкованого рядка матриці.
34. Перевірити, чи складається вектор з попарно різних компонентів. Підрахувати кількість рядків матриці, що складаються з попарно різних елементів.
35. Перевірити, чи утворюють компоненти вектора довжиною m перестановку чисел від 1 до m. Підрахувати, скільки рядків у даній матриці розміром $n \times m$ є перестановками чисел від 1 до m
36. Перевірити, чи містить вектор лише непарні компоненти. Знайти серед рядків матриці, що містять лише непарні елементи, рядок з найбільшою характеристикою (див. вправу 32).
37. Серія – це послідовність, що складається з однакових елементів. Знайти: а) довжину найбільшої серії серед рядків матриці; б) рядки з найдовшою серією.
38. Два вектори схожі, якщо збігаються множини їхніх компонент. Написати функцію для перевірки, чи схожі два вектори. Визначити, які рядки матриці: а) найбільш схожі; б) найменш схожі.
39. Написати функцію для обчислення скалярного добутку двох векторів. Визначити рядок матриці, скалярний добуток якого з першим рядком матриці максимальний.
40. За даною числовою квадратною матрицею A порядку m і вектором b довжиною m отримати вектор: а) Ab; б) $(A^3 - E)b$.

41*. За даною булевою квадратною матрицею A порядку m обчислити матрицю A^n. Урахувати, що квадратні матриці є півгрупою відносно множення, тому можна скористатися алгоритмом швидкого піднесення до степеня.

42*. Написати функцію для розв'язання методом Гаусса системи лінійних рівнянь. Застосувати її для розв'язку систем:
$10x_1 + x_2 + x_3 = 12;\ 4x_1 + 0.24x_2 - 0.08x_3 = 8;$
$2x_1 + 10x_2 + x_3 = 13;\ 0.09x_1 + 3x_2 - 0.15x_3 = 9;$
$2x_1 + 2x_2 + 10x_3 = 14;\ 0.04x_1 - 0.082x_2 + 4x_3 = 20$ [19, 33].

43*. За даною квадратною матрицею A порядку m знайти обернену матрицю A^{-1}.

3.5. Тип покажчиків

- Специфікатор покажчиків
- Операції з покажчиками
- Індексація покажчиків і масиви
- Масиви покажчиків
- Функції виділення пам'яті
- Моделювання динамічних масивів

Ключові слова: *тип покажчиків, покажчик, покажчики на об'єкти, функції загального призначення типу* **`void*`**, *операція взяття адреси* **`&`**, *розіменування покажчика* **`*`**, *константа* **`NULL`**, *зведення покажчиків, операції індексації, збільшення та зменшення покажчиків, адресна арифметика, масив покажчиків, динамічний масив, функції динамічного виділення пам'яті:* **`malloc`**, **`calloc`** *та* **`free`**, *моделювання динамічних масивів, надвеликі масиви.*

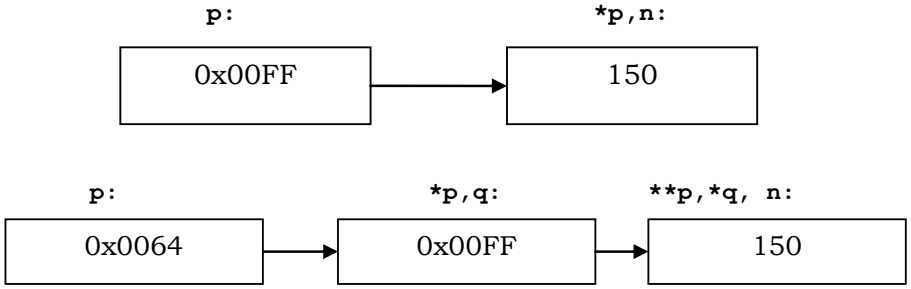
Тип покажчиків складають адреси полів об'єктів у ОП певного фіксованого типу з адресою порожнього поля **`NULL`**, що є препроцесорною константою з нульовим значенням і використовується для подання ситуації, коли покажчик не адресує об'єкт даного типу. Константи та змінні типу покажчиків називаються *покажчиками*.

Кажуть, що покажчик *посилається* на об'єкт, який він адресує. Якщо об'єктом, що адресує покажчик **`p`**, є інший покажчик, який теж

адресує певний об'єкт, то кажуть, що **p** *опосередковано посилається* на об'єкт (*непряма адресація*).

Отримати доступ до об'єкта, що адресується покажчиком **p**, можна за допомогою імені ***p** (див. операцію розіменування в підрозд. 3.5.2).

Зв'язки між покажчиками й об'єктами, на які вони посилаються, зручно подавати графічно. Наприклад: а) покажчик **p** посилається на цілу змінну **n=150**, розміщену за адресою **0x00FF**; б) покажчик **p** посилається на цілий покажчик **q**, розміщений за адресою **0x0064**, який, у свою чергу, посилається на ту саму змінну **n=150**. Тоді в першому випадку імена ***p** та **n** є синонімами. У другому випадку покажчик **p** непрямо посилається на **n**, імена **n**, ***q** і ****p** – синоніми. Ці факти графічно подаються так:



Кожному типу **T** відповідає тип покажчиків на нього – **T***. Наприклад, тип покажчиків **char*** складають адреси байтів із кодами символів, тип **int*** – адреси двобайтних слів із двійковими цілими числами тощо.

Для подання покажчиків використовують шістнадцяткові беззнакові константи: 0X00FF (=255), 0x00FF (=255), 0x7FFF (=32767). Структура покажчиків залежить від архітектури процесора. Зазвичай ОП має сторінкову організацію, тобто розбита на сторінки (сегменти) фіксованого розміру, наприклад 2^{16} байти, кожна з яких має порядковий номер (базу). Адреса поля тоді має структуру пари (*база*, *зсув*). У межах фіксованого сегмента вказують тільки зсув поля відносно його початку: від 0x0000 до 0x7FFF.

Розрізняють *покажчики на об'єкти, функції* та *загального призначення* типу **void***. Останні ще називають *безтиповими*. Вони сумісні з будь-якими покажчиками на об'єкт і використовуються в основному в прототипах і описах функцій. Покажчики на функції розглядаються також у підрозд. 3.6.

Покажчики застосовуються у двох різних напрямах:

а) по-перше, вони забезпечують прямий доступ до об'єктів програмних даних (як статичних, так і динамічних) і тим самим – можливість "ручного" керування пам'яттю;

б) по-друге, можливості адресної арифметики й непрямої адресації використовують для оптимізації коду програми.

Нагадаємо, що динамічними називаються дані, зв'язування яких з об'єктами в пам'яті відбувається не до, а в процесі виконання програми.

Увага! При роботі з покажчиками слід уникати невизначених, тобто ненульових, покажчиків, що не посилаються на реальний об'єкт даного типу ►

Приклад 3.38. Невизначений покажчик:

```
void main (void)
{double *cptr; /*cptr – покажчик типу double*/

   *cptr=1.0;
      …
}
```

Оператор присвоювання `*cptr=1.0;` записує число `1.0` у пам'ять за адресою, на яку вказує покажчик `cptr`. Це може мати непередбачувані наслідки, оскільки значення покажчика `cptr` у програмі ще не ініціалізувалося, отже, запис потрапить у випадкову ділянку пам'яті ■

Вважається коректною обов'язкова ініціалізація покажчика під час його опису. Як мінімум це має бути його обнулення. Наприклад, `int *p=NULL;`.

3.5.1. СПЕЦИФІКАТОР ПОКАЖЧИКІВ

Синтаксис специфікатора покажчика:

```
<специфікатор-покажчика>::=<покажчик> <прямий-специфікатор>
<покажчик>::=*[<кваліфікатор-типу>] [<покажчик>]
```

Прямий специфікатор (див. підрозд. 3.13) може бути ідентифікатором, специфікатором масиву, функції й покажчика, узятим у круглі дужки.

Приклад 3.39. Специфікатори покажчика:

Опис	Тип
`int * p;`	Покажчик на цілі
`int *p[];`	Масив покажчиків на цілі
`int *p();`	Функція зі значенням типу, який є покажчиком на цілі
`int (*p)();`	Покажчик на функцію із цілими значеннями
`const int *p;`	Покажчик на цілі константи
`int * const p;`	Константний покажчик на цілі
`char **p;`	Покажчик на символьні покажчики

■

Список кваліфікаторів типу містить їхній перелік (див. підрозд. 3.13), де кваліфікатор **restrict** використовується тільки з покажчиками й повідомляє компілятору, що в певний період часу покажчик є єдиним засобом доступу до об'єкта, на який він посилається. Жодний інший покажчик чи змінна не мають звертатись до об'єкта, якщо він адресується покажчиком із кваліфікатором **restrict**. Наприклад, якщо адресу динамічного масиву подати **restrict**-покажчиком:

```
int * restrict dm;
dm=(int*)calloc(100000,sizeof(int));
```

то до його елементів можна звертатись лише через покажчик **dm**.

Розглянемо інший приклад, а саме прототип стандартної функції strcpy (див. табл. 3.12):

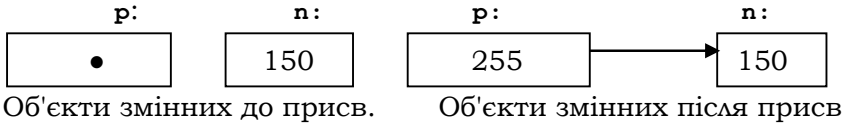
```
char *strcpy(char *s1,const char *s2);
```

яка копіює рядок **s2** за адресою **s1**. Кваліфікатор **const** указує, що адреса рядка **s2** у процесі виклику функції не змінюється. Однак сам рядок може змінитися при копіюванні, якщо поля рядків **s1** та **s2** перетинаються. Параметри **s1** та **s2** із кваліфікатором **restrict**: **char *strcpy(char* restrict s1,const char* restrict s2);** будуть забороняти таку ситуацію.

Компілятор оптимізує вирази з покажчиками, які не змінюються (з метою пришвидшення виконання програми). Без кваліфікатора **restrict** оптимізація могла б мати непередбачувані наслідки.

3.5.2. ОПЕРАЦІЇ З ПОКАЖЧИКАМИ

Операція *взяття адреси* змінної **&** повертає адресу свого операнда. Наприклад, у результаті присвоювання **p=&n;** покажчик **p** отримає значення, що є адресою цілої змінної **n** і дорівнює адресі першого з двох байтів поля, в якому зберігається значення змінної. Нехай **n** зберігається в полі з адресою 0x00FF, а її значення дорівнює **150**. Тоді після присвоювання змінна **p** отримає значення **255** і буде посилатися на змінну **n**:



Об'єкти змінних до присв. Об'єкти змінних після присв.

Друга операція для роботи з покажчиками позначається символом * і називається *розіменуванням*. Її ми вже розглядали на початку

даного підрозділу. Вона створює змінну для забезпечення доступу до об'єкта, на який посилається покажчик. У наведеному вище прикладі операція розіменування повертає змінну з ім'ям ***p**, яка є синонімом змінної **n**. Звернення **n** та ***p** до цілого об'єкта за адресою **255** у всіх припустимих контекстах є еквівалентними (доти, доки покажчик зберігатиме значення **255**).

Приклад 3.40. Операції взяття адреси й розіменування:

Вираз	Значення
`p=NULL;`	Обнулення покажчика
`p= =NULL`	Порівняння **p** із нулем після обнулення поверне 1
`!p`	Заперечення **p** після обнуління поверне 1
`*p<0;`	Порівняння **n** із нулем
`printf("%d\n", *p);`	Виведення значення **n** (= 150)
`printf("%p\n", p);`	Виведення значення покажчика (= 0x00FF)
`*p=0;`	Обнулення змінної **n**
`n=*p+2;`	Збільшення **n** на 2

■

Покажчик можна зводити до іншого типу. Перетворення здійснюються за участю безтипових покажчиків і без них. У мові С допускається присвоювання значень безтипового покажчика покажчику довільного типу (і навпаки) без явного перетворення типу. До безтипових покажчиків звертаються, якщо тип об'єкта невідомий. Вони дозволяють посилатися на довільну ділянку пам'яті, незалежно від розміщених там об'єктів. Їхнє використання як параметрів функції дозволяє передавати у функцію покажчик на об'єкт довільного типу.

Перетворення інших типів покажчиків мають бути завжди явними (тобто за допомогою операції зведення типів). Однак варто враховувати, що перетворення одного типу покажчика до іншого може викликати непередбачувану поведінку програми.

Приклад 3.41. Некоректне перетворення:

```
double d=10.5;
int *ip;
ip=(int*) &d; /*синтаксично вірне перетворення, але
                призводить до втрати інформації*/ ■
```

Розмір покажчиків часто не збігається з розмірами типів **int** і **long**. У С99 тип **intptr_t** є цілочисловим і достатнім для збереження покажчика на об'єкт. Покажчики на функції й дані можуть мати суттєво різні подання й розміри.

Адресна арифметика. Над покажчиками можна виконувати унарні операції збільшення ++ і зменшення -- у префіксному й постфіксному варіантах. При цьому значення покажчика після цих операцій збільшується або зменшується на довжину об'єкта, зв'язаного з покажчиком. Нехай `Val(e)` – значення виразу `e`. Тоді для покажчика `p` типу `T`: `Val(p++)=Val(p--)=Val(p)`, `Val(++p)=Val(p)+sizeof(T)`, `Val(--p)=Val(--p)= Val(p)-sizeof(T)`.

У бінарних операціях додавання й віднімання можуть брати участь покажчик і ціле. При цьому результатом операції буде знов покажчик того самого типу: $p \pm i = p \pm (i * \texttt{sizeof(T)})$.

В операції віднімання можуть брати участь два покажчики одного типу. Результат такої операції має цілий тип і дорівнює кількості об'єктів між зменшуваним і від'ємником. Результат буде від'ємним, якщо перша адреса менша ніж друга.

Два покажчики одного типу можна порівнювати в операціях **==**, **!=**, **<**, **<=**, **>**, **>=**. Значення покажчиків розглядаються як цілі числа.

Приклад 3.42. Адресна арифметика:

```
int i=3;
int *ptr, *ptr1, *ptr2, v[20];

ptr1=ptr=&v[4];       /*=&v[4]*/
Зн(ptr++)             /*=&v[4]*/
Зн(ptr--)             /*=&v[4]*/
Зн(--ptr)             /*=&v[3]*/
ptr2=ptr1+(i+4);      /*=&v[11]*/
ptr1=ptr2-i;          /*=&v[8]*/
i=ptr1-ptr2;          /*=-3*/
i=ptr2-ptr1;          /*=3*/
(prt1>=ptr2)?1:2      /*=2*/
```

■

3.5.3. ІНДЕКСАЦІЯ ПОКАЖЧИКІВ І МАСИВИ

Операція *індексації* покажчику **p** типу **T** і цілому **i** ставить у відповідність змінну з індексом $p[i] = *(p + i)$. Якщо пригадати, що ім'я масиву є покажчиком (посилається на перший його компонент з індексом 0), то стане зрозумілим, що змінні з індексом є не чим іншим, як результатом індексації імені масиву відповідними індексами.

Отже, існують два методи звернення до компонентів масиву: змінні з індексами та з використанням адресної арифметики. Останній є

більш універсальним і саме він часто використовується у функціях обробки символьних масивів-рядків.

У наступному прикладі наведені різні версії стандартної функції **strlen** для підрахунку довжини рядка з використанням адресної арифметики та індексації масиву.

Приклад 3.43. Функція **strlen**:

```
//strlen: версія 1
int strlen (char *s)
{int i;
for (i=0; *s!='\0'; s++) i++;
return i;
}

//strlen: версія 2
int strlen (char *s)
{int i;
for (i=0; *s; s++) i++; /*вираз *s трактується як булевий*/
return i;
}

//strlen: версія 3
int strlen (char s[])
{int i;
for (i=0; s[i]!='\0'; i++);
return i;
}
```

■

У табл. 3.12 наведені деякі стандартні функції для роботи з рядками. Їхні прототипи містяться в бібліотеці **<string.h>**.

Таблиця 3.12. **Стандартні функції для роботи з рядками**

Функція	Прототип	Семантика
strcpy	**char *strcpy(char *s1,const char *s2);**	Копіювання **s2** у **s1**
strcat	**char *strcat(char *s1,const char *s2);**	Конкатенація **s1** та **s2**
strlen(s1)	**long strlen(char *s1);**	Повертає довжину рядка **s1**
strcmp(s1,s2)	**int strcmp(const char *s1,const char *s2);**	Повертає 0, якщо **s1** = **s2**, від'ємне значення, якщо **s1** < **s2**, і додатне, якщо **s1** > **s2**

Функція	Прототип	Семантика
`strchr(s1,ch)`	`char *strchr(const char *s1, int c);`	Повертає покажчик на перше входження символу **c** у рядок **s1**
`strstr(s1,s2)`	`int strstr(const char s1,const char *s2);`	Повертає покажчик на перше входження рядка **s2** у рядок **s1**
`strtok(s1,s2)`	`char* strtok(char s1, const char *s2);`	Повертає покажчик на першу лексему в рядку **s1**, обмежену символами рядка **s2**. Щоб знайти другу й наступні лексеми в рядку **s1**, необхідно викликати `strtok(Null,s2);`

Така програма ілюструє роботу наведених функцій:

```
int main(void)
{
    char word[80],word_1[80];
    gets(word);
    gets(word_1);
    printf("Length: %d %d\n", strlen(word),strlen(word_1));
    if(!strcmp(word,word_1)) printf("Strings is the same\n");
    strcat(word,word_1);
    printf("%s\n",word);
    strcpy(word,"Checking.\n");
    printf(word);
    if(strchr("Hello", 'h')) printf("Hello contain h\n");
    if(strstr("Hello", 'he')) printf("Hello contain he");
    printf(strtok("Hello contain\the"," \n\t");/*виведе Hello*/
printf(strtok(null, " \n\t"); /*виведе contain*/
printf(strtok(null, " \n\t"); /*виведе he*/

    return 0;
}
```

3.5.4. МАСИВИ ПОКАЖЧИКІВ

Як і об'єкти будь-яких інших типів, покажчики можуть бути зібрані в масив. Нижченаведені оператори описують масиви з покажчиками на об'єкти типу `int` та `float`:

```
int var;       /*звичайна ціла змінна*/
int array[10]; /*звичайний масив*/
```

```
int *v[20];    /*масив покажчиків*/
int *pd[]={&array[0],&array[2],&array[4]}/*ініціалізований
           масив покажчиків*/
v[2]=&var;  /*присвоює адресу цілої змінної var третьому
              елементу масиву v*/
float *w[20];  /*масив дійсних покажчиків*/
```

При передаванні масиву покажчиків у функцію передаються його ім'я й довжина. Наприклад, щоб вивести вектор дійсних чисел, який подається масивом покажчиків, можна застосувати функцію

```
void printArray(int n, float *q[])/*q – масив довжиною n дійс-
них покажчиків*/
{
for(int i=0; i<n; i++)
        printf("%f", *q[i]);
}
```

Виклик `printArray(20, w);` виведе значення, що подає масив покажчиків `w`.

Є задачі, в яких необхідно мати справу відразу з кількома різними порядками елементів масиву (напр., задачі пошуку інформації в таблицях тощо), а зберігати одночасно кілька впорядкованих по-різному копій масиву обтяжливо або взагалі неможливо. Універсальним виходом із такої ситуації є зберігання не впорядкованих копій масиву, а тільки масивів покажчиків (індексів) елементів, які відповідають розташуванню елементів масиву в тому чи іншому порядку.

Приклад 3.44. Одночасне сортування числового масиву за зростанням і спаданням:

```
#define n 6
int main()
{
float array[n]={5.0, 2.0, 1.0, 3.0, 6.0, 4.0};
float *pmin[n], /*масив pmin[n]: буде задавати порядок у масиві
array за зростанням*/
*pmax[n], /*масив pmax[n]: буде задавати порядок у масиві array
за спаданням*/
*temp;
    int i, j;
    for (i=0; i<n; i++) pmin[i]=pmax[i]=&array[i];
    for (i=0; i<n-1; i++)
          for (j=i+1; j<n; j++)
          {
                if (*pmin[i]<*pmin[j])
```

```
                {
                        temp=pmin[i];
                        pmin[i]=pmin[j];
                        pmin[j]=temp;
                }
                if (*pmax[i]>*pmax[j])
                {
                        temp=pmax[i];
                        pmax[i]=pmax[j];
                        pmax[j]=temp;
                }
        }
        printf("\n за спаданням: \n");
           printArray(n, pmin);
        printf("\n за зростанням: \n");
        printArray (n, pmax);

return 0;
}
```

■

Масиви покажчиків використовуються при роботі з рядками. Останні зазвичай ініціалізуються літералами – константами типу символьних покажчиків. Наприклад, така функція виводить повідомлення про помилку з номером **num**:

```
void syntaxError(int num)
{
static char *err[]={
"Не можна відкрити файл\n",
"Помилка при читанні\n",
"Помилка при записі\n",
};
printf("%s", err[num]);
}
```

3.5.5. ФУНКЦІЇ ВИДІЛЕННЯ ПАМ'ЯТІ

Прототипи функцій для роботи з пам'яттю містяться в бібліотеці **<stdlib.h>**. Для виділення пам'яті служать функції **void *malloc(size_t n)**, **void *calloc(size_t n, size_t m)** та **void *realloc(void* ptr, size_t n)**, де **size_t** – цілий тип, розмір якого залежить від компілятора. Перша виділяє в купі поле пам'яті, достатнє для розміщення об'єкта довжиною **n** байтів, і повертає його безти-

пову адресу, яка може бути явно перетворена до будь-якого потрібного типу з урахуванням вирівнювання. Якщо в купі недостатньо місця, то функція повертає значення **NULL**. Якщо параметр $n = 0$, то функція повертає значення **NULL** або інший безтиповий покажчик.

Друга функція **calloc** виділяє в купі поле пам'яті, достатнє для розміщення масиву довжиною **n** із компонентами довжиною **m** байтів, і повертає його безтипову адресу, яка може бути явно перетворена до типу компонент з урахуванням вирівнювання. При виділенні пам'яті її комірки обнулюються. Якщо в купі недостатньо місця, то функція повертає **NULL**.

Третя – **realloc** – змінює розмір поля, виділеного раніше однією зі стандартних функцій зі збереженням даних. За необхідності дані копіюються в нову область пам'яті. Якщо операцію здійснити неможливо, то функція повертає **NULL**, а пам'ять залишається без змін. Для **ptr=NULL** функція працює як **malloc(0)**. Виклик **realloc(ptr,0)** повертає безтиповий покажчик і звільняє заняту пам'ять. Якщо обсяг нової пам'яті менший ніж попередньої, то зайві байти втрачаються.

Функція **void free(char *ptr)** звільняє область купи, розподілену раніше функціями **malloc** та **realloc**. Аргументом її є покажчик, повернутий однією із цих функцій. Виклик **free(NULL)** не здійснює жодних дій.

Функція **void cfree(char *ptr)** робить те саме, але для покажчиків, повернутих функцією **calloc**.

3.5.6. МОДЕЛЮВАННЯ ДИНАМІЧНИХ МАСИВІВ

У підрозд. 3.4.2 динамічні масиви були визначені як масиви зі змінною довжиною. Пам'ять під такі масиви виділяється не під час компіляції, а в процесі виконання програми, коли з'являється конкретна довжина масиву.

Завдяки операції індексування динамічні масиви легко можна промоделювати неявно. Для цього за допомогою функцій **calloc** або **malloc** виділяється поле в купі з необхідною кількістю елементів даного типу **T** і присвоюється покажчику типу **T**, який потім шляхом індексування забезпечує прямий доступ до відповідних ділянок динамічного масиву [31]. Це також відкриває шлях до використання надвеликих масивів, розмір яких може становити кілька сегментів ОП. Справа в тім, що вся пам'ять, виділена автоматичним даним під час ви-

[31] Іноді кажуть, що ці функції породжують масив елементів у динамічній пам'яті.

клику функції, обмежена зазвичай одним сегментом, розмір якого залежить від реалізації мови (часто це $2^{16} = 65536$ байтів).

Динамічний одновимірний масив **a[n]** із **n** компонентами типу **float** можна промоделювати таким чином:

```
int n;
float *a;
scanf("%d", &n);
a=(float*)(calloc(n,sizeof(float));
```

Для $n = 30000$ та **sizeof(float)=4** розмір масиву **a** становитиме $30000 \times 4 = 120000$ байтів.

Наступні два приклади містять зразки обробки неявних динамічних масивів.

Приклад 3.45. Функція **in_out** вводить динамічний масив **v**, а потім виводить його значення на екран у зворотному порядку:

```
double * in_out(float *v, unsigned long n)
{
    for (int i=0; i<n; i++)
    {     printf("v[%d]=", i);
          scanf("%f", &v[i]);
    }
    for (i=n-1; i>=0; i--)
          printf("\t v[%d]=%f", i, v[i]);
    return v;
}
```

Функція **Re_allocArray** збільшує граничний розмір динамічного масиву **v** і додає до нього новий елемент:

```
#defne ARR_INCREMENT 128 /*кількість компонентів, на яку збільшується масив v*/

int Re_allocArray (float *v, double new_el, unsigned long *lim,
unsigned long *count)
/* *lim – гранична кількість компонентів динамічного масиву*/
/* *count – поточна кількість компонентів динамічного масиву*/

{if (*count<*lim) v[*count++]=new_el;
     else {
     int new_lim=*lim+ ARR_INCREMENT;
     if (double *new_arr=realloc(v, new_lim*sizeof(double)))==NULL)
            {printf("No memory");}
```

```
        else {
                v=new_arr;
                *lim=new_lim;
                v[*count++]=new_el;
              }
          }
        return *count;
}
```

Виклик функції **Re_allocArray (a, 3.1428, &n, &n)** додасть до масиву a число 3.1428 ■

Для створення двовимірного динамічного масиву спочатку потрібно виділити пам'ять для масиву покажчиків на одновимірні масиви, а потім – під самі масиви.

Приклад 3.46. Створення двовимірного динамічного масиву **v[n][m]:**

```
{   double **v;
    int n, m;
    scanf("%d %d", &n, &m);
    v=(double**)calloc(m, sizeof(double*));
    for (i=0; i<n; i++)
          v[i]=(double*)calloc(m, sizeof(double));
…
}■
```

Якщо виникає необхідність працювати з надвеликими одновимірними масивами з кількістю елементів більше **sizeof(size_t)**, то для їх моделювання застосовують саме двовимірні динамічні масиви.

***Література для СР:** покажчики – [55, 102, 132, 133]; надвеликі масиви – [55, 102, 122].

Контрольні запитання та вправи

1. Що таке покажчик і тип покажчиків?
2. Які бувають покажчики?
3. Навести визначення операції взяття адреси та розіменування покажчика.
4. Що таке константа **NULL**?
5. Як зводяться покажчики?
6. Що таке індексація покажчика?
7. Який існує зв'язок між масивами й покажчиками?
8. Що таке збільшення та зменшення покажчиків?
9. Сформулювати операції адресної арифметики.
10. Як описується масив покажчиків?

11. Що таке динамічний масив?
12. Як описується динамічний масив?
13. Дати визначення функцій динамічного виділення пам'яті.
14. Як моделюються динамічні масиви за допомогою покажчиків?
15. Що таке надвеликий масив?
16. Як моделюються надвеликі масиви?
17. Якого значення набуде змінна **p** після виконання оператора **{int p=3, *a=&p, q=2; *a*=p-=q+=2;}**?
18. Написати варіанти бібліотечних (**<string.h>**) функцій **strlen**, **strcpy**, **strcat**, **strncpy**, **strncat**, **ctrncmp**, використовуючи: а) індексацію; б) розіменування покажчиків.
19. Написати функції **int atoi(char *p)**, **itoa(int n**, **char *p)** та **long htol(char *p)**, які перетворюють, відповідно: а) послідовність десяткових цифр зі знаком або без нього на ціле типу **int**; б) ціле типу **int** на послідовність символів – знака й значущих десяткових цифр; в) послідовність шістнадцяткових цифр із префіксом **0x** або **0X** (якому може передувати знак) на ціле типу **long**.
20. Написати функцію **int getint(int *pn)**, що здійснює введення з клавіатури у вільному форматі цілого числа й подання його цілим типу **int**. Функція повертає прочитане ціле через свій параметр. Вона має сигналізувати про успішність (неуспішність) введення числа, а також про досягнення кінця вхідного потоку.
21. Написати функцію **int getdouble(double *pa)** – аналог функції **getint** – для введення дійсних чисел.
22. Що виведуть фрагменти програм:

```
char *ch[]={"12345", "ABCDE", "67800", "3401"};
```

а)

```
main()
{char **chp[]={ch+2,ch,ch+1};
char ***chpp=chp;

printf ("%s", **chpp);
printf ("%s", **++chpp+2);
}
```

б)

```
main()
{char **chp[]={ch+2,ch,ch+1,ch+3};
char ***chpp=chp;
```

```
printf ("%s", **chpp++);
printf ("%s", ***++chpp+3);
}
?
```

23. Написати програму калькулятору для обчислення арифметичних виразів, поданих в ОПЗ: а) для цілих виразів (використати функцію **getint** із вправи 20); б) для дійсних виразів (використати функцію **getdouble** із вправи 21). Додати в калькулятор стандартні арифметичні функції й можливість запам'ятовувати в однолітерних змінних проміжні результати обчислень і використовувати їх у подальших обчисленнях. Вирази вводяться з клавіатури.
24. Лексемою називається: а) послідовність символів, розташована між порожніми символами тексту (табуляція, пробіл, кінець рядка, кінець файла); б) слово латинського алфавіту; в) ідентифікатор; г) літерал; д) десяткове натуральне число; е) двійкове число; є) ціле число; ж) дійсне число; з) вісімкове число; и) шістнадцяткове число. Написати функцію: а) **int token (char *s, char *t)**, що здійснює пошук першої лексеми в рядку **s** і повертає її в рядку **t**. Функція повертає **n>0**, якщо лексему знайдено і **n** – її довжина; інакше – **0**, б) те саме, що й у а), але скористатися стандартною функцією **strtok.**
25. За допомогою функції: а) **int token (char *s, char *t)** (вправа 25); б) **strtok** знайти всі ідентифікатори С-програми, що 1) вводиться з клавіатури; 2) розміщена в символьному рядку.
26. У прикл. 3.44 показано, як можна реалізувати паралельне сортування числового масиву за кількома різними порядками. Реалізувати бінарний пошук елемента в масиві, порядок елементів у якому задається окремим масивом: а) індексів; б) покажчиків.
27. Написати варіанти функцій із передаванням результатів через параметри-покажчики для: а) обчислення площі й периметра трикутника за трьома його сторонами; б) відшукання за цілим $n > 7$ цілих a, b таких, що $5a + 3b = n$ та $a + b$ є мінімальним.
28. Керування купою. Написати функцію **char *alloc(int n)** для виділення поля вільної пам'яті (купи) заданого розміру. Результатом роботи процедури має бути **NULL**, якщо поле такого розміру не можна виділити. Написати функцію **void afree(char *p)** для звільнення пам'яті (повернення у вільну зону виділеного раніше поля). Купа задається статичним зовнішнім масивом з елементами типу

char s і розглядається як стек, вершина якого є початком вільної частини купи, а елементи нижче становлять задіяну зону купи. Таким чином, і виділення пам'яті, і її звільнення жорстко прив'язані до вершини стека, а вказані функції реалізують стратегію "останнім прийшов – першим пішов". Вершину стека задати статичним зовнішнім покажчиком, що рухається в межах стека.

29. Дано надвеликий одновимірний символьний масив розміром 250 Мбайт. Знайти частоту входжень у ньому кожного символу кодової таблиці ASCII.

3.6. Функції

- Специфікатори функцій
- Виклики функцій
- Рекурсивні функції
- Покажчики як аргументи й результат роботи функцій
- Функція **main** із параметрами
- Функції й покажчики функцій як параметри функцій
- Функції зі змінною кількістю параметрів
- Вбудовані функції
- Стандартна бібліотека*

Ключові слова: *специфікатор функції, специфікатори імені функції, виклик функції, рекурсивна функція, пряма рекурсія, непряма рекурсія, глибина рекурсії, покажчик функції, масив покажчиків функцій, параметри функції* **main,** *покажчики функцій як параметри, функція зі змінною кількістю параметрів, стандартна бібліотека, функція швидкого сортування масиву* **qsort.**

Поняття функції як основної структурної одиниці С-програм уже розглядалось у підрозд. 3.1. Там само зазначалось, що воно є важливим засобом декомпозиції програм. У належно вибраних функціях завдяки їхній блочній структурі вдається приховати несуттєві для інших підпрограм деталі, зробити їх зрозумілішими, полегшити подальшу модифікацію програм. Одного разу розроблені функції можна повторно використовувати в інших програмах.

Функції, як і масиви, мають подвійну семантику – абстрактну й реалізаційну. З одного боку, ім'я функції позначає певний алгоритм обробки даних, а з іншого – воно є покажчиком у системі спеціальних типів функціональних покажчиків. Кожен із таких типів визначається типами параметрів і значень функцій. Покажчик конкретної функції задається в заголовку функції ідентифікатором і є константою зі значенням адреси входу в машинний код даної функції. З такими покажчиками можна працювати так само, як і зі звичайними даними-константами: їх можна присвоювати, організовувати у вигляді масивів, передавати як параметри, повертати як значення функцій тощо. Подібні покажчики, як і покажчики-масиви, не можуть бути змінені в програмі, оскільки рівень доступу до них, як і до будь-якої іншої константи, – лише для читання. Кожен із покажчиків функцій, у тому числі й стандартних, має бути явно описаний у програмі. Для декларації стандартних функцій до програми підключають заголовні файли з їхніми прототипами.

У програмі можна визначати функціональні покажчики-змінні. Таким змінним можна присвоювати значення інших покажчиків функцій даного типу, викликати функцію за даним покажчиком тощо.

3.6.1. СПЕЦИФІКАТОРИ ФУНКЦІЙ

Загальний вигляд прототипу й опису функції, а також семантика оператора їхнього виклику розглядалися в підрозд. 3.1.3, 3.3.1. Зупинимось тепер детальніше на специфікаторах функцій і покажчиків на них.

У С99 поряд із сучасними формами для узгодження з першими версіями С збережено й традиційні специфікатори функцій, в яких типи параметрів декларуються за межами заголовка функції. Наприклад:

```
int hash (char* s){…} /*сучасна форма*/
int hash (s) char* s; {…} /*традиційна форма*/
```

Традиційна форма вважається застарілою (у мові С++ вона взагалі не використовується), тому ми на ній зупинятися не будемо.

Специфікатор функцій має вигляд:

```
<специфікатор-функції>::=<прямий-специфікатор> \
        \([<список-параметрів>] | <список-параметрів>, …\)
<прямий-специфікатор>::=<простий-специфікатор>|
\(<специфікатор-імені>\)|
<специфікатор-функції>|
<специфікатор-масиву>
```

```
<список-параметрів>::=<параметр>{,<параметр>} | <void> | <порожньо>
<параметр>::=[<кваліфікатор_типу>]<тип> (<специфікатор-імені> |
<абстрактний-специфікатор>)
<порожньо>::=ε
```
[32]

У С99 параметри обов'язково декларуються з типом[33]. Тип параметра може бути будь-яким, за винятком **void**. При цьому параметри функціонального типу **T** і параметри-масиви типу **T** будуть замінюватися компілятором на покажчики типу **T**. Існує три різновиди списку параметрів: а) порожній або, що те саме, одноелементний **void**; б) із фіксованою кількістю параметрів; в) зі змінною кількістю параметрів. У випадках б) і в) параметр може бути з іменем або без такого. Абстрактний специфікатор – це специфікатор імені з вилученим елементом, який задає безпосередньо ім'я. Наприклад, параметрам з іменами **int n** та **int p[]** відповідають параметри з абстрактними специфікаторами відповідно **int** та **int []**. У списку зі змінною кількістю параметрів спочатку розташовується його фіксована частина, а неоголошені параметри замінюються трьома крапками (див. підрозд. 3.6.7).

Приклад 3.47. Прототипи функцій і описів функціональних покажчиків:

Специфікатор	Тип *f*
`char f1 ();`	Символьна функція без параметрів
`void f2 (void);`	Функція без параметрів, що не повертає значення
`void f3 (double, int*);`	Функція з двома параметрами – дійсним і цілим покажчиками, яка не повертає значення. Специфікатори абстрактні
`void f4 (double d, int *p);`	Та сама функція, але специфікатори – з іменем
`int *f5 ();`	Функція без параметрів зі значеннями – цілими покажчиками
`int (*f6)();`	Покажчик цілих функцій без параметрів
`int f7 (int n,double x[]);`	Ціла функція з двома параметрами – цілим і дійсним масивами
`int f8 (int n,double *x);`	Ціла функція з двома параметрами – цілим і дійсним покажчиками
`int (*const f9)(void);`	Константний покажчик цілих функцій без параметрів

[32] ε – порожнє слово.

[33] У попередніх версіях параметр без типу вважався цілим.

Специфікатор	Тип *f*
`int f10 (int n, int (*g)(void));`	Ціла функція з двома параметрами – цілим і покажчиком цілих функцій без параметрів
`int f11 (int n, int g(void));`	Ціла функція з двома параметрами – цілим і цілою функцією без параметрів
`int (*f12 (void))(void);`	Функція без параметрів зі значеннями-покажчиками цілих функцій без параметрів

■

Специфікатори імені задають ім'я даного й уточнюють його тип. Вони можуть мати досить складну структуру (компонуватися з інших специфікаторів). Особливо це стосується описів функцій і функціональних покажчиків, в яких необхідно специфікувати типи значень і параметрів функцій. У таких випадках внутрішні специфікатори беруться зазвичай у дужки, щоб підкреслити всю структуру. Для спрощення подібних записів серед виразів-специфікаторів передбачені пріоритети, які дозволяють у деяких випадках не писати зайві дужки. Специфікатори функцій і масивів мають вищий пріоритет, ніж специфікатори покажчиків. Наприклад, конструкція `T*f()` еквівалентна виразу `T*(f())`, де `f` – функція без параметрів, що повертає покажчик типу `T`, а не виразу `T(*f )()`, в якому `f` – покажчик функції з порожнім списком параметрів і типом значень `T`.

Спростити складні випадки специфікаторів, зробити їхню структуру прозорішою дозволяє застосування синонімів для типів внутрішніх специфікаторів.

Приклад 3.48. Пріоритети і спрощення специфікаторів:

Опис	Значення
`T*f ();`	Еквівалентне `T*(f());` `f` – функція без параметрів зі значеннями – покажчиками типу `T`
`T*f [5];`	Еквівалентне `T*(f[5]);` `f` – масив покажчиків типу `T`* довжиною 5
`int (*f [])(int *,…);`	Еквівалентне `int (*(f[ ])) (int *,…);` `f` – масив покажчиків цілих функцій зі змінним списком параметрів
`int (*(*(*f))())[5])();`	`f` – покажчик функцій без параметрів зі значенням – покажчиком на п'ятиелементний масив покажчиків цілих функцій без параметрів

Опис	Значення
`typedef int (* func_ptr)();`	**func_ptr** – тип покажчиків цілих функцій без параметрів
`typedef func_ptr (* mass)[5];`	**mass** – тип покажчиків на п'ятиелементні масиви типу **func_ptr**
`mass (*f)();`	Еквівалентне вищенаведеному оператору опису **f**

■

3.6.2. ВИКЛИКИ ФУНКЦІЙ

Як ми вже знаємо, для обчислення значення функції на конкретних аргументах її необхідно викликати. Виклик функції полягає в передаванні їй фактичних параметрів і виконанні модифікованого тіла. Синтаксично він може бути оформлений у вигляді виразу-виклику або оператора виклику функції:

```
<виклик-функції>::=<вираз-виклик>|<вираз-виклик>;
<вираз-виклик>::=(<константний-покажчик-функції> |\
|<покажчик-функції>|(*<покажчик-функції>\)) \
([<список-фактичних-параметрів>]\)
<список-фактичних-параметрів>::=<фактичний-параметр>
{,<фактичний-параметр>
<фактичний-параметр>::=<вираз>
```

Оскільки в процесі передавання фактичних параметрів їхні значення зводяться до типу формальних, то типи формальних і фактичних параметрів мають бути узгодженими (див. підрозд. 3.2.4).

Приклад 3.49. Виклики функцій:

Специфікатор	Виклик
`int * ip;`	
`int nn;`	
`double dd;`	
`double xx[25]);`	
`extern int func(void);`	
`int (*ff (void))(void);`	
`char f1 ();`	`nn=f1();`
`char f2 (void);`	`nn=f2();`
`void f3 (double, int *);`	`f3 (1.0, &nn);`
`void f4 (double, int*p);`	`f4 (1.0, &nn);`
`int *f5 ();`	`ip=f5 ();`
`int (*f6)();`	`nn=f6 (); nn=(*f6) ();`
`int f7 (int n,double x[]);`	`nn=f7 (25,xx);`
`int f8 (int n,double*x);`	`nn=f8 (25,xx);`
`int (* const f9)(void);`	`nn=f9 ();`

`int f10 (int n, int (*g)(void));`
`int f11(int n, int g(void));`
`int (*f12(void))(void);`
	`nn=f10 (25, func);`
	`nn=f11 (25, func);`
	`ff=f12 ();`

■

3.6.3. РЕКУРСИВНІ ФУНКЦІЇ

У мові програмування С тіло функції може містити виклики інших функцій, що задекларовані вище в тексті програми. Це означає, що функція може містити виклики й себе самої, оскільки її заголовок є її прототипом і передує тілу функції – такі виклики називаються *рекурсивними*. Функція з рекурсивними викликами називається *рекурсивною.* Рекурсивні функції відкривають шлях до реалізації індуктивних означень функцій та їхніх систем (див. підрозд. 2.6).

Рекурсія, при якій функція викликає сама себе, називається *прямою*. Рекурсивними також будуть дві функції, що викликають одна одну (*непряма рекурсія*). Непряма рекурсія виникає при реалізації систем індуктивно визначених функцій. Виклик рекурсивної функції виконується так само, як і виклик будь-якої іншої, нерекурсивної, тобто послідовно реалізуються всі три етапи: ініціалізація параметрів і локальних даних, виконання модифікованого тіла й вихід із функції.

Глибиною рекурсії виклику функції, як ми вже знаємо, називається максимальна кількість розпочатих і не закінчених рекурсивних викликів при виконанні даного виклику. Коли виконується виклик із глибиною рекурсії **n**, то при виконанні найглибше розташованого виклику в автоматичній пам'яті одночасно зберігається **n** екземплярів локальних даних і параметрів функції. Тому, якщо **n** буде занадто великим, пам'яті, що надається процесу виконання програми, може не вистачити. *Загальною кількістю* вкладених викликів, породжених викликом рекурсивної функції, називається кількість викликів, виконаних між початком і завершенням виклику. Від цієї кількості залежить часова складність програми $F_A(n)$.

Рекурсивні функції є природним засобом обробки індуктивно визначених структур даних. Покажемо, як програмуються деякі індуктивно визначені функції з підрозд. 2.6.2.

Приклад 3.50. Програмування індуктивно визначених функцій:

1)
```
/*Швидке піднесення до степеня n>=0*/
double spow1 (double a, unsigned int n)
{
if (n==0) return 1;
if (n%2) return a*spow1(a, n/2)*spow1(a, n/2);
```

```
 else return spow1(a, n/2)*spow1(a, n/2);
 }
double spow2 (double a, unsigned int n)
 {double p;
 p=spow2(a, n/2);
 if (n==0) return 1;
 if (n%2) return (a*y * y);
 else return (y*y);
 }
```

2) **/*Комутативне розбиття числа ІВФ$_{19}$ */**

```
 long q(long n, long m)
 {
 if (n==1 || m==1) return 1;
 return (m>=n?1+q(n,n-1):q(n-m,m)+q(n,m-1));
 }

 long f(long n)
 {return q(n,n);
 }
```

3) **/*розкладання натурального числа на прості множники ІВФ$_{17}$ */**

```
 void g(long x, long y)
 {
 if (x==1) printf("1");
 else if (x<y*y) printf("%dl", x);
 else if (x%y==0) {printf("%dl:", y); g(x/y, y);}
 else g(x,y+2);
 }
 long f(long x)
 {if (x%2==0) {printf("2:"); f(x/2);}
 else g(x,3);
 }
```

Наприклад, виклик **f(24)** виведе на екран:

```
2:2:2:3:1
```

4) Ханойські вежі (ІВФ$_{15}$). Результуюча послідовність пар операцій буде подана у стандартному вихідному потоці **stdout** у вигляді:

$< x_1, y_1 > < x_2, y_2 > \ldots$

```
 /*Ханойські вежі ІВФ15 */
void hanoi (short n, short x, short y)
 {
 if (n<1 || x<1 || y<1 || x>3 || y>3)
```

```
{
printf("Помилка в даних"); return;
}
if (n==1) printf("<%d,% d>",x,y);
else {
hanoi (n-1, x,6-x-y);
printf("<%d,% d>", x,y);
hanoi (n-1, 6-x-y, y);
}
}
```

■

Складніші приклади програмування й застосувань рекурсивних функцій можна знайти в розд. 4. Нерекурсивний варіант швидкого піднесення до степеня див. у прикл. 3.31.

3.6.4. ПОКАЖЧИКИ ЯК АРГУМЕНТИ Й РЕЗУЛЬТАТ РОБОТИ ФУНКЦІЙ

Функції можуть повертати покажчики, можна передавати покажчики як аргументи функцій. Якщо як фактичний параметр передати функції змінну, то функція отримає доступ до її копії, а не до самої змінної. Тому всі маніпуляції з копією в тілі функції не будуть мати жодного впливу на саму змінну – фактичний параметр. Що ж робити, якщо треба змінити змінну, а не її копію? Одним з варіантів є використання замість змінних покажчиків, що задають їхні адреси, як аргументів функції. Наприклад, у програмі сортування методом бульбашки (див. прикл. 4.3) необхідно поміняти місцями два елементи, що порушують порядок серед компонентів масиву. Це можна зробити за допомогою такої функції:

```
void swap(int *pa, int *pb)
{
int temp=*pa;
    *pa=*pb;
    *pb=temp;
}
```

Тоді, якщо потрібно поміняти місцями, наприклад, сусідні елементи масиву `a` з індексами `i` та `i+1`, то достатньо викликати функцію `swap` і передати їй покажчики на ці елементи:

```
swap(&a[i], &a[i+1]); або swap(a+i, a+i+1);
```

Покажчики як аргументи використовуються зазвичай у функціях, які мають повертати більше одного значення. Можна сказати, що **swap** повертає два значення – нові значення змінних, на які посилаються аргументи.

Інша обставина, що спонукає до використання аргументів-покажчиків, суто економічна. У процесі побудови модифікованого тіла функції виділяється локальна пам'ять, куди копіюються значення аргументів, при цьому витрачаються такі важливі обчислювальні ресурси, як час роботи процесора та стек. Тому замість складних структур даних – структур, об'єднань, їхніх похідних – краще передавати у функцію покажчики на них. Тоді у стек при виклику будуть копіюватися лише адреси даних, а не самі дані, що набагато ефективніше. Наприклад, нехай задано структуру

```
struct date {
    int day;
int month;
unsigned year;
};
```

Можна написати таку функцію для відображення дати:

```
void printDate(struct date d)
{
    printf("%02d/%02d/%04d", d.day, d.month, d.year);
}
```

У процесі виклику функції буде виділено автоматичну пам'ять для збереження аргументу – структури **d**. Однак оптимальнішим був би такий варіант функції **printDate** з аргументом – покажчиком на структуру:

```
void printDate(struct date *pd)
{
    printf("%02d/%02d/%04d", pd->day, pd->month, pd->year);
}
```

Класичними функціями, що повертають покажчики, є функції динамічного розподілу пам'яті, які повертають адреси полів або константу **NULL** у випадку помилки – функції виділення пам'яті **malloc**, **calloc** і **realloc** (див. підрозд. 3.5.5).

Як зазначалося вище, функція не може повертати масив або функцію, але може повертати покажчик на будь-який тип, у тому числі – на масив або функцію. Розглянемо кілька подібних прикладів.

Нехай потрібно організувати вибірково виконання кількох варіантів дій, кожна з яких подана певною функцією й вибирається на підставі аналізу вхідної інформації. Подібне меню можна оформити у вигляді функції з порожнім списком параметрів, що, аналізуючи отриману зі вхідного потоку інформацію, повертає як результат адресу функції для відповідного варіанта дій.

Приклад 3.51. Покажчики функцій як результат функції.

Нехай є два варіанти дій, що задаються функціями **f1** та **f2** із прототипами

```
int f1(void);
int f2(void);
```

Функція для реалізації меню має прототип:

```
int (*menu(void))(void);
```

```
#include <stdio.h>

int f1(void);
int f2(void);

int (*menu(void))(void);

int main(void)
{
    int (*r) (void); /*опис покажчика на цілі функції з порожнім
списком параметрів, такими функціями є f1 та f2*/
    while (1)
    {
          r=menu(); /*виклик меню*/
          if (!r) {
                printf("\n The End");
                return 0;
          }
          printf("\t%d", r());
          return 0;
    }
}

int f1(void)
{
    return 1;
}
```

```
int f2(void)
{
    return 2;
}

/*результатом роботи функції menu є одна з функцій f1, f2 або
покажчик NULL*/
int (*menu(void))(void)
{
    int choice;
    printf("Pick the menu item (1 or 2), pls:");
    scanf("%d",&choice);
    switch (choice)
    {
    case 1: return f1;
    case 2: return f2;
    default: return NULL;
    }
}
```

■

Якщо варіантів вибіркових дій більше ніж два, то систему меню можна реалізувати за допомогою масиву покажчиків функцій.

Приклад 3.52. Масив покажчиків функцій:

```
#include <stdio.h>
int f1(void);
int f2(void);
int f3(void);

/*Ініціалізований масив покажчиків на функції*/
 int (*menu_items[3])(void)=
{
f1,
f2,
f3
}; /*ініціалізація масиву покажчиків на функції*/

int (*menu(void))(void);
int main(void)
{
    int (*r) (void); /*опис покажчика на цілі функції з порожнім
списком параметрів, такими функціями можуть бути f1, f2 та f3*/
    while (1)
    {
          r=menu(); /*виклик меню*/
```

```
            if (!r) {
                  printf("\n The End");
                  return 0;
            }
            printf("\t%d", r());
            return 0;
      }
}

int f1(void)
{
    return 1;
}

int f2(void)
{
    return 2;
}

int f3(void)
{return 3;
}


/*результатом роботи цієї функції є функція*/
int (*menu(void))(void)
{
    int choice;
    printf("Pick the menu item (1, 2 or 3), pls:");
    scanf("%d",&choice);
    if((choice<3)&&(choice>=0))
          return menu_items[choice];
    else return NULL;
}
```

■

3.6.5. ФУНКЦІЯ `main()` ІЗ ПАРАМЕТРАМИ

Функція **main** може мати параметри – символьні рядки, що формуються в командному рядку запуску програми. Для доступу до них у функції **main()** використовуються два параметри: перший – для означення кількості переданих рядків, другий – самих рядків. Загальноприйняті (але не обов'язкові) імена цих параметрів – **argc** та **argv**. Заголовок функції **main** із параметрами має вигляд

```
int main(int argc, char *argv[], [char * argp[]])
```

Параметр **argc** має тип **int**, його значення формується автоматично в процесі аналізу командного рядка й дорівнює кількості символьних рядків у ньому, починаючи з імені самої програми. Рядки розділяються пробілом. Якщо символьний рядок містить символ-пробіл, то його подають у вигляді літерала. Параметр **argv** – це масив символьних покажчиків на рядки, його довжина збігається з **argc**. Функція **main()** може мати й третій параметр **argp**, який служить для передавання їй параметрів ОС.

Приклад 3.53. Програма, що друкує значення фактичних параметрів функції **main()**:

```
#include <stdio.h>

int main(int argc, char *argv[], char * argp[])
/*
argc – кількість параметрів
argv – вектор параметрів-рядків
argp – вектор змінних середовища*/
{
    int i;
    char *p[];

    /*друкування значень параметрів*/
    for(i=0; i<argc; i++)
          printf("arg %i:%s\n", i, argv[i]);

    /*друкування значень змінних середовища*/
    for(p=argp; *p !=(char*)0; p++)
          printf("%s\n", *p);

    return 0;
}
```

■

3.6.6. ФУНКЦІЇ Й ПОКАЖЧИКИ ФУНКЦІЙ ЯК ПАРАМЕТРИ ФУНКЦІЙ

Функції й покажчики функцій можуть використовуватись як параметри інших функцій (функції **f10** та **f11** із прикл. 3.47). Розглянемо кілька прикладів детальніше.

Приклад 3.54. Обчислити наближене значення інтеграла $\int_{-1}^{1} e^{-x^2} dx$:

Лістинг.

```
  #include <math.h>
  #include <stdio.h>
  /*Прототип функції для обчислення інтеграла методом прямокутників*/
  /*Покажчик функцій f використовується як параметр функції*/
  double integral(double (*f)(double), double a, double b, int n);

  /*Опис фактичного параметра – підінтегральної функції-експоненти. Її тип має бути узгодженим із прототипом параметра f функції integral*/
   double f1(double x)
  {
     return exp(-x*x);
  }

  int main(void)
  {
  /*Покажчик f1 використовується як перший аргумент виклику функції integral*/
     printf("\n integral=%f", integral(f1,-1.0,1.0,100));
     return 0;
  }

  double integral(double (*f)(double), double a, double b, int n)
  {
     int i;
     double h, s;
     for (i=0, h=(b-a)/n, s=0; i<n; i++)
           s+=f(a+i*h);
     return s*h;
  }
  ■
```

Іншим прикладом функції з покажчиками в параметрах є бібліотечна функція швидкого сортування масиву, визначена в заголовку **<stdlib.h>**:

```
  void qsort(void *base, size_t nelem, size_t width, int (*fcmp)(const void *p1, const void *p2));
```

Розглянемо її параметри:

base – покажчик на початок таблиці (масиву) елементів, що сортуються (адреса нульового елемента масиву);

nelem – кількість елементів, що сортуються, у таблиці (ціла величина, не більша за розмір масиву) – сортуються перші **nelem** елементів від початку масиву;

width – розмір одного елемента таблиці (масиву) у байтах;

fcmp – покажчик на функцію порівняння, яка отримує як параметри два покажчики **p1**, **p2** на елементи таблиці й повертає залежно від результату порівняння ціле число:

якщо ***p1<*p2** – то від'ємне ціле;

якщо ***p1==*p2** – то **0**;

якщо ***p1>*p2** – то додатне ціле.

Алгоритм роботи цієї функції буде розглянуто в підрозд. 4.1.6.

Застосуємо функцію **qsort** для лексикографічного впорядкування масиву покажчиків на рядки. Зауважимо, що рядки в процесі сортування не міняють своїх положень, змінюються лише значення покажчиків у масиві.

Приклад 3.55. Лексикографічне впорядкування рядків:

```
#include <stdio.h>
#include <stdlib.h> /*для функції qsort()*/
#include <string.h> /*для порівняння рядків strcmp()*/

/*функція для порівняння*/
int comparePC(const void *a, const void *b);

/*друкування таблиці p розміром n*/
void printPC(int n, char *p[]);

int main(void)
{
 char *pc[]={
           "One",
           "Two",
           "Three",
           "Four",
           "Five",
           "Six",
           "Seven",
           "Eight",
           "Nine",
           "Ten"
 };

 /*обчислення розміру таблиці*/
 int n=sizeof(pc)/sizeof(pc[0]);
```

```
    printf("\n До сортування:");
    printPC(n, pc);

    /*виклик функції qsort()*/
    qsort((void *)pc, n, sizeof(pc[0]), comparePC /*покажчик на
функцію порівняння*/);
    printf("\n\n Після сортування:");
    printPC(n, pc);

    int i;
    scanf("%d", &i);
    return 0;
   }

   /*функція для порівняння*/
   int comparePC(const void *a, const void *b)
   {
      unsigned long *pa=(unsigned long *)a,
                        *pb=(unsigned long *)b;
      return strcmp((char *)*pa, (char *)*pb);
   }

   /*друкування таблиці p розміром n*/
   void printPC(int n, char *p[])
   {
      for(int i=0; i<n; i++)
   printf("\np[%d]: point: %p\tvalue: %s", i, p[i], p[i]);
   }
```

Результат виконання програми:

```
До сортування:
p[0]: point: 0042B0C8 value: One
p[1]: point: 0042B0D0 value: Two
p[2]: point: 0042B708 value: Three
p[3]: point: 0042B734 value: Four
p[4]: point: 0042B728 value: Five
p[5]: point: 0042B71C value: Six
p[6]: point: 0042B2E4 value: Seven
p[7]: point: 0042B48C value: Eight
p[8]: point: 0042B3A8 value: Nine
p[9]: point: 0042BB54 value: Ten

Після сортування:
p[0]: point: 0042B48C value: Eight
p[1]: point: 0042B728 value: Five
```

```
p[2]: point: 0042B734 value: Four
p[3]: point: 0042B3A8 value: Nine
p[4]: point: 0042B0C8 value: One
p[5]: point: 0042B2E4 value: Seven
p[6]: point: 0042B71C value: Six
p[7]: point: 0042BB54 value: Ten
p[8]: point: 0042B708 value: Three
p[9]: point: 0042B0D0 value: Two
■
```

3.6.7. ФУНКЦІЇ ЗІ ЗМІННОЮ КІЛЬКІСТЮ ПАРАМЕТРІВ

Прототип функції зі змінним списком параметрів має вигляд

```
T f(<список-явних-параметрів>, …);
```

Кількість (не менше одного) і тип явних параметрів фіксовані й відомі до компіляції. З погляду композиційної семантики функції зі змінною кількістю параметрів є еквітонними параметризованими X – арними функціями ($X-Y$ -операторами) (див. підрозд. 2.1.2).

Існує кілька механізмів для визначення кількості й типів неявних параметрів. Найпростіший – коли тип останнього явного параметра визначає єдиний тип решти з нефіксованої частини списку, а конкретна їхня кількість визначається за допомогою певного стопера (напр., вона передається явно або відоме унікальне значення останнього параметра).

Визначення типу неявних параметрів і перехід від одного параметра до іншого здійснюється за допомогою покажчиків.

Приклад 3.56. Змінні списки параметрів функцій:

1) **/*підрахунок суми параметрів функції*/**

```
long suma(int n,…)
{
    int *p=&n; /*адреса списку параметрів*/
    long total=0;
    for (; n;n--) total+=*(++p); /*++p – перехід до наступного
                                   параметра*/
    return total;
}
int main (void)
{
    printf("\n %ld", suma(2,4,6));
```

```
    return 0;
}
2)  /*підрахунок добутку параметрів функції, список яких закінчується 0.0*/

double prod (double arg,…)
{
    double prod1=1.0;
    double *p=&arg; /*адреса списку параметрів*/
    long total=0;
    if (*p==0) return 0.0;
    for (; *p; p++) prod1*=*p;
    return prod1;
}

int main (void)
{
    printf("\n %f", prod(1.1, 2.0, 3.3, 5.1, 0.0));
    return 0;
}
```

■

У загальному випадку для забезпечення прямого доступу до неявних параметрів функції зі змінною кількістю параметрів різного типу й мобільності програм використовують спеціальні макроси, задекларовані в заголовних файлах **<varargs.h>** і **<stdarg.h>**. Основні з них –

```
void va_start (va_list parg, last);
type va_arg(va_list parg, type);
void va_end (va_list parg);
```

– містяться у **<stdarg.h>**.

Порядок роботи з ними такий. У функції зі змінною кількістю параметрів описується спеціальний покажчик **ap** із типом **va_list**, який використовується для звертання до параметрів. Викликом макросу **va_start (ap, last);** де **last** – ім'я останнього явного параметра в заголовку функції, покажчик **ap** установлюється на перший неявний параметр. Для послідовного звертання до наступних неявних параметрів застосовується виклик макросу **va_arg(ap, type);**, що повертає значення поточного параметра типу **type**, при цьому **ap** автоматично пересувається до наступного параметра. Тип **type** має бути або одним зі стандартних (**int, double** тощо), або ім'ям типу, визначеним оператором **typedef**. Як і в попередньому випадку, необхідно передбачити спосіб обмеження кількості параметрів. Коли всі

параметри опрацьовані, перед виходом із функції викликають макрос **va_end (ap);** для обнулення змінної **ap** та інших змінних макросів.

Розглянемо приклад організації доступу до списку неявних параметрів певної функції **void f(char *s,…)** за допомогою спеціального форматного символьного рядка **s**, елементи якого відповідають першим латинським літерам у назві типу кожного з неявних параметрів. Аналогічно працюють і деякі стандартні функції – наприклад **printf ()** та **scanf()**.

Нехай типами неявних параметрів функції **f** є **_Bool, char,double** та **long**. Якщо список фактичних неявних параметрів у виклику **f** складається, наприклад, з п'яти елементів та їхні типи – **char, double, _Bool, long** та **long**, то такому списку відповідає форматний рядок **"cdBll"**:

```
#include <stdarg.h>
void f(const char *s,…)
{_Bool bx; char cx; double dx; long lx;
 va_list ap;    /*покажчик на поточний неявний параметр*/
 char *ss=s; /*покажчик для визначення типа параметра*/
 va_start (ap,s); /*ініціалізація ap*/

while(*ss){
   switch (*ss++){ /*обробка поточного параметра*/
       case 'B':bx= va_arg(ap, _Bool);…;break;
       case 'c':cx= va_arg(ap, char);… ;break;
       case 'd':dx= va_arg(ap, double);… ;break;
          case '1':lx= va_arg(ap, long);… ;break;
                    }
                }
va_end (ap);
}

int main()
{double d1=1/3,d2=2e-3;
  char c1='a';
   f("ddc" ,d1,d2,c1); /*обробка трьох параметрів*/
   f("lcdcd",0L,'0',d2,c1,5.0); /*обробка п'ятьох параметрів*/
   return 0;
}
```

3.6.8. ВБУДОВАНІ ФУНКЦІЇ

Вбудовані функції вперше з'явилися в C99. Ознакою їх є службове слово **inline** у заголовку функції. Саме це слово походить від оптимізаційного методу внутрішньорядкового кодування (англ. – inline expansion), що застосовується на етапі генерації коду, коли виклик функції замінюється

копією машинного коду її тіла. При цьому зникає необхідність у службових командах, які обслуговували б виклик функції. Коли виникає потреба в подібній оптимізації, то функцію визначають як вбудовану. Однак таке визначення не імперативне для компілятора, воно скоріше є запитом, побажанням, і може бути компілятором проігноровано. Основна вимога до вбудованої функції: її опис має бути видно в місці виклику. Будь-яка статична функція задовольняє цю вимогу, оскільки всі її виклики містяться в одному файлі з її описом. У загальному випадку виклик зовнішньої функції відбувається з файла, що не містить її опис, а тільки прототип із класифікатором **extern**. Тоді можна створити заголовний файл з описом даної вбудованої функції й підключити його до файла, де вона викликається. У самому ж файлі необхідно декларувати дану функцію як зовнішню і вбудовану. Розглянемо приклад.

Приклад 3.57. Вбудована функція **max(x,y)**:

```
/*Заголовний файл max.h*/
/*містить опис вбудованої функції max*/

inline max (double x, double y)
 {  return ((x<y)? y : x);
 }

/*Файл max.c містить виклики функції max із файла max.h*/
 #include "max.h"
/*Визначення вбудованої функції max як зовнішньої*/
extern inline max (double, double);
 …
 printf ("%f", max(a,b));
 ….
```

■

3.6.9. СТАНДАРТНА БІБЛІОТЕКА*

Бібліотека мови С містить велику кількість функцій, макросів, типів, констант. Вона може змінюватися від системи до системи, але ядро функції (стандартна бібліотека) реалізується в усіх системах. Стандартна бібліотека не є частиною мови С, але закладений у ній набір функцій, макросів, типів становить системне середовище, що підтримує стандарт С. Коротко оглянемо стандартні бібліотеки, затверджені в стандартах С89 та С99.

Стандартні функції, типи й макроси стандарту С89 декларуються в заголовних файлах (табл. 3.13).

Таблиця 3.13. **Заголовні файли**

<assert.h>	Визначає макрос **assert()**
<ctype.h>	Обробка символів
<errno.h>	Видача повідомлень про помилки
<float.h>	Задає межі значень із плаваючою точкою, що залежать від реалізації
<limits.h>	Задає різні обмеження, що залежать від реалізації
<locate.h>	Підтримує локалізацію
<math.h>	Різні визначення, що використовуються математичною бібліотекою
<setjmp.h>	Підтримує нелокальні переходи
<signal.h>	Підтримує обробку сигналів
<stdarg.h>	Підтримує списки вхідних параметрів функції зі змінною кількістю аргументів
<stddef.h>	Визначає деякі найчастіше використовувані константи
<stdio.h>	Підтримує систему введення-виведення
<stdlib.h>	Мішані оголошення
<string.h>	Підтримує функції обробки рядків
<time.h>	Підтримує функції, що звертаються до системного часу

У стандарт C99 було додано деякі заголовні файли (табл. 3.14).

Таблиця 3.14. **Нові заголовні файли C99**

<complex.h>	Підтримує арифметичні операції з комплексними числами
<fenv.h>	Надає доступ до прапорців стану обчислювача, що виконують операції з плаваючою точкою, а також до інших сигналів цього обчислювача
<inttypes.h>	Визначає стандартний набір імен цілочислових типів, підтримує функції, що працюють із цілими значеннями найбільшої розрядності
<iso646.h>	Доданий у 1995 р.; визначає макроси, що відповідають різним операторам, наприклад **&&** та ^
<stdbool.h>	Підтримує логічні типи даних; визначає макрос **bool**, що сприяє сумісності з мовою C++
<stdint.h>	Задає стандартний набір імен цілочислових типів; цей файл включений у заголовок **<inttypes.h>**
<tgmath.h>	Визначає макроси для родового (абстрактного) типу чисел із плаваючою точкою
<wchar.h>	Доданий у 1995 р.; підтримує функції обробки багатобайтних слів і двобайтних символів
<wctype.h>	Доданий у 1995 р.; підтримує функції класифікації багатобайтних слів і двобайтних символів

Основні функції введення-виведення (файл **<stdio.h>**), математичні функції (файл **<math.h>**), функції для роботи з рядками (файл **<string.h>**) і покажчиками (файл **<stdlib.h>**) розглядалися раніше.

***Література для СР:** С-функції [55, 102, 132, 133]; ознайомитися з іншими функціями, макросами, типами й константами бібліотеки мови С можна в [55, 102, 136].

Контрольні запитання та вправи

1. Визначити структуру специфікатора функції.
2. Визначити структуру специфікатора імені функції.
3. Який вигляд має виклик функції?
4. Що таке формальні й фактичні параметри функції?
5. Які функції називаються рекурсивними?
6. Що таке глибина рекурсії?
7. Чим небезпечні великі глибина рекурсії й загальна кількість рекурсивних викликів?
8. Що таке пряма й непряма рекурсія?
9. Як описується покажчик і масив покажчиків функцій?
10. Що таке параметри функції **main** і яка їхня роль у ній?
11. Як описуються покажчики функцій у якості параметрів?
12. Як відбувається ініціалізація параметрів функцій зі змінною кількістю параметрів?
13. Описати параметри функції швидкого сортування масиву **qsort**.
14. Який тип мають змінні **p** та **q**:

```
void (*p)(int x, int g(void));
int q (int *n, int (*g) (int, char));
char  **q(void);   int (*p)(int x);
char  **q(int x);  int (*p)(const int x);
int *p[] (int n,double *x int, (*q (void))(void) ?
```

15. Написати функцію, що за дійсними змінними **x** та **y** знаходить значення $z=\begin{cases} y+2, & 0<x\le 4 \\ x-y, & x\le 0 \\ x+y, & x>4 \end{cases}$.
16. Дано цілі числа $m,n,p,q\neq 0$. Знайти їхні найбільший спільний дільник $\text{НСД}(m,n,p,q)$ і найменше спільне кратне $\text{НСК}(m,n,p,q)$.
17. Напишіть рекурсивну функцію, що переставляє елементи рядка у зворотному порядку.

18. Написати рекурсивну функцію:
    а) для друкування десяткових цифр числа типу **int** у оберненому порядку;
    б) **int strcmp(char *x, char *y)**, яка порівнює два рядки;
    в) **int strlencmp(char *x, char *y)**, яка порівнює довжину двох рядків (повертає: 1, якщо |x|>|y|; 0, якщо |x|=|y|; –1, якщо |x|<|y|);
    г) **int strpre (char *x, char *y)**, яка перевіряє, чи є другий рядок префіксом першого;
    д) **int strch(char *x, char c)** для обчислення частоти входжень символу в рядок.
19. Елімінувати рекурсію у функції (прикл. 3.50), що знаходить: а) комутативне розбиття числа; б) розкладання числа на прості множники.
20*. Елімінувати рекурсію у функції: а) **hanoi** (прикл. 3.50) [1]; б) 91-й функції Мак-Картні (підрозд. 2.6.2). У б) обчислити значення функції для $n = \overline{1,200}$.
21. Побудувати трикутник Паскаля, що складається з коефіцієнтів бінома Ньютона $(a+b)^n$. Навести рекурсивний і нерекурсивний варіанти розв'язку [33].
22. Задача про ферзів, що домінують на дошці. Побудувати всі можливі розташування мінімальної кількості ферзів, що: а) атакують усі вільні поля дошки розміром $n \times n$; б) те саме, що й у а), але ферзі не атакують один одного.
23. Задача про коней (слонів), що домінують на дошці 8×8. Побудувати можливе розташування: а) 12 коней, що атакують усі вільні поля дошки; б) те саме, що й у а), але йдеться про 8 слонів.
24. Застосувати функцію **qsort** для впорядкування масиву рядків за їхньою довжиною.
25. Реалізувати бінарний пошук елемента в довільному впорядкованому масиві, порядок на елементах якого задає функція-предикат **Compare(x,y)**, а розташування елементів масиву згідно з даним порядком – окремий масив: а) індексів; б) покажчиків.
26. У командному рядку задано масив цілих чисел. Відсортувати його й реалізувати рекурсивний бінарний пошук елемента у відсортованому масиві. Нерекурсивний алгоритм бінарного пошуку див. у підрозд. 4.1.2.
27. Написати функцію **double scalar(double *a, double *b, int n)** для обчислення скалярного добутку двох векторів довільної довжини. Знайти скалярний добуток двох

векторів, елементи яких вводяться з клавіатури. Довжина векторів задається в командному рядку.

28. Написати варіант калькулятора із вправи 23 із підрозд. 3.5 для виразу, що задається в командному рядку. При цьому кожному операнду й символу операції відповідає окремий аргумент.
29. Дано п'ять функцій: **`double f1(int i)`**, …, **`double f5(int i)`**, що задають математичні функції: $f1(x)=\sin(x*x)$; $f2(x)=\cos(x)*\sin(x)$; $f3(x)=x*x/2$, $f4(x)=\ln(x*x/2)/x$, $f5(x)=\mathrm{tg}(\pi x/4)$. Реалізувати табуляцію цих функцій у таблиці, у шапці якої відобразити номер функції, інтервал $[a,b]$ і крок табуляції **`d`** (вводяться з клавіатури).
30. Описати функцію зі змінною кількістю цілих параметрів, що знаходить: а) НСД аргументів, б) НСК аргументів.
31. Описати функцію зі змінною кількістю різнотипних параметрів, що знаходить: а) середнє арифметичне аргументів; б) середнє геометричне аргументів.

3.7. Тип структур

- Специфікатори структур
- Операції на структурах
- Бітові поля
- Тип об'єднань

Ключові слова: *тип структур, структура, поле структури, специфікатор типу структур, тег структури, ініціалізатор структури, тип об'єднання, бітове поле, операції на структурах, операції стрілка та крапка, складена змінна.*

Тип структур складають вектори з поіменованими компонентами певних фіксованих типів. Дані типу структур називаються *структурами*[34], а компоненти структур – *полями*. На відміну від масивів, типи полів усередині структури можуть бути різними, але не функціональними (не типами функцій і покажчиків на них).

Структури використовують для збереження таблиць із різнотипною складноструктурованою інформацією й організації сукупностей по-

[34] У деяких мовах (Кобол, ПЛ/1, Паскаль тощо) структури називаються записами.

в'язаних між собою об'єктів. Для зв'язування в одне ціле таких об'єктів їх визначають у вигляді структур зі спеціальними полями-покажчиками, які й задають відповідні зв'язки.

Також допускаються структури з полями-покажчиками свого власного типу, що дозволяє моделювати в мовах програмування індуктивно визначені об'єкти, наприклад такі, як списки, дерева, графи тощо, робота з якими описується в підрозд. 4.3.

3.7.1. СПЕЦИФІКАТОРИ СТРУКТУР

Специфікатор типу структур має вигляд:

```
<специфікатор-типу-структур>::=struct [<ім'я-тегу-структури>] \
 {<список-полів>}|struct <ім'я-тегу-структури>
< ім'я-тегу-структури>::=<ідентифікатор>
<список-полів>::=<поле>{,<поле>}
<поле>::=<просте-поле> | <бітове-поле>
<просте-поле>::=<опис-поля>
```

У структурі повинне бути принаймні одне поле. Опис поля має структуру оператора опису змінних без специфікатора класу пам'яті та кваліфікаторів типу. Сукупність усіх полів структури складає її *тег*, який зазвичай іменують. Тег з іменем фактично є певним об'єктом, що асоціюється зі структурним типом. Імена полів у межах тегу унікальні, але вони не конфліктують з іменами змінних, функцій інших об'єктів поза структурою. Поля структури можуть мати будь-який негнучкий тип.

Приклад 3.58. Структура для подання дати в році:

```
struct date {
     int day;     /*поле, що задає день */
     int month;    /*поле, що задає місяць */
     int year;     /*поле, що задає рік */
      }  d;
struct date d2, Tab[10], *p; /*змінна-структура d2 типу date,
масив структур (таблиця) Tab типу date, покажчик p типу date*/
```

Змінні **d**, **d2** визначені як структури, кожна з яких складається із трьох полів – **day**, **month** та **year**, змінна **p** – покажчик на таку структура, а змінна **Tab** – масив із десяти таких структур. Змінні **d2**, **Tab** та **p** описані за допомогою тегу **date** ■

Як і у випадку масивів, декларація структур може задавати неповний тип. Наприклад, можна декларувати тільки ім'я тегу, а не сам

тег. Цього достатньо, щоб зробити невидимим у даному блоці інший тег, і необхідно для декларації двох тегів, які містять посилання один на одного. Коректними є такі визначення структур у блоці:

```
{struct first_node; /*неповний тип*/
 struct second_node {struct first_node *left; …};/*повний  тип*/
 struct first_node {struct second_node *right; …}; /*повний тип*/
}
```

Перша декларація, окрім того що декларує ім'я тегу структури, відміняє в блоці дію будь-яких раніше визначених тегів з іменем **first_node.**

Декларація неповного типу може існувати й усередині опису структури від першої появи нового тегу до того моменту, коли опис стає повним. Це дозволяє визначати структурні типи з полями-покажчиками на себе. Наприклад, для реалізації цілих лінійних списків використовують структури вигляду:

```
struct node {
int data;
struct node *next; /*node: неповний тип*/
};                              /*node: повний тип*/
```

Деталі відповідної реалізації можна знайти в підрозд. 4.3.1.

Компоненти структури розміщуються в загальному полі структури один за одним з урахуванням правил вирівнювання, що залежать від компілятора й архітектури процесора. Розриви між сусідніми полями, які можуть виникати внаслідок вирівнювання, не впливають на значення операції **sizeof**, що повертає довжину загального поля.

Змінні-структури можна ініціалізувати в процесі опису. *Ініціалізатор структури* має вигляд:

```
<ініціалізатор-структури>::={<ініціалізатор-поля> \
                                    {,<ініціалізатор-поля>}
                             }
```

Ініціалізатор поля має відповідати типу поля. Стандарт С99 дозволяє неконстантні ініціалізатори. Як і у випадку масивів, існують правила для скорочення запису ініціалізаторів. Зокрема, якщо ініціалізаторів полів менше ніж потрібно, то використовуються їхні значення за умовчанням. Якщо ж таких ініціалізаторів більше, то це викликає помилку в роботі компілятора.

Приклад 3.59. Описи змінних-структур з ініціалізацією:

```
struct date d={1, 9, 2008};
int n=9; struct date d={1, n}; /*еквівалентно опису struct date
                                  d={1, 9, 0}; */
struct date d={1, n, 2008, 0}; /*некоректний опис: константа 0
                                  зайва*/
struct data {char *row, *col; int matr [2][3]}s={"abc",NULL,
{{2,2,2},{3,3,3}}}; ■
```

3.7.2. ОПЕРАЦІЇ НА СТРУКТУРАХ

На однотипних структурах можна виконувати операції присвоювання, взяття адреси структури та її окремих полів і операції крапка '.' та стрілка '**->**', що забезпечують доступ до полів структури.

Операція крапка **.** за іменем структури **s** та її полем **v** будує *складену змінну* **s.v**, яка є *l*-виразом і зв'язана із цим полем. Складена змінна має тип свого поля й забезпечує всі можливості доступу, визначені для змінних даного типу. Складена змінна має тип поля й забезпечує всі можливості доступу до нього, визначені для даних цього типу.

Операція стрілка **->** за покажчиком **p** на структуру та її полем **v** будує *l*-вираз **p->v**, зв'язаний із цим полем у структурі, на яку посилається покажчик. Формально це можна записати так:

$$\mathbf{p->v} \overset{def}{=} \mathbf{(*p).v},$$

де **(*p)** є розіменуванням покажчика **p**.

Приклад 3.60. Для змінних, описаних у прикл. 3.58, 3.59, можливі такі вирази:

```
d.day=1;
d.month=9;
d2.day=d.day;
Tab [3].day=5;
(*s.row=='a')=1
s.matr[1][1]=0;
p->day=2;
*p=d;
p.month++; ■
```

За стандартом С99 тип полів структури не може бути гнучким, але певна структура **s** може містити недовизначений масив вигляду **T x[ ]** за умови, що цей масив є останнім полем у ній. Тоді звернення до змінних з індексами **s.x[0], s.x[1],** … синтаксично припустимі, але необхідно додатково забезпечити їхню семантичну коректність.

Приклад 3.61. Структура `Vector` для зображення динамічного масиву та його довжини:

```
struct Vector {int n; float mas[];} *p;
int m;/*реальна довжина масиву*/
…
/*виділення пам'яті для одного екземпляра структури Vector і
елементів масиву*/

p=malloc (sizeof(struct Vector)+m*sizeof(float));
p->n=m;
p->mas=(float *) p++; /*адреса динамічного масиву*/
…
/*збільшення компонентів динамічного масиву на 1*/
for (int i=0; i<(p->n);i++) p->mas[i]++;
…
■
```

Операції порівняння для однотипних структур як векторів не визначені, але їх можна порівнювати покомпонентно, наприклад:

```
(d2.day= =d.day)&&(d2.month= =d.month)&&(d2.year= =d.year).
```

Щоб спростити визначення даних структурних типів, останні часто іменують за допомогою оператора `typedef`. Структури можуть передаватися параметрам функцій, функції – повертати структури як результати.

Приклад 3.62. Функція для додавання комплексних чисел:

```
typedef struct {
double real, imag;
} complex;   /*визначено новий  тип complex*/
complex x, y, z;  /*опис змінних типу complex*/

/*Функція Add_complex: із комплексними параметрами й комплекс-
ним значенням*/
//Повертає суму c3=c1+c2
complex Add_complex (complex c1, complex c2)
{
          complex c3=c1;
          c3.real+=c2.real;
          c3.imag+=c2.imag;
          return c3;
}
…
z=Add_complex (x,y) ;      /* z=x+y*/
■
```

У наступному прикладі показано, як обробляються таблиці, подані у вигляді масиву структур. Тут і далі під таблицями будемо розуміти кортежі однотипних структур.

Приклад 3.63 [55]. Підрахунок частоти входжень службових слів у текст С-програми. У програмі ініціалізується масив **keytab** структур із двома полями: у першому розташований покажчик на службове слово, у другому – частота його входжень у дану програму. Масив **keytab** містить усі службові слова мови С, розташовані за визначенням у лексикографічному порядку, який потрібен для застосування бінарного пошуку в таблиці:

```
  #include <stdio.h>
  #include <ctype.h>
  #include <string.h>
  //Опис та ініціалізація таблиці: у першому стовпчику розташова-
ні // покажчики на службові слова, у другому – частота їхніх вхо-
джень // у текст

  struct key {char *keyword; int keycount;} keytab []={
      {"break", 0},
      {"case", 0},
      {"char", 0},
                …
      {"unsigned", 0},
      {"while", 0}
  };
  # define MAXWORD 20
  # define LETTER 'a'
  # define DIGIT     '0'

  int getword(char *w, int lim); /*Функція читання чергового сло-
ва*/
  int type(int c); /*Визначення типу символу*/
  int binary (char *word); /*Бінарний пошук у таблиці слова*/

  int main()
  {
  int n, t;
  char word[MAXWORD];
  while((t=getword(word, MAXWORD))!=EOF)
  if(t==LETTER)
  {
  if((n=binary(word))>=0)
  keytab[n].keycount++;
  };
  int count=(sizeof(keytab)/sizeof(struct key));
  for(n=0; n<count; n++)
```

```
   {
      if(keytab[n].keycount>0)
            printf("%4d%s\n", keytab[n].keycount, keytab[n].keyword);
   };
      return 0;
   }

   /*бінарний пошук у таблиці*/
   int binary (char *word)
   {
   int cond, low=0, high=(sizeof(keytab)/sizeof(struct key))-1;
   int mid;
   while(low<=high)
   {
   mid=(low+high)/2;
   struct key k=keytab[mid];
   //cond=;
   if((cond=strcmp(word, k.keyword))<0)
   high=mid-1;
   else
   if (cond>0) low=mid+1;
   else return (mid);
   }
   return (-1);
   }
   /*getword - читає з клавіатури в рядок w черговий ідентифіка-
тор, довжина якого обмежена числом lim */
   int getword(char *w, int lim)
   {
   int c, t;
   if(type(c=*w++=getch())!=LETTER){
   *w='\0'; return(c);}
   while(--lim>0){
   t=type(c=*w++=getch());
   if(t!=LETTER && t!=DIGIT) {ungetch(c); break;}
   }
   *(w-1)='\0';
   return(LETTER);
   }

   int type(int c)
   {
   if(isalpha(c)) return(LETTER);
   else
   if(isdigit(c)) return(DIGIT);
   else return(c);
   }
   ■
```

3.7.3. БІТОВІ ПОЛЯ

Часто трапляється, що один чи кілька компонентів структури не перевищують певного невеликого цілого значення, наприклад 1 (випадок бітових прапорців). З метою економії пам'яті було б доцільно пакувати такі об'єкти в одне машинне слово (типу **int**). В інших ситуаціях виникає необхідність отримати прямий доступ не до всього слова, а до окремих його бітів. Такі можливості надають *бітові поля*:

<бітове-поле>::=[<декларація-цілого-поля>]:<константа-довжина-поля>

У загальному випадку тип структури з бітовим полем виглядає так:

```
struct
 { ..
 unsigned x: n1; /*x:беззнакове бітове поле довжиною n1*/
signed int y: n2;   /*y: знакове бітове поле довжиною n2*/
int z: n3;          /*z: просте бітове поле довжиною n3*/
     : n4;     /*z: неіменоване бітове поле довжиною n4*/
…
}
```

До неіменованих полів доступ відсутній. За їхньою допомогою вирівнюють адреси сусідніх компонент. Нульова довжина поля означає, що наступне бітове поле розпочнеться з нового слова.

Приклад 3.64. Бітові поля:

а) Визначення чотирьох прапорців і бітового поля довжиною 8:

```
struct flag {unsigned int less:1;
       unsigned int eq:1;
       unsigned int gr:1;
                                :5;
        unsigned int fl:1;
                                :0;
       unsigned int reg:8;

}  cmp;
Структура flag:
```

1 1 1 5 1 7 8

Структура **flag** потребує двох слів, затемнені поля довжиною 5, 7 та 8 не використані. Поле **reg** автоматично розміщується на початку другого слова.

б) Пакування десяткових цифр:

```
struct pdigit {
              unsigned int a:4;
       unsigned int b:4;
       unsigned int  c:4;
       unsigned int d:4;
   } pd, longdec[256];
```

Структура pdigit:

a	B	C	d
4	4	4	4

У структурі **pd** подаються чотири десяткові цифри, доступні за іменами **pd.a, pd.b, pd.c, pd.d**, а в масиві **longdec** – 1024 цифри. Оператори присвоювання **pd.a=5; pd.b=6; pd.c=7; pd.d=8;** записують у змінну **pd** двійково-десяткове число 5678 ■

Знакові бітові поля автоматично розміщуються на границях слів, при цьому деякі частини слів можуть бути невикористаними.

Використання бітових полів має певні обмеження:

- не можна отримати адресу бітового поля;
- немає масивів бітових даних;
- програми з бітовими полями значною мірою є машинно-залежними.

3.7.4. ТИП ОБ'ЄДНАНЬ

Тип об'єднань є прямою сумою кількох типів. Його дані називаються *об'єднаннями*. Змінні типу об'єднання можуть набувати значень кількох різних типів. У мові С вони трактуються як структури, усі поля яких розміщуються не в окремих послідовних ділянках пам'яті, а в одній спільній ділянці, достатній для збереження найбільшого поля. Це означає, що всі вони мають одну спільну адресу. Звідси походить і друга назва даних цього типу – *суміші*. Як наслідок, є можливість доступу до однієї й тієї самої ділянки пам'яті за допомогою змінних різних типів. Передбачивши, наприклад, в об'єднанні поле – символьний масив, можна працювати з кожним окремим байтом ділянки, де зберігається значення об'єднання. Можна також отримати доступ до окремих бітів ділянки – у С99 в об'єднаннях допускаються й бітові поля. За допомогою об'єднань можна економити пам'ять за умови, що обробка полів здійснюється в асі послідовно.

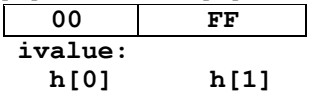
Синтаксис опису об'єднань, як і всі обмеження на нього, а також засоби доступу до полів ті самі, що й у випадку структур. Необхідно

тільки в описі об'єднань службове слово **struct** замінити на службове слово **union.**

Приклад 3.65. Об'єднання двобайтного масиву й цілої змінної:

```
union mix {unsigned char h[2];
           int   ivalue;
           }
```

Масив **h** дозволяє розкласти поле **ivalue** на дві однобайтні частини – старшу **h[0]** і молодшу **h[1]**. Нехай у полі **ivalue** записано **255**. Тоді **h[0]=0x0** і **h[1]=0xFF**



Застосуємо тип **mix** для друкування ASCII- та SCAN-кодів символів, що вводяться з клавіатури:

```
#include <bios.h>
#include <stdio.h>

int main()
{
union mix exc;
unsigned char scn, asc;

do /*цикл до Cntr+ Z*/
{
printf ("\n");
exc.ivalue=bioskey(0); /*bioskey(0): повертає черговий розширений код символу*/
asc=exc.h[0];
scn=exc.h[1];
printf ("%4d || %4d", scn, asc);
}
while (asc!=26 || scn !=44) /*26 та 44 є ASCII- та SCAN-кодами розширеного символу Cntr+ Z*/
return 0;
}
```

■

У мові С не передбачено стандартних засобів для запам'ятовування того поля об'єднання, в якому було зроблено останній запис, тобто для фіксації поточного активного поля. Загальний спосіб розв'язання цієї проблеми – введення спеціального інформаційного поля зліченного типу, що разом з об'єднанням утворювало б структуру й у кожний

момент часу інформувало про активне поточне поле всередині об'єднання. Для об'єднання **data** подібна структура й організація контролю за активним полем могли б виглядати так:

```
enum mix_tag{mix_h, mix_ivalue};
struct MIX {
enum mix_tag flag; /*flag: індикатор активного поля в об'єднан-
ні data*/
union mix {unsigned char h[2];
int ivalue;
 } data;
} exc;
```

Нехай у полі **data** структури **exc** необхідно змінити масив **h**, наприклад, записати в **h[0]** число **25**. Тоді спочатку робимо активним масив, а потім здійснюємо відповідне присвоювання:

```
exc.flag=mix_h;
exc.data.h[0]=25;
```

Якщо там само необхідно змінити значення поля **ivalue** на **1024**, то дії аналогічні:

```
exc.flag=mix_ivalue;
exc.data.ivalue=1024;
```

***Література для СР:** Структури, об'єднання – [55, 102, 132, 133].

Контрольні запитання та вправи

1. Що таке тип структур?
2. Що таке структура?
3. Як виглядає специфікатор типу структур?
4. Що таке тег структури?
5. Як ініціалізуються структури?
6. Навести приклади структур із неповним типом.
7. Визначити усі операції на структурах.
8. Який зв'язок між операціями стрілочка та крапка?
9. Що таке тип об'єднання?
10. Як виглядає специфікатор типу об'єднання?
11. Що спільного між структурами й об'єднаннями?
12. Чим відрізняються структури й об'єднання?
13. Як описуються бітові поля?
14. Для чого застосовують бітові поля?
15. Які існують обмеження на застосування бітових полів?
16. Що виведе такий фрагмент програми:

```
void main()
{struct   A {int i; char *s; struct A *p;};
static  struct    A  b[]={{1,"XYZ",b+2},{2,"ABC",b},
{3,"PQR",b+1},
{4,"PRQ",b}} ;
struct A *ptr=b;
printf ("%s",  b[--(ptr->p ->i)+1].s);
}?
```

17. Описати структуру й написати функцію, що порівнює дві структури:
    а) книга: містить інформацію про автора, назву книжки, рік видання, видавництво й кількість сторінок;
    б) студент: містить інформацію про ПІБ студента, рік вступу, рік народження, адресу й телефон;
    в) товар: містить інформацію про назву товару, виробника, кількість одиниць товару в наявності;
    г) клієнт банку: містить інформацію про ПІБ клієнта, номер рахунку, кількість грошей на рахунку;
    д) телефонний номер: містить інформацію про ПІБ абонента, номер телефону, адресу абонента.
18. Описати таблицю у вигляді масиву структур для даних, оголошених у вправі 17, і написати функції для введення й виведення таких таблиць.
19. Для описаних у вправі 18 таблиць знайти й вивести на екран інформацію:
    а) дані про книжку із заданою назвою;
    б) дані про всіх студентів із заданим роком вступу;
    в) дані про всі товари заданого виробника;
    г) дані про всіх клієнтів банку, на рахунку яких кількість грошей менша ніж зазначена;
    д) дані про абонента за ПІБ.
20. Структура **`struct date{int day;int month;int year;int year_day; char mon_name[4];}`** подає інформацію про дату: номер місяця, рік, порядковий номер дня в році й назву місяця. Написати функції: а) **`void day_of_year(struct date *pd)`** для знаходження порядкового номера дня в даті **`*pd`** і занесення його в поле **`(*pd).year_day;`**
    б) **`void month_day(struct date *pd)`** для пошуку місяця й числа в даті **`*pd`** за порядковим його номером і занесення їх у відповідні поля структури **`*pd`**.
21. Раціональні числа можна подати у вигляді структур типу:

```
typedef struct {long n/*=чисельник*/;
long d /*=знаменник*/;
} racio;
```

    Написати функції для основних арифметичних операцій над раціональними числами й відношень на них. Напри-

клад, для операції додавання відповідна функція могла б мати такий вигляд:

```
racio add(racio a, racio b)
{
racio temp; long com;
long gcd(long,long);
temp.n=a.n*b.d+a.d*b.n;
temp.d=a.d*b.d;
com=gcd(temp.n,temp.d); /*функція gcd обчислює НСД (x,
y)*/
temp.n/=com;
temp.d/=com;
return temp;
}
```

22. Знайти індекси двох однакових елементів матриці. Для зменшення часової складності програми матрицю подати як об'єднання двовимірного й одновимірного масивів.
23. Знайти найбільший елемент у тривимірному масиві. Для зменшення часової складності програми вхідний масив подати як об'єднання тривимірного й одновимірного масивів.
24. Реалізувати цілу довгу: а) двійкову; б) десяткову арифметики. Для компактного подання довгих десяткових цілих використати масиви структур із бітовими полями **pdigit** із прикл. 3.64. Написати функції для взаємного перетворення чисел форматів а) та б).
25. Скористатися довгою арифметикою із вправи 24 для обчислення надвеликих чисел 2^{100} та 100!.

3.8. Файли

- Потоки й файли
- Консольні функції введення-виведення
- Обробка файлів
- Перенаправлення потоків

Ключові слова: *файл, потік, потокове введення-виведення, введення-виведення нижнього рівня, введення-виведення для консолі й портів, потоки* **stdin**, **stdout**, **stderr**, *текстовий потік, двійковий потік, покажчик файла, відкривання й закривання файла, перенаправлення потоків, стандартні функції:* **getchar**, **putchar**, **getch**, **getche**, **gets**, **puts**, **scanf**, **printf**, **sscanf**, **sprintf**, **fopen**,

fclose, **fputc**, **fgetc**, **fgets**, **fputs**, **fseek**, **ftell**, **fprintf**, **fscanf**, **feof**, **ferror**, **fflush**, **rewind**, **remove**, **fread**, **fwrite**, **freopen**.

Збереження й обробка інформації на зовнішніх пристроях здійснюються за допомогою файлових типів даних.

3.8.1. ПОТОКИ Й ФАЙЛИ

Файли – це послідовності однотипних компонентів із певними засобами доступу й обробки.У мові С усі файли розглядаються як послідовності символів (байтів). Бібліотеки С підтримують три рівні обробки таких файлів: *потокове введення-виведення*, *введення-виведення нижнього рівня* та *введення-виведення для консолі й портів*.

На перших двох рівнях обмін даними між ОП і файлом здійснюється побайтно фіксованими частинами – блоками, що формуються і зберігаються в спеціальних системних полях – *буферах*. При читанні з файла дані спочатку блоками заносяться в буфер, а потім звідти за допомогою функцій введення-виведення передаються програмі. При записі у файл функції введення-виведення попередньо заносять дані (побайтно) у буфер, і при його заповненні весь блок разом передається у файл.

Незважаючи на специфіку фізичних файлів і операцій введення-виведення для різних пристроїв (екран, клавіатура, диск, принтер тощо), у потоковому введенні-виведенні завдяки буферизації підтримується єдиний механізм доступу до даних. У програмах даний механізм реалізують спеціальні об'єкти – *потоки*. Неформально можна сказати, що потік – це поіменована сукупність стандартних універсальних засобів буферизації, пов'язана з певним фізичним файлом. Потоки бувають двох типів: текстові та двійкові. При роботі з ними можливі такі дії:

1) відкривання й закривання потоку (функції **fopen**, **fclose**);

2) введення-виведення: символів, рядків, форматованих даних, порцій даних довільної довжини (функції **getc**, **putc**, **fgets**, **fputs**, **fprintf**, **fscanf**);

3) аналіз помилок операцій введення-виведення;

4) керування буферизацією потоку (розмір буфера тощо);

5) керування буферним покажчиком (індексом потоку), який задає поточну позицію в потоці.

У стандарті С *текстовий потік* організований у вигляді рядків, кожен з яких закінчується символом нового рядка **'\n'**; у кінці останнього рядка цей символ не обов'язковий. У текстовому потоці на вимогу базового середовища можуть відбуватися певні перетворення символів. Наприклад, символ **LF** нового рядка може бути замінений

парою символів – **CR** і **LF**. Таким чином, може не бути однозначної відповідності між символами, що пишуться (зчитуються), і тими, які зберігаються на зовнішньому пристрої.

Двійковий потік – це послідовність байтів, що взаємно однозначно відповідає байтам на зовнішньому пристрої. Перетворення символів (як у попередньому випадку) не відбувається, проте наприкінці двійкового потоку може додаватись обумовлена програмою кількість нульових байтів. Ці байти можуть використовуватися в блоці пам'яті для повного заповнення сектора на диску.

Коли С-програма розпочинає роботу автоматично, для неї відкриваються три стандартних потоки: **stdin** (для клавіатури), **stdout** (для екрана) і **stderr** (для виведення повідомлень про помилки на екран). Розглянемо спочатку *консольні* функції введення-виведення – так називають функції, що виконують операції введення з клавіатури й виведення на екран.

3.8.2. КОНСОЛЬНІ ФУНКЦІЇ ВВЕДЕННЯ-ВИВЕДЕННЯ

Читання й запис символів. Найпростіший механізм введення інформації – читання по одному символу з клавіатури за допомогою функції **getchar**:

```
int getchar(void)
```

Функція реалізована у вигляді макросу **getc(stdin)** і повертає значення коду чергового символу (якщо він є) з потоку **stdin** чи **EOF** при досягненні кінця файла. Виведення символу в поточну позицію курсору на екрані можна здійснювати за допомогою функції **putchar**:

```
int putchar(int c);
```

Функція реалізована у вигляді макросу **putc(c, stdout);** і повертає **c** чи **EOF** у випадку помилки.

Приклад 3.66. Посимвольне копіювання на екран рядка, введеного із клавіатури:

```
#include<stdio.h>

int main()
{
    char c;
/*getchar читає символи з потоку stdin, який має буфер на один рядок. Повертає значення тільки після введення всього рядка, який закінчується символом'\n'(клавіша <Enter>)*/
```

```
    while((c=getchar())!='\n')
          putchar(c);
    return 0;
}
```

■

Як зазначалося вище, функція **getchar** має одну особливість – вона повертає значення тільки після остаточного заповнення буфера клавіатури, яке завершується введенням символу **CR** за допомогою клавіші **<Enter>**. Таким чином, щоб ввести один або кілька символів, необхідно натиснути відповідні клавіші, а потім – клавішу **<Enter>**. Функція при кожному виклику вводить тільки один символ, тому збереження в буфері цілого рядка може призвести до того, що в черзі на введення залишаться один або кілька символів, а в інтерактивному середовищі це досить незручно. Іншою незручністю може стати ехо-відображення на екрані введених символів. Альтернативами функції **getchar**, які можуть застосовуватися як інтерактивні й не використовують буферизацію введеної інформації, є функції

```
int getch(void);
int getche(void);
```

Вони визначені в заголовному файлі **<conio.h>**. Перша повертає один введений із консолі символ без виведення його на екран, друга робить те саме, але символ на екран відображає.

Приклад 3.67. Із клавіатури вводиться послідовність символів. Підрахувати, скільки в ній слів, тобто послідовностей, розташованих між порожніми символами (пробіл, $\backslash n$, $\backslash t$, EOF) або на її початку – до першого порожнього символу.

Проілюструємо на цьому прикладі, як для аналізу й обробки послідовностей символів (програми-фільтри) використовуються скінченні автомати. Ідея застосування автомата полягає в тому, щоб за допомогою станів у процесі введення символів контролювати скінченну кількість фіксованих ситуацій і, залежно від стану, реагувати на введений символ (як того вимагає умова задачі). У нашому випадку таких ситуацій дві: 1) поточний символ перебуває в межах чергового слова, 2) за його межами. Можливі дії – вводити черговий символ (**g**) і збільшувати лічильник слів (**m**++), якщо розпочалося введення нового слова. Наш автомат має два стани: **IN** (=1) та **OUT** (=0). Перший означає, що процес перебігає в поточному слові, а другий – поза ним. Нехай a та b позначають довільні непорожній і порожній символи (пробіл, $\backslash n, \backslash t$). У таблиці переходів

автомата за символом / показана реакція програми на ситуацію. При переході в стан **STOP** автомат закінчує роботу.

Стан\Вхідний символ	A	B	**EOF**
IN	**IN** /g	**OUT** /g	**STOP**
OUT	**IN** /g, m++	**OUT** /g	**STOP**

Для подання станів вводиться змінна **state**, яка набуває одного зі значень – **IN** або **OUT**. Програма за допомогою перемикачів моделює переходи автомата й реакцію на них згідно з таблицею переходів. Як тільки черговий символ розпочинає нове слово (при переході від стану **OUT** у стан **IN**), лічильник слів **m** збільшується на 1. Читання чергового символу зі вхідного потоку відбувається в усіх ситуаціях, тому виклик функції **getchar()** винесено за межі перемикачів у кінець тіла циклу.

Лістинг.

```
   #include <stdio.h>
   #define IN 1
   #define OUT 0

/*count_word: обчислення кількості введених із клавіатури слів*/
    long count_word(void)
     {int ch, state=OUT, m=0;
       ch=getchar();/*читання першого символу зі вхідного потоку*/
 while (ch !=EOF)
     {
     if (ch==' ' || ch=='\n' || ch=='\t')/*прочитано порожній
                                                        символ*/
     switch (state){
     case IN : state=OUT;break;
     case OUT:
                  }
     else /*прочитано непорожній символ*/
    switch (state){
     case OUT : state=IN; m++;break;
     case IN:
                  }
     ch=getchar();/*читання чергового символу зі вхідного потоку*/
    }
     return m;
     }

    void main (void)
    {printf("\n%ld", count_word());
    }
```

■

Читання й запис рядків. У файлі **<stdio.h>** описані функції для зчитування з клавіатури й відображення на екран рядків символів:

```
char *gets(char *s);
int puts(const char *s);
```

Перша функція читає з клавіатури рядок символів, включаючи пробіли й табуляції, поки не зустріне символ нового рядка, що замінюється нульовим символом (**\0**). Послідовність прочитаних символів запам'ятовується в області пам'яті, що адресується аргументом **s**. Цією областю є зазвичай масив. Необхідно слідкувати, щоб пам'яті вистачало для зберігання введеного рядка. Якщо введення успішне, то функція повертає **s**, у випадку кінця файла чи помилки – нуль. Друга функція виводить на екран заданий рядок і символ нового рядка, а при успішному завершенні повертає ненульове додатне значення, інакше – **EOF**.

Приклад 3.68. Застосування функцій **gets** та **puts** для введення з клавіатури й виведення на екран рядка:

```
#include<stdio.h>
#define MAXLINE 128
int main()
{
    char str[MAXLINE];
    puts ("Введіть рядок:");
    gets(str);
 …
 puts (str);

    return 0;
}
```

■

Форматовані функції читання й запису. У **<stdio.h>** описані функції **scanf** та **printf**, що виконують форматоване введення й виведення, тобто можуть читати й відображати дані в різних форматах:

```
int scanf (const char * restrict format, ...);
int printf (const char * restrict format, ...);
```

Обидві функції мають змінну кількість параметрів, перший з яких – форматний рядок – є обов'язковим і захищеним від зовнішніх впливів кваліфікаторами типу **const** та **restrict**. Він керує всім процесом форматування і введення-виведення інформації.

Функція **printf** здійснює виведення на екран відформатованої інформації. Повертає кількість записаних символів або, у випадку помилки, – від'ємне значення. Форматний рядок містить два види елементів: звичайні символи, що виводяться на екран, та *специфікатори перетворення*, які викликають перетворення до відповідного вигляду й виведення інших параметрів згідно з порядком їхнього розташування в операторі виклику функції. Форматний рядок може містити:

- пробіли `' '`, табуляції **`'\t'`**, символи нового рядка **`'\n'`** та інші керівні символи;
- звичайні символи, але не **`'\'`** та **`'%'`**, які просто виводяться;
- специфікатори перетворення.

Останні мають вигляд:

`%[<прапорці>][<ширина>][.<точність>][F|N|h|l|L] <специфікатор-перетворення>`

Прапорці (у довільному порядку):

-	Текст, що виводиться, вирівнюється по лівому краю. Весь порожній простір, що залишився праворуч, заповнюється пробілами. За умовчанням текст вирівнюється по правому краю
+	Числові значення друкуються зі знаком
пробіл	Перед додатними числовими значеннями виводиться пробіл, а перед від'ємними – знак мінус
0	Числа мають доповнюватися ліворуч нулями до всієї ширини поля
#	Указує на одну з таких форм виведення: для **o** першою цифрою має бути **0**; для **x** чи **X** ненульовому результату повинні передувати **0x** чи **0X**; для **e, E, f, g** та **G** виведення має обов'язково містити десяткову точку; для **g** і **G** завершальні нулі не відкидаються

Ширина. Задає мінімальну ширину поля виведення. Перетворений аргумент друкується в полі, розмір якого не менше зазначеної ширини, а за потреби – у полі більшого розміру. Якщо кількість символів аргументу менша за ширину поля, то поле доповнюється пробілами ліворуч (праворуч, якщо текст вирівняний по лівому краю). Однак, якщо значення ширини починається з 0, то доповнення здійснюється цифрами 0.

Як ширину поля можна вказати символ *, що приводить до використання значення наступного аргументу типу **int** у списку аргументів як ширини поля.

Точність. Точка відділяє ширину від точності. За точкою йде ціле значення. Значення модифікатора точності залежить від типу даних. Якщо його використовують для даних із рухомою точкою, то він визна-

чає кількість виведених дробових десяткових розрядів. Наприклад, **%10.4f** означає, що ширина поля буде не менше 10 символів, причому для дробових десяткових розрядів буде відведено чотири позиції.

Якщо модифікатор точності застосовується з форматами **%g** або **%G**, то визначає кількість значущих цифр.

При введенні рядків цей модифікатор визначає максимальну довжину поля. Наприклад, **%5.7s** означає, що довжина виведеного рядка становить мінімум п'ять і максимум сім символів. Якщо рядок довший ніж максимальна довжина поля, то кінцеві символи виводитися не будуть.

Для цілих типів модифікатор точності визначає мінімальну кількість цифр, що виводяться. Для досягнення необхідної кількості цифр додається деяка кількість початкових нулів.

Якщо замість точності вказати символ *, то за її числове значення буде взято значення наступного аргументу типу **int** у списку аргументів.

Перетворення:

Символ	Опис
d, i	Десяткове ціле зі знаком
o	Беззнакове вісімкове число (без **0** ліворуч)
x, X	Беззнакове шістнадцяткове число (без **0x** чи **0X** ліворуч)
u	Беззнакове десяткове ціле
c	Символ
s	Рядок символів
f	Десяткове з плаваючою точкою
e, E	Експоненційне зображення відповідно з позначеннями **e** чи **E**
g, G	Залежно від того, яке виведення буде коротшим, використовується **%e (%E)** чи **%f**
p	Виводить покажчик
n	Аргумент, що відповідає цьому специфікатору, має бути покажчиком на цілочислову змінну. Специфікатор дозволяє зберегти у змінній кількість виведених символів
%	Виводиться символ **%**

Приклад 3.69. Програма, що демонструє можливості специфікаторів перетворення функції **printf ()**:

```
#include<stdio.h>
#define pi 3.141592654

int main()
{
    printf("%10s\n", "hello, world");
```

```
    printf("%20s\n", "hello, world");
    printf("%-20s\n", "hello, world");
    printf("%20.10s\n", "hello, world");
    printf("%-20.10s\n", "hello, world");
    printf("%.10s\n", "hello, world");
    printf("%10f\n", pi);
    printf("%20f\n", pi);
    printf("%-20f\n", pi);
    printf("%20.10f\n", pi);
    printf("%-20.10f\n", pi);
    printf("%.10f\n", pi);

    return 0;
}
```

У результаті виконання програми на екран будуть виведені рядки:

```
hello, world
 hello, world
hello, world
 hello, wor
hello, wor
hello, wor
 3.141593
 3.141593
3.1415926540
3.1415926540
■
```

Функція **scanf** читає дані з клавіатури під керуванням форматного рядка, перетворює їх у внутрішню форму згідно зі специфікаторами формату та присвоює по порядку аргументам, кожен з яких має бути виразом і задавати покажчик відповідного типу. Завершує роботу, коли вичерпується форматний рядок. Повертає кількість перетворених і введених елементів або **EOF** при досягненні кінця файла чи у випадку помилки.

Форматний рядок має вигляд:

```
<форматний рядок>::=[<пробіл>] [<табуляція>] [<новий рядок>] \
[<літерал>] [<специфікатор-перетворення>]
<специфікатор-перетворення>::=%[*][<ширина>]       [N       | F]
[h | l | L] [<тип>]
<ширина>::=<число>
<тип>::=% | c | d | D | e | E | f | F | g | G |i |
|I | n | o | O | p | s | u | U | x | X | [
```

Форматний рядок може містити пробіли, символи табуляції й нового рядка, літерали без зовнішніх лапок і специфікатори перетворення.

Керують виведенням тільки специфікатори перетворення, решта елементів є фільтрами – вони пропускають із клавіатури лише свої коди, які читаються, але без наслідків, тобто фактично ігноруються.

Специфікатори й типи перетворень:

Специфікатори перетворень:

*****	Блокує присвоювання значення аргументу, що відповідає вхідному полю
[ширина]	Максимальна кількість символів вхідного поля, що враховуються під час введення інформації. При цьому використовується не більше зазначеної кількості символів
[N|F]	Указує адресу аргументу як **near** (близький) (**N**) чи **far** (далекий) (**F**)
[h|l|L]	Тип даних аргументів: **short** –для **(h), long int, double** – для **(l), long double** – для **(L)**

Типи перетворень:

Код	Значення
%a	Значення з плаваючою точкою
%c	Символ
%d	Десяткове ціле число
%i	Ціле число як у десятковому, так і у вісімковому чи шістнадцятковому форматі
%e	Число з плаваючою точкою
%f	Число з плаваючою точкою
%g	Число з плаваючою точкою
%o	Вісімкове число
%s	Рядок
%x	Шістнадцяткове число
%p	Покажчик
%n	Ціле значення, що дорівнює кількості вже зчитаних символів
%u	Десяткове ціле число без знака
%[]	Набір символів, що скануються. Наприклад, **%[XYZ]** означає сканувати лише **X**, **Y** та **Z**; **%[A-Z]** – сканувати символи від **A** до **Z**
%%	Перетворення відсутнє. Читається й запам'ятовується знак відсотка

Приклад 3.70. Виведення на екран десяткового й шістнадцяткового подань введеного цілого:

```
#include<stdio.h>

int main()
```

```
{
    int i;
    puts("Введіть десяткове ціле у форматі: i: ц…ц ");
    scanf("i: %d", &i); /*Без префікса "i:" число не буде прочи-
тане!*/
    printf("\nДесяткове подання i:%d || Шістнадцяткове подання
i: %x\n", i, i);

    return 0;
}
```

Після введення числа 10 програма виведе на екран рядок символів:

```
Десяткове подання i: 10 || Шістнадцяткове подання i: 0xA
■
```

Форматовані читання з рядків і запис у рядки здійснюють функції

```
int sscanf (char *s, const char* restrict format, …);
int sprintf (char* restrict s, const char* restrict format, …);
```

Вони є аналогами відповідних функцій **scanf** та **printf**. Відмінність полягає тільки в тому, що стандартні вхідний і вихідний потоки в них замінені на рядок **s**, тобто символи читаються з рядка **s** і записуються в нього. При читанні досягнення кінця рядка рівноцінне ситуації з досягненням кінця файла.

3.8.3. ОБРОБКА ФАЙЛІВ

Потік зв'язується з конкретним файлом у процесі його відкривання. Як тільки потік (файл) відкрито, можна проводити обмін інформацією між ним і програмою. Файл від'єднується від певного потоку (розривається зв'язок між файлом і потоком) за допомогою операції закривання. Кожний потік має керівну структуру типу **FILE**, що містить усю необхідну інформацію для роботи з ним. Вона визначається за допомогою покажчика на структуру типу **FILE: FILE *fp;**. Змінна **fp** зображує потік у подальшій роботі з файлом.

Опис типу **FILE**, а також прототипи більшості функцій і констант файлової системи містяться в заголовному файлі **<stdio.h>**. Окрім типу **FILE**, у ньому розташовані описи типів **size_t**, **fpos_t**, **FILE**, **size_t** та **fpos_t** – цілі без знака – і визначаються кілька спеціальних макросів, зокрема такі, як **NULL**, **EOF**, **FOPEN_MAX**, **SEEK_SET**, **SEEK_CUR** та **SEEK_END**. Макрос **NULL** визначає порожній (**null**) покажчик. Макрос

EOF, часто визначений як **-1**, є значенням, що повертають тоді, коли функція намагається виконати читання після закінчення файла. Константа **FOPEN_MAX** визначає ціле значення, рівне максимальній кількості одночасно відкритих файлів. Інші макроси використовуються разом із функцією **fseek**, що реалізує прямий доступ до файла.

Розглянемо основні функції файлової системи. Їхні аналоги для обробки широких символів можна знайти в [133].

Відкривання й закривання потоків. Відкриває потік і зв'язує його з файлом функція **fopen**:

```
FILE *fopen(const char *filename, const char *mode);
```

Якщо відкривання відбулося успішно, то функція повертає *покажчик файла*, інакше – значення **NULL**. Параметр **filename** задає ім'я файла й може містити інформацію про шлях до нього у файловій системі. Параметр **mode** визначає режим відкривання файла:

Режим	Двійковий	Текстовий	Семантика
«r»	«rb»	«rt»	Відкриває існуючий файл для читання
«w»	«wb»	«wt»	Створює новий файл для запису. Перезаписує існуючий файл із таким самим іменем
«a»	«ab»	«at»	Відкриває існуючий файл для додавання в його кінець нової інформації
«r+»	«r+b»	«r+t»	Відкриває існуючий файл для читання й запису
«w+»	«w+b»	«w+t»	Створює новий файл для читання й запису. Перезаписує існуючий файл із таким самим іменем
«a+»	«a+b»	«a+t»	Відкриває існуючий файл для додавання в його кінець нової інформації. Якщо файл не існує, то створюється новий із таким самим іменем

Максимальна кількість одночасно відкритих файлів визначається **FOPEN_MAX**. Її точне значення залежить від компілятора, але має бути не менше 8.

Закриває відкритий потік функція `int fclose(FILE *stream);`

Якщо потік, заданий параметром-покажчиком файла, був відкритий для запису, то перед закриванням у файл записуються дані, що містяться в буферах потоку. Після закривання файла його покажчик стає беззмістовним. Якщо закривання відбулося успішно, то функція повертає 0, інакше – значення **EOF**.

Приклад 3.71. Відкривання й закривання файлів:

1)
```
FILE *fp;
fp=fopen("file1.txt", "a+"); /*відкривання файла file1.txt для модифікації й доповнення*/
```

2) При відкриванні файла можливі помилкові ситуації. Наприклад, помилково записане ім'я файла, що відкривається для читання тощо. Тоді **fopen** повертає **NULL**:

```
if ((fp=fopen("file2.txt", "r")==NULL)
{perror("помилка при відкриванні файла file2.txt"); exit(0);}
```

3)
```
fclose (fp); /*закривання файла file1.txt*/
```

4)
```
if (fclose (fp))
{perror("помилка при закриванні файла file2.txt"); exit(0);}
```

■

Читання й запис символів у файл

```
int fputc(int c, FILE *stream);
int fgetc(FILE *stream);
int ungetc (int c, FILE *stream);
```

Перша функція записує заданий символ **c** у потік **stream**. У випадку успіху повертає значення **c**, при помилці – значення **EOF**. Друга читає черговий символ із заданого потоку **stream**. У випадку успіху повертає значення символу, при помилці – **EOF**. Функція **ungetc** повертає символ **c** у вхідний потік **stream**, щоб він міг бути повторно прочитаним функціями **fgetc**, **getc** та **getchar**. Якщо повертається кілька символів, то вони будуть доступними, як у стеку – останнім повернутий є першим доступним.

Є також функції **putc** та **getc**, повністю аналогічні наведеним функціям. Їх включено до стандарту С99 для сумісності зі старими версіями С.

Читання й запис рядків у файл здійснюють функції

```
char *fgets(char *s, int n, FILE *stream);
int fputs(const char *s, FILE *stream);
```

Перша послідовно читає із заданого потоку **stream** символи включно до символу нового рядка, але не більше **n-1** символів. Прочитані символи розміщуються за адресою **s**. До кінця рядка приєднується символ нового рядка, якщо він прочитаний, а також нульовий символ. У випадку успіху повертає значення **s**, при помилці – нульовий покажчик. Друга записує в потік **stream** рядок символів, розташований за адресою **s**. Функція передає символи, поки не зустріне нульовий символ. У випадку успіху повертає кількість записаних символів, при помилці – **EOF**.

Приклад 3.72. Читання рядків довільного файла з виведенням їх на екран. Після завершення читання виводиться загальна кількість прочитаних рядків:

```
#include <stdio.h>
#include <errno.h>

void main (void)
{
char string[256],file_name[128];
    int line_number=0;
    FILE*fptr;
    puts("Введіть ім'я файла");
    gets(file_name);
    if((fptr=fopen(file_name, "r"))!=NULL)
    {
          for (;fgets(string,255,fptr)!=NULL;line_number++)
          fputs(string,stdout);
          printf("\n Прочитано %d рядків файла
                                    %s\n",line_number,file_name);
          fclose(fptr);
    }
 else perror("\nПомилка при відкриванні файла");
}
```

■

Перевірка ознаки кінця файла здійснюється функцією

```
int feof (FILE *stream);
```

Повертає ненульове значення, якщо буферний покажчик потоку міститься за останнім байтом файла. Зазвичай це відбувається в результаті попередньої операції введення. Повертає **0**, якщо буферний покажчик не міститься в кінці файла.

Функції форматованого читання з файла й запису у файл:

```
int fscanf (FILE *stream, const char *format[,address, …]);
int fprintf(FILE *stream, const char *format[,argument, …]);
```

Дані функції зі змінною кількістю параметрів є аналогами консольних функцій **scanf** та **printf** і працюють не зі стандартними, а з довільними відкритими потоками. Перша читає символи з потоку **stream**, відкритого для читання, перетворює їх згідно зі специфікаторами формату на двійкову форму й результати записує в поля, задані адресними аргументами. Друга здійснює виведення в потік **stream**, відкритий для запису, відформатованої інформації. Роль форматного рядка в обох функціях така сама, як і в їхніх аналогах.

Функції для роботи з буферним покажчиком:

```
long int ftell(FILE *stream);
int fseek(FILE *stream, long offset, int whence);
int rewind(FILE *stream);
```

Функція **ftell** повертає поточну позицію буферного покажчика потоку **stream**, з якого починається наступна операція введення-виведення. Зокрема, це значення можна передати функції **fseek**. Використовується також для текстових файлів, але тоді значення, що повертається, не обов'язково буде подавати байтове зміщення у файлі. Функція **fseek** установлює поточну позицію буферного покажчика потоку **stream**, з якої почнеться наступна операція читання чи запису. У випадку успіху повертає нуль, при виникненні помилки – ненульове значення. Тут **long offset** для двійкових файлів – кількість байтів, на яку потрібно зсунути файловий покажчик у напрямку, указаному **whence**. Для текстових файлів це значення має бути нулем чи значенням, що повертається функцією **ftell.** Щоб перемістити файловий покажчик в оберненому напрямку (до початку файла), потрібно встановити значення **offset** від'ємним. Для переміщення файлового покажчика на **offset** байтів від початку файла потрібно встановити **whence** рівним **SEEK_SET** (або 0) відносно поточної позиції – **SEEK_CUR** (або 1); на задану кількість байтів від кінця – **SEEK_END** (або 2).

Функція **rewind** установлює буферний покажчик на початок файла. Зазвичай використовується для повторного читання, починаючи з першого байта, чи для підготовки запису нових даних із початку існуючого файла.

Запис буферів у файл:

```
int fflush(FILE *stream);
```

Функція записує у файл поточний зміст буфера для заданого потоку без закривання файла. Цю функцію слід періодично викликати в усіх програмах, що довго працюють із відкритими файлами.

Повідомлення про помилкові ситуації в потоці:

```
int ferror(FILE *stream);
```

Функція повертає істину (ненульове значення), якщо під час обробки даного потоку виникали помилки. Якщо помилки не були виявлені, то повертається нуль (хибно).

Видалення файла:

```
int remove(const char *filename);
```

Функція видаляє файл **filename** з файлової системи за умови, що він існує. У випадку успіху повертає нуль, інакше повертає –1 і встановлює значення змінної **errno** рівним **ENOENT** (файл не знайдено) або **EACCES** (доступ заборонено). Зазвичай реалізується як макрос, що викликає функцію **unlink**.

Функції для читання й запису даних із потоку:

```
size_t fread (void *ptr, size_t size, size_t n, FILE *stream);
size_t fwrite(void *ptr, size_t size, size_t n, FILE *stream);
```

Реалізують введення-виведення нижнього рівня. Функція **fread** читає дані з файлового потоку, відкритого у двійковому режимі, починаючи з поточного файлового покажчика. Після читання даних цією функцією файловий покажчик розміщується безпосередньо за останнім прочитаним байтом. Кількість прочитаних байтів дорівнює результату функції, помноженому на кількість байтів у елементі. У випадку невдачі або якщо файловий покажчик розташований за кінцем файла функція повертає нуль. Тут **void *ptr** – покажчик на приймаючий буфер, **size_t size** – розмір у байтах одного елемента, що читається з файлового потоку, **size_t n** – кількість таких елементів.

Функція **fwrite** записує один чи кілька байтів у файловий потік, відкритий у двійковому режимі, і повертає кількість записаних елементів. Кількість записаних байтів дорівнює результату функції, помно-

женому на значення аргументу **size** (розмір одного елемента). Тут **void *ptr** – покажчик на дані, що записуються, **size_t size** – розмір у байтах одного елемента, **size_t n** – кількість елементів.

Приклад 3.73. Для ілюстрації роботи деяких описаних вище функцій розглянемо програму:

```
  /*fopen, fgets, ftell, fseek, fread, fclose, rewind, ferror,
fputs*/

  #include <stdio.h>
  #include <stdlib.h>
  #define FNAME "TEST.C"

  int main(int argc, char *argv[])
  {
      FILE *inf; /*вхідний файловий потік*/
      long pos, pos1; /*позиція у файлі*/
      char buffer[128];

      char c;
      /*відкрити текстовий файл для читання у двійковому режимі*/
      inf=fopen(FNAME, "rb");
      if(inf==NULL){
            printf("Помилка при відкриванні файла", FNAME);
            exit(1);
      }
      /*читає перший рядок*/
      fgets(buffer, 128, inf);
      printf("10-й рядок==%s\n", buffer);
      pos=ftell(inf);
      printf("Позиція у файлі==%ld\n", pos);
      /*покажчик на початок файла*/
      rewind(inf);
      pos1=ftell(inf);
      printf("Позиція після rewind==%ld\n", pos1);

      /*Шукати pos байтів від початку файла*/
      if(fseek(inf, pos-1, SEEK_SET)!=0)
            printf("Помилка позиціонування");
      else{
            /*читання символу з файла*/
            fread(&c, 1, 1, inf);
            printf("pos-й символ==%c", c);
      }

      /*намагаємося писати у відкритий для читання файл,
      тобто моделюємо помилку*/
```

```
    puts("Намагаємося писати у відкритий для читання файл");

    fputs("Моделюємо файлову помилку", inf);
    if(ferror(inf))
          printf("Виникла помилка файлового потоку!");

    /*закривання файла*/
    fclose(inf);

    return 0;
}

/*fopen, fwrite, rewind, fread, fclose*/
#include <stdio.h>
#include <stdlib.h>

int array[100]; /*масив цілих нулів*/
#define FNAME "TEMP.C"
int main(int int argc, char *argv[])
{
    /*створити файл у режимі w+b*/
    FILE *tmpf=fopen(FNAME, "w+b");
    if(!tmpf){
          perror("Не можна відкрити файл");
          exit(1);
    }
    /*записати 100 цілих у файл*/
    for(int i=0; i<100; i++)
          fwrite(&i, sizeof(int), 1, tmpf);
    /*прочитати цілі числа в масив*/

    /*установити файловий покажчик на початок файла*/
    rewind(tmpf);
    fread(&array, sizeof(int), 100, tmpf);
    puts("Масив після читання з диска:\n");
    for(int i=0; i<100; i++)
          printf("%8d", array[i]);
    /*закрити файл*/
fclose(tmpf);
    return 0;
}
```

■

3.8.4. ПЕРЕНАПРАВЛЕННЯ ПОТОКІВ

Пригадаємо, що на початку виконання програми автоматично відкриваються три потоки: **stdin** (стандартний потік введення), **stdout** (стандартний потік виведення) і **stderr** (стандартний потік помилок).

Зазвичай ці потоки зв'язані з консоллю. Проте в середовищах, що підтримують перенаправлення введення-виведення, вони можуть бути перенаправлені операційною системою на інший пристрій.

Функції консольного введення-виведення насправді спрямовують результати своїх операцій на один із потоків – **stdin** або **stdout**, і по суті, кожна з них є спеціальною версією відповідної файлової функції.

Оскільки імена стандартних потоків є покажчиками файлів типу **FILE***, то вони можуть використовуватися для застосування загальних файлових операцій при введенні-виведенні на консоль. Наприклад, функція **putchar** може бути визначена таким чином:

```
int putchar(char c)
{
 return putc(c, stdout);
}
```

Узагалі, у ролі покажчиків файлів потоки **stdin**, **stdout** та **stderr** можна застосовувати в будь-якій функції, де використовується змінна типу **FILE***. Наприклад, для введення рядка з консолі можна застосувати такий виклик файлової функції **fgets**:

```
char str[255];
fgets(str, 80, stdin);
```

Зазначимо, що **stdin**, **stdout** і **stderr** – це не змінні у звичайному розумінні та їм не можна присвоювати значення за допомогою функції **fopen()**. Через те, що на початку роботи програми покажчики файлів створюються автоматично, наприкінці роботи вони й закриваються автоматично. Тому не потрібно намагатися закрити їх самостійно.

Операції введення-виведення на дискових файлах можна виконувати за допомогою функції консольного введення-виведення, наприклад **printf**. Справа в тому, що всі функції консольного введення-виведення виконують свої операції з потоками **stdin** та **stdout**, а в середовищах, що підтримують перенаправлення потоків введення-виведення, це рівносильне тому, що **stdin** або **stdout** можуть бути перенаправлені на інший пристрій. Коли С-програма завершується, то всі перенаправлені потоки повертаються в стани, які були встановлені за умовчанням. Таке перенаправлення здійснює функція

```
FILE *freopen(const char *filename, const char *mode, FILE
*stream);
```

Вона закриває поточний відкритий файловий потік і пов'язує його з новим файлом. Зазвичай використовується для перенаправлення потоків

stdin, **stdout**, **stderr**. Тут **filename** – покажчик на рядок імені нового файла, пов'язаного з файловим потоком. Файл відкривається в режимі **mode**; цей параметр може набувати тих самих значень, що й відповідний параметр функції **fopen()**. Якщо функція **freopen()** виконана успішно, то вона повертає **stream**, а якщо зустрілися помилки – то **NULL**:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    FILE *outf;
    //перенаправити stderr на тимчасовий вихідний файл
    outf=freopen("tmp.txt", "w", stderr);
    if(outf==NULL) {
          puts("Помилка відкривання тимчасового файла");
          exit(1);
    }
    fputs("Перевірка повідомлення про помилку,\n", stderr);
    fputs("записаного у файл за допомогою stderr\n", stderr);
    fclose(outf);
    printf("Перегляньте файл tmp.txt");

    return 0;
}
```

Перенаправлення стандартних потоків за допомогою **freopen** у деяких випадках може бути корисним, наприклад, при налагодженні. Однак виконання дискових операцій введення-виведення на перенаправлених потоках **stdin** та **stdout** не дуже ефективне порівняно з використанням таких функцій, як **fread** або **fwrite**.

*** Література для СР:** обробка файлів – [32, 55, 102, 133].

Контрольні запитання та вправи

1. Що таке введення-виведення для консолі й портів?
2. Які функції є консольними функціями введення-виведення?
3. Як ввести символ із клавіатури?
4. Яка різниця в дії функцій **getchar**, **getch** та **getche**?
5. Як вивести символ на екран?
6. Як прочитати рядок символів із клавіатури?
7. Як вивести на екран рядок **"Hello, world!"**?
8. Що таке форматовані введення й виведення?

9. Описати форматний рядок функції **printf**.
10. Описати форматний рядок функції **scanf**.
11. Як діють функції **sscanf** та **sprintf**?
12. Що таке потік і файл?
13. Що таке потокове введення-виведення?
14. Що таке покажчик файла і для чого він використовується?
15. Як відкрити файловий потік?
16. Які є режими відкривання файла?
17. Як закрити файловий потік?
18. Як записати символ у потік?
19. Як отримати символ із заданого потоку?
20. Як записати рядок у потік?
21. Як отримати рядок із заданого потоку?
22. Як отримати файловий покажчик?
23. Як установити нову позицію покажчика файла?
24. Як установити файловий покажчик на початок файла?
25. Як здійснити форматоване текстове виведення чисел у файл? Яка роль при цьому форматного рядка?
26. Як прочитати з файла п'ять даних різних числових типів?
27. Як перевірити, чи досягнутий кінець файла?
28. Як перевірити, чи виникали помилки під час обробки файлового потоку?
29. Як видалити файл?
30. Описати функції для читання й запису даних **fread** та **fwrite**.
31. Що таке перенаправлення потоків?
32. Написати програму-фільтр, яка малі латинські літери замінює на великі. Текст вводиться з клавіатури й виводиться на екран. Скористатися функціями **islower(c)** та **toupper(c)**. Перша перевіряє, чи є символ малою латинською літерою, друга повертає код відповідної літери на верхньому регістрі.
33. Підрахувати кількість прочитаних із клавіатури: а) символів; б) символів табуляції (**'\t'**), пробілів (**' '**) і рядків (**'\n'**). Для завершення процесу читання ввести символ **'Ctrl+Z'**.
34. Написати програму-фільтр, яка копіює введені з клавіатури символи на екран і при цьому: а) прибирає зайві пробіли; б) розміщує по одному слову на рядок. В обох випадках підрахувати кількість введених із клавіатури слів.
35. Не всі зовнішні пристрої вміють інтерпретувати символ табуляції **'\t'**. Написати програми-фільтри для вилучення символу **'\t'** текстового файла й заміни його відповідною кількістю пробілів ' ' і навпаки – для вставки там це можливо символ табуляції і вилученням відповідної кількості про-

білів. Позиції для табуляції задаються константою **n** у командному рядку. Стандартне значення константи – 8.

36. Із клавіатури читається послідовність рядків. Надрукувати її в лексикографічному порядку.
37. Написати програму з іменем **tail**, що друкує **n** останніх введених рядків. За умовчанням **n=5**. Однак **n** може бути задано й у командному рядку. Командний рядок > **tail** – **n** запускає друкування останніх **n** рядків. У кожний момент зберігати в пам'яті не більше **n** рядків.
38. Написати програму-фільтр для копіювання текстового файла. Рядки тексту мають довжину не більше 80. Порожні рядки й рядки із символів-пробілів вилучаються. Вхідний файл: а) вводиться з клавіатури; б) міститься на диску.
39. Порівняти два файли й надрукувати перший рядок, в якому вони розрізняються.
40. Написати: а) довільну програму, б)* програму з лінійною часовою складністю пошуку текстових файлів, що містять входження даного слова. Слово та імена файлів подаються в командному рядку.
41. Написати програму для друкування кількох файлів. Кожний файл повинен розпочинатися з нової сторінки, містити заголовок і мати свою нумерацію сторінок. Розміри сторінки задаються у командному рядку.
42. Написати функцію для: а) копіювання файла в список його елементів у ОП; б) переписування у файл елементів заданого списку.
43. Написати програму для пошуку спільних рядків двох упорядкованих лексикографічно файлів. Програма має виводити рядки у три колонки: у першу – ті, що є тільки в першому файлі; у другу – ті, що є тільки в другому; у третю – спільні рядки обох файлів.
44. Написати програму, що будує дерево пошуку з чисел, поданих у файлі **sf.txt**, і виводить його: а) у префіксному; б) постфіксному; в) інфіксному порядках у файл **df.txt**.
45. Прокоментувати й налагодити програму:

```
#include <stdio.h>
#include <stdlib.h>
#define  N   5

int array[N];
#define FNAME "TEMP.C"

void main()
{
```

```
FILE *tmpf=fopen(FNAME, "w+b");  /* …  */
if(!tmpf){perror("Не можна відкрити файл");
exit(1);
}
for(int i=0; i<N; i++)
fwrite(&i, sizeof(int), 1, tmpf);  /* …  */

rewind(tmpf);  /* …  */
fread(&array, sizeof(int), 100, tmpf);  /* …  */
puts("Масив після читання з диска:\n");
for(int i=0; i<100; i++)printf("%8d", array[i]);

fclose(tmpf);

}
```

46*. Написати функції **int fwritef(FILE* fp,char *format,…)** та **int freadf(FILE* fp, char *format,…)** для запису в потік і читання з потоку послідовностей різнотипних даних. Скористатися макросами з бібліотеки **<stdarg.h>** для роботи з неоголошеними аргументами функції (див. підрозд. 3.6.7). Функції повертають значення так само, як їхні аналоги – функції **fprintf** та **fscanf**.

3.9. Препроцесор

- Директиви препроцесора
- Макропідстановки
- Включення файлів
- Умовна компіляція
- Директиви **#error**, **#line** та **#pragma**

Ключові слова: *препроцесор, директива препроцесора, простий і параметризований макроси, включення файлів, умовна компіляція, операції перетворення на рядок* **#** *і конкатенації* **##**, *директиви*: **#define**, **#endif**, **#ifdef**, **#line**, **#elif**, **#error**, **#ifndef**, **#pragma**, **#else**, **#if**, **#include**, **#undef**.

Препроцесор є складовою системи програмування мови С. Він отримує на вхід текст С-програми *з директивами,* виконує їх і вилучає з тексту. Окрім цього, фаза препроцесування включає:

1) відкидання пари символів: **** та **\n** (склеювання сусідніх рядків програми);

2) розбиття програми на лексеми. Коментарі при цьому замінюються одиничними пробілами;

3) заміну в символьних константах і літералах керівних послідовностей на їхні еквіваленти. Сусідні літерали конкатенуються.

Розглянемо приклад препроцесорної обробки С-програми, що не містить директив препроцесора. Лексеми розміщуються в дужках **<...>**.

Приклад 3.74. Нехай на вхід препроцесора надходить програма:

```
int main(void)
{
     printf("\n"); /*вихід із програми*/
}
```

Препроцесор поверне послідовність лексем:

<int>	Службове слово
< >	Пробіл
<main>	Ідентифікатор
<(>	Роздільник
<void>	Ідентифікатор
<)>	Роздільник
<{>	Роздільник
< >	Пробіл (один замість трьох)
<printf>	Ідентифікатор
<(>	Роздільник
<10>	Код символу **'\n'**
<)>	Роздільник
<; >	Роздільник
< >	Пробіл замість коментаря
<}>	Роздільник

■

3.9.1. ДИРЕКТИВИ ПРЕПРОЦЕСОРА

Директиви задають певні синтаксичні перетворення тексту програми й мають вигляд:

#<ім'я-директиви> <лексема-операнд>{<лексема-операнд>}

Вони розміщуються в окремих рядках і розпочинаються символом **'#'**. Область дії директив – від неї до кінця текстового файла. Якщо безпосередньо перед закінченням рядка поставити символ **'\'**, то директива продовжиться на наступний рядок програми.

У мові С визначено такі директиви препроцесора:

```
#define          #endif          #ifdef          #line
#elif            #error          #ifndef         #pragma
#else            #if             #include        #undef
```

3.9.2. МАКРОПІДСТАНОВКИ

Макрос (скорочення від макропідстановка) – це механізм заміни заданого ідентифікатора (імені макросу), можливо параметризованого, на пов'язану з ним послідовність символів. Макроси задаються директивою, що має такий вигляд:

```
<макрос>::=#define (<простий-макрос>|
<параметризований-макрос>)
<простий-макрос>::=<ім'я-макросу> <тіло-макросу>
<параметризований-макрос>::=
<ім'я-макросу>\(<параметр>{,<параметр>}\)<тіло-макросу>
<і'мя-макросу>::=<ідентифікатор>
<параметр>::=<ідентифікатор>
<тіло-макросу>::=<послідовність-символів>
```

Тіло макросу може бути довільним фрагментом С-програми, у тому числі й порожнім. У випадку *простого макросу* всі подальші входження його імені в текст програми як лексеми препроцесор замінить на тіло макросу – відповідну послідовність символів. Якщо тіло порожнє, то всі входження імені макросу в текст програму усуваються. Якщо ім'я макросу є частиною літерала, то заміна не відбудеться. Наприклад:

```
# define begin {/*усі входження begin замінити на {*/
# define end} /*усі входження end замінити на }*/
main()
begin
 int_begin=0;
  if (…) begin …… end
        else return 0;
end
```

На виході препроцесора цей фрагмент матиме вигляд:

```
main()
{int_begin=0;
  if(…) {……}
         else return 0;
}
```

Допускаються ланцюжки макропідстановок. Наприклад:

```
# define X 10
# define Y X+2
int main()
 {
int i;
          …
i=Y;
printf("X   Y");
          ...
 }
```

Тоді препроцесор спочатку здійснює таке перетворення:

```
 # define Y 10+2
 int main()
 {
int i;
          …
i=Y;
printf("X   Y");
....
 }
```

На виході препроцесора цей фрагмент матиме вигляд:

```
 int main()
 {
int i;
          …
i=10+2;
printf("X   Y");
 ...
 }
```

Нехай є програма, в якій визначаються масив і кілька процедур, що отримують доступ до нього. Замість того, щоб задавати розмір масиву безпосередньо в тексті програми, доцільно за допомогою директиви **#define** ввести для нього ім'я й конкретне значення. Тоді для зміни розміру масиву необхідно буде лише змінити це значення в директиві й перекомпілювати програму:

```
#define SIZE 100
/* ... */
float vect[SIZE];
```

```
/* ... */
for(i=0; i<SIZE; i++) printf("%f", vect[i]);
/* ... */
for(i=0; i<SIZE; i++) x=+vect[i];
```

Розмір масиву **vect** визначається іменем макросу **SIZE**, тому для зміни розміру потрібно лише змінити визначення **SIZE**. У результаті при перекомпіляції програми всі звернення до імені макросу будуть автоматично змінені.

Щоб скористатися параметризованим макросом, його необхідно викликати. Виклик макросу має вигляд:

```
<виклик-макросу>::=
<ім'я_макросу>(<фактичний_параметр>{,<фактичний-параметр>})
<фактичний_параметр>::=<послідовність-символів>
```

Кількість фактичних параметрів у виклику макросу має збігатися з кількістю його параметрів. Дія директиви полягає в заміні всіх його викликів на модифіковане тіло. Модифікація тіла – це заміна всіх входжень параметрів у ньому на їхні фактичні значення. Наприклад:

```
#define MAX(X,Y) (((X)>(Y))?(X):(Y))
 main()
 {
int i;
          …
i=MAX(i, 100)
....
 }
```

На виході препроцесора цей фрагмент матиме вигляд:

```
main()
 {int i;
 … i=(((i)>(100))?(i):(100)) ....
 }
```

Увага! Під часу виклику макросу можуть виникати проблеми, якщо в його тілі параметри чи деякі підвирази не взято в дужки ►

Розглянемо простий макрос і опис змінної за його участю:

```
#define N (-2)
    int p=4-N;
```

Якщо константу **-2** у тілі макросу не взяти в дужки, то для компілятора буде отримано опис **int p=4--2**;, який є помилковим.

Розглянемо інший параметризований макрос:

```
#define sqr(X) X*X
```

Тут виклик макросу **sqr(a+2)** у коді програми буде замінено препроцесором на вираз **a+2*a+2**, який не збігається з **(a+2)*(a+2)**! Здавалося б, правильним має бути такий варіант макросу:

```
#define sqr(X) (X)*(X)
```

Однак візьмемо вираз **1/sqr(a)**. Після препроцесування він перетвориться на **1/(a)*(a)** і після скорочення буде збігатися з **1**, а не з **1/(a*a)**, як мало б бути. Правильно було б записати макрос так:

```
#define sqr(X) ((X)*(X)).
```

Макроси з формальними параметрами мають суттєву перевагу перед функціями – вони швидше виконуються (немає потреби витрачати ресурси на виклик функцій). Проте, якщо макрос із формальними параметрами має великий розмір, то дублювання коду може занадто збільшити розмір машинної програми.

Область дії директиви **#define** можна обмежити директивою **#undef:**

```
# undef <ім'я-макросу>
```

Директива відміняє дію раніше визначеного макросу з даним іменем. Наприклад:

```
#define SIZE 100
float vect[SIZE];
# undef SIZE
/*тут і далі SIZE вже не визначено*/
```

Є дві операції препроцесора: **#** та **##**. Вони застосовуються разом із директивою **#define**.

Операція **#** називається *перетворенням на рядок* (перетворює запис аргументу на рядок). Розглянемо програму:

```
#include <stdio.h>
#define mkstr(s)#s

int main()
{
  printf(mkstr(TEST));
```

```
  return 0;
}
```

Препроцесор перетворює виклик `printf(mkstr(TEST));` на виклик `printf("TEST");`.

Операція `##` називається *конкатенацією* (склеює дві лексеми в одну). Розглянемо програму:

```
#include <stdio.h>
#define cont(a,b) a##b
int main()
{
  int x1=10;
  printf("%d", cont(x,1));
  return 0;
}
```

Препроцесор перетворює виклик `printf("%d", concat(x,1));` на виклик `printf("%d", x1);`.

3.9.3. ВКЛЮЧЕННЯ ФАЙЛІВ

Директива має два варіанти:

1) `#include <file>` – для файла із системної бібліотеки;
2) `#include "file"` – для довільного файла (указується його ім'я).

`<file>` задає ім'я файла за правилами ОС і може містити його маршрут у файловій системі. Якщо ім'я файла зазначене в лапках, то його пошук здійснюється відповідно до заданого маршруту, а за його відсутності – у поточному каталозі. Якщо ім'я файла задане в кутових дужках, то пошук відбувається в стандартних директоріях ОС, що задають командою `PATH`. Наприклад:

```
#include <stdio.h>
#include "c:\MyFile.h"
```

Заголовні файли можна включати в довільному порядку і скільки завгодно разів.

Файли, імена яких розташовані в директивах `#include`, можуть, у свою чергу, містити інші директиви `#include`. Вони називаються *вкладеними директивами* `#include`. Кількість припустимих рівнів вкладеності в різних компіляторах може бути різною. У стандарті С89 передбачено, що компілятори мають допускати не менше 8 таких рівнів, а в стандарті С99 – не менше 15 рівнів вкладеності.

Директива **#include** широко використовується для включення в програму заголовних файлів, що містять прототипи бібліотечних функцій, тому більшість С-програм розпочинаються з неї.

3.9.4. УМОВНА КОМПІЛЯЦІЯ

#if, #else, #endif, #ifdef, #ifndef

Є кілька директив, що дають можливість вибірково компілювати частини коду програми. Подібну компіляцію називають *умовною*. Вона може бути застосована, наприклад, для компілювання програми, яка має різні версії:

```
#if <ціл-конст-вираз>
 … text-1 ….
 #else
 … text-2 …
 #endif
```

Якщо значення цілого виразу істинне, то компілюється **text-1**, інакше – **text-2**.

Інша форма:

```
#if <ціл-конст-вираз>
 … text ….
#endif
```

Якщо значення виразу істинне, то компілюється **text**.

Значення константних виразів мають бути обчислені під час компіляції, тому в них можуть бути тільки раніше визначені препроцесорні та звичайні константи, але не змінні:

```
/*Простий приклад директиви #if*/
#include <stdio.h>

#define MAX 100
int main(){
#if
printf("Компілює фрагмент для MAX>50");
#else
printf("Компілює фрагмент для MAX≤ 50");
#endif
return 0;
}
```

Ця програма виводить на екран повідомлення:

```
Компілює фрагмент для MAX≤50
```

Директива **#elif** означає **"else if"**. Після **#elif** міститься **<ціл-конст-вираз>**; якщо він істинний, то компілюється відповідний блок і більше не перевіряються жодні вирази **#elif**, якщо ні – перевіряється наступний **#elif**. У загальному вигляді:

```
#if <ціл-конст-вираз_1>
 … text-1 …
#elif <ціл-конст-вираз_2>
… text-2 …
…
#elif <ціл-конст-вираз_N>
… text-N …
#endif
```

Відповідно до стандарту С89 директиви **#if** та **#elif** можуть містити 8 рівнів вкладеності, а стандарт С99 дозволяє використовувати 63 рівні. При вкладеності кожна директива **#endif, #else** або **#elif** приєднується до найближчої директиви **#if** або **#elif**. Наприклад, правильним є такий фрагмент коду:

```
#if MAX>100
  #if SERIAL_VERSION
    int port=198;
  #elif
    int port=200;
  #endif
#else
  char out_buffer[100];
#endif
```

Інший спосіб умовної компіляції – це використання директив **#ifdef** та **#ifndef**, що відповідно означають **"if defined"** ("якщо визначено") та **"if not defined"** ("якщо не визначено"). Загальний вигляд **#ifdef**:

```
#ifdef <ім'я_макросу>
… text-1 ….
#endif
```

Блок коду буде компілюватися, якщо ім'я макросу було визначене раніше в директиві **#define**. Загальний вигляд директиви **#ifndef**:

```
#ifndef <ім'я_макросу>
… text-1 ….
#endif
```

Блок коду буде компілюватися, якщо ім'я макросу ще не визначене в директиві **#define**.

В **#ifdef** та **#ifndef** можна використовувати директиви **#else** або **#elif**:

```
#ifdef <ім'я_макросу>
… text-1 ….
#elif <ім'я_макросу_2>
… text-2 ….
…
#elif <ім'я_макросу_N>
… text-N ….
#endif
```

Приклад такого використання наведено в програмі:

```
#include <stdio.h>
#define TED 10

int main()
{
      #ifdef TED
              printf("Привіт, Тед\n");
      #else
              printf("Привіт, хто-небудь \n");
      #endif
      #ifndef RALPH
              printf("A RALPH не визначений \n");
      #endif
      return 0;
}
```

У стандарті С89 допускається не більше 8 рівнів вкладеності **#ifdef** та **#ifndef**, у С99 – не більше 63.

3.9.5. ДИРЕКТИВИ #error, #line ТА #pragma

Директива **#error** змушує компілятор припинити компіляцію. Її загальний вигляд:

```
#error <повідомлення-про-помилку>
```

Наприклад: **#error TEST ERROR**

Коли зустрічається директива **#error**, то виводиться повідомлення про помилку.

Директива **#line** змінює вміст **__LINE__** та **__FILE__**, які є зарезервованими ідентифікаторами в компіляторі. У першому зберігається номер рядка коду, що компілюється в даний момент. Другий ідентифікатор – це рядок, що містить ім'я вихідного файла, який компілюється. Директива **#line** має вигляд:

```
#line <номер> <"ім'я-файла">,
```

де **номер** – додатне ціле число, що стає новим значенням **__LINE__**, а необов'язкове **ім'я-файла** – будь-який припустимий ідентифікатор файла, що стає новим значенням **__FILE__**.

Директива **#pragma** – це обумовлена реалізацією директива, що дозволяє передавати компілятору різні інструкції. Її можливості зазначаються в документації для компілятора.

***Література для СР:** препроцесор – [55, 102, 132, 133].

Контрольні запитання та вправи

1. Що таке препроцесор?
2. Що таке директива препроцесора?
3. Що таке простий макрос?
4. Що таке параметризований макрос?
5. Як викликається параметризований макрос?
6. Для чого використовуються макроси? Навести приклади застосування макросів.
7. Пояснити різницю між функцією й параметризованим макросом.
8. Що таке умовна компіляція?
9. Описати директиви умовної компіляції.
10. Описати операції перетворення в рядок # та конкатенації ##. Навести власні приклади їх застосування.
11. Як діють директиви **#line**, **#error** та **#pragma**?
12. Порівняти значення викликів стандартної функції **sqr** і макросу **#define sqr(X) ((X)*(X))** на аргументі **a++,** а також значення змінної **a** після цих викликів. Пояснити розбіжності в значеннях.
13. Написати макрос **tree_node (data_type)** для опису структури вузла бінарного дерева, в якому тип вершини задається параметром **data_type**.

14. Знайти й виправити помилки в директивах 1)-3):
```
#include \<stdio.h> #include <stdlib>
#define begin {;/*усі входження begin замінити на {*/
#define end} /*усі входження end замінити на}*/
```
15. Навести приклад застосування умовної компіляції для організації виведення налагоджувальної інформації на екран.

3.10. Порівняльний аналіз мов С та С++

- Стандартні бібліотеки С++
- Область дії змінних
- Простір імен
- Специфікатори класів пам'яті
- Введення-виведення в мові С++
- Робота з функціями в С++
- Константи й типи даних мови С++
- Посилання Керування пам'яттю

Ключові слова: *стандарти мови,* С++*, простір імен,* **std**, *оператори* **namespace** *та* **using**, *специфікатор* **mutable**, *бібліотека* **iostream**, *операції стандартного введення* **>>** *і виведення* **<<**, *об'єкти* **cin**, **cout** *та* **cerr**, *бібліотеки для роботи з файлами:* **fstream**, **ofstream** *та* **ifstream**, *методи* **seekg()** *та* **seekp()**, *прототип функцій у* С++, *аргументи функцій за умовчанням, вбудована функція, перевантажена функція, посилання, посилання як параметр і результат роботи функцій, оператори* **new()**, **delete()**, *тип* **bool**, *бібліотека рядків* **<string>**.

Програми, наведені до цього, створювалися на основі підпрограм. Такий *процедурно-орієнтований підхід* не завжди забезпечує потрібний рівень надійності та гнучкість програмування. Ускладнення розв'язуваних задач стимулювало розвиток іншого підходу – *об'єктно-орієнтованого*. Цей підхід уже згадувався у вступі й підрозд. 2.2.2, 2.7.3.

ООП на сьогодні є абсолютним лідером у прикладному програмуванні. Разом із тим у системному програмуванні досі переважає процедурна парадигма, і основною мовою є мова С. При взаємодії системного та прикладного рівнів ОС дедалі більше застосовуються мови ООП.

Мова С++ була створена Б. Страуструпом у 1979 р. як об'єктно-орієнтований варіант мови С. Надалі обидві мови розвивалися, впливаючи одна на одну. У 1998 р. був прийнятий ANSI/ISO-стандарт мови С++, побудований на основі С89, а в 1999 р. – новий ANSI/ISO-стандарт для мови С – С99. Ця версія мала нові можливості, деякі з них були запозичені зі стандарту С++ 1998 р., а деякі – нові. У С99 є незначна кількість конструкцій, не сумісних зі стандартом С++ 1998 р. Опис їх можна знайти в [133]. Якщо їх відкинути, то отримаємо власно С, що дає можливість писати С-програми таким чином, щоб вони задовольняли стандарти обох мов С/С++. Отже, мову С++ умовно можна розділити на три частини:

С++ = С + додаткові (не ООП) можливості + ООП

Розпочнемо розгляд мови С++ із другої частини, тобто з нових, не об'єктно-орієнтованих, її можливостей.

3.10.1. СТАНДАРТНІ БІБЛІОТЕКИ С++

Як і в С, у С++ немає операцій, що забезпечують введення-виведення, роботу з рядками, здійснюють математичні розрахунки тощо. Усі ці операції виконуються за рахунок використання набору бібліотечних функцій, що підтримуються компілятором. У С++ існують два основні типи бібліотек: бібліотеки С та С++. Бібліотеки в С++, як і в С, активізуються за допомогою директиви препроцесора **`#include`**, але їхні імена не мають розширення **`.h`**. Однак його додання не викличе помилки.

Значна частина бібліотеки С++ заснована на бібліотеці шаблонів **`STL`** (англ. – Standard Template Library). Вона надає такі важливі засоби, як контейнери – вектори (**`std::vector<>`**), списки **`(std::list<>)`**, стеки **`(std::stack<>)`**, черги **`(std::queue<>)`**, рядки **`(std::string)`**, множини (**`std::set<>`**) тощо – та ітератори, що дають доступ до контейнерів та інших інструментів. Детальніше з рядками можна ознайомитися нижче, а з контейнерними типами – у підрозд. 4.3.5.

У стандартній С++ уся інформація, пов'язана зі стандартною бібліотекою, визначена в просторі імен **`std`**. Для отримання прямого доступу до елементів стандартної бібліотеки потрібно перед назвою елемента зазначати ім'я простору, наприклад **`std::cout`**, або після включення потрібного заголовку використати оператор **`using`**:

```
using namespace std;.
```

Зауважимо, що деякі застарілі компілятори С++ не підтримують команду **`namespace`** і заголовки без зазначення розширення. При ви-

користанні заголовка в традиційному стилі С усі імена, визначені цим заголовком, розміщуються в глобальному просторі імен, а не в просторі **std**, тому оператор **using** не потрібен.

3.10.2. ОБЛАСТЬ ДІЇ ЗМІННИХ

У мові С++ і версії С99 локальні змінні можна оголошувати практично в довільному місці блока, головне, щоб змінна була оголошена перед її використанням:

```
int f()
{
    int x=10;
    x++;
int n=100;
for(int i=0; i<n; i++)
{
    x+=n*i;
}
return x;
}
```

Глобальні змінні мають оголошуватися поза всіма функціями, у тому числі й **main()** (зазвичай такі змінні розташовуються на початку файла, перед функцією **main()**).

3.10.3. ПРОСТІР ІМЕН

У мові С++ можна створювати простори імен оператором **namespace**. Простір імен визначає деяку декларативну область, що обмежує дію імен. Простори імен потрібні для уникнення колізій між пакетами, які мають збіжні імена глобальних змінних, функцій і типів:

```
<простір-імен>::=namespace <ім'я_простору_імен> {
<фрагмент-програми>
}
```

На імена, оголошені всередині деякого простору імен, можуть напряму посилатися інші оператори всередині цього простору.

Приклад 3.75. Робота з простором імен:

```
namespace MyNamespace {
const int a=5;
```

```
int count;
typedef int**T;
void f(x) {return a*x};
...
}
```

Поза простором імен для звернення до змінної додається ім'я простору:

```
MyNamespace::a, MyNamespace::count, MyNamespace::T, MyNamespace::f.
```

Якщо в деякому файлі потрібно звернутися до змінних, визначених у іншому просторі імен, то використовують оператор **using**:

```
using namespace MyNamespace; /*використання простору MyNamespase*/

using MyNamespace::T; /*використання змінної T із простору MyNamespase*/
```

■

Спеціальним випадком є безіменний простір імен. Усі імена, описані в ньому, доступні в поточному файлі й більше ніде:

```
namespace {
           ...
          }
```

3.10.4. СПЕЦИФІКАТОРИ КЛАСІВ ПАМ'ЯТІ

Мовою С++ успадковані від мови С такі специфікатори класів пам'яті: **extern**, **auto**, **register**, **static**, що використовуються для зміни способу створення пам'яті для змінних. У мові С++ при використанні специфікатора **static** для даних – членів класу – створюється лише одна копія цих членів, яка сумісно використовується всіма об'єктами. До того ж у С++ з'явився новий специфікатор **mutable**, що дозволяє специфікованому таким чином атрибуту об'єкта, кваліфікованому як **const**, бути модифікованим.

3.10.5. ВВЕДЕННЯ-ВИВЕДЕННЯ В МОВІ С++

Частиною стандартної бібліотеки С++ є бібліотека **<iostream>**, що реалізована як ієрархія класів і забезпечує базові можливості введення-виведення. Для її використання потрібна директива **#include <iostream>** і відкриття доступу до відповідного простору імен: **using namespace std.**

Введення з клавіатури називається *стандартним введенням* і прив'язане до об'єкта **cin**. Стандартне виведення на екран прив'язане до об'єкта **cout**. *Стандартне виведення помилок* прив'язане до об'єкта **cerr**. У мові С замість них використовуються стандартні потоки **stdin**, **stdout** та **stderr**. Щоб значення потрапило в стандартний потік, використовується операція *виведення* **<<**. Наприклад:

```
int i_x=10;
cout<<"i_x=="<<i_x<<'\n';
```

Замість символу **'\n'** можна використовувати стандартний операнд **endl**, що виводить символ переходу на новий рядок і здійснює дозаписування решти символів із буфера виведення:

```
int i_x=10;
cout<<"i_x=="<<i_x<<endl;
```

Для читання значення зі стандартного потоку застосовується операція *введення* **>>**. Наприклад:

```
string name;
int age;
…
cout<<"Введіть ім'я та вік:";
cin>>name>>age;
```

Файлове введення-виведення. Бібліотека **iostream** підтримує також файлове введення й виведення (аналогічно перенаправленню потоків у С). Для використання файлів потрібно підключити ще одну бібліотеку:

```
#include <fstream>.
```

Для роботи з файлами використовуються такі класи: **ofstream**, **ifstream**, **fstream**.

Перед відкриванням файла для виведення потрібно оголосити об'єкт типу **ofstream**:

```
/*потік виведення, відкритий у режимі за умовчанням*/
ofstream outfile("temp.out");
```

Файл типу **ofstream** за умовчанням відкривається в режимі виведення **ios_base::out** або режимі дозаписування **ios_base::app**:

```
/*потік виведення, відкритий у режимі дозаписування*/
ofstream outfile("temp.out", "ios_base::app");
```

Перед відкриванням файла для введення потрібно оголосити об'єкт типу **ifstream**:

```
ifstream outfile("temp.in");
```

Для перевірки коректності відкривання файла перевіряємо об'єкт **outfile**:

```
if(!outfile)
{//файл не відкрито
…
}
```

Для закривання файла потрібно викликати функцію (член класу) **close()**.

Об'єкти класів **ofstream** та **ifstream** можна визначати без зазначення імені файла. Пізніше до цього об'єкта можна приєднати ім'я файла функцією **open()**:

```
ifstream f1;
…
f1.open("file1");
…
f1.close();
```

Об'єкт класу **fstream** може відкривати файл для введення чи виведення. Наприклад, відкриємо файл для введення й дозаписування:

```
fstream f2("file2", ios_base::in|ios_base::app);
```

Об'єкт цього класу можна позиціонувати за допомогою методів **seekg()** (для читання) чи **seekp()** (для запису).

Для позиціонування у фіксовану позицію у файлі використовується формат команди

```
seekg(pos_type position);
```

Для зміщення відносно поточної позиції використовується формат команди

```
seekg(off_type position, ios_base::seekdir dir);
```

Тут **dir** може набувати таких значень:

ios_base::beg – від початку файла;

ios_base::cur – від поточної позиції;

ios_base::end – від кінця файла.

Приклад 3.76. Позиціонування:

```
fstream f2("file2", ios_base::in|ios_base::app);
f2.seekp(4); /*позиціонування на четвертий символ*/
f2.seekp(10, ios_base::beg); /*зміщення на 6 символів від початку файла*/
```

■

Розглянемо приклад програми, що працює з файлами.

Приклад 3.77.

Лістинг.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main()
{
    int i;
    ifstream infile("c:\\TEMP\\file1.txt");
    ofstream outfile("c:\\TEMP\\file2.txt");
    if((!infile)||(!outfile))
    {
          cerr<<"Error.\n";
          return -1;
    }
    string word;
    while(infile>>word)
          outfile<<word<<' ';
    outfile.close();
    infile.close();

    fstream iofile("c:\\TEMP\\file3.txt");
    if(!iofile)
    {
          cerr<<"Error.\n";
          return -1;
    }

    iofile.seekp(6, ios_base::beg);
    iofile<<"!!!!"<<' ';

    iofile.seekp(6, ios_base::end);
    iofile<<"????"<<' ';
    iofile.close();
```

```
    return 0;
}
```

Значення файлів:

```
C:\temp\file1.txt
aaaaaaa

bbbb c dddd   eeeeeee fff ggg

C:\temp\file2.txt
aaaaaaa bbbb c dddd eeeeeee fff ggg

C:\temp\file3.txt /*до запуску програми*/
слово1 слово2    слово3   слово4   слово5

C:\temp\file3.txt /*після виконання програми*/
слово1???? o2    слово3   слово4   слово5
```

■

3.10.6. РОБОТА З ФУНКЦІЯМИ В С++

Прототипи функцій. У мові С++ усі функції повинні мати прототип. У С99 для задання прототипу функції, що не має параметрів, замість їхнього списку використовується порожній тип **void,** наприклад **int f(void);**. У С++ уживання **void** у даному випадку не обов'язкове (дозволене, але зайве). Якщо функція поряд із фіксованими параметрами має й нефіксовані, то потрібно користуватися, як і в С, трьома крапками. Наприклад: **int printf(const char* fmt, ...);**

Аргументи функцій за умовчанням. У С++ один або кілька останніх аргументів функції можуть задаватися за умовчанням. Ця властивість використовується, якщо при виклику функції потрібна лише частина параметрів, а не всі. Нехай нам потрібна функція, що повертає суму п'яти цілих чисел, наприклад:

```
int sum(int x1, int x2, int x3, int x4, int x5)
{
    return x1+x2+x3+x4+x5;
}
```

Для виклику функції лише з трьома аргументами потрібно доповнити нулями ті аргументи, що не використовуються:

```
int x=2, y=3, z=10, rez ;
rez=sum(x, y, z, 0, 0);
```

Такий виклик функції виглядає неприродним. Ситуація спроститься, якщо в прототипі функції записати

```
int sum(int x1, int x2, int x3=0, int x4=0, int x5=0);
```

Тоді ми можемо використовувати виклики:

```
int rez;
rez=sum(2, 3, 10); /* x1==2, x2==3, x3==10, x4==0, x5==0*/
rez=sum(2, 3); /* x1==2, x2==3, x3==0, x4==0, x5==0*/
rez=sum(2, 3, 10, 4); /* x1==2, x2==3, x3==10, x4==4, x5==0*/
rez=sum(2, 3, 10, 4, 5); /* x1==2, x2==3, x3==10, x4==4,
x5==5*/
```

Вбудовані функції. Як і мова C, C++ містить вбудовані функції `inline`. Оголошуючи вбудовану функцію, ми даємо команду компілятору там, де це можливо, вставляти тіло функції безпосередньо в місце її виклику. Методи класів також можуть бути вбудованими.

Приклад 3.78. Опис вбудованої функції.

Лістинг.

```
#include <iostream>
using namespace std; /*робить видимими імена простору std, зокрема cout*/
inline int max(int a, int b);
int main()
{
    int x, y, z;
cout<<"Input x:";
cin>>x;
cout<<"Input y: ";
cin>>y;
z=max(x, y);
cout<<"max("<<x<<","<<y<<")=="<<z<<endl;
return 0;
}
//команда компілятору за можливості
//вбудувати функцію в місце виклику
inline int max(int a, int b)
{
    if(a>b)
return a;
else return b;
}
```

■

Перевантаження функцій. У мові С++ функції можуть бути *перевантажені*. Це означає, що дві чи більше функцій можуть мати однакові імена, але відрізнятися кількістю чи типами параметрів (саме аналізуючи аргументи функцій, компілятор С++ визначає, яка функція викликається). Типи значень, що повертаються, можуть збігатися.

Наведемо простий приклад програми для демонстрації того, як перевантаження функцій спрощує текст програми.

Приклад 3.79. Перевантаження функцій.

Лістинг.

```
#include <iostream>//для cout
using namespace std; /*робить видимими імена простору std, зокрема cout*/

int cube(int a);
double cube(double a);

int main()
{
    int i_x=5;
    double d_x=10.2;

    cout<<cube(i_x)<<"\n";
    cout<<cube(f_x)<<"\n";
    return 0;
}

int cube(int a)
{
    return a*a*a;
}

double cube(double a)
{
    return a*a*a;
}
```

■

3.10.7. КОНСТАНТИ Й ТИПИ ДАНИХ МОВИ С++

У мові С++, як і в С, виділяють базові й похідні типи даних. Деякі типи даних мови С++ успадковані від С, тому тут вони розглядатися не будуть. Розглянемо нові типи даних мови С++ і зміни у визначенні типів, запозичених із С.

bool. У C++ визначено дві булеві константи: **true** та **false**, а також додано новий тип **bool**, що має значення **true** та **false**. Операції порівняння повертають тип **bool**. Зокрема, предикати після **if**, **while** зводяться до типу **bool**. Зауважимо, що у версії C99 мови C введено аналогічний тип під назвою **_Bool**, але не визначено булеві константи (у **<stdbool.h>** визначено макроси **true** та **false**).

Рядки. У C++ підтримуються два типи рядків: вбудований тип, успадкований від C – символьні масиви **char[]** та **wchar_t[]** для широких символів і, відповідно, класи **string** та **wstring** зі стандартної бібліотеки C++. Насправді класи **string** та **wstring** інстанційовані з шаблонного класу **basic_string** оголошеннями

```
typedef basic_string<char> string:
typedef basic_string<wchar_t> wstring:
```

C++-рядки визначені в бібліотеці **<string>.** Усі імена, визначені в ній, входять до стандартного простору імен **std.**

У класі **string** є кілька конструкторів. Наведемо приклади створення об'єктів класу **string** різними способами:

```
string str;  //створює порожній рядок
string str(cstr);  //створює рядок за C-рядком
string str(cstr, len); //створює рядок із len символів C-рядка
string str(num, ch);  //створює рядок із num символів ch
string str(s);  //створює рядок-копію за рядком s
string str(s, ind);  /*створює рядок із символів рядка s, починаючи з індексу ind*/
string str(str, ind, len); /*створює рядок символів рядка str, починаючи з індексу ind, довжина створюваного рядка не більше len*/
string str(beg, end, s); /*створює рядок з інтервалу [beg, end) символів рядка s*/
```

Для об'єктів класу **string** визначені такі операції:

Операція	Дія
=	Присвоювання. Визначено три типи присвоювання: рядку присвоюється значення C++-рядка; значення C-рядка; значення символу
+	Конкатенація
==	Рівність
!=	Нерівність
<	Менше
<=	Менше чи дорівнює
>	Більше
>=	Більше чи дорівнює

Операція	Дія
`[]`	Індексація
`<<`	Виведення
`>>`	Введення
`+=`	Додавання

Методи класу **`string`**:

Метод	Призначення
`append`	Додає символи в кінець рядка
`assign`	Призначає нові значення символам рядка
`at`	Повертає посилання на елемент у зазначеній позиції в рядку
`begin`	Повертає ітератор, що адресує перший елемент у рядку
`c_str`	Перетворює вміст рядка на С-рядок
`capacity`	Повертає найбільшу кількість елементів, що можуть бути записані в рядок, без додаткового виділення пам'яті
`clear`	Видаляє всі елементи рядка
`compare`	Порівнює рядок із зазначеним рядком, щоб визначити, чи є два рядки лексикографічно рівними
`copy`	Копіює максимальну кількість символів від індексованої позиції в рядку
`data`	Перетворює вміст рядка в масив символів
`empty`	Перевіряє, чи містить рядок зазначені символи
`end`	Повертає ітератор, що адресує позицію, наступну за останнім елементом у рядку
`erase`	Видаляє елемент чи зазначений діапазон елементів у рядку
`find`	Шукає найбільш ліве входження зазначеного рядка в даному рядку, починаючи із заданої позиції. Повертає позицію входження чи **`npos`**, якщо входження не знайдене
`find_first_not_of`	Шукає перший символ у рядку, який не є символом зазначеного рядка
`find_first_of`	Шукає перший символ у рядку, який є символом зазначеного рядка
`find_last_not_of`	Шукає останній символ у рядку, який не є символом зазначеного рядка
`find_last_of`	Шукає останній символ у рядку, який є символом зазначеного рядка

Метод	Призначення
insert	Вставляє елемент, елементи чи діапазон елементів у рядку в зазначену позицію
length	Повертає поточну кількість елементів у рядку
max_size	Повертає максимальну кількість елементів, яку може містити рядок
push_back	Додає елемент у кінець рядка
rbegin	Повертає ітератор першого елемента в зміненому рядку
rend	Повертає ітератор символу, що йде за останнім у зміненому рядку
replace	Заміняє елементи в рядку в зазначених позиціях символами, скопійованими з іншого рядка чи із С-рядка
reserve	Установлює місткість рядка не меншою ніж зазначене число
resize	Визначає новий розмір рядка, за необхідності додає чи видаляє елементи
rfind	Шукає останнє входження символу чи рядка, починаючи з позиції
size	Повертає поточну кількість елементів у рядку
substr	Копіює підрядок рядка, що складається максимум із зазначеної кількості символів і розпочинається із зазначеної позиції
swap	Обмінює вміст двох рядків

Тут **size_type** – визначений у **std** синонім типу **unsigned int**, **npos** – константа, що задає максимально можливе число. Методи визначені для різних аргументів (перевантаження), тому типи аргументів не зазначаються, за необхідності з ними можна ознайомитися в документації.

Приклад 3.80. Робота з класом **string**:

```
#include <iostream>
#include <string>

int main()
{
   using namespace std;
   string s1 ("one"); //створення рядка
   string s2 ("two"); //створення рядка

   cout<<"s1="<<s1<<"."<<endl;
```

```
    cout<<"s2="<<s2<<"."<<endl;

    s1.swap (s2); //обмін значеннями рядків

    cout<<"s1.swap (s2):"<<endl;
    cout<<"s1="<<s1<<"."<<endl;
    cout<<"s2="<<s2<<"."<<endl;

    cout<<"s1[2]="<<s1.at(2)<<endl; /*виведення на екран значен-
ня 2-го елемента рядка (елементи нумеруються з 0)*/

    s1.at(2)='O'; /*присвоювання 2-му елементу рядка значення*/

    cout<<"s1[2]="<<s1.at(2)<<endl; /*виведення на екран значен-
ня 2-го елемента рядка (елементи нумеруються з 0)*/

    cout<<"s1="<<s1<<endl; /*виведення на екран рядка s1*/

    cout<<"s1.find(n)="<<s1.find("n")<<endl; /*виведення на ек-
ран позиції символу n у рядку s1*/

    cout<<"s1.find(w)="<<s1.find("w")<<endl; /*виведення на ек-
ран позиції символу w у рядку s1*/

    s1.clear(); //очищення рядка
    cout<<"s1="<<s1<<endl;

    return 0;
}
```

Результат виконання програми:

```
s1=one
s2=two
s1.swap (s2):
s1=two
s2=one
s1[2]=o
s1[2]=O
s1=twO
s1.find(n)=4294967295
s1.finf(w)=1
s1=
```

■

3.10.8. ПОСИЛАННЯ

У мові С++ є спеціальні покажчики – посилання, що, на відміну від звичайного покажчика, постійно прив'язані до об'єкта, на який посилаються. Посилання ініціалізується покажчиком на існуючий об'єкт даного типу.

Посилання завжди вказує на одну й ту саму адресу.

Увага! При зверненні до посилання операція розіменування * проводиться автоматично ►

```
<специфікатор_посилання>::=<посилання> <прямий-специфікатор>
<посилання>::=&[<кваліфікатори-типу>] [<посилання>]
```

Посилання зазвичай використовуються як змінні, параметри й результати функцій.

Посилання як змінні. Наведемо програму, яка використовує посилання як змінні.

Приклад 3.81.

Лістинг.

```
#include <iostream> // для cout
using namespace std; /*робить видимими імена простору std, зокрема cout*/
int main()
{
    int i_x=10;
    int i_y=5;
    int *p_ix=&i_x; /*покажчику присвоєно адресу змінної i_x*/
    int &r_ix=i_x; //посилання, асоційоване з i_x
    int *p      =&r_ix; /*покажчику присвоєно адресу r_ix*/

    cout<<"i_x=="<<i_x<<'\n';
    cout<<"*p_ix=="<<*p_ix<<'\n';
    cout<<"r_ix=="<<r_ix<<'\n';
    cout<<"*p=="<<*p<<'\n';

    r_ix=22;
    cout<<"i_x=="<<i_x<<'\n';
    cout<<"*p_ix=="<<*p_ix<<'\n';
    cout<<"r_ix=="<<r_ix<<'\n';
    cout<<"*p=="<<*p<<"\n";

    i_x=33;
```

```
    cout<<"i_x=="<<i_x<<'\n';
    cout<<"*p_ix=="<<*p_ix<<'\n';
    cout<<"r_ix=="<<r_ix<<'\n';
    cout<<"*p=="<<*p<<'\n';

    p_ix=&i_y;
    cout<<"i_x=="<<i_x<<'\n';
    cout<<"*p_ix=="<<*p_ix<<'\n';
    cout<<"r_ix=="<<r_ix<<'\n';
    cout<<"*p=="<<*p<<"\n";

    return 0;
}
```

У результаті роботи програми маємо:

```
i_x==10
*p_ix==10
r_ix==10
*p==10
i_x==22
*p_ix==22
r_ix==22
*p==22
i_x==33
*p_ix==33
r_ix==33
*p==33
i_x==33
*p_ix==5
r_ix==33
*p==33
```

■

Посилання як параметри й результат роботи функцій. Посилання можуть бути параметрами й результатом роботи функції.

Приклад 3.82.

Лістинг.

```
#include <iostream>//для cout
#define SIZE 10
using namespace std; /*робить видимими імена простору std, зокрема cout*/

double array[SIZE];
```

```
  //функція, що демонструє роботу з посиланнями
  double &f(double &x, int index); /*посилання як аргумент і ре-
зультат*/
  //ініціалізація масиву
  void setArray();
  //друкування масиву
  void printArray();
  int main()
  {
      setArray();
      printArray();
      double x=0;
      double &p=array[1];
      cout<<"x=="<<x<<'\n';
      cout<<"p=="<<p<<'\n';
      f(p, 6)=x; //присвоювання функції посиланню
      cout<<"x=="<<x<<'\n';
      cout<<"p=="<<p<<'\n';
      printArray();
      return 0;
  }

  double &f(double &x, int index)
  {
      x=22; //присвоювання аргументу посиланню
      if((index>=SIZE)||(index<0))
            index=0;
      return array[index];
  }

  void setArray()
  {
      for(int i=0; i<SIZE; i++)
            array[i]=i;
  }

  void printArray()
  {
      for(int i=0; i<SIZE; i++)
            cout<<"array["<<i<<"]=="<<array[i]<<'\n';
  }
```

У результаті виконання програми матимемо:

```
array[0]==0
array[1]==1
array[2]==2
```

```
array[3]==3
array[4]==4
array[5]==5
array[6]==6
array[7]==7
array[8]==8
array[9]==9
x==0
p==1
x==0
p==22
array[0]==0
array[1]==22
array[2]==2
array[3]==3
array[4]==4
array[5]==5
array[6]==0
array[7]==7
array[8]==8
array[9]==9
■
```

3.10.9. КЕРУВАННЯ ПАМ'ЯТТЮ

Замість функцій **malloc()** та **free(),** що залишилися для сумісності із C, мова C++ пропонує альтернативні засоби – оператори **new()** та **delete().**

Виділення пам'яті для одного об'єкта (кількох однотипних об'єктів) типу **<type>** здійснює оператор **new <type>** (**new <type> [<ціле_число>]**). В обох випадках результат – покажчик типу **<type>*.**

Звільнення пам'яті для одного об'єкта (кількох об'єктів) здійснює оператор **delete <name>** (**delete []<name>**). Якщо аргументом є **NULL,** то в обох випадках він діє як порожній оператор, інакше руйнує об'єкт, на який посилається покажчик **<name>,** і звільняє область пам'яті, виділену для нього раніше оператором **new()** (руйнує кожен з об'єктів масиву, на який посилається покажчик **<name>,** і звільняє область пам'яті, виділену для цього масиву раніше операцією **new <type>[<ціле число>]**).

У всіх наведених вище операторах аргумент може братися в круглі дужки – наприклад, **new (<type>).**

Приклад 3.83. Керування пам'яттю:

```
double *dp=new double; //можна писати new(double)
*dp=25.3;
delete dp; //можна писати delete (dp)
```

■

***Література для СР:** мова С++ – [55, 57, 64, 70, 73, 102, 124].

Контрольні запитання та вправи

1. Які стандарти мови С++ ви знаєте?
2. Що таке простір імен і як його використовувати?
3. Що таке оператори **namespace** та **using**?
4. Що таке клас пам'яті **mutable**?
5. Описати тип **bool.**
6. Описати основні операції й методи класу рядків **string.**
7. Які особливості введення-виведення в мові С++?
8. Що таке бібліотека **<iostream>**?
9. Які є бібліотеки для роботи з файлами?
10. Визначити операції стандартного введення **>>** і виведення **<<**.
11. Що таке об'єкти **cin**, **cout** та **cerr**?
12. Яке призначення методів **seekg()** та **seekp()**?
13. Порівняти прототип функцій у мовах С та С++.
14. Як задавати аргументи функцій за умовчанням?
15. Що таке перевантаження функцій, яке його призначення й застосування?
16. Наведіть означення посилання, його відмінності від покажчика.
17. Навести приклади застосувань посилань як параметрів і результатів роботи функцій.
18. Які методи керування пам'яттю в мові С++ ви знаєте?
19. Порівняти в С++ два файли й надрукувати перший рядок, в якому вони відрізняються.
20. Написати С++-програму пошуку текстових файлів, що містять входження даного слова. Слово та імена файлів вводяться з клавіатури.
21. Написати С++-програму для друкування кількох файлів. Кожний файл має розпочинатися з нової сторінки, містити заголовок і мати свою нумерацію сторінок.
22. Написати С++-програму для пошуку спільних рядків двох упорядкованих лексикографічно файлів. Програма має виводити рядки в три колонки: у першу – ті, що є тільки в першому файлі; у другу – ті, що є тільки в другому; у третю – спільні рядки обох файлів.

23. Написати С++-програму для заміни в англійському тексті входження артикля "a" на "the".
24. У заданому тексті програми видалити коментарі (визначені за правилами С++).
25. Написати С++-програму для заміни в тексті входження символу ‘1' на слово "один".
26. У тексті програми замінити кожний ідентифікатор **begin** на ‘**{**', а ідентифікатор **end** – на ‘**}**'.

3.11. Об'єктно-орієнтовані можливості С++

- Інкапсуляція, класи та об'єкти
- Ініціалізація, присвоювання та знищення класу
- Спадкування
- Поліморфізм

Ключові слова: *клас, атрибути, методи, об'єкт класу, публічні, захищені та приватні атрибути й методи, вбудовані методи, конструктор, деструктор, публічне, захищене й приватне спадкування, клас-нащадок, базовий клас, множинне спадкування, раннє й пізнє зв'язування, віртуальний метод, абстрактний клас.*

3.11.1. ІНКАПСУЛЯЦІЯ, КЛАСИ ТА ОБ'ЄКТИ

У мові С основним засобом структуризації даних є структури. Наприклад, структура, призначена для роботи з датою, має вигляд:

```
struct CDate {
int day;
int month;
int year;
} ;
```

Мова С++ дозволяє опустити ключове слово **struct** перед **TDate** при оголошенні змінних. С++-програмі, що працює з такою структурою, будуть потрібні функції відображення й порівняння дат тощо.

Лістинг.

```
struct CppDate{
    int day;
```

```
    int month;
    int year;
};

/*порівняння дат: 0 – рівні, 1 – перша більша, -1 – друга більша*/
int comparison(CppDate *dt1, CppDate *dt2);

//друкування дати
void printDate(CppDate *dt);

//установлення дати
void setDate(int d, int m, int y);

/*порівняння дат: 0 – рівні, 1 – перша більша, -1 – друга більша*/
int comparison(CppDate *dt1, CppDate *dt2)
{
    if(dt1->year>dt2->year) return 1;
    if(dt1->year<dt2->year) return -1;
    if(dt1->month>dt2->month) return 1;
    if(dt1->month<dt2->month) return -1;
    if(dt1->day>dt2->day) return 1;
    if(dt1->day<dt2->day) return -1;
    return 0;
}

//друкування дати
void printDate(CppDate *tm)
{
    char buffer[32];
    sprintf(buffer, "Date: %02d.%02d.%04d\n",
tm->day, tm->month, tm->year);
    cout<<buffer;
}

//установлення дати
void setDate(CppDate *tm, int d, int m, int y)
{
    tm->day=d;
    tm->month=m;
    tm->year=y;
}
■
```

При такій роботі з даними структурного типу виникає низка проблем. Наприклад, поля структур типу **CppDate** не захищені від некоректних модифікацій і використання. Немає загального механізму об-

меження доступу до зміни значень елементів структури. Як наслідок це може викликати внесення неправильних дат (напр.: `{35, 35, 35000}`). Звичайно, можна замість `int` використовувати деякі його обмеження, але це не може адекватно розв'язати проблему різної кількості днів у місяцях тощо. Крім того, нехай після реалізації певної кількості функцій із використанням цієї структури ми вирішили, що зручніше буде змінити спосіб збереження часу, а саме зберігати кількість днів, починаючи від фіксованої дати. Для цього потрібно:

• змінити структуру `CppDate`, видаливши старі атрибути й додавши довге ціле (можна взагалі не використовувати структуру);

• змінити всі функції, що використовують `CppDate`, а також модифікувати всі оператори, які оперують компонентами структури `CppDate`.

Для великих програм (кілька тисяч рядків) здійснити такі зміни досить складно, знадобиться повторне налагодження.

У процедурному програмуванні внутрішнє подання даних визначається на ранній стадії розробки, що обмежує свободу внесення змін. Використовуючи ООП, можна усунути подібні проблеми. В ООП форма зображення даних може змінюватися з обмеженим впливом на код.

Основним засобом організації даних у мові С++ є класи. Зовні цей тип даних схожий на тип структур, однак полями (членами) у класі можуть бути не лише дані, але й методи (функції):

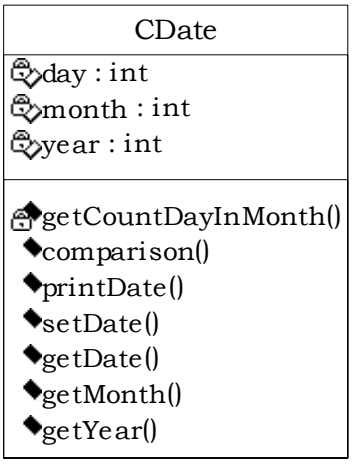
```
   <клас>::=class <ім'я-класу>[:<список-батьківських класів>] \
   {<список-секцій>};
<батьківський-клас>::=(public  <ім'я-класу>|private  <ім'я-класу>|
protected <ім'я-класу>)
   <секція>::=(public: <список-атрибутів-і-методів>|
   protected: <список-атрибутів-і-методів>|
   private: <список-атрибутів-і-методів>)
   <арибут>::=<опис-змінної>
   <метод>>::=<прототип-функції>|<опис-функції>
```

Клас інкапсулює свої елементи. Атрибути й методи (функції) бувають публічними (`public`), захищеними (`protected`) і приватними (`private`). До публічних атрибутів і методів є повний доступ для зміни й читання як із самого класу, так і ззовні. До захищених і приватних атрибутів і методів не можна звертатися ззовні класу для збереження цілісності даних класу. Спроба неправильного звернення викликає помилку компіляції. До таких атрибутів і методів можуть звертатися лише функції-члени класу, а також функції-друзі (див. підрозд. 3.12.3) і функції-члени класів-друзів. Зокрема, до захищених атрибутів можна звертатися із самого класу й класів-нащадків, а до закритих – лише із самого класу.

У C++ тип структури аналогічний типу класу, відмінність лише в тому, що за умовчанням атрибути й методи – члени у структурі публічні, а в класі – приватні.

Більшість класів краще оголошувати в заголовних файлах, які потім можуть включатися в численні програмні модулі для сумісного використання класів.

Якщо останню програму переписати, використовуючи класи, то утвориться клас, зображений UML-діаграмою:



Перепишемо програму мовою C++, використовуючи класи.

Приклад 3.84.

Лістинг.

```
class CppDate{
private:
    int day;
    int month;
    int year;
    //кількість днів у місяці m року y; якщо рік і місяць неко-
ректні, то 0
    int getCountDayInMonth(int m, int y);

public:
    /*порівняння дат: 0 – рівні, 1 – перша більша, -1 – друга
більша*/
    int comparison(CppDate dt1, CppDate dt2);
    //друкування дати
    void printDate();

    //установлення дати
```

```
    void setDate(int d, int m, int y);
    //повернення дня
    int getDay();
    //повернення місяця
    int getMonth();
    //повернення року
    int getYear();
};

/*порівняння дат 0 – рівні, 1 – перша більша, -1 – друга більша*/
int CppDate::comparison(CppDate dt)
{
    if(year>dt.year) return 1;
    if(year<dt.year) return -1;
    if(month>dt.month) return 1;
    if(month<dt.month) return -1;
    if(day>dt.day) return 1;
    if(day<dt.day) return -1;
    return 0;
}

//друкування дати
void CppDate::printDate()
{
    char buffer[32];
    sprintf(buffer, "Date: %02d.%02d.%04d\n", day, month, year);
    cout<<buffer;
}

//ініціалізує дату – за умови некоректних даних присвоює нульові елементи
void CppDate::setDate(int d, int m, int y)
{
    if((d>=1)&&(d<=getCountDayInMonth(m, y)))
    {
          day=d;
          month=m;
          year=y;
    }
    else
          {
                day=0;
                month=0;
                year=0;
          }
}
```

```
  //кількість днів у місяці m року y; якщо рік і місяць некоректні, то 0
  int CppDate::getCountDayInMonth(int m, int y)
  {
      if((y>=1900)&&(y<=3000))
      {
      switch (m)
  {
  case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
  return 31;
  case 4:
            case 6:
            case 9:
            case 11:
  return 30;
            case 2:
                  if(!(y%4)&&((y%100)))
                        return 29;
                  else
                        return 28;
  default:
  return 0;
  }
      }
      else
            return 0;

  }

  int CppDate::getDay()
  {
      return day;
  }

  int CppDate::getMonth()
  {
      return month;
  }

  int CppDate::getYear()
  {
```

```
    return year;
}
```

Зауважимо: атрибути **day, month, year** приватні, – це гарантує, що жодним іншим чином у програмі, окрім методів, описаних у класі (тут **setDate()**), не можна отримати доступ до них. У такий спосіб гарантується перевірка правильності введення даних. Для зміни подання дати (напр., кількості днів від заданої дати) потрібно лише внести зміни в клас і відредагувати методи для роботи з новою структурою. Якщо при цьому зберегти інтерфейс методів, то зовнішнє застосування класу не зміниться ■

Приклад 3.85. Використання вбудованих (**inline**) методів.

У заданому вище описі класу в його тілі зазначено лише заголовки методів (функцій), а можна описати в тілі класу весь метод (він називається *вбудованим* – **inline**). Наприклад, у розглянутому вище прикладі доцільно описати клас таким чином:

Лістинг.

```
class CppDate{
private:
    int day;
    int month;
    int year;
    //кількість днів у місяці m року y; якщо рік і місяць не ко-
ректні, то 0
    int getCountDayInMonth(int m, int y);

public:
    /*порівняння дат: 0 – рівні, 1 – перша більша, -1 – друга
більша*/
    int comparison(CppDate dt2);

    //друкування дати
    void printDate();

    //установлення дати
    void setDate(int d, int m, int y);

    //повернення дня (inline-метод)
    int getDay()
    {
        return day;
    }
    //повернення місяця (inline-метод)
```

```
      int getMonth()
      {
            return month;
      }
      //повернення року (inline-метод)
      int getYear()
      {
            return year;
      }
  };
```

Метод класу можна також описати як вбудований поза описом класу:

```
  class CppDate{
  private:
      int day;
      int month;
      int year;
  /*кількість днів у місяці m року y, якщо рік і місяць некоректні, то 0*/
      int getCountDayInMonth(int m, int y);

  public:
      /*порівняння дат: 0 – рівні, 1 – перша більша, -1 – друга більша*/
      int comparison(CppDate dt2);

      //друкування дати
      void printDate();

      //установлення дати
      void setDate(int d, int m, int y);

      //повернення дня (inline-метод)
      int getDay();
      //повернення місяця (inline-метод)
      int getMonth();
      //повернення року (inline-метод)
      int getYear();
  };

  //повернення дня (inline-метод)
      inline int CppDate::getDay();
      {
            return day;
      }
      //повернення місяця (inline-метод)
```

```
inline int CppDate::getMonth()
{
      return month;
}
//повернення року (inline-метод)
inline int CppDate::getYear();
{
      return year;
}
```

■

3.11.2. ІНІЦІАЛІЗАЦІЯ, ПРИСВОЮВАННЯ ТА ЗНИЩЕННЯ КЛАСУ

Використання для ініціалізації об'єкта класу функцій, подібних **setDate()**, може призвести до помилок, оскільки програміст після створення об'єкта може забути викликати цю функцію або викликати її кілька разів. У С++ є можливість описати функцію, явно призначену для ініціалізації об'єктів. Така функція називається *конструктором* і має таке саме ім'я, як клас. Для класу можна описати кілька конструкторів, тобто перевантажити конструктор.

Аналогічно можна описати *деструктор* – метод, що викликається при знищенні об'єкта класу.

Приклад 3.86. Конструктори й деструктори:

Лістинг.

```
class CTree{
private:
    CTree *m-pLeft;
    CTree *m-pRight;
    int m-iEl;
public:
//конструктор 1
    CTree(int el);
//конструктор 2
CTree();
//конструктор 3
    CTree(int el, CTree *left, CTree *right);
    //деструктор
    ~CTree();
…
};
//конструктор 1
```

```
CTree::CTree(int el)
{
    m-iEl=el;
    m-pLeft=NULL;
    m-pRight=NULL;
}
//конструктор 2
CTree::CTree()
{
    m-iEl=0;
    m-pLeft=NULL;
    m-pRight=NULL;
}
//конструктор 3
CTree::CTree(int el, CTree *left, CTree *right)
{
    m-iEl=el;
    m-pLeft=left;
    m-pRight=right;

}
//деструктор
CTree::~CTree()
{
    delete m-pLeft;
    delete m-pRight;
}
…
■
```

За умовчанням кожен клас має конструктор без параметрів і деструктор. Конструктор за умовчанням викликає конструктори всіх атрибутів, а деструктор – їхні деструктори.

3.11.3. СПАДКУВАННЯ

Розглянемо другу властивість ООП, що реалізується в C++ – *спадкування*.

Для створення класів із доданою функціональністю вводять спадкування. За допомогою класів-нащадків можна забезпечити загальний інтерфейс для кількох різних класів таким чином, щоб інші частини програми могли працювати з об'єктами цих класів однаково.

Клас-нащадок має атрибути (поля) і методи (функції-члени) базового класу, але не має права звертатися до його приватних (**private**) атрибутів і методів. Клас-нащадок може додавати свої поля й функції або перевизначати функції базового класу.

За умовчанням конструктор нащадка без параметрів викликає конструктор базового класу, а потім виконується сам. Інші конструктори (крім конструктора без параметрів) доводиться визначати кожного разу заново або явно описувати виклик конструктора базового класу. Деструктор нащадка – навпаки, після виконання свого тіла викликає деструктор базового класу.

Нащадок можна використовувати всюди, де використовується базовий клас. Наприклад, якщо метод як параметр потребує об'єкт базового класу, то можна передати йому об'єкт класу-нащадка, але не навпаки.

У мові С++ спадкування буває публічним (**public**), захищеним (**protected**) і приватним (**private**). Специфікатори доступу членів базового класу змінюються в нащадках таким чином:

- при **public**-спадкуванні члени базового класу зберігають свій статус;
- при **protected**-спадкуванні публічні члени базового класу стають захищеними членами нащадка;
- при **private**-спадкуванні всі члени базового класу в нащадку стають **private**.

Приклад 3.87. Типи спадкування:

```
class CA {
...
};

class CB:public CA {
...
};

class CC:protected CA{
...
};

class CppD:private CA{
...
};
```

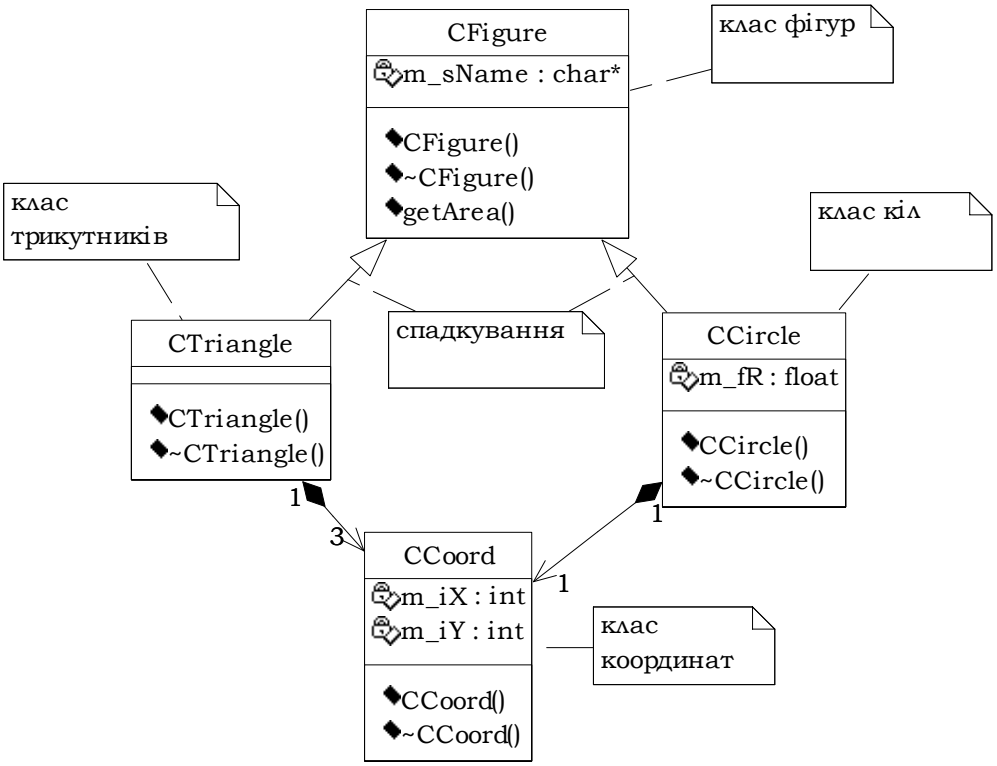
■

Найчастіше зустрічається публічне спадкування. Однією з його основних переваг є те, що покажчик на класи-нащадки може бути не-

явно перетворений на покажчик на базовий клас. Детальніше про це буде сказано при вивченні перетворення типів у мові С++.

Приклад 3.88. Спадкування. Знаходження площі геометричних фігур.

Напишемо програму для роботи з геометричними фігурами на площині (трикутником і колом). Потрібно знайти їхню площу. Зрозуміло, що і трикутник, і коло є фігурами й можуть мати певні спільні методи. Тому зручно скористатися такими структурами даних:



Як видно з діаграми, у даному випадку кілька класів (**`CTriangle`**, **`CCircle`**) спадкуються від одного базового (**`CFigure`**).

Опишемо класи для роботи із цими фігурами в мові С++:

Лістинг.

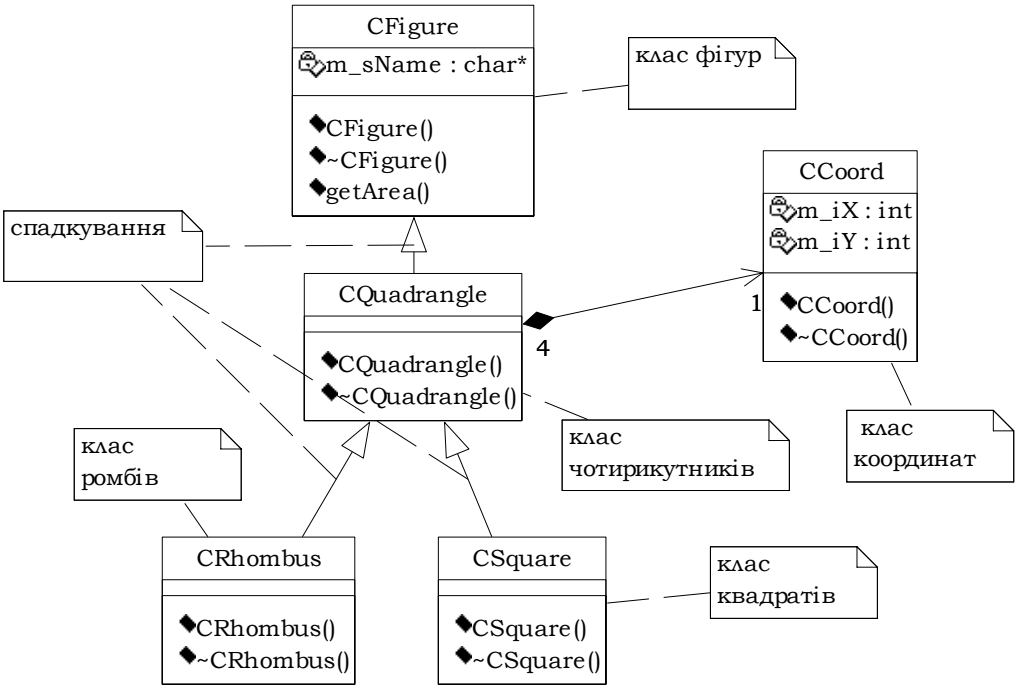
```
//клас координат
class CCoord {
public:
    int m-iX;
    int m-iY;
//конструктор
```

```
    CCoord(int x, int y) {
          m-iX=x;
          m-iY=y;
    }
};

//клас фігур
class CFigure{
protected:
    char *m-sName;
public:

//конструктор
    CFigure(const char *s) {
          m-sName=strdup(s);
    };
//деструктор
    ~CFigure() {
          delete m-sName;
    }
//знаходження площі
    float getArea();
};
//клас трикутників
class CTriangle:public CFigure {
private:
    CCoord *m-pA;
    CCoord *m-pB;
    CCoord *m-pC;
public:
//конструктор
    CTriangle(CCoord *a, CCoord *b, CCoord *c);
//деструктор
    ~CTriangle();
//знаходження площі
    float getArea();//площа
};
//клас кіл
class CCircle:public CFigure {
private:
    CCoord *m-pO;
    float m-fR;
public:
    CCircle(CCoord *o, float r);
    ~CCircle();
//знаходження площі
    float getArea();//площа
};
```

```
CTriangle::CTriangle(CCoord *a, CCoord *b, CCoord *c)
: CFigure("Triangle")//виклик конструктора базового класу
{
    m-pA=new CCoord(a->m-iX, a->m-iY);
    m-pB=new CCoord(b->m-iX, b->m-iY);
    m-pC=new CCoord(c->m-iX, c->m-iY);
}

CTriangle::~CTriangle()
{
    delete m-pA;
    delete m-pB;
    delete m-pC;
}
float CTriangle::getArea()//площа
{
    floata=(m-pA->m-iX*(m-pB->m-iY - m-pC->m-iY));
    float b=(m-pB->m-iX*(m-pC->m-iY - m-pA->m-iY));
    float c=(m-pC->m-iX*(m-pA->m-iY - m-pB->m-iY));
    //модуль
    float d=a+b+c;
    if(d>0)
          return (d/2);
    else
          return ((-1)*d/2);
}

CCircle::CCircle(CCoord   *o,   float   r):   CFigure("Square")
    //виклик конструктора базового класу
{
    m-pO=new CCoord(o->m-iX, o->m-iY);
    m-fR=r;
}

CCircle::~CCircle()
{
    delete m-pO;
}

float CCircle::getArea()
{
    return m-fR*m-fR*3.1415;
}
```

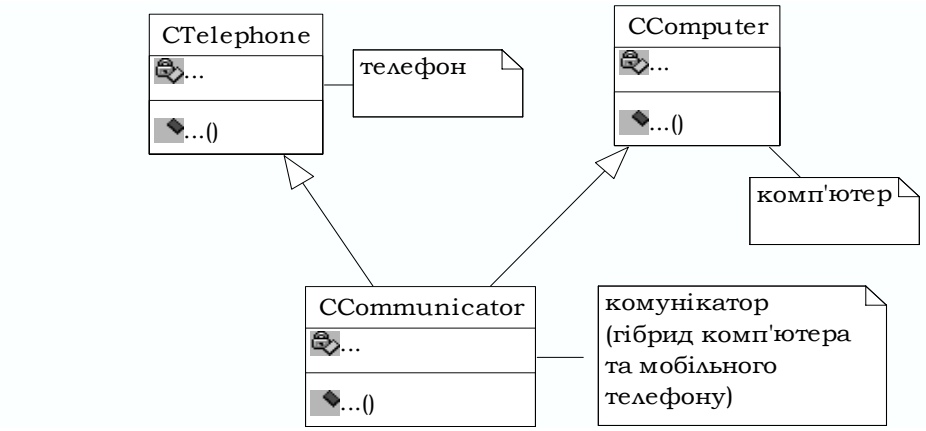
■

Клас-нащадок, у свою чергу, сам може бути базовим:



Множинне спадкування. У мові С++ клас може успадковуватися від кількох класів (*множинне спадкування*). Такий клас володіє атрибутами й методами всіх його предків. Перевагою цього підходу є гнучкість, однак множинне спадкування – джерело потенційних помилок, що можуть виникнути через наявність однакових імен методів у базових класах.

Приклад 3.89. Множинне спадкування:



У мові C++ це буде виглядати так:

```
class CTelephone {
…
};
class CComputer {
…
};
class CCommunicator:public CTelephone, public CComputer {
…
};
```

Проблема може виникнути, якщо кілька базових класів містять однойменний метод, а в класі-нащадку він не перевизначений:

```
class CTelephone {
…
public:
    void connection();
…
};
class CComputer {
…
public:
    void connection();
…
};

class CCommunicator:public CTelephone, public CComputer {
…
//void connection(); не визначено!!!
};

void f(CCommunicator *p)
{
p->connection();//ПОМИЛКА! НЕОДНОЗНАЧНІСТЬ!
}
```

У C++ такі неоднозначності зазвичай усуваються введенням у клас-нащадок додаткової функції:

```
class CCommunicator:public CTelephone, public CComputer {
…
```

```
public:
void connection()
{
CTelephone::connection();
CComputer::connection();
}
};
```
■

Зауважимо, що більшість сучасних об'єктно-орієнтованих мов програмування (C#, Java, Delphi та ін.) не підтримують множинне спадкування. Натомість вони підтримують можливість одночасно успадковуватися від базового класу й реалізовувати методи кількох інтерфейсів одним класом. Цей механізм дозволяє багато в чому замінити множинне спадкування – методи інтерфейсів потрібно перевизначати явно, що виключає помилки при спадкуванні функціональності однакових методів різних класів.

3.11.4. ПОЛІМОРФІЗМ

Як зазначалося вище, поліморфізм забезпечується тим, що в класі-нащадку перевизначають методи батьківського класу. Це забезпечує збереження незмінним інтерфейсу батьківського класу й дозволяє здійснювати зв'язування імені методу в коді з різними класами (з об'єкта якого класу здійснюється виклик, з того класу й береться метод із заданим іменем). Такий механізм називається *динамічним* (*пізнім*) *зв'язуванням* – на відміну від *статичного* (*раннього*) зв'язування, що здійснюється на етапі компіляції.

Цей механізм уже було використано вище при вивченні спадкування (приклад про знаходження площ фігур, класи **CFigure, CTriangle, CCircle, метод float getArea(); //площа**). Проте в тому випадку можливі були лише заздалегідь визначені виклики:

```
CCoord *a=new CCoord(1, 1);
CCircle *sq=new CCircle(a, 10);
cout<<"\nArea:"<<sq->getArea();
```

Однак поліморфні виклики спричинили б помилку.

Приклад 3.90. Помилкові поліморфні виклики:

```
//ранній поліморфізм
void f(CFigure *p)
{
```

```
//викличе метод базового класу
    cout<<"\nArea:"<<p->getArea();
}

або

CCoord *c1=new CCoord(10, 10);
CCoord *c2=new CCoord(100, 100);
CCoord *c3=new CCoord(25, 25);

//пізній поліморфізм
CFigure *p=new CTriangle(c1, c2, c3);
//викличе метод базового класу
    cout<<"\nArea:"<<p->getArea();
```

■

Для застосування поліморфізму введемо невеликі зміни в опис відповідних методів, оголосивши функцію для обчислення площі віртуальною.

Приклад 3.91. Правильні поліморфні виклики:

```
class CFigure{
protected:
    char *m-sName;
public:
    CFigure(const char *s) {
          m-sName=strdup(s);
    };
    ~CFigure() {
          delete m-sName;
    }
    double getPerimeter();
    /*const=0; - означає, що метод не реалізується в цьому кла-
сі; у такому разі метод називається чистою віртуальною функцією*/
    virtual float getArea() const=0;
};

class CTriangle:public CFigure {
private:
    CCoord *m-pA;
    CCoord *m-pB;
    CCoord *m-pC;
public:
    CTriangle(CCoord *a, CCoord *b, CCoord *c);
    ~CTriangle();
    virtual float getArea() const; //площа
};
```

```
class CCircle:public CFigure {
private:
    CCoord *m-pO;
    float m-fR;
public:
    CCircle(CCoord *o, float r);
    ~CCircle();
    virtual float getArea() const; //площа
};
virtual float CCircle::getArea() const //площа
{
//без змін
}

virtual float CTriangle::getArea() const //площа
{
//без змін
}
//усе інше без змін
```

У даному випадку яка саме функція буде застосована визначається на етапі компіляції ■

Абстрактним називається клас, який містить хоча б один суто віртуальний метод. Об'єкти таких класів створювати заборонено. Абстрактні класи використовуються як інтерфейси.

***Література для СР:** мова С++ – [57, 64, 70, 73, 96, 98, 102, 124].

Контрольні запитання та вправи

1. Як інкапсуляція подана в С++?
2. Що таке клас у С++?
3. Що таке атрибути й методи класу?
4. Що таке конструктор і деструктор класу та як їх описати в мові С++?
5. Як спадкування подано в С++?
6. Що таке публічне, захищене й приватне спадкування?
7. Що таке клас-нащадок і базовий клас?
8. Що таке множинне спадкування?
9. Які проблеми можуть виникнути при застосуванні множинного спадкування та як їх уникнути?
10. Як поліморфізм поданий у С++?
11. Що таке раннє й пізнє зв'язування?
12. Що робить метод віртуальним?
13. Що таке абстрактний клас?

14. Переписати програму з прикл. 3.84 для класу, де дата зображується кількістю днів від заданої дати (напр., `1.01.1900`).
15. Подайте задану діаграму класів (прикл. 3.85) у мові C++ і реалізуйте відповідні методи.
16. Створити клас «Трикутник» із полями-сторонами. Визначити методи зміни сторін, обчислення кутів, обчислення периметра. Створити клас-нащадок «Рівносторонній трикутник», який має поле площі. Визначити метод обчислення площі.
17. Написати функції для основних арифметичних операцій над раціональними числами й відношеннями на них (див. вправу 21, підрозд. 3.7). Раціональні числа подати у вигляді класу з атрибутами цілих типів, що задають чисельник і знаменник.
18. Створити базовий клас «Пара цілих чисел» з операціями перевірки на рівність і множення на число. Реалізувати операцію додавання пар за формулою $(a,b)+(c,d)=(a+b.c+d)$.

    Визначити клас-нащадок «Гроші» з полями «гривні й копійки». Перевизначити операцію додавання й визначити методи віднімання й ділення грошових сум.
19. Написати програму для роботи з геометричними фігурами (трикутник, круг, прямокутник, квадрат, ромб). У програмі створити абстрактний базовий клас «Фігура» з віртуальними методами обчислення площі й периметра та його нащадків – «Трикутник», «Круг», «Прямокутник», «Квадрат», «Ромб».
20. Створити абстрактний клас «Трикутник» із віртуальними методами обчислення площі й периметра. Поля даних повинні мати дві сторони й кут між ними. Визначити класи-нащадки «Прямокутний трикутник» і «Рівнобедрений трикутник» зі своїми функціями обчислення площі й периметра.
21. Напишіть програму, що дає можливість зберігати і зчитувати з диска, шукати за зразком, додавати нову й видаляти чи редагувати вже існуючу інформацію для задач:
    - каталог книг у бібліотеці;
    - список студентів у групі;
    - список товарів у магазині ;
    - список клієнтів банку;
    - телефонний довідник.

22. Реалізувати калькулятор числових константних арифметичних виразів. Додати в нього можливість вводити власні функції, запам'ятовувати константи, використовуючи засоби мови C++ (перевантаження операторів, бібліотеку STL, простори імен).

23. Реалізувати й увести в калькулятор додаткові алгебри (комплексні, раціональні числа, вектори, матриці) як окремі класи.
24. Реалізувати введення й виведення елементів комплексних чисел, перевантаживши оператори <<&>> в **oistream**.
25. Реалізувати просту текстову віконну систему, створивши відповідну ієрархію класів.

3.12. Інші засоби С++

- Шаблони
- Перетворення типів у мові С++
- Друзі
- Перевантаження операцій
- Виключення. Блок **try-catch-throw**

Ключові слова: *шаблон функції, вбудований, зовнішній і перевантажений шаблони, конкретизація* (*інстанціювання*), *шаблон класу, стандартна бібліотека шаблонів STL, примусове перетворення типів зі специфікаторами:* **static-cast**, **dynamic-cast**, **const-cast**, **reinterpret-cast**, *знижувальне зведення, друзі* (**friend**), *функції- й методи-друзі, перевантаження операцій, виключення, виняткова ситуація, обробка виключення, конструкції* **try**, **catch** *та* **throw**.

3.12.1. ШАБЛОНИ

У мові С++ можна створювати *шаблони* (узагальнені функції), використовуючи ключове слово **template**.

Шаблони функцій описують їхні загальні властивості, зазначають специфікації для функцій і класів, але не деталі реалізації. Шаблони особливо корисні в бібліотеках класів. Зазвичай вони розміщуються в заголовних файлах.

У шаблонів можуть бути параметри-типи й параметри-константи, що є фіксованими константними виразами. Будь-який із типів **int**, **double**, **char ***, **vector<int>**, **list<double>** є припустимим аргументом шаблона:

```
<оголошення-шаблона>::=template <<список-параметрів>>
<функція-з-параметрами-зі-списку-параметрів>
```

Тут зовнішні кутові дужки (<, >) є частиною визначення шаблона.

Якщо шаблон функції має кілька параметрів-типів, то кожному з них повинно передувати ключове слово **class** чи **typename**.

Подібно до звичайних функцій, шаблон функції може бути оголошений як **inline** чи **extern**, а також перевантажений.

Процес підстановки типів і значень замість параметрів називається *конкретизацією* (*інстанціюванням*) шаблона.

Визначимо функцію **max** для довільних типів.

Приклад 3.92. Визначення функції **max** через шаблони:

```
template <class T>
T max(T a, T b)
{
    if(a>b)
return a;
else return b;
}

template <class T>
inline
T min(T a, T b)
{
    if(a<b)
return a;
else return b;
}
```

Викликати ці функції можна таким чином:

```
max(3, 10);/*конкретизація екземпляра функції max, де T int*/
max(3.5, 5.4);/*конкретизація екземпляра функції max, де T double*/
max('a', 'b');/*конкретизація екземпляра функції max, де T  char*/
max(string("sss"),   string("rrr"));/*конкретизація   екземпляра
функції max, де T string*/
min(3, 10);/*конкретизація екземпляра функції min, де T int*/
min(3.5, 5.4);/*конкретизація екземпляра функції min, де T double*/
min('a', 'b');/*конкретизація екземпляра функції min, де T char*/
min(string("sss"),   string("rrr"));/*конкретизація   екземпляра
функції min, де T string*/
```

■

У шаблонах можна використовувати більше одного типу.

Приклад 3.93. Шаблон функції з кількома параметрами-класами:

```
template <class T1, class T2>
T1 fun(T1 a, T2 b)
```

```
//повертаємо значення типу T1, аргументи типів T1 та T2
{
//тіло функції

}
```
■

Приклад 3.94. Шаблон функції з параметром-класом і параметром-константою:

```
template <class T, int size>
T max1(T (&arr) [size])
{
/*параметризована функція для пошуку мінімального значення в
масиві*/
    T max-val=arr[0];
for(int i=1; i<size; ++i)
    if(arr[i]>max-val)
          max-val=arr[i];
return max-val;
}
```

Тут **T** – тип елементів масиву **arr** і локальної змінної **max-val**, а також тип значення, що повертається функцією; **size** задає розмір масиву ■

Шаблони можуть задавати не тільки функції, але й класи. *Шаблон класу* задає каркас узагальненого класу для наступної його реалізації.

Приклад 3.95. Шаблон класу:

```
template<class T1, class T2>
class CA{
private:
T1* m-pV1;
T2 * m-pV2;

public:
    CA(int n)
{
    m-pV1=new T1;
m-pV2=new T2;
}
~CA()
{
    delete v1;
}
T1 *f1(int n, T2 v);
```

```
void f2(T1 v1,T2 v2);
};

template<class T1, class T2>
T1 * CA<T1, T2>::f1(int n, T2 v)
{
T1 * p;
//….
return p;
}

template<class T1, class T2>
void CA<T1, T2>::f2(T1 v1,T2 v2)
{
//….
}
```
■

У C++ введено стандартну бібліотеку шаблонів **STL** (англ. – Standard Template Library), що визначає шаблони й функції для векторів (одновимірних масивів довільної довжини), множин, асоціативних масивів (**map**), списків, символьних рядків, потоків введення-виведення тощо.

3.12.2. ПЕРЕТВОРЕННЯ ТИПІВ У МОВІ C++

Виділяють явні й неявні перетворення типів. Неявні істотно не відрізняються від таких у мові C (див. підрозд. 3.2.4). Форма явного перетворення типів, що використовується в мові C, збережена в стандарті C++ для сумісності з програмами, написаними мовою C. До того ж стандарт мови C++ пропонує нові операції явного перетворення типів за допомогою специфікаторів **static-cast**, **dynamic-cast**, **const-cast** та **reinterpret-cast**.

Перетворення типів виконується для значень змінних при обчисленні виразів і впливає на тип результату, але не змінює тип змінних, що містяться у виразі:

```
<перетворення-типів>::=<cast-name> <тип>(<вираз>);
<cast-name>::=static-cast|dynamic-cast|const-cast|reinterpret-cast
```

Тут **<тип>** – це тип, до якого зводиться **<вираз>**. Ця форма явного перетворення типів прийнятніша за таку в мові C, оскільки зазначені оператори містять додаткову інформацію про типи, що беруть участь у перетворенні.

Специфікатор **static-cast** указує на перетворення споріднених типів, тобто такі, що можуть здійснюватися неявно, на основі правил за умовчанням. Використовується для уникнення попередження компілятора про можливу втрату точності.

Приклад 3.96. Перетворення типів **static-cast**:

```
double d=55.0;
int i=static-cast<int>(d);
```

■

Крім того, покажчик на **void*** можна перетворювати на покажчик довільного типу, арифметичне значення – на **enum**, а базовий клас – на похідний.

Приклад 3.97. Перетворення типів **static-cast** із використанням **void***:

```
void *p;
int *pi=static-cast <int*>(p);
```

■

Специфікатор **dynamic-cast** використовується для динамічного перетворення типів, що реалізується на етапі виконання, і безпечного зведення покажчика на базовий клас до покажчика на похідний (*знижувальне зведення*).

Увага! Це єдиний формат перетворення, що перевіряє, чи підтримується об'єктом заданий інтерфейс (до якого зводять); якщо ні – то результат 0 ►

Отже, перетворення **dynamic-cast** виконує відразу дві операції: перевіряє, чи можливе перетворення, а якщо можливе – то здійснює його. Таке перетворення типів, що виконується під час роботи програми, безпечніше за інші.

Приклад 3.98. Перетворення типів **dynamic-cast**:

```
class A {...};
class B:public A {...};
class C:public A {...};

void f(A* pa)
{
    B* pb=dynamic-cast<B*>(pa);
    if(pb){
          /*якщо pa є об'єктом типу B, то виконується перетво-
рення
           і працюємо з об'єктом pb*/
          …
```

```
      }
      else {
            C* pc=dynamic-cast<C*>(pa);
            if(pc) {
                  /*якщо pa є об'єктом типу C, то виконується пе-
ретворення
                   і працюємо з об'єктом pc*/
                  …
            }
            else
            {
                  //pa не є об'єктом класів B чи C
                  …
            }
      }
  }
```

■

Специфікатор **const-cast** виконує зведення константних виразів до неконстантних і навпаки.

Приклад 3.99. Перетворення типів **const-cast**:

```
extern char *str-copy(char*);
const char *c-str;
char *str=str-copy(const-cast<char*>(c-str));
```

■

Будь-яке інше використання **const-cast** викликає помилку на етапі компіляції.

reinterpret-cast – працює з внутрішніми зображеннями об'єктів. Правильність цієї операції цілком залежить від програміста.

Приклад 3.100. Перетворення типів **reinterpret-cast**:

```
complex *pCom=reinterpret-cast <complex*>(2000);
```

■

3.12.3. ДРУЗІ

Одна з основних переваг ООП – інкапсуляція даних і методів у класах. Однак іноді зручно дозволити деяким методам чи класам доступ до приватних і закритих даних і методів деякого класу. У мові C++ можна обійти правила інкапсуляції за допомогою *друзів*, хоча при цьому й виникає деякий ризик порушення цілісності даних.

Дружні класи. У класі можна оголосити інший клас дружнім. Один клас (в якому оголошується друг) надає іншому (другу) можливість доступу до всіх своїх закритих елементів.

Отже, якщо клас **CA** – друг класу **CB**, то всі його методи можуть звертатися до довільного атрибута чи методу класу **CB**. Синтаксично це виглядає таким чином:

Приклад 3.101. Дружні класи:

```
class CB {
    public:
…
private:
    int m-iCount;
    int f1();
protected:
    void f2();
…
friend class CA;
}

class CA {
…
}
```

■

При використанні дружніх класів варто дотримуватися таких правил:

1) *У класі мають бути перелічені всі його друзі.*

2) *Дружній клас може бути оголошений як до, так і після класу, в якому він описаний як друг.*

3) *Похідні від дружніх класів не успадковують прав спеціального доступу.*

Можна оголосити також взаємно дружні класи.

Приклад 3.102. Взаємно дружні класи:

```
class CA {
…
friend class CB;
}

class CB {
…
friend class CA;
}
```

■

Функції-друзі та дружні методи. Функції-друзі – це функції, що не є функціями-членами, але мають доступ до захищених і власних атрибутів і методів класу. Вони повинні бути описані в тілі класу як **friend**.

Приклад 3.103. Опис дружньої функції:

Лістинг.

```
#include <iostream>

using namespace std;
class CC {
friend void Show (CC &c1);
private:
    char *s1;
public:
    CC() {s1="TEST";};
};

int main()
{
    CC c1;

    Show(c1);
    system("pause");
    return 0;
}

void Show (CC &c1)
{
    cout<<c1.s1<<'\n';
}
```

■

Дружня функція може бути елементом класу (методом). Зазвичай оголошується дружнім метод іншого класу. Цей метод має доступ до закритих і захищених елементів класу, в якому функція оголошена дружньою.

Приклад 3.104. Опис дружнього методу:

Лістинг.

```
#include <iostream>
using namespace std;

class CE;
class CD {
    friend void Show (CE &c1);
private:
```

```
    char *s1; //доступний у класі CD
public:
    CD() {s1="TEST CD";};
    void Show(CE &c1);
};
class CE {
    friend void CD::Show (CE &c1);
private:
    char *s1; //доступний у класах CE та CD::Show
public:
    CE() {s1="TEST CE";};
};

int main()
{
    CD c1;
    CE c2;
    c1.Show(c2);
    system("pause");
    return 0;
};

void CD::Show (CE &c1)
{
    cout<<c1.s1<<'\n';
}
```

■

3.12.4. ПЕРЕВАНТАЖЕННЯ ОПЕРАЦІЙ

Перевантаження операцій дозволяє визначати дії для об'єктів класу у виразах, що використовують звичайні операції, такі як `+`, `-` тощо.

Наведемо список операцій, що можуть бути перевантажені:

*	/	+	-	%	^	&	|
~	!	,	=	<	>	<=	>=
++	--	<<	>>	==	!=	&&	||
*=	/=	%=	^=	&=	|=	+=	-=
<<=	>>=	->	->*	[]	()	new	delete

Операції `+, -, *, &` можуть бути перевантажені як для бінарних, так і для унарних виразів. Операції `=, (), [], ->` потрібно реалізовувати нестатичними методами.

Перевантаження оголошується так само, як і звичайна дружня функція чи звичайний метод класу.

Приклад 3.105. Перевантаження операцій.

Напишемо програму, що використовує перевантаження бінарних операцій +, – та <:

Лістинг.

```
#include <iostream>
using namespace std;

class CA {
private:
    int m-iX;
    int m-iY;
public:
    CA() {
          m-iX=0;
          m-iY=0;
    }
    CA(int x, int y) {
          m-iX=x;
          m-iY=y;
    }
    CA operator+(CA &a);
    CA operator-(CA &a);
    int operator<(CA &a);

    void show() {cout<<m-iX<<","<<m-iY<<endl;}
};

//перевантаження +
CA CA::operator+(CA &a)
{
    return CA(m-iX+a.m-iX, m-iY+a.m-iY);
}

//перевантаження -
CA CA::operator-(CA &a)
{
    return CA(m-iX – a.m-iX, m-iY – a.m-iY);
}

//перевантаження <
//тут вважаємо меншим той елемент, в якого сума компонент менша
int CA::operator<(CA &a)
{
    return ((m-iX+m-iY)<(a.m-iX+a.m-iY));
}
```

```
int main()
{
    CA a=CA(6, 2);
    CA b=CA(3, 4);
    CA c=CA(0, 0);
    c=a+b;
    a.show();
    b.show();
    c.show();
    cout<<"min a, b"<<endl;
    if(a<b)
          a.show();
    else b.show();
    return 0;
}
```

У результаті роботи програми на екрані побачимо:

```
6, 2
3, 4
9, 6
min a, b
3, 4
```

■

3.12.5. ВИКЛЮЧЕННЯ. БЛОК `try-catch-throw`

Виключення, або *виняткова ситуація* (англ. – exception handling), – це аномальна поведінка програми під час її виконання, наприклад: ділення на 0, вихід за межі масиву або звільнення вільної пам'яті. Такі виключення порушують нормальний хід роботи програми й на них потрібно негайно реагувати. У С++ є вбудовані засоби для їхньої генерації та обробки.

Обробка виключень – це механізм, призначений для обробки помилок під час виконання.

При винятковій ситуації керування передається деякому оброблювачу на вищий рівень і забезпечується можливість *нелокального виходу*, тобто передавання керування в деяку віддалену точку програми.

Для ініціювання виняткової ситуації оператор надсилає об'єкт, що описує її сутність. У мові С++ виняткові ситуації зображуються об'єктами: об'єкт може бути літеральним значенням, рядком, об'єктом класу тощо. При ініціюванні виняткової ситуації одразу завершується виконання функції.

Приклад 3.106. Використання оператора **throw**, що ініціює виключення:

```
class CTestError {
public:
   CTestError(){};
   ~CTestError(){};
   const char *showReason() const {return "Exception in
CTestError class.";}
};
…
int fun(int i)
{
if(i==0)  //якщо виконується умова
          throw 1;//то ініціюється виняткова ситуація типу int
    if(i==1)//якщо виконується умова, то
          throw CTestError();//надсилає об'єкт класу CTestError
    //якщо немає проблем, то
return i;
}
```

Ця функція повертає цілочислове значення лише у випадку, коли не виникли виняткові ситуації; якщо така виникла, то вона повертає об'єкт **CTestError** ■

Для обробки виняткової ситуації параметризований *оператор-оброблювач виключень* **catch** перехоплює умову, надіслану деяким іншим процесом.

Приклад 3.107. Використання оператора **catch**:

```
catch(CTestError e) {
//обробка виняткової ситуації для класу CTestError
};
catch(int) {
//обробка виняткової ситуації для int
};
```
■

Для підготовки програм до майбутніх перехоплень виключень під час експлуатації їх випробовують (за допомогою *конструкції* **try**) на одному чи кількох процесах, що можуть викликати виняткову ситуацію.

Приклад 3.108. Використання **try** …**catch**:

```
int i=0;
…
try {
```

```
    int x fun(i); //виклик функції
}
catch(CTestError e) {
//виклик функції обробки виняткової ситуації
    cout<<e.showReason<<endl;
}
catch(int k) {
//виклик функції обробки виняткової ситуації
    cout<<"ERROR"<<k<<endl;
}
```

■

***Література для СР:** мова С++ – [57, 64, 70, 73, 96, 98, 102, 124].

Контрольні запитання та вправи

1. Що таке шаблони функцій і як їх використовувати?
2. Що таке вбудований, зовнішній і перевантажений шаблони?
3. Що таке конкретизація (інстанціювання) шаблону?
4. Описати стандартну бібліотеку шаблонів STL.
5. Які особливості зведення типів у мові С++?
6. Які особливості примусового зведення типів зі специфікаторами: **static-cast**, **dynamic-cast**, **const-cast** та **reinterpret-cast**?
7. Що таке знижувальне зведення типу?
8. Яке призначення дружніх класів, методів і функцій?
9. Що таке перевантаження операцій?
10. Які операції можуть бути перевантажені?
11. Що таке виняткова ситуація?
12. Що таке виключення та його обробка?
13. Охарактеризувати конструкції **throw**, **catch** та **try**.
14. Написати програму, що задає як шаблон функції знаходження мінімуму, максимуму й суми й використовує ці методи для аргументів різних типів.
15. Написати програму, що дає можливість зберігати та зчитувати з диска, додавати нову й видаляти чи редагувати вже існуючу інформацію. Перевантажте оператор порівняння для відповідних класів:
    - каталог книг у бібліотеці;
    - список студентів у групі;
    - список товарів у магазині;
    - список клієнтів банку;
    - телефонний довідник.
16. Подати раціональні числа у вигляді класу з атрибутами цілих типів, що задають чисельник і знаменник. Реалізувати за до-

помогою перевантаження відповідних операторів основні арифметичні операції над раціональними числами й відношення на них.

17. Написати функцію для обчислення скалярного добутку двох векторів довільної довжини. Знайти скалярний добуток двох векторів, елементи яких вводяться з клавіатури.
18. У калькуляторі із вправи 22, підрозд. 3.11 передбачити детектор арифметичних помилок (ділення на 0 тощо), реалізувавши його за допомогою блока **`try-catch`**.
19. Реалізувати в калькуляторі з попередньої вправи додаткові алгебри (комплексні числа, раціональні числа, вектори) як окремі класи з детектором можливих помилок.
20. Для наведених функцій виконати перевірку параметрів, що передаються, і згенерувати виключення у випадку помилки:
    - площа трикутника за трьома сторонами: $S = \sqrt{p(p-a)(p-b)(p-c)}$, де $p = (a+b+c)/2$;
    - обчислення кореня лінійного рівняння $ax + b = 0$;
    - обчислення периметра трикутника;
    - переведення годин і хвилин у секунди;
    - обчислення кореня квадратного рівняння $ax^2 + bx + c = 0$;
    - обчислення суми геометричної прогресії $S_n = (a_0 - a_n r)/(1-r)$.

Розділ IV

АЛГОРИТМИ

Складність і обсяги задач, що постають перед розробниками програмних систем, постійно збільшуються. Тому, незважаючи на стрімке зростання потужності обчислювальних систем, застосування ефективних алгоритмів для розв'язання задач є і буде актуальним.

Цей розділ присвячено вивченню алгоритмів для розв'язання таких важливих прикладних класів задач, як пошук і сортування, алгоритми на графах, числові алгоритми, обробка таблиць, лексичний і синтаксичний аналіз програм тощо.

4.1. Сортування й пошук

- Послідовний пошук у масиві
- Бінарний пошук
- Бульбашкове сортування
- Сортування вставленням
- Сортування злиттям
- Швидке сортування
- Вибір методу сортування

Ключові слова: *задачі пошуку й сортування, ключ, таблиця, рядок таблиці, послідовний пошук, бінарний (дихотомічний, діленням навпіл) пошук, бульбашкове сортування, сортування вставленням, сортування злиттям, швидке сортування.*

Розглянемо питання, пов'язані з ефективним сортуванням і пошуком у таблицях. При розв'язанні багатьох важливих задач попередня відсортованість таблиць суттєво підвищує швидкодію алгоритмів.

Сортування кортежу – це його впорядкування за зростанням чи спаданням. Задачі сортування кортежів є одними з найпоширеніших

у програмуванні. Методи їхнього розв'язання застосовують і для сортування загальних послідовностей.

Сформулюємо загальну задачу сортування. Нехай A – довільна *базова множина*, а $(K,\leq)$ – довільна лінійно впорядкована множина елементів, які називаються *ключами*. Пов'яжемо з кожним базовим елементом певний ключ, і нехай $f: A \to K$ означає дану відповідність. Таким чином, $f(a)$ є ключем елемента $a \in A$. Зазвичай базові елементи мають певну структуру і ключами стають значення певного фіксованого елемента в ній. Типовим прикладом структурованих базових елементів є вектори фіксованої довжини з різнотипними компонентами. Як ми вже знаємо, у програмуванні подібні вектори називаються структурами, їхні координати – полями, а кортежі таких однотипних структур – *таблицями*. У таблицях структури називаються *рядками*. За ключі рядків беруть значення певного фіксованого поля. Порядок на ключах елементів природно переноситься й на самі елементи, а саме: для довільних $a,b \in A: a \leq b \Leftrightarrow f(a) \leq f(b)$. Кортеж базових елементів $(a_1,...a_n)$ називається *відсортованим*, якщо $1 \leq \forall i < n \;\; a_i \leq a_{i+1}$.

Наприклад, є таблиця рядків про студентів факультету, що містить інформацію про прізвище, ім'я та по-батькові (ПІБ) студента, рік народження, курс тощо, яку необхідно відсортувати за курсом. Тоді ключем буде курс, а решта полів міститиме супутню інформацію.

Задача сортування. Задано довільний кортеж базових елементів $(a_1,...,a_n)$. Необхідно знайти таку його перестановку $(a'_1,...,a'_n)$, після якої їхні ключі $f(a'_i)$ розмістяться в неспадному порядку: $1 \leq \forall i < n \;\; f(a'_i) \leq f(a'_{i+1})$.

Сортування називається *стійким*, якщо воно задовольняє додаткову умову: елементи з однаковими ключами залишаються в попередньому порядку.

Виділяють два класи методів сортування: *внутрішні*, коли кортеж зберігається в ОП, і *зовнішні*, коли він розташований на зовнішніх пристроях.

Розглянемо спочатку важливу задачу пошуку елементів у кортежі.

Задача пошуку. Задано довільний кортеж базових елементів $(a_1,...,a_n)$ і ключ $k \in K$. Необхідно знайти всі або один з елементів a_i в кортежі з ключем k, що задовольняє певну умову.

4.1.1. ПОСЛІДОВНИЙ ПОШУК У МАСИВІ

Найпростіший спосіб подання кортежу – це масив. *Послідовний* (*лінійний*) пошук у масиві передбачає послідовний перегляд усіх його елементів у порядку їхнього розташування, поки не знайдеться елемент a_i такий, що $f(a_i) = k$. Якщо достеменно невідомо, чи є такий елемент у масиві, то необхідно стежити за тим, щоб пошук не вийшов за межі масиву.

Приклад 4.1. Лінійний пошук:

Лістинг. **s_search.c**

```
/*послідовний пошук у символьному масиві a[] довжиною n*/

/*a – масив, в якому ведеться пошук, n – кількість елементів
масиву, k – ключ для пошуку*/
int sequential_search(char *a, int n, char k)
{
 register int i;
 for(i=0; t<n; ++t)
 if(k==a[i]) return i;
 return -1; /*ключ не знайдено*/
}
```

■

Кількість порівнянь у алгоритмі лінійного пошуку становить O(n), де n – кількість елементів у масиві. При цьому максимальна кількість порівнянь збігається із n (коли елемент відсутній у масиві або розташований на останньому місці).

4.1.2. БІНАРНИЙ ПОШУК

Для впорядкованих масивів існують значно ефективніші алгоритми пошуку, але й для таких масивів можна застосовувати алгоритми послідовного пошуку. *Бінарний пошук* (інші назви – дихотомічний, логарифмічний, пошук діленням навпіл) полягає в тому, що ключ k порівнюється з ключем середнього елемента масиву. Якщо ці значення рівні, то шуканий елемент знайдено, в іншому випадку пошук продовжується в одній із двох половин списку (якщо ключ k більше ключа середнього елемента – то в правій половині, інакше – у лівій).

Для знаходження елемента бінарним пошуком необхідна кількість порівнянь становить $O(\log_2 n)$, де n – кількість елементів у масиві.

Приклад 4.2. Бінарний пошук:

Лістинг. **b_search.c**

```
/*бінарний пошук у символьному масиві a[] довжиною n*/

/*a – масив, в якому ведеться пошук, n – кількість елементів
масиву, k – ключ для пошуку*/

int binary_search(char *a, int n, char k)
{
/*low i high – перший і останній індекси поточного підмасиву, в
якому ведеться пошук, mid – його середина*/
 int low, high, mid;
 low=0; high=n-1;
 while(low<=high) {
    mid=(low+high)/2;
    if(k<a[mid]) high=mid-1;
       else if(k>a[mid]) low=mid+1;
            else return mid; /*ключ знайдено*/
 }
 return -1;
}
```

■

4.1.3. БУЛЬБАШКОВЕ СОРТУВАННЯ

Серед усіх методів сортування масивів найвідомішим і найпростішим для реалізації й розуміння є обмінний алгоритм *бульбашкового сортування* (англ. – bubble sort).

Алгоритм полягає в неодноразовому проходженні масиву, що сортується. За кожне проходження всі пари сусідніх елементів послідовно порівнюються, і якщо порядок у парі неправильний, то елементи її міняються місцями. Проходження повторюються доти, доки на черговому не виявиться, що обміни більше не потрібні, отже, масив відсортовано. При проходженні алгоритму елемент, що стоїть не на своєму місці, "спливає" на потрібну позицію, як бульбашка у воді, звідки й походить назва алгоритму.

Приклад 4.3. Бульбашкове сортування:

Лістинг. **bubble.c**

```
/*Бульбашкове сортування масиву a[] довжиною n*/
  void bubbleSort(float a[], int n)
  {
  char is=1;
  int m=0;
```

```
int i;
float c;
  while(is)
  {   is=0;
      for (i=m+1; i<=n; i++)
      if (a[i]<a[i-1])
      {    /*обмін елементами*/
         c=a[i];
           a[i]=a[i-1];
           a[i-1]=c;
           is=1;
      }
  }
}
```

Сортування методом бульбашки для вхідного масиву [4.0, 5.0, 3.0, 10.0, 1.0]:

```
[4.0   5.0   3.0   10.0  1.0]
[4.0   3.0   5.0   10.0  1.0]
[4.0   3.0   5.0   1.0   10.0]
[3.0   4.0   5.0   1.0   10.0]
[3.0   4.0   1.0   5.0   10.0]
[3.0   1.0   4.0   5.0   10.0]
[1.0   3.0   4.0   5.0   10.0]
```

■

У найкращому випадку (масив уже відсортований) алгоритм вимагає $n-1$ порівнянь і переміщень елементів. При першому проходженні здійснюється $n-1$ порівнянь, при другому – $n-2$, на кожному наступному кроці кількість порівнянь зменшується на 1. Складність у найгіршому випадку задається формулою

$$\sum_{i=n-1}^{1} i = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \frac{n^2-n}{2} \approx \frac{1}{2}n^2 = \mathrm{O}\left(n^2\right).$$

Отже, алгоритм бульбашкового сортування ефективний лише для невеликих масивів, і його складність сягає $\mathrm{O}\left(n^2\right)$ порівнянь і переміщень елементів.

4.1.4. СОРТУВАННЯ ВСТАВЛЕННЯМ

Метод *сортування вставленням* простий у реалізації, ефективний на невеликих наборах даних або на частково відсортованих даних, алгоритм його стійкий. На кожному кроці алгоритму вибираємо один з елементів і вставляємо його на потрібну позицію в раніш відсорто-

ваному початку масиву доти, доки набір вхідних даних не буде вичерпано. Вибір чергового елемента зі вхідного підмасиву – довільний.

Приклад 4.4. Сортування вставленням:

Лістинг. **insert.c**

```
/*сортування вставленням масиву a довжиною n*/

void insertSort (float a[], int n)
{
int i, j;
float value;
    for (i=1; i<n; i++) {
          value=a[i];
for (j=i-1; (j>=0) && (a[j]>value); j--) {
    a[j+1]=a[j];
}
a[j+1]=value;
}
}
```

Тут обидва списки, вхідний і шуканий, розміщуються в масиві a, причому початковий список займає частину a з індексами від i до n, а вихідний – від 0 до i-1.

Для вхідного масиву [4.0, 5.0, 3.0, 10.0, 1.0] при сортуванні методом вставлень маємо:

Вихідний список:	**Вхідний список:**
[]	**[4.0 5.0 3.0 10.0 1.0]**
[4.0]	[5.0 3.0 10.0 1.0]
[4.0 5.0]	[3.0 10.0 1.0]
[3.0 4.0 5.0]	[10.0 1.0]
[3.0 4.0 5.0 10.0]	[1.0]
[1.0 3.0 4.0 5.0 10.0]	**[]**

■

Розглянемо найгірший випадок застосування алгоритму сортування вставленням. Першим вставляється другий елемент списку. Він порівнюється з одним елементом. Кожний наступний елемент, що вставляється, порівнюється з усіма попередньо вставленими. Отже, i-й елемент, що вставляється, порівнюється з i попередніми, і цей процес повторюється $n-1$ разів. Складність цього алгоритму визначається формулою $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \frac{n^2-n}{2} \approx \frac{1}{2}n^2 = \mathrm{O}\left(n^2\right)$.

4.1.5. СОРТУВАННЯ ЗЛИТТЯМ

У всіх вищерозглянутих алгоритмів сортування є суттєвий недолік – час їхнього виконання має порядок $O\left(n^2\right)$. Потрібно шукати кращі алгоритми.

Попередній алгоритм діяв за принципом: додаємо елементи один за одним до відсортованої частини масиву. Застосуємо інший підхід – "розділяй і володій", і побудуємо за його допомогою значно швидший алгоритм.

Нагадаємо, що ідея принципу "розділяй і володій" (див. підрозд. 2.2.2) полягає в: 1) розбитті задачі на менші підзадачі; 2) розв'язанні вибраних підзадач; 3) розв'язанні шляхом комбінації отриманих результатів загальної задачі.

Для задачі сортування ці три етапи можуть виглядати таким чином. Спочатку ми розбиваємо масив навпіл, потім сортуємо кожну половину окремо. Після цього здійснюємо злиття двох відсортованих масивів (див. прикл. 3.36). Подібний процес можна зробити рекурсивним. Рекурсивне розбиття задачі на менші буде відбуватися доти, доки розмір підмасиву не досягне одиниці, а такий масив за означенням є впорядкованим.

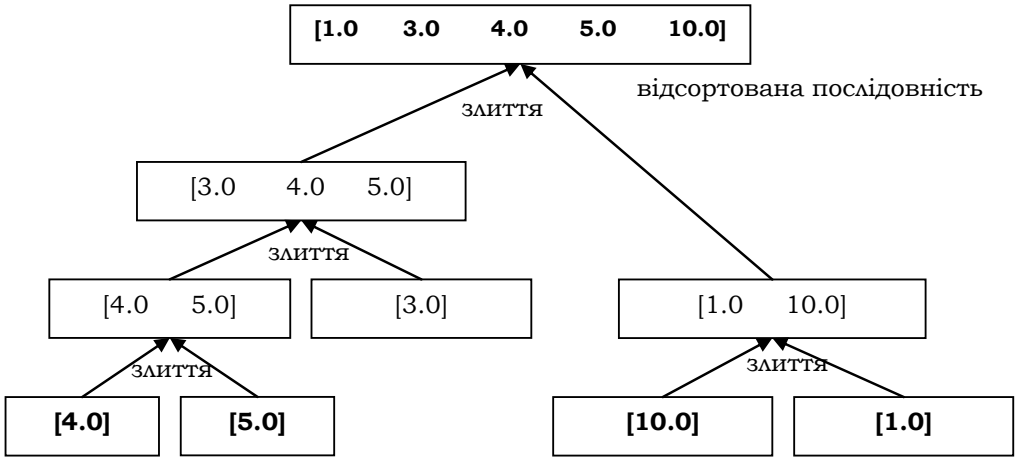
Приклад 4.5. Сортування злиттям:

Лістинг. **mergeSort.c**

```
/*злиття двох відсортованих підмасивів масиву v з індексами від
left до mid і від mid+1 до right у впорядкований глобальний масив
b і копіювання його назад у v*/

void merge(float v[], int left, int mid, int right)
{
    int i, m1, m2;
    i=m1=left;
    m2=mid+1;
    float b[right-left+1];
     while (m1<=mid && m2<=right)
          if (v[m1]<v[m2])
                b[i++]=v[m1++];
          else b[i++]=v[m2++];
          while (m1<=mid) b[i++]=v[m1++];
          while (m2<=right) b[i++]=v[m2++];
          for (i=left; i<=right; i++) v[i]=b[i];
}
```

```
  /*упорядкування за зростанням ділянки масиву v від left до
right*/
  /*масив v, left – початок ділянки, що сортується, right – кі-
нець ділянки, що сортується, mid – середина*/
  void mergeSort(float v[], int left, int right, int mid)
  {
      /*список порожній*/
      if(left==right) return;

      /*список складається з одного елемента – відсортований*/
      if(left==right-1) return;
      mergeSort(v, left, mid, (left+mid)/2); /*перша половина*/
      mergeSort(v, mid+1, right, (mid+right+1)/2); /*друга половина*/

      /*злиття двох відсортованих фрагментів масиву v (від left до
mid) та (від mid+1 до right)*/
      merge(v, left, mid, right);
  }
```

Сортування вхідного масиву [4.0, 5.0, 3.0, 10.0, 1.0] методом злиття:



■

Складність алгоритму оцінюється як $O(n\log_2 n)$, оскільки за одне проходження виконується n порівнянь, а всього потрібно $\log n$ проходжень. Для великих n сортування злиттям значно ефективніше за сортування вставленням чи методом бульбашки, які вимагають часу $O(n^2)$.

Розглянемо ще один відомий метод сортування, що вимагає $O(n\log_2 n)$ часу – *швидке сортування.*

4.1.6. ШВИДКЕ СОРТУВАННЯ

Алгоритм швидкого сортування був розроблений Ч. Хоаром. Цей алгоритм, як і попередній, базується на принципі "розділяй і володій". Його трьома етапами є: 1) вибір деякого елемента, що називається *опорним*; 2) поділ масиву на два підмасиви шляхом перестановки його елементів так, щоб усі елементи, менші чи рівні опорному, виявилися ліворуч від нього, а всі елементи, більші за нього – праворуч; 3) рекурсивне впорядкування вибраних підмасивів. Базою рекурсії є підмасиви, що складаються з одного чи двох елементів. Алгоритм завжди завершується, оскільки за кожну ітерацію він ставить принаймні один елемент на його остаточне місце.

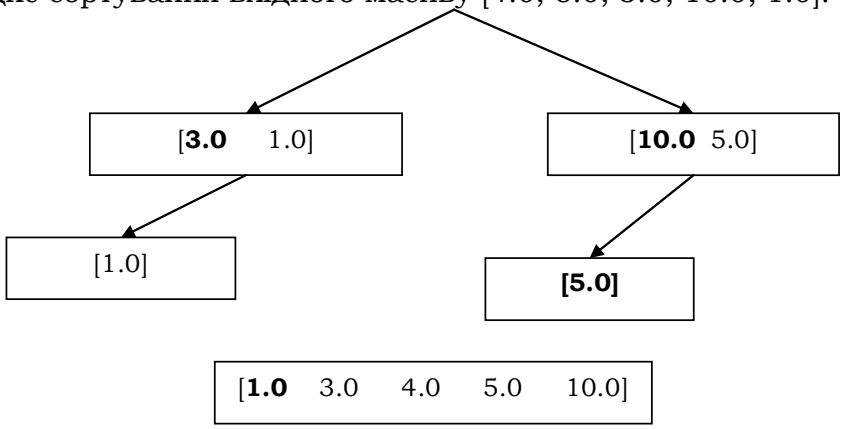
Приклад 4.6. Швидке сортування:

Лістинг. **QSort.c**

```
/*Швидке упорядкування за зростанням масиву a від елемента left
до елемента right*/

void qSort(float a[], int left, int right)

/*масив a, left – початок ділянки, що сортується, right – кі-
нець ділянки, що сортується*/

{
if(first<last)
{
int i=left; /*змінна для перегляду масиву ліворуч*/
int j=right; /*змінна для перегляду масиву праворуч*/
float point=a[left]; /*опорний елемент*/

while(i<j) {
/*Шукаємо ліворуч елемент, який >= опорної точки*/
while(a[i]<=point && i<right)
i++;
/*Шукаємо праворуч елемент, який <= опорної точки*/
while(a[j] >= point && j>left)
j--;

if(i<j)
swap(a[i], a[j]); /*обмін елементами*/
}

swap(a[left], a[j]); /*обмін елементами*/

/*Сортуємо лівий і правий підмасиви*/
```

```
qSort(a, left, j-1);
qSort(a, j+1, right);
}
}
```

Швидке сортування вхідного масиву [4.0, 5.0, 3.0, 10.0, 1.0]:



■

Обґрунтування часових оцінок швидкого сортування можна знайти в [8, 60, 65]. Швидке сортування дає в середньому $O(n\log_2 n)$ порівнянь і обмінів при впорядкуванні n елементів. Проте, якщо значення опорної точки на кожному кроці дорівнює найбільшому елементу, то цей алгоритм вимагає $O(n^2)$ порівнянь. Тому слід уважно обирати метод визначення опорної точки (за можливості з урахуванням природи масиву, що сортується). На практиці цей алгоритм виявляється одним із найкращих серед алгоритмів з оцінкою $O(n\log_2 n)$. Швидке сортування, на відміну від сортування злиттям, не вимагає додаткової пам'яті та зберігає ефективність для систем із віртуальною пам'яттю.

4.1.7. ВИБІР МЕТОДУ СОРТУВАННЯ

Для швидкого сортування даних можна скористатися стандартною функцією **`qsort()`**, що входить до стандартної бібліотеки C (див. підрозд. 3.6.6). Однак різні підходи до сортування мають різні характеристики. Незважаючи на те, що деякі способи сортування можуть бути в середньому кращими за інші, жоден алгоритм не є ідеальним для всіх випадків. Тому широкий набір алгоритмів сортування – корисний додаток у інструментарії будь-якого програміста.

Пояснимо коротко, чому та сама функція **qsort()** не забезпечує універсального розв'язку всіх завдань сортування. По-перше, функцію **qsort()** неможливо застосувати в усіх ситуаціях – вона сортує лише масиви в пам'яті, але не може сортувати дані, що зберігаються, наприклад, у зв'язаних списках. По-друге, **qsort()** – параметризована функція, завдяки чому вона може обробляти широкий набір типів даних, але робить це повільніше, ніж еквівалентна функція, розрахована на якийсь один фіксований тип даних. Незважаючи на те, що зазвичай оптимальним є швидке сортування, воно не є кращим у всіх випадках. Наприклад, при сортуванні дуже малих списків (менше 100 елементів) додатковий обсяг роботи, пов'язаний із рекурсивними викликами швидкого сортування, може нівелювати його переваги. У таких рідких випадках один із простих методів сортування (навіть метод бульбашки) може працювати швидше. До того ж, якщо відомо, що список уже майже впорядкований або є вимоги щодо стійкості алгоритму, то якийсь інший алгоритм може виявитися прийнятнішим за швидке сортування.

***Література для СР:** алгоритми сортування й пошуку – [8, 60, 65, 77, 95, 125].

Контрольні запитання та вправи

1. Сформулювати задачі пошуку й сортування кортежів.
2. Що таке ключі елементів і яка їхня роль у задачах пошуку й сортування?
3 Що таке таблиця й рядок таблиці?
4. Що таке ключове поле рядка?
5. Сформулювати алгоритм послідовного пошуку.
6. Що таке бінарний пошук?
7. Порівняти часову складність послідовного й бінарного пошуків.
8. Що таке бульбашкове сортування?
9. Що таке сортування вставленням?
10. Порівняти часову складність бульбашкового сортування й сортування вставленням.
11. Що таке злиття двох послідовностей?
12. Сформулювати алгоритм сортування злиттям. Яка його часова складність?
13. Сформулювати алгоритм швидкого сортування.
14. Які переваги й недоліки застосування функція **qsort()**?
15. Порівняти часову складність сортування злиттям і швидкого сортування.

16. Дано послідовність цілих чисел $a_1,..,a_n$. Упорядкувати її за зростанням удосконаленим методом вставлення. Переглядати послідовно $a_2,..,a_n$ і кожний новий елемент вставляти на відповідне місце в уже впорядковану сукупність $a_1,..,a_i$. Це місце визначається методом бінарного пошуку.
17. Дано послідовність цілих чисел $a_1,..,a_n$. Упорядкувати за зростанням тільки додатні її члени, решту залишити на своїх місцях.
18. Замінимо лінійний порядок на ключах частковим. Як відсортувати масив із такими ключами?
19. Дано послідовність $a_1,..,a_n$: 1) цілих чисел; 2) раціональних чисел, поданих у вигляді структур (див. вправу 21 із підрозд. 3.7); 3) двовимірний числовий масив. Упорядкувати їх за спаданням за допомогою функції **`qsort()`** (див. підрозд. 3.6.6). У п. 3) порівнюється середнє квадратичне рядків.
20. Упорядкувати надвеликий одновимірний цілочисловий масив розміром 500 Мбайт.
21. Надвеликий одновимірний символьний масив розміром 1 Гбайт зберігається у файлі **`bigarray.dat`**. Упорядкувати його за зростанням.

4.2. Лінійна алгебра

- Прямий алгоритм множення матриць
- Алгоритм Винограда для множення матриць
- Алгоритм Штрассена для множення матриць
- Системи лінійних рівнянь. Метод Гаусса – Жордана
- Розріджені матриці

Ключові слова: *додавання й віднімання матриць, прямий алгоритм множення матриць, алгоритм Винограда, алгоритм Штрассена, метод Гаусса – Жордана, розріджена матриця, списокове й повне рядкове зображення RR(C) розрідженої матриці.*

Матриці означені в підрозд. 1.4.4 як кортежі фіксованої довжини з однотипними елементами – теж кортежами фіксованої довжини. Матриці й операції з ними застосовуються в різноманітних задачах, таких як моделювання фізичних об'єктів, наукові та економічні розра-

хунки, комп'ютерна графіка тощо. Зокрема, комп'ютерна графіка вимагає великого обсягу обчислень із багаточленами й матрицями. Ці обчислення зазвичай виконуються для кожної точки екрана, тому навіть незначне поліпшення ефективності окремих алгоритмів може привести до помітного прискорення всієї програми.

Найпростіший спосіб подання матриці – у вигляді двовимірного масиву. Числа розміщуються в матриці в рядках і стовпчиках:

```
int matrInt[4] [3];
float matrFloat [4] [3];
```

Дві матриці однакового розміру можна поелементно додати чи відняти одну від одної. Алгоритми додавання й віднімання матриць реалізуються двома вкладеними циклами.

Приклад 4.7. Додавання й віднімання матриць:

Лістинг.

```
/*додавання й віднімання матриць*/
#define Size1 5
#define Size2 5
… … …

double MatrA[Size1][Size2];
double MatrB[Size1][Size2];
double MatrSum[Size1][Size2];   /*сума матриць*/
double MatrRemainder[Size1][Size2]; /*різниця матриць*/

int i,j,k;

for (i=0; i<Size1; i++){
     for (j=0; j<Size2; j++){
             MatrSum[i][j] = MatrA[i][j]+ MatrB[i][j];
             MatrRemainder[i][j] = MatrA[i][j]- MatrB[i][j];
     }
}
```

■

4.2.1. ПРЯМИЙ АЛГОРИТМ МНОЖЕННЯ МАТРИЦЬ

Якщо кількість стовпчиків у першій матриці збігається з кількістю рядків у другій, то ці матриці можна перемножити (див. підрозд. 1.4.4).

Прямий алгоритм множення матриць реалізується трьома вкладеними циклами.

Приклад 4.8. Прямий алгоритм множення матриць:

Лістинг.

```
/*множення матриць*/
#define count1 5
#define count2 5
#define count3 5

double a[count1][count2];
double b[count2][count3];
double matrMult [count1][count3];
…
int i,j,k;

for(int i=0; i<count1; i++){
    for(int j=0; j<count3; j++){
          matrMult[i][j]=0;
          for(int k=0; k<count2; k++){
                matrMult[i][j]+=a[i][k]*b[k][j];
          }
    }
}
```

Зазначимо, що присвоювання `matrMult[i][j]+=a[i][k]*b[k][j]` реалізується ефективніше, ніж звичайне `MatrMult[i][j]=MatrMult[i][j]+ a[i][k]*b[k][j].`

Часова складність (за кількістю операцій) даного алгоритму становить `count1`$\times$`count2`$\times$`count3` ■

При множенні квадратних матриць розміром $n \times n$ кількість виконаних операцій має порядок $\mathrm{O}\left(n^3\right) = \mathrm{O}\left(n^{\log_2 8}\right)$.

Перевагою запропонованого алгоритму є простота, але він повільний. Розглянемо ефективніші алгоритми.

4.2.2. АЛГОРИТМ ВИНОГРАДА ДЛЯ МНОЖЕННЯ МАТРИЦЬ

Кожен елемент добутку двох матриць є скалярним добутком відповідних рядка і стовпчика. Розглянемо скалярний добуток двох векторів – $V = (v_0, v_1, v_2, v_3)$ та $W = (w_0, w_1, w_2, w_3)$: $V \circ W = v_0 w_0 + v_1 w_1 + v_2 w_2 + v_3 w_3 = (v_0 + w_1)(v_1 + w_0) + (v_2 + w_3)(v_3 + w_2) - v_0 v_1 - v_2 v_3 - w_0 w_1 - w_2 w_3$. Праву частину останнього виразу можна попередньо вирахувати й запам'ятати для кожного рядка першої матриці й кожного стовп-

чика другої. Це дозволяє виконувати для кожного елемента лише перші два множення та п'ять додавань, а також наступні два додавання.

Ш. Виноград для множення матриці G розміром $n \times m$ на матрицю H розміром $m \times k$ запропонував алгоритм, названий його іменем. Результат записується в матрицю R розміром $n \times k$:

$$a_i = \sum_{l=0}^{m/2} g_{i\,2l} g_{i\,2l+1}\,;\; b_j = \sum_{l=0}^{m/2} h_{2l\,j} h_{(2l+1)\,j}\,;$$

$$g_{ij} = \sum_{l=0}^{m/2} ((g_{i\,2l+1} + h_{2l\,j})(g_{i\,2l} + h_{(2l+1)\,j})) - a_i - b_j + (g_{i\,m-1} h_{m-1\,j})(m - \text{непарне}).$$

Приклад 4.9. Алгоритм Винограда:

Лістинг.

```
/*множення матриць методом Винограда*/
#define count1 5
#define count2 5
#define count3 5

double a[count1][count2];
double b[count2][count3];

double matrMult [count1][count3]; //результуюча матриця
double rowFactor[count1];  //сума добутків у кожному рядку
double columnFactors[count3]; //сума добутків у кожному стовпчику
//...
int i, j, k, d=(count2)/2;

//обчислення columnFactors для a
void calcRowFactors()
{
    for(i=0; i<count1; i++){
          rowFactor[i]=a[i][0] * a[i][1];
                for(j=1; j<d; j++){
                      rowFactor[i]+=a[i][2*j] * a[i][2*j+1];
          }
    }
}

//обчислення columnFactors для b
void calcColumnFactors()
{
    for(i=0; i<count3; i++){
          columnFactors[i]=b[0][i] * b[1][i];
          for(j=1; j<d; j++){
                columnFactors[i]+=b[2*j][i]* b[2*j+1][i];
          }
```

```
    }
}

//обчислення матриці matrMult
void mullMatr()
{
    for (i=0; i<count1; i++){
          for(j=0; j<count3; j++){
                matrMult[i][j]=-rowFactor[i] - columnFactors[j];
                for (k=0; k<d; k++){
                matrMult[i][j]+=(a[i][2*k+1]+b[2*k][j])*(a[i][2*k]+
b[2*k+1][j]);
                }
          }
    }
}

//обчислення матриці matrMult для випадку непарної розмірності
void mullMatr_1()
{
    if ((2 *((count2)/2))!=count2) {
          for(i=0; i<count1; i++){
                for(j=0; j<count3; j++){
                  matrMult[i][j]+=a[i][count2-1] * b[count2-1][j];
                }
          }
    }
}

int main()
{
  //... ініціалізація матриць a та b

  calcRowFactors();
  calcColumnFactors();
  mullMatr();
  mullMatr_1();
  // ...
  system("PAUSE");
  return 0;
}
```

■

Увага! Дана програма діє некоректно при `count2=1` ►

Верхня оцінка часової складності алгоритму Винограда для квадратних матриць – $O\left(n^{2.375477}\right)$.

4.2.3. АЛГОРИТМ ШТРАССЕНА ДЛЯ МНОЖЕННЯ МАТРИЦЬ

Розглянемо відкритий Штрассеном рекурсивний алгоритм, що множить дві $(n \times n)$-матриці за час $O\left(n^{\log_2 7}\right) = O\left(n^{2.81}\right)$.

Алгоритм Штрассена діє за принципом "розділяй і володій". Він працює з квадратними матрицями, розмір яких є степенем двійки, але настільки ефективний, що іноді доцільно розширити матриці до квадратних (додаванням нульових рядків чи стовпчиків), і при цьому він все одно дає виграш.

Нехай потрібно обчислити добуток двох матриць $(n \times n)$: $C = AB$. Нехай n є точним степенем двійки, тоді розділимо кожну з матриць A, B та C на 4 блоки розміром $(n/2 \times n/2)$. Перепишемо рівняння $C = AB$ таким чином:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} e & g \\ f & h \end{pmatrix},$$

де $r = ae + bf$, $s = ag + bh$, $t = ce + df$, $u = cg + dh$.

Тут відбувається вісім множень матриць розміром $n/2$, тому в такому вигляді алгоритм, як і послідовний алгоритм, працює за час $\mathrm{O}\left(n^{\log_2 8}\right)$. Алгоритм Штрассена пропонує схему, за якої в рекурсивному алгоритмі можна обійтися лише сімома множеннями матриць розміром $n/2$, тим самим скоротивши час до $\mathrm{O}\left(n^{\log_2 7}\right) = \mathrm{O}\left(n^{2.81}\right)$.

Алгоритм Штрассена множить дві матриці $(n \times n)$ A та B таким чином:

$x_1 = ag - ah$;

$x_2 = ah + bh$;

$x_3 = ce + de$;

$x_4 = df - de$;

$x_5 = ae + ah + de + dh$;

$x_6 = bf + bh - df - dh$;

$x_7 = ae + ag - ce - cg$.

$r = x_5 + x_4 - x_2 + x_6 = ae + de - d + bf$;

$s = x_1 + x_2 = ag + bh$;

$t = x_3 + x_4 = ce + df$;

$u = x_5 + x_1 - x_3 - x_7 = cg + dh$.

Отже, при множенні двох матриць 2×2 алгоритм виконує 7 множень і 18 додавань. На практиці алгоритм Штрассена застосовується порівняно рідко через суттєву величину константи, що міститься в асимптотичному виразі для часу його роботи. Його застосування виправдане для великих матриць (від 45×45) з малою кількістю нулів.

4.2.4. СИСТЕМИ ЛІНІЙНИХ РІВНЯНЬ. МЕТОД ГАУССА – ЖОРДАНА

Системи лінійних рівнянь застосовуються в різноманітних задачах. Таку систему можна записати у вигляді матричного рівняння.

Система лінійних рівнянь із n рівнянь із n невідомими має вигляд

$$a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 + \ldots + a_{0,n-1}x_{n-1} = b_0$$
$$a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,n-1}x_{n-1} = b_1$$
$$\ldots$$
$$\ldots$$
$$\ldots$$
$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 + \ldots + a_{n-1,n-1}x_{n-1} = b_{n-1}.$$

Розглянемо метод Гаусса – Жордана для розв'язання систем лінійних рівнянь.

Систему лінійних рівнянь можна зобразити у вигляді матриці розміром $n+1\times n$:

$$\begin{pmatrix} a_{0,0} & \ldots & a_{0,n-1} & b_0 \\ a_{1,0} & \ldots & a_{1,n-1} & b_1 \\ \ldots & \ldots & \ldots & \ldots \\ a_{n-1,0} & \ldots & a_{n-1,0} & b_{n-1} \end{pmatrix}.$$

Алгоритм Гаусса – Жордана:

1. Вибираємо перший стовпчик ліворуч, в якому є принаймні одне ненульове значення.
2. Якщо верхнє число в цьому стовпчику – нуль, то весь перший рядок матриці міняємо з тим рядком, де у стовпчику немає нуля.
3. Усі елементи першого рядка ділимо на верхній елемент вибраного стовпчика.
4. Від усіх рядків, окрім першого, віднімаємо перший рядок, помножений на перший елемент кожного рядка, щоб отримати в першій позиції кожного з решти рядків нулі.

5. Викреслюємо перший рядок і перший стовпчик. Повторюємо процедуру (кроки 1-5) $n-1$ разів, поки не отримаємо верхню трикутну матрицю.

6. Віднімаємо від передостаннього рядка останній, помножений на відповідний коефіцієнт так, щоб у останньому рядку залишилася лише одиниця на головній діагоналі.

7. Повторюємо крок 6 для всіх наступних рядків. У результаті отримуємо одиничну матрицю й розв'язок на місці вільного вектора.

Систему лінійних рівнянь будемо подавати в текстовому файлі (коефіцієнти через пробіл, рядки з нового рядка) у вигляді

$a_{0,0}a_{0,1} \ \ldots a_{0,n-1} \ b_0$

…

$a_{n\text{-}1,0} \ a_{n\text{-}1,1}\ldots a_{n\text{-}1,n\text{-}1} \ b_{n\text{-}1}$.

Приклад 4.10. Метод Гаусса – Жордана.

Лістинг.

```
/*метод Гаусса – Жордана*/
#include <stdio.h>
#include <process.h>

float **a, *b, *x;
int n; /*розмір матриці*/
char filename[256]; /*ім'я файла з матрицею; файл зберігається
в одній директорії з програмою*/
FILE* InFile=NULL;

/*вивільняє пам'ять*/
void freeMatrix(){
int i;
 for(i=0; i<n; i++){
    delete [] a[i];
 }
 delete [] a;
 delete [] b;
 delete [] x;
}

void countNumLines(){
 /*підраховує кількість рівнянь у системі*/
 int nelf=0;
 do{
    nelf=0;
 while(fgetc(InFile)!='\n' && !feof(InFile)) nelf=1;
```

```
      if(nelf) n++;
    } while(!feof(InFile));
  }

  void allocMatrix(){
  /*виділяє пам'ять для масивів, що зберігають коефіцієнти рів-
нянь, вільні члени й розв'язки*/
    int i,j;
    x=new float[n];
    b=new float[n];
    a=new float*[n];
    if(x==NULL || b==NULL || a==NULL){
      printf("\nNot   enough   memory   to   allocate   for   %d
equations.\n", n);
      exit(-1);
    }
    for(i=0; i<n; i++){
      a[i]=new float[n];
      if(a[i]==NULL){
            printf("\nNot   enough   memory   to   allocate   for   %d
equations.\n", n);
      }
    }
    for(i=0; i<n; i++){
      for(j=0; j<n; j++){
            a[i][j]=0;
      }
      b[i]=0;
      x[i]=0;
    }
  }
  /*підставляє розв'язки в систему та друкує поруч те, що отриму-
ємо в лівій частині рівняння. Для порівняння друкує поруч вільні
члени. Два стовпчики мають збігатися*/
  void testSolve(){
   //test that ax=b
   int i=0,j=0;
   printf("\n");
   for(i=0; i<n; i++){
      float s=0;
      for(j=0; j<n; j++){
            s+=a[i][j]*x[j];
      }
      printf("%f\t%f\n", s, b[i]);
    }
  }
```

```
/*зчитує з файла коефіцієнти й вільні члени в масиви*/
void readMatrix(){
 int i=0,j=0;
for(i=0; i<n; i++){
    for(j=0; j<n; j++){
          fscanf(InFile, "%f", &a[i][j]);
    }
    fscanf(InFile, "%f", &b[i]);
 }
}

/*виводить систему лінійних рівнянь на екран*/
void printMatrix(){
int i=0,j=0;
 printf("\n");
 for(i=0; i<n; i++){
    for(j=0; j<n; j++){
          printf("%+f*X%d", a[i][j], j);
    }
    printf("=%f\n", b[i]);
 }
}

/*друкує результат*/
void printResult(){
 int i=0;
 printf("\n");
 printf("Result\n");
 for(i=0; i<n; i++){
    printf("X%d=%f\n", i, x[i]);
 }
}
/*робить так, щоб на головній діагоналі не було нулів*/
void diagonal(){
 int i, j, k;
 float temp=0;
 for(i=0; i<n; i++){
 if(a[i][i]==0){
     for(j=0; j<n; j++){
     if(j==i) continue;
     if(a[j][i] !=0 && a[i][j]!=0){
           for(k=0; k<n; k++){
                temp=a[j][k];
                a[j][k]=a[i][k];
                a[i][k]=temp;
           }
```

```
            temp=b[j];
            b[j]=b[i];
            b[i]=temp;
            break;
      }
      }
 }
 }
}

int main(){
 int i=0,j=0, k=0;
 do{
    printf("\nInput filename:");
    scanf("%s", filename);
    InFile=fopen(filename, "rt");
 }while(InFile==NULL);
 countNumLines();
 allocMatrix();
 rewind(InFile);
readMatrix();
diagonal();
 fclose(InFile);
 printMatrix();
for(k=0; k<n; k++){
    for(i=k+1; i<n; i++){
          if(a[k][k]==0){
          printf("\nSolution is not exist.\n");
          return 0;
      }
      float M=a[i][k]/a[k][k];
      for(j=k; j<n; j++){
          a[i][j]-=M * a[k][j];
      }
      b[i]-=M*b[k];
 }
 }
 printMatrix();
 for(i=n-1; i>=0; i--){
    float s=0;
    for(j=i; j<n; j++){
          s+=a[i][j]*x[j];
    }
    x[i]=(b[i] - s)/a[i][i];
 }
```

```
 InFile=fopen(filename, "rt");
 readMatrix();
 fclose(InFile);
 printMatrix();
 testSolve();
 printResult();
 freeMatrix();
 scanf("%d", &i);
 return 0;
}
```

■

4.2.5. РОЗРІДЖЕНІ МАТРИЦІ

Часто доводиться виконувати операції з матрицями, більшість елементів яких дорівнюють нулю. Такі матриці називаються *розрідженими*. Операції над ними та їх збереження можна реалізувати ефективніше ніж у загальному випадку. Складність алгоритмів роботи з розрідженими матрицями зазвичай визначається кількістю ненульових елементів матриці.

При збереженні розріджених матриць дотримуються двох основних вимог: збереження переважно ненульових елементів (інколи з невеликою кількістю нульових) і зручність оперування матрицями. Найбільшого поширення набули два способи зображення розрідженої $m \times n$ матриці A. Перший – *повне рядкове зображення* RR(C) (від англ. Row-wise Representation Complete). У ньому замість одного двовимірного масиву використовуються три одновимірні:

1) **AN** – масив ненульових елементів матриці A;

2) **JA** – масив індексів стовпчиків ненульових елементів матриці A;

3) **IA** – масив покажчиків; його i-та компонента вказує, з якої позиції масивів **AN** та **JA** починається опис i-го рядка матриці A. Остання компонента додаткова й указує на номер першої вільної позиції в масивах **AN** та **JA**.

У загальному випадку опис **r**-го рядка матриці A зберігається в компонентах від **IA[r]** до **IA[r + 1]-1** масивів **AN** та **JA**. Якщо **IA[r + 1]=IA[r]**, то це означає, що **r**-й рядок – нульовий. Кількість елементів у масиві **IA** на одиницю більше кількості рядків початкової матриці, а в масивах **JA** та **AN** – дорівнює кількості ненульових елементів початкової матриці. Цей формат називають повним, оскільки зображена вся матриця A.

Залежно від того, як записуються в кожному рядку індекси стовпчиків у масиві **JA** (за порядком зростання чи ні), розрізняють упорядковані й невпорядковані зображення.

Приклад 4.11. Зображення розрідженої матриці:

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 3 & 0 & 0 & 2 \\ 0 & 0 & 0 & 2 & 0 & 1 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 4 & 0 & 0 \\ 7 & 6 & 0 & 0 & 8 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 9 \\ 0 & 0 & 5 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Тут

AN=(1, 3, 2, 2, 1, 4, 4, 7, 6, 8, 1, 9, 5),
JA=(1, 5, 8, 4, 6, 2, 6, 1, 2, 5, 8, 8, 3),
IA=(1, 4, 6, 8, 12, 13, 14) ■

Можна задати зображення у стовпчиковому форматі. Детально цей формат розглядати не будемо, оскільки його можна подати як рядкове зображення транспонованих матриць.

Другий спосіб подання розрідженої числової матриці $n \times m$ – *списковий*. Він полягає в побудові $n + m$ списків, вузли яких подають ненульові елементи рядків і стовпчиків матриці, і двох масивів довжиною n та m, що задають голови цих списків. Кожен вузол – це п'ятірка $(i, j, a[i, j], ptr1, ptr2)$, де i – відповідно номер рядка, j – номер стовпчика, $a[i, j]$ – сам ненульовий елемент матриці, $ptr1$ – покажчик на наступний ненульовий елемент у i-му рядку, $ptr2$ – покажчик на наступний ненульовий елемент у j-му стовпчику.

Вибір структури даних для подання розрідженої матриці визначається алгоритмом її обробки. Наприклад, для додавання матриць зручно задати їх у RR(C)-форматі. Тоді виконання операції зведеться до операції злиття ненульових елементів однакових за номером рядків обох матриць, якщо елементи в них мають різні позиції, і додавання елементів, розташованих на спільних позиціях. Обчислення добутку $y = Ax$ полягатиме в підсумовуванні елементів масиву x, що задаються елементами масиву **JA**, з ваговими коефіцієнтами, заданими елементами масиву **AN**, тобто $y_i = \sum_{j:\, a_{ij} \neq 0} a_{ij} x_j$.

***Література для СР:** алгоритми лінійної алгебри – [65, 77, 151].

Контрольні запитання та вправи

1. Описати алгоритми додавання, віднімання й прямого множення матриць. Яка їхня часова складність?
2. Продемонструвати роботу алгоритму Штрассена при множенні матриць $\begin{pmatrix} 1 & 5 \\ 3 & 2 \end{pmatrix}$ та $\begin{pmatrix} 0 & 2 \\ 10 & 4 \end{pmatrix}$.
3. Написати програму, що обчислює добуток матриць методом Винограда. Вхідними параметрами є розмірність матриці й сама матриця.
4. Протестувати програму з прикл. 4.10. Скористатися системами рівнянь із вправи 42 підрозд. 3.4.
5. Що таке RR(C)-зображення розрідженої матриці?
6. Що таке списковe зображення розрідженої матриці?
7. Для розрідженої матриці A побудувати RR(C)-зображення:

а) $A = \begin{pmatrix} 0 & 0 & 5 & 0 & 4 & 0 & 0 \\ 0 & 0 & 6 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 9 & 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$; б) $A = \begin{pmatrix} 1 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 \\ 1 & 5 & 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$;

в) $A = \begin{pmatrix} 0 & 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 5 & 0 & 0 \\ 6 & 0 & 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$; г) $A = \begin{pmatrix} 0 & 0 & 0 & 9 & 8 & 0 & 0 & 7 \\ 0 & 0 & 6 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 3 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$;

д) $A = \begin{pmatrix} 0 & 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 4 & 5 & 0 \end{pmatrix}$; е) $A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$.

8. Для RR(C)-зображення відновити матрицю A (з точністю до нульових стовпчиків праворуч):
   а) `AN=(1, 13, 5, 7, 9, 1, 2), JA=(1, 2, 4, 1, 3, 5, 6), IA=(1, 4, 6, 7, 8);`
   б) `AN=(1, 2, 3, 4, 5, 6), JA=(1, 2, 3, 4, 5, 6), IA=(1, 4, 5, 6, 6);`
   в) `AN=(9, 10, 5, 3, 2, 1, 6), JA=(1, 5, 7, 3, 2, 1, 7), IA=(1, 3, 3, 5, 6, 7);`

г) **AN=(1, 2, 5, 4, 3, 2, 9), JA=(1, 5, 4, 1, 3, 5, 4), IA=(1, 3, 5, 7, 8);**

д) **AN=(1, 1, 2, 4, 6, 2, 5), JA=(1, 2, 4, 1, 3, 5, 6), IA=(1, 4, 6, 8, 8);**

е) **AN=(5, 3, 5, 3, 5, 3), JA=(1, 2, 4, 1, 3, 5), IA=(1, 3, 5, 6, 7).**

9. Написати програму, яка на вході отримує зображення матриці у вигляді двовимірного масиву, а на виході видає її RR(C)-зображення.
10. Написати програму, яка на вході отримує RR(C)-зображення матриці, а на виході видає цю матрицю у вигляді двовимірного масиву.
11. Написати програму, що на вході отримує два RR(C)-зображення розріджених матриць однакової розмірності й підраховує їхні:
    а) суму;
    б) різницю;
    в) добуток.

    Результат подати у вигляді RR(C)-зображення. Визначити кількість використаних дій додавання, віднімання, множення.
12. Написати клас, що реалізує основні операції над розрідженими квадратними матрицями дійсних чисел: введення-виведення, додавання, віднімання, множення, множення на вектор, обернення, тріангуляція. Матриці подаються за допомогою списків.

4.3. Списки, стеки, черги й бінарні дерева

- Списки
- Стеки
- Черги
- Бінарні дерева
- Контейнерні типи

Ключові слова: *динамічні структури даних, зв'язаний список, голова списку, однозв'язний лінійний список, двозв'язний список, загальний список, стек і черга як масив і список, операції* **empty, pop** *та* **push** *для стеків і черг, бінарне дерево, обхід бінарного дерева, послідовні контейнерні типи векторів, списків і двосторонніх черг, асоційовані контейнерні типи відношень і множин, ітератор, стандартні узагальнені алгоритми*

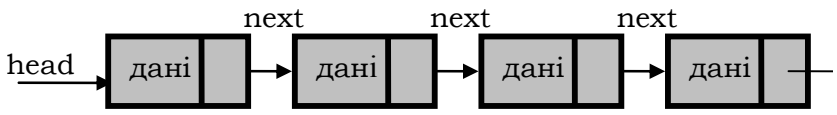
Часто доводиться працювати з множинами, що змінюють потужність у процесі виконання програми. Для реалізації таких множин потрібні спеціальні структури даних, які отримали назву *динамічних*. Найчастіше елемент динамічної структури даних – це структура, що містить поля, одне з яких розглядається як *ключ*, призначений для ідентифікації елемента, а інші – як необхідна інформація, що зберігається разом із ключем, у тому числі й для зв'язку з іншими елементами множини. Пошук елемента множини відбувається за ключем.

Розглянемо основні динамічні структури даних, які застосовуються для реалізації динамічних множин – зв'язані списки, стеки, черги й бінарні дерева.

4.3.1. СПИСКИ

Списки означені в підрозд. 1.4.4. Вони реалізуються за допомогою *зв'язаних списків*, які можуть бути однозв'язними та двозв'язними. Елементом однозв'язного списку є структура, що окрім ключа та іншої інформації містить покажчик на наступний елемент списку. Покажчиком в останньому елементі списку є **NULL**. Двозв'язний список містить покажчик як на наступний, так і на попередній елементи списку. Вибір типу списку залежить від конкретної задачі.

Однозв'язний лінійний список. У такому списку інформаційне поле кожного з елементів містить тільки ключ – певний атом. Сам атом має фіксовану (нединамічну) структуру. Це може бути число, слово, таблиця тощо. У такому списку можна пересуватися лише в одному напрямку – від *голови* (першого елемента) у його кінець. Щоб задати лінійний список, достатньо вказати адресу його голови:



Покажемо, як реалізуються основні операції для лінійних списків.

Приклад 4.12. Опис лінійного списку:

Лістинг.

```
struct Item {
 int data;          /*інформаційне поле*/
 struct Item *next; /*покажчик на наступний елемент*/
};
typedef struct Item *Itemptr; /*введення імені Itemptr для типу
покажчиків на структуру Item*/
```

```
Itemptr HeadList; /*покажчик на голову списку*/
```

■

Послідовне перебирання елементів лінійного списку здійснюється просто: почати з початку та йти за покажчиками. Наведена нижче функція виводить на екран усі значення зі списку.

Приклад 4.13. Друкування елементів лінійного списку:

Лістинг.

```
/*друкування значень елементів списку*/
void display(Itemptr start)
{
 while(start) {
 printf("%d\t", start->data);
 start=start->next;
 }
printf("\n");
}
```

■

При виклику функції **display** параметр **start** задає голову списку, потім функція переходить до наступного елемента, на який указує поле **next**. Процес припиняється, коли **next** дорівнює нулю.

Для отримання елемента зі списку потрібно просто пройти низкою посилань. Розглянемо приклад функції пошуку за ключовим полем **data**.

Приклад 4.14. Пошук за ключем:

Лістинг.

```
/*пошук попередника елемента*/
Itemptr getPrev(Itemptr start,Itemptr elem)
{
 if(elem){
     Itemptr prev=NULL;
     Itemptr p=start;
     while((p)&&(p!=elem)) {
           prev=p;
           p=p->next;
     }
     return prev;
 } else return NULL; /*елемент не знайдено*/
}

/*пошук попередника елемента за значенням; результат – NULL,
якщо елемент не знайдено або він у голові списку*/
```

```
Itemptr getPrevElem(Itemptr start, int n)
{
 Itemptr prev=NULL;
 while(start) {
 if(n==start->data) return prev;
 prev=start;
    start=start->next;
 }
 return NULL; /*елемент не знайдено*/
}

/*пошук елемента за його значенням*/
Itemptr search(Itemptr start, int n)
{
 while(start) {
 if(n==start->data) return start;
 start=start->next;
 }
 return NULL; /*елемент не знайдено*/
}

/*пошук останнього елемента*/
Itemptr getLast(Itemptr start)
{
while(start->next)
    start=start->next;
    return start;
}
■
```

Розглянемо вставлення елементів у лінійний список. Нехай потрібно вставити елемент у кінець списку.

Приклад 4.15. Вставлення й вилучення елемента в лінійному списку:

Лістинг.

```
/*додавання нового елемента зі значенням n у кінець списку*/
Itemptr addItem (Itemptr start, int n)
{
    Itemptr p;

    p=(Itemptr)malloc(sizeof(Item));
    p->data=n;
    p->next=NULL;
    if(!start) start=p;
    else{
```

```
            start=getLast(start);
            start->next=p;
      }
      return p;
   }

   /*вилучення елемента з однозв'язного списку, розташованого за
елементом prevItem; якщо видаляється голова списку, то
prevItem=NULL; результат 1 – видалення відбулося, 0 – видалення не
виконане*/
   int deleteItem(Itemptr start,Itemptr prevItem)
   {
      if(start) /*список не порожній*/
      {
            Itemptr p;
      /*видаляється голова списку*/
            if(!prevItem) {
                  p=start;
                  start=start->next;
                  free(p);
                  return 1;
            }
            else{
                  p=prevItem->next;
                  if(p) /*попередник не останній*/{
                        prevItem->next=p->next;
                        free(p);
                        return 1;
                  } else return 0;
            }
      }
      else return 0;
   }
   ■
```

При роботі з упорядкованим списком зручно додавати елемент до списку відразу в потрібну позицію.

Приклад 4.16. Пошук елемента у відсортованому списку:

Лістинг.

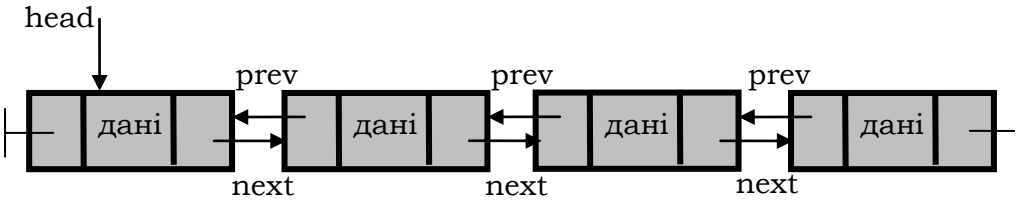
```
   /*пошук елемента за значенням у відсортованому списку; резуль-
тат – адреса його попередника; NULL – якщо елемент перший у списку
або його не знайдено*/
   Itemptr getPrevPositionElem(Itemptr start, int n)
   {
```

```
    Itemptr prev=NULL;
    while((start)&&(start->data<n)) {
    prev=start;
       start=start->next;
    }
    return prev; /*елемент не знайдено*/
   }

   /*вставлення елемента зі значенням у відсортований список у по-
трібну позицію*/
   void addElemSort(Itemptr start,int n)
   {
       Itemptr prev=getPrevPositionElem(start, n);
       Itemptr p=(Itemptr)malloc(sizeof(Item));
       p->data=n;
       p->next=NULL;
       if(!prev) /*елемент перший*/{
             p->next=start;
             start=p;
       } else /*не перший*/{
             p->next=prev->next;
             prev->next=p;
       }
   }
```

■

Недоліком однозв'язного списку є те, що в ньому не можна рухатися у зворотному напрямку, тому частіше використовують двозв'язний список.

Двозв'язний список. У двозв'язному списку кожен елемент інформації містить покажчики на наступний і попередній елементи. Таким списком можна пересуватися в довільному напрямку – як у початок, так і в кінець:



Проілюструємо роботу з двозв'язним списком.

Приклад 4.17. Оголошення двозв'язного списку:

Лістинг.

```
typedef struct item {
 int data;
 struct item *next; /*покажчик на наступний елемент*/
 struct item *prev; /*покажчик на попередній елемент*/
} Item;

typedef Item *Itemptr;
Itemptr HeadList; /*голова списку*/
```

■

Послідовне перебирання елементів двозв'язного списку здійснюється так само просто, як і у випадку однозв'язного. Наведена нижче функція виводить на екран усі імена зі списку.

Приклад 4.18. Виведення значень елементів двозв'язного списку:

Лістинг.

```
/* Виведення значень елементів списку*/
void display(Itemptr start)
{
 while(start) {
 printf("%d\t", start->data);
 start=start->next;
 }
printf("\n");
}
```

■

При виклику функції **display** параметр **start** задає голову списку. Після цього функція переходить до наступного елемента, на який указує поле **next**. Процес припиняється, коли **next** дорівнює нулю.

Розглянемо приклад функції пошуку за ключовим полем **data** (на відміну від однозв'язного списку потрібно шукати не попередника, а сам елемент).

Приклад 4.19. Пошук, додавання й вилучення елемента:

Лістинг.

```
/*пошук елемента за ключем; результат – NULL, якщо елемент не
знайдено*/
Itemptr getElem(Itemptr start, int n)
{
 while(start) {
 if(n==start->data) return start;
    start=start->next;
```

```
    }
    return NULL; /*елемент не знайдено*/
   }

   /*пошук останнього елемента*/
   Itemptr getLast(Itemptr start)
   {
       while(start->next)
       start=start->next;
       return start;
   }

   /*пошук елемента за значенням у відсортованому списку; резуль-
тат – адреса його попередника; NULL – якщо елемент у голові списку
або його не знайдено*/
   Itemptr getPrevPositionElem(Itemptr start, int n)
   {
    Itemptr prev=NULL;
    while((start)&&(start->data<n)) {
    prev=start;
       start=start->next;
    }
    return prev; /*елемент не знайдено*/
   }

   /*додавання нового елемента зі значенням n у кінець двозв'язно-
го списку*/
   Itemptr addItem (Itemptr start, int n)
   {
       Itemptr p;
       p=(Itemptr)malloc(sizeof(Item));
       p->data=n;
       p->next=NULL;
       p->prev=NULL;
       if(!start) start=p;
       else{
             start=getLast(start);
             start->next=p;
             p->prev=start;
       }
       return p;
   }

   /*Видалення елементів із двозв'язного списку спрощується порів-
няно з однозв'язним*/
   /*видалення елемента із двозв'язного списку за його адресою;
   результат 1 – видалення відбулося, 0 – видалення не виконане*/
```

```
int deleteItem(Itemptr start,Itemptr delItem)
{
    if((start)&&(delItem)) /*список не порожній*/
    {
          Itemptr p, p1;
          p=delItem->prev;
          p1=delItem->next;
          if(p) /*елемент не перший*/
                p->next=p1;
          else start=p1;
          if(p1) /*елемент не останній*/
                p1->prev=p;
          free(delItem);
          return 1;
    }
    else return 0;
}
```

■

Розглянемо роботу з упорядкованим двозв'язним списком.

Приклад 4.20. Пошук і додавання елемента:

Лістинг.

```
/*пошук елемента за значенням у відсортованому списку; результат – адреса його попередника; NULL – якщо елемент у голові списку або його не знайдено*/
Itemptr getPrevPositionElem(Itemptr start, int n)
{
 Itemptr prev=NULL;
 while((start)&&(start->data<n)) {
 prev=start;
    start=start->next;
 }
 return prev; /*елемент не знайдено*/
}

/*вставлення елемента зі значенням у відсортований список у потрібну позицію*/
void addElemSort(Itemptr start,int n)
{
    Itemptr prev=getPrevPositionElem(start, n);
    Itemptr p=(Itemptr)malloc(sizeof(Item));
    p->data=n;
    p->next=NULL;
    if(!prev) /*елемент перший*/{
          p->next=start;
```

```
            p->prev=prev;
            start=p;
      } else /*не перший*/{
            p->next=prev->next;
            p->prev=prev;
            prev->next=p;
      }
   }
```

■

Загальні списки. На відміну від лінійних списків, елементами яких є атоми, елементами загальних списків є інші списки, тобто це списки списків. Для їхнього подання використовують загальні зв'язані списки. Структури цих списків мають два поля. Перше – інформаційне – містить покажчик на голову певного списку, а друге – поле зв'язку – покажчик на наступний елемент списку. У разі відсутності останнього значення цього поля – **NULL**.

Приклад опису загального списку:

```
   struct Item {
    void *data;           /*інформаційне поле*/
    struct Item *next; /*покажчик на наступний елемент*/
   };
   typedef struct Item *Itemptr; /*введення імені Itemptr для типу
покажчиків на структуру Item*/
   Itemptr HeadList; /*покажчик на голову списку*/
```

■

Завдяки суто індуктивній природі загальних списків основу їх обробки становлять рекурсивні функції.

4.3.2. СТЕКИ

Стеки визначені в підрозд. 1.4.4 як кортежі однотипних елементів із відповідними операціями на них. Нагадаємо, що операціями зі стеками є: **read** – читання вершини, **pop** – усунення вершини, **push(c)** – додавання в кінець стеку нової компоненти, **empty** – перевірка стеку на порожність. У реалізаціях стеків перші дві операції об'єднуються в одну з ім'ям **pop**.

Стек реалізують у вигляді масиву або лінійного списку. Перший варіант наведено на рис. 4.1. Цілий масив **int stack[MAX]** подає елементи стеку, а змінна-індекс **int p** – його вершину, а саме наступний за вершиною елемент масиву. При такій реалізації **p==0** означає, що стек порожній.

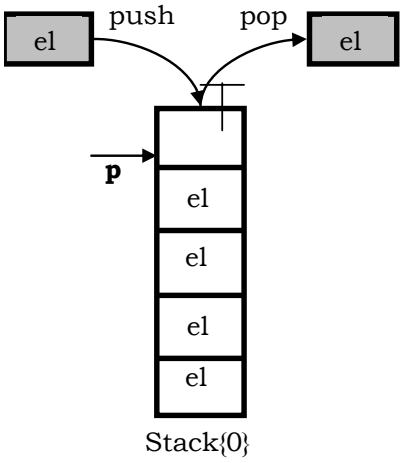


Рис. 4.1

Приклад 4.21. Стек цілих чисел у вигляді масиву:

Лістинг.

```
int stack[MAX];
int p=0; /*вершина стеку*/

/*функція empty – перевірка стеку на порожність*/
int empty ()
{return p<=0;}

/*функція push додає елемент у кінець стеку*/
void push(int i)
{
 if(p>=MAX) {
      printf("Стек повний\n");
      return;
 }
 stack[p]=i;
 p++;
}

/*функція pop знімає елемент із вершини стеку і повертає його
як значення*/
int pop(void)
{
if (empty ()) {
      printf("Стек порожній\n");
      return 0;
 }
```

```
 p--;
 return stack[p];
}
```

Нижче наведені різні варіанти обробки стеку:

Початковий вміст стеку	Функція та її аргумент	Результат	Вміст стеку після операції
NULL (порожній)	push(A)	void	A
A	push(B)	void	B A
B A	push(C)	void	C B A
C B A	pop()	C	B A
B A	pop()	B	A
A	push(D)	void	D A
D A	empty()	0	D A
D A	pop()	D	A
A	pop()	A	NULL
NULL	empty()	1	NULL

Стек можна реалізувати за допомогою лінійного списку однотипних структур, в якому функції **pop()**, **push()** та **empty()** працюють із головою списку, а покажчик вершини стеку встановлено на неї. Прикладом застосування такого стеку може бути перетворення арифметичного виразу на форму ОПЗ ■

Приклад 4.22. Перетворення виразу на форму ОПЗ:

Лістинг.

```
#include <stdio.h>
#include <stdlib.h>
/*Опис структури (елемента стеку операцій)*/
typedef struct st{
    char c;
    struct st *next;
} St;
typedef St *Stptr;

Stptr push(Stptr, char);
char pop(Stptr*);
int PRIOR(char);
int main()
{
```

```
      struct st *OPERS=NULL;
      char a[80], outstring[80];
      int k, point;

      do{
            puts("Введіть вираз, що завершується '='):");
            fflush(stdin);
                  /*Введення арифметичного виразу*/
            gets(a);
            k=point=0;
                  /*Повторюємо, поки не '=' */
            while(a[k]!='\0'&&a[k]!='=')
            {
                  /*Якщо черговий символ – ')', то виштовхуємо зі
стеку у вихідний рядок*/
            if(a[k]==')') {
            /*усі знаки операцій до найближчої дужки, що відкрива-
ється*/
                  while((OPERS->c)!='(')
outstring[point++]=pop(&OPERS);
            /*Видаляємо зі стеку дужку, що відкривається*/
                  pop(&OPERS);
            }
      /*Якщо черговий символ – літера, то переписуємо її у вихід-
ний рядок*/
            if(a[k]>='a'&&a[k]<='z') outstring[point++]=a[k];
            /*Якщо черговий символ – '(', то заносимо його у стек*/
            if(a[k]=='(') OPERS=push(OPERS, '(');
            if(a[k]=='+'||a[k]=='-'||a[k]=='/'||a[k]=='*')
            /*Якщо черговий символ – знак операції, то: */
            {/*якщо стек порожній, то записуємо в нього операцію*/
            if(OPERS==NULL) OPERS=push(OPERS, a[k]);
            /*якщо не порожній*/
            else
  /*якщо пріоритет нової операції більший за пріоритет операції
на вершині стеку, то заносимо нову операцію у стек*/
                  if(PRIOR(OPERS->c)<PRIOR(a[k]))
OPERS=push(OPERS, a[k]);
      /*якщо пріоритет менше, то переписуємо у вихідний рядок усі
операції з більшим чи рівним пріоритетом*/
            else {
            while((OPERS!=NULL)&&(PRIOR(OPERS->c)>=PRIOR(a[k])))
                  outstring[point++]=pop(&OPERS);
                  /*записуємо у стек нову операцію*/
```

```
            OPERS=push(OPERS, a[k]);
            }
            }
            /*Переходимо до наступного символу вхідного рядка*/
            k++;
            }
            /*після розгляду всього виразу*/
            while(OPERS!=NULL)
      /*Переписуємо всі операції зі стеку у вихідний рядок і друкуємо його*/
                  outstring[point++]=pop(&OPERS);
            outstring[point]='\0';
            printf("\n%s\n", outstring);
            fflush(stdin);
            puts("\nПовторити(y/n)?");
      } while(getchar()!='n');
      return 0;
  }
  /*Функція push записує у стек (на вершину якого вказує HEAD)
   символ a. Повертає покажчик на нову вершину стеку*/
  Stptr push(Stptr HEAD, char a)
  {
      Stptr PTR=(Stptr)malloc(sizeof(St));
      if(PTR==NULL) {
            puts("Не вистачає пам'яті");
            exit(-1);}
      PTR->c=a;
      PTR->next=HEAD;
      return PTR;
  }

  /*Функція pop видаляє символ із вершини стеку.
   Повертає символ, що видаляється*/
  char pop(Stptr *HEAD)
  {
      Stptr PTR;
      char a;
      if(*HEAD==NULL) return '\0';
      PTR=*HEAD;
      a=PTR->c;
      *HEAD=PTR->next;
      free(PTR);
      return a;
  }
```

```
/*Функція prior повертає пріоритет арифметичної операції*/
int PRIOR(char a)
{
    switch(a)
    {
          case '*':
          case '/':
          return 3;

          case '-':
          case '+':
          return 2;

          case '(':
          return 1;
    }
}
```

■

4.3.3. ЧЕРГИ

Черги означені в підрозд. 1.4.4 як кортежі однотипних елементів із відповідними операціями. Нагадаємо, що операціями з чергами є: **read** – читання вершини, **pop** – усунення голови черги, **push(c)** – додавання в кінець черги нової компоненти, **empty** – перевірка черги на порожність. У реалізаціях черг, як і стеків, перші дві операції об'єднуються в одну.

Функцію додавання елемента до черги позначимо **enqueue**, а функцію читання й вилучення елемента з черги – **dequeue**.

Тип черги складають послідовності однотипних компонент із доступом до першої (голова черги) та останньої компоненти. У черги є голова та хвіст. Структуру даних черги можна порівняти з чергою в магазині. Елемент, що додається до черги, опиняється в її хвості останнім, а елемент, що видаляється з черги, розташований у її голові, як той покупець, що відстояв найдовше.

Черги, як і стеки, подають за допомогою масивів і зв'язаних списків. Розглянемо перший варіант (рис. 4.2). Цілий масив **int queue[MAX]** зображує елементи черги, змінна-індекс **int rpos** – її голову, змінна-індекс **int spos** – наступний за останнім елемент хвоста черги в масиві. При такій реалізації **spos==0** означає, що черга порожня.

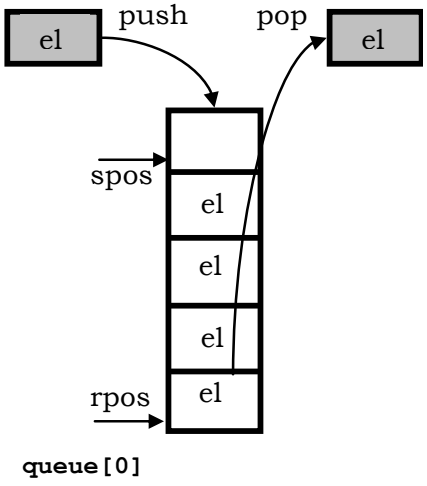


Рис. 4.2

Приклад 4.23. Черга на основі масиву цілих чисел:

Лістинг.

```
int queue[MAX];
int spos=0; /*індекс наступного вільного місця в черзі*/
int rpos=0; /*індекс елемента, що підлягає читанню*/

/*функція enqueue додає елемент у кінець черги*/
void enqueue (int i)
{
 if(spos==MAX) {
 printf("Список переповнено \n");
 return;
 }
 queue[spos]=i;
 spos++;
}

/*функція dequeue знімає перший елемент черги*/
int dequeue ()
{
 if(rpos==spos) {
 printf("Список порожній\n");
 return 0;
 }
 rpos++;
 return queue[rpos-1];
}
```

Варіанти обробки черг:

Початковий вміст черги	Функція та її аргумент	Результат	Вміст черги після операції
NULL (порожня)	enqueue(A)	Void	A
A	enqueue(B)	Void	A B
A B	enqueue(C)	Void	A B C
A B C	dequeue()	A	B C
B C	dequeue()	B	C
C	enqueue(D)	Void	C D
C D	dequeue()	C	D
D	dequeue()	D	NULL

■

4.3.4. БІНАРНІ ДЕРЕВА

Розглянемо реалізацію структури даних, що називається *бінарним деревом* (див. підрозд. 1.4.4). Незважаючи на те, що існує багато різних типів дерев, бінарні відіграють особливу роль, оскільки у відсортованому стані дозволяють дуже швидко виконувати вставлення, видалення й пошук. Кожен елемент бінарного дерева складається з інформаційної частини й покажчиків на лівий і правий елементи.

Більшість функцій, що працюють із деревами, рекурсивні внаслідок індуктивної побудови самих дерев. Спосіб упорядкування дерева залежить від того, як до нього здійснюється доступ. Процес почергового доступу до кожної вершини називається *обходом* дерева (англ. – tree traversal).

Бінарні дерева подаються за допомогою структур із покажчиками на себе:

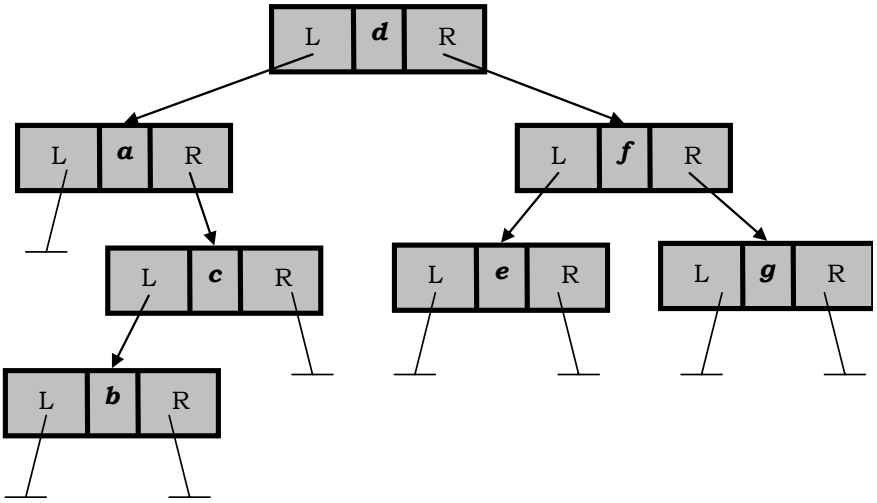
```
struct item {/*вузол числового дерева*/
 int data; /*число*/
 struct item *left; /*ліве піддерево*/
 struct item *right; /*праве піддерево*/
 } Item;
typedef Item *Itemptr;
tepedef Itemptr *Itemref;
```

При зв'язуванні різних структур типу **Item** утворюється бінарна деревоподібна структура. Ідучи за покажчиками **left**, **right**, можна обійти всі вузли дерева (здійснити його обхід).

Існують прості, але ефективні алгоритми обходу бінарного дерева. Основними є *прямий, внутрішній* і *обернений* порядки обходу. При

внутрішньому обході опрацьовується спочатку ліве піддерево, потім – корінь, а потім – праве піддерево. При прямому обході спочатку опрацьовується корінь, потім – ліве піддерево, а потім – праве. При оберненому обході спочатку опрацьовується ліве піддерево, потім – праве і, нарешті, – корінь. Проілюструємо описані стратегії обходу дерева.

Приклад 4.24. Стратегії обходу бінарного дерева:



Пряма стратегія: d, a, c, b, f, e, g.

Внутрішня стратегія: a, b, c, d, e, f, g.

Обернена стратегія: b, c, a, e, g, f, d ■

Бінарне дерево називається *деревом пошуку* (див. підрозд. 1.4.4), якщо для кожної його вершини вершини лівого піддерева менші за неї, а правого – більші.

Продемонструємо роботу з бінарним словарним деревом пошуку (лінійний порядок – лексикографічний).

Приклад 4.25. Бінарне дерево пошуку: прямий обхід.

Лістинг.

```
#include <stdio.h>
#include <string.h>

typedef struct item {/*вузол дерева*/
    char *data; /*покажчик на текст*/
    struct item *left; /*ліве піддерево*/
    struct item *right; /*праве піддерево*/
 } Item;
```

```
typedef Item *Itemptr;
Itemptr root; /*покажчик на глобальний корінь дерева*/

void search(Itemptr *tree, const char *s);
void preOrder(Itemptr node);

int main()
{
    int done=0;
    char s[12];

    puts("Tree demonstration");
    while(!done){
          printf("Data:");
          gets(s);
          done=(strlen(s)==0);
          if(!done)
                search(&root, s);
    }
    puts("\nPREORDER:\n");
    preOrder(root);

    return 0;
}

/* search: пошук елемента, що вже існує, чи створення нового*/
void search(Itemptr *tree, const char *s)
{
    Itemptr p;
    int i;

    if(*tree==NULL) {
          p=(Itemptr)malloc(sizeof(Item));
          p->left=NULL;
          p->right=NULL;
          p->data=strdup(s);
          *tree=p;
    } else {
          p=*tree;
          i=strcmp(s, p->data);
          if(i<0)
                search(&p->left, s);
          else if(i>0)
                search(&p->right, s);
          else {
                puts("Duplicate data!");
                printf("%s\t", p->data);
                puts("");
```

```
            }
        }
    }

    /*пряма стратегія*/
    void preOrder(Itemptr node)
    {
        if(node!=NULL){
              printf("%s\t", node->data); /*замість printf може бути
довільний процес, що потрібно виконати з елементом*/
              preOrder(node->left);
              preOrder(node->right);
        }
    }
```

■

Збалансовані дерева – це дерева, в яких для будь-якої вершини кількість елементів у лівому й правому піддеревах відрізняється не більше ніж на 1. Рекурсивна функція **CreateTS** будує збалансоване числове дерево пошуку з вершинами, що належать заданому інтервалу:

```
    typedef struct item2 {/*вузол числового дерева*/
        int data; /*число у вузлі*/
        struct item *left; /*ліве піддерево*/
        struct item *right; /*праве піддерево*/
     } Item2;
    typedef Item2 *Itemptr2;

    /*CreateTS: побудова збалансованого числового дерева пошуку з
вершинами, що належать інтервалу [n,m]*/

    Itemptr2 CreateTS(int n,int m)
    {Itemptr2 root=(Itemptr2)malloc(sizeof(*Itemptr2)); /*корінь
дерева*/
       int mid=(n+m)/2; /*середина інтервалу*/

        if (n<m) return NULL;
          else {if (!root) {printf("No mamory."); return NULL;}
                root->data=mid;
                root->left=CreateTS(n,mid-1);
                root->right=CreateTS(mid+1,m);
                }
    return root;
    }
```

■

4.3.5. КОНТЕЙНЕРНІ ТИПИ

Ми розглянули реалізацію динамічних структур даних засобами мови С. Для роботи з ними в мові С++ існують *контейнерні типи*, що входять до складу її стандартної бібліотеки.

Послідовний контейнер містить упорядкований набір елементів одного типу. Основними типами контейнерів є вектор (**vector**), список (**list**) і двостороння черга (**deque**). Тип **deque** забезпечує ту саму функціональність, що й **vector**, але особливо ефективно реалізує операції вставлення й видалення першого елемента. Усе зазначене нижче для контейнерного типу **vector** буде стосуватися й типу **deque**.

При використанні послідовних контейнерів варто дотримуватися таких рекомендацій: якщо потрібен довільний доступ до елементів, то краще застосовувати вектор; при завчасно відомій кількості елементів – тип **vector**; за необхідності вставляти чи видаляти елементи всередині контейнера – **list**; якщо не потрібно вставляти чи видаляти елементи на початку контейнера, то краще застосовувати **vector**, а не **deque**.

Асоціативний контейнер ефективно реалізує операції перевірки існування й вилучення елементів.

Два основних асоціативних контейнера – це відношення (**map**) і множина (**set**). Контейнер **map** складається з пар ключів-значень, де ключ використовується для пошуку елемента, а значення – для збереження інформації.

Елемент контейнера **set** містить тільки ключ. Цей контейнер ефективно реалізує операцію перевірки існування.

В асоціативних контейнерах (**map**, **set**) не може бути дублікатів. Для підтримки дублікатів існують контейнери **multimap** та **multiset**.

Для визначення об'єкта контейнерного типу потрібно ввести відповідний заголовок:

```
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>
```

Після цього можна визначити об'єкт контейнерного типу:

```
vector<string> svec; //вектор елементів типу string
list<int> ilist; //список елементів типу int
map<string, int> simap;
set<string> sset;
```

Крім того, можна задати розмір масиву як константою, так і виразом:

```
extern int get_size();

const int vect_size=25;
vector<string> svec1(vect_size);
list<int> ilist1(get_size());
```

Для визначення розміру контейнера існує метод **size()**.

Можна змінити попередньо заданий розмір контейнера в процесі роботи:

```
svec1.resize(3*svec1.size());
```

Кожен елемент контейнера ініціалізується значеннями за умовчанням (**int** – значення 0). Також можна вказати початкове значення всіх елементів:

```
vector<string> svec2(vect_size, "aaa");
list<int> ilist2(get_size(), 100);
```

При такому визначенні спочатку контейнери порожні. Для перевірки на порожність контейнера існує метод **empty**:

```
Ilist.empty() // true - порожній
```

Для вставлення елемента в кінець контейнера використовують метод **push_back()**:

```
string txt;
while (cin>>txt)
svec.push_back(txt);
```

Метод **push_front()** використовують для додавання елемента в початок списку:

```
int i=1;
ilist.push_front(i);
```

Для видалення останнього елемента контейнера існує метод **pop_back()**:

```
svec.pop_back();
ilist.pop_back();
```

Для перебору елементів контейнера (як послідовного, так і асоціативного) існують ітератори:

```
  vector<string>::iterator iter=svec1.begin(); /*повертає ітератор, що вказує на перший елемент*/
  vector<string>::iterator iter1;
  list<int>>::iterator iter2=ilist1.end(); /*повертає ітератор, що вказує на останній елемент*/
```

Переміщення ітератора:

```
  ++iter; /*переміщення ітератора на наступний елемент контейнера*/
  --iter; /*переміщення ітератора на попередній елемент контейнера*/
```

Взяття значення елемента контейнера за ітератором:

```
  *iter;

  for(iter1=svec2.begin(); iter1 !=svec2.end(); ++iter1)
      cout<<*iter1<<endl;
```

Для вставлення елемента в довільну позицію послідовного контейнера існує метод **insert()**:

```
  string str("abc"); /*створення об'єкта string зі значенням "abc"*/
  svec2.insert((svec2.begin()+svec2.end())/2, str); /*вставлення елемента str на позицію, що передує (svec2.begin()+svec2.end())/2, тобто всередину контейнера*/
```

Для усунення елемента в заданій позиції (чи елементів із діапазону) існує дві форми методу **erase()**:

```
  svec2.erase(iter1); /*усунення елемента, на який указує ітератор iter1*/
  svec2.erase(svec2.begin(), svec2.end()); /*усунення елементів від svec2.begin() до svec2.end() (у даному випадку усунення всіх елементів)*/

      svec=svec2; /*після присвоювання контейнер svec містить елементи контейнера svec2 і тільки їх, навіть якщо попередньо в цих контейнерів були різні розміри*/
      svec2.swap(svec); /*обмін значеннями контейнерів*/
  list<int>::iterator first;
  first=find(ilist.begin(), ilist.end(), 23); /*пошук значення 23 у діапазоні від ilist.begin() до ilist.end()*/
```

Контейнер **list** підтримує також операції **sort()** та **merge()**.

Стандартні узагальнені алгоритми. У стандартній бібліотеці шаблонів STL визначені узагальнені алгоритми для роботи з контейнерними типами (табл. 4.1). Для використання алгоритмів, що не змінюють елементів послідовності модифікуючих і сортувальних алгоритмів, у програму потрібно ввести заголовний файл **<algorithm>**, а для обчислювальних алгоритмів – заголовний файл **<numeric>** (табл. 4.2).

Усі алгоритми реалізовані як шаблони функцій (див. підрозд. 3.12) і обробляють вхідну послідовність елементів контейнера, задану парою ітераторів **[first, last)** (останнім елементом, що обробляється, є **last-1**). Багато алгоритмів реалізовано в кількох варіантах, для різних наборів параметрів, тому тут наведені лише їхні назви. Деякі з перелічених тут алгоритмів уже згадувалися вище.

Таблиця 4.1. **Алгоритми, визначені в <algorithm>**

Функція	Призначення
adjacent_find	Пошук першої пари збіжних елементів
binary_search	Перевірка входження заданого елемента в упорядкований інтервал
Copy	Копіювання вхідного інтервалу в новий контейнер
copy_backward	Копіювання вхідного інтервалу в новий контейнер у оберненому порядку
count	Підрахунок кількості елементів у інтервалі
count_if	Підрахунок кількості елементів у інтервалі, що відповідає заданому предикату
equal	Бінарний пошук у інтервалі інтервалу елементів, що дорівнюють заданому елементу
equal_range	Бінарний пошук у впорядкованому інтервалі інтервалу елементів, що дорівнюють заданому елементу
fill	Заповнює інтервал заданим значенням
fill_n	Заповнює контейнер, починаючи з *n*-го елемента, заданим значенням
find	Заповнює перші *n* елементів заданим значенням
find_end	Пошук у одному інтервалі останнього входження іншого
find_first_of	Пошук у першому інтервалі одного зі значень другого інтервалу
find_if	Пошук у інтервалі елемента, що задовольняє предикат
for_each	Застосовує задану функцію (функтор) до всіх елементів інтервалу

Продовження табл. 4.1.

Функція	Призначення
`generate`	Заповнює інтервал значеннями, що обчислені заданою функцією (функтором)
`generate_n`	Заповнює перші *n* елементів контейнера значеннями, що обчислені заданою функцією (функтором)
`includes`	Перевіряє, чи входить один упорядкований інтервал у інший
`inplace_merge`	Злиття на місці
`iter_swap`	Обмінює значення елементів, що адресовані парою ітераторів, але не модифікує самі ітератори
`lexicographical_compare`	Лексикографічно попарно порівнює два інтервали
`lower_bound`	Бінарний пошук у впорядкованому інтервалі першого елемента, більшого чи рівного заданому
`make_heap`	Конвертує елементи з діапазону в купу, в якій перший елемент є найбільшим і для якої критерій сортування може бути визначений бінарним предикатом
`max`	Порівняти два елементи. Порівнює два об'єкти й повертає більший із двох, де критерій може бути визначений бінарним предикатом
`max_element`	Знаходить перше входження найбільшого елемента в зазначеному діапазоні, де критерій пошуку може бути визначений бінарним предикатом
`merge`	Запис об'єднаної послідовності в нову
`min`	Порівняти два елементи. Порівнює два об'єкти й повертає менший із двох, де критерій може бути визначений бінарним предикатом
`min_element`	Знаходить перше входження найменшого елемента в зазначеному діапазоні, де критерій пошуку може бути визначений бінарним предикатом
`mismatch`	Пошук перших незбіжних елементів у двох послідовностях
`next_permutation`	Отримати наступну перестановку
`nth_element`	Елементи, менші за *n*-й, переміщуються ліворуч від нього, а більші – праворуч
`partial_sort`	Сортування початкової частини інтервалу
`partial_sort_copy`	Сортування початкової частини інтервалу з копіюванням результату в новий контейнер

Продовження табл. 4.1.

Функція	Призначення
`partition`	Переставлення елементів, що задовольняють заданий предикат, на початок інтервалу
`pop_heap`	Видаляє найбільший елемент від початку купи до останнього положення в діапазоні й потім формує нову купу з елементів, що залишилися
`prev_permutation`	Отримати попередню перестановку
`push_heap`	Додає елемент із кінця діапазону до існуючої купи, що складається з елементів, які передують діапазону
`random_shuffle`	Випадкова перестановка елементів інтервалу
`remove`	Видалити з інтервалу всі елементи, що дорівнюють заданому
`remove_copy`	Видалити з інтервалу всі елементи, що дорівнюють заданому, результат скопіювати в новий контейнер
`remove_copy_if`	Видалити з інтервалу всі елементи, що задовольняють заданий предикат, результат скопіювати в новий контейнер
`remove_if`	Видалити з інтервалу всі елементи, що задовольняють заданий предикат
`replace`	Замінити всі елементи інтервалу, що дорівнюють заданому, на нові
`replace_copy`	Замінити всі елементи інтервалу, що дорівнюють заданому, на нові, результат скопіювати в новий контейнер
`replace_copy_if`	Замінити всі елементи інтервалу, що задовольняють заданий предикат, на нові, результат скопіювати в новий контейнер
`replace_if`	Замінити всі елементи інтервалу, що задовольняють заданий предикат, на нові
`reverse`	Переставити елементи інтервалу в оберненому порядку
`reverse_copy`	Переставити елементи інтервалу в оберненому порядку, результат скопіювати в новий контейнер
`rotate`	Циклічна перестановка елементів інтервалу
`rotate_copy`	Циклічна перестановка елементів інтервалу, результат скопіювати в новий контейнер
`search`	Пошук в одному інтервалі першого входження іншого
`search_n`	Пошук інтервалу з n елементів, що дорівнюють заданому
`set_difference`	Різниця множин-інтервалів

Закінчення табл. 4.1.

Функція	Призначення
`set_intersection`	Перетин множин-інтервалів
`set_symmetric_difference`	Симетрична різниця множин-інтервалів
`set_union`	Об'єднання множин-інтервалів
`sort`	Сортування інтервалу
`sort_heap`	Конвертує купу до діапазону, що сортується
`stable_partition`	Перестановка елементів, що задовольняють заданий предикат, на початок інтервалу зі збереженням відносного порядку елементів
`stable_sort`	Перестановка елементів, що задовольняють заданий предикат, на початок інтервалу
`swap`	Обмінює значення об'єктів
`swap_ranges`	Обмінює значення із двох інтервалів
`transform`	Застосовує задану функцію (функтор) до заданого інтервалу (інтервалів) з отриманням нової послідовності елементів
`unique`	Видаляє з інтервалу однакові сусідні елементи, залишаючи єдиний
`unique_copy`	Видаляє з інтервалу однакові сусідні елементи, залишаючи єдиний; результат скопіювати в новий контейнер
`upper_bound`	Бінарний пошук у впорядкованому інтервалі першого елемента, більшого за заданий

Таблиця 4.2. **Алгоритми, визначені в заголовному файлі `<numeric>`**

Функція	Призначення
`accumulate`	Обчислює суму значень елементів послідовності з діапазону **`[first, last)`** з початковим значенням. Наприклад, якщо задана послідовність {1,2,2,4,5} і початкове значення 0, то результатом роботи алгоритму буде 14. Операція додавання може бути замінена іншою бінарною операцією, що передається як параметр
`adjacent_difference`	Обчислює часткову різницю. Створює нову послідовність, в якій значення кожного елемента (крім першого, що рівний першому елементу вихідного інтервалу) дорівнює різниці поточного елемента та його попередника у вихідній послідовності із заданого діапазону. Наприклад, якщо маємо {1, 3, 4, 5, 5, 7}, то результатом буде {1, 2, 1, 1, 0, 2}. Замість різниці у вихідну послідовність може бути записаний результат деякої іншої узагальненої процедури, яка є бінарною операцією

Закінчення табл. 4.2.

Функція	Призначення
`inner_product`	Знаходить скалярний добуток двох послідовностей з інтервалів. Наприклад, якщо послідовності {1, 2, 4} та {1, 1, 2}, то результатом буде 1*1+2*1+4*2=11. Замість операцій додавання та множення як параметри можна передати інші бінарні операції
`partial_sum`	Обчислює часткову суму. Створює нову послідовність із послідовності, обмеженої діапазоном `[first, last]`: у новій послідовності значення кожного елемента дорівнює сумі всіх попередніх, включаючи й поточний. Наприклад, з послідовності {0, 1, 2, 2} буде створена послідовність {0, 1, 3, 5}. Замість операції суми можна використовувати іншу бінарну операцію

***Література для СР:**. динамічні структури даних і алгоритми на них – [8, 60, 65, 77, 95, 124, 125].

Контрольні запитання та вправи

1. Що таке динамічні структури даних?
2. Що таке зв'язаний список?
3. Як реалізується однозв'язний список?
4. Які основні операції над однозв'язними списками?
5. Як реалізується двозв'язний список?
6. Як реалізуються стеки й операції на них?
7. Як реалізуються черги й операції на них?
8. Яка відмінність між стеком і чергою?
9. Яка відмінність зв'язаного списку від стеку й черги?
10. Як реалізується бінарне дерево?
11. Скільки всього є варіантів обходу бінарного дерева?
12. Що таке контейнер та ітератор?
13. Що таке послідовний контейнерний тип векторів?
14. Що таке послідовний контейнерний тип списків?
15. Що таке послідовний контейнерний тип двосторонніх черг?
16. Що таке асоційовані контейнерні типи відношень і множин?
17. Що таке стандартні узагальнені алгоритми?
18. Написати клас, що реалізує булеву алгебру множин цілих чисел з операціями та предикатами: введення-виведення, об'єднання, перетин, різниця, симетрична різниця, включення, рівність, приналежність елемента множині. Множини подати у вигляді лінійного списку елементів.

19. Написати функції, що реалізують алгебру багаточленів від однієї змінної із цілими коефіцієнтами. Операції алгебри: введення-виведення, додавання, множення на одночлен і багаточлен, ділення двох багаточленів (з остачею й часткою у вигляді багаточленів). Багаточлени подати як лінійний список пар (коефіцієнт, степінь змінної).
20. Циклічним називається список, останній елемент якого містить не **NULL**, а показує на його голову. Реалізувати основні операції для роботи з такими списками.
21. Написати програму для розв'язання відомої задачі: n солдатів стали в коло. Із k-го солдата починається рахування. Через p солдатів той, на кого випало, виходить із кола, а рахування починається знову з наступного солдата до тих пір, поки не залишиться останній. Знайти його номер. Застосувати циклічний список.
22. Написати програму для перетворення арифметичного виразу на форму ОПЗ, застосовуючи стек у вигляді масиву.
23. Реалізувати обчислення арифметичного виразу у формі ОПЗ.
24. Написати програму **tail**, що друкує n останніх введених рядків. За умовчанням $n = 5$, але n може бути задано й у командному рядку. Командний рядок > **tail - n** викликає друкування останніх **n** рядків. У кожний момент зберігати в пам'яті не більше **n** рядків. Застосувати: а) чергу; б) циклічний список у вигляді масивів.
25. Написати функції для роботи із чергами на основі лінійного списку, які здійснюють:
    а) заміну k-го вузла в черзі;
    б) вилучення k-го вузла із черги;
    в) конкатенацію кількох черг;
    г) розбиття черги на кілька черг;
    д) створення копії черги;
    е) визначення довжини черги;
    є) пошук вузла(ів) черги із заданими властивостями.
26. Написати програму на основі однозв'язного списку, що дозволяє додавати й видаляти дані списку, шукати інформацію, зберігати список у файл і зчитувати його з файла для таблиць:
    а) каталог книг у бібліотеці;
    б) список студентів у групі;
    в) список товарів у магазині ;
    г) список клієнтів банку;
    д) телефонний довідник.
27. Для таблиць із попереднього завдання написати програму на основі двозв'язного списку, що дозволяє додавати й видаля-

ти дані списку, шукати інформацію за ключем, зберігати список у файл і зчитувати його з файла.

28. Для таблиць із вправи 26 написати програму на основі бінарного дерева пошуку (порядок лексикографічний за ключовим полем), що дозволяє додавати й видаляти дані списку, шукати інформацію за ключем, зберігати список у файл і зчитувати його з файла.
29. Для лінійного однозв'язного списку цілих чисел від кожного додатного числа відняти найменше з додатних. Процедуру продовжувати, поки всі елементи списку не стануть від'ємними. Зазначити кількість ітерацій у процедурі.
30. Написати функцію, що читає з клавіатури дужковий вираз і перевіряє його на приналежність мові Діка: а) для дужок (,); б) для дужок (,),[,],{,}. Використати стек у динамічній пам'яті.
31. Написати функції для роботи із загальними списками з атомами-ідентифікаторами: 1) створення й виведення списку; 2) пошуку всіх атомів списку; 3) пошуку частоти входжень усіх атомів у список; 4) усунення й заміни одного атома на інший.
32. Для завдань 18-30 використати відповідні контейнерні типи. Проаналізувати, який із варіантів програми простіший для реалізації й розуміння.

4.4. Динамічні таблиці

- Таблиці як масиви структур
- Таблиці у вигляді зв'язаних списків
- Підвищення ефективності при пошуку в таблицях
- Пряма адресація
- Хеш-таблиці з колізіями
- Відкрите хешування
- Закрите хешування
- Таблиці у вигляді дерев пошуку

Ключові слова: *таблиця як масив, рядок таблиці, операції з таблицями, додавання й видалення рядка, пошук рядка, таблиця як зв'язаний список рядків, хеш-таблиця, хеш-функція, таблиця з прямою адресацією, колізія, відкрите й закрите хешування, лінійне хешування.*

У підрозд. 4.1 розглядалася задача пошуку елементів у таблицях фіксованої довжини. Ми вже знаємо, що таблиця – це кортеж векторів певної фіксованої структури. Самі вектори називаються рядками таблиці й подаються зазвичай структурами. Наприклад, якщо таблиця має вигляд

Прізвище, ім'я, по-батькові	Курс	Середній бал
Іваненко Іван Іванович	2	4.25
Петренко Петро Петрович	1	3.5
Сидоров Сидір Сидорович	1	5

то її рядок можна описати таким чином:

```
typedef struct student {
    char *key;
    int rate;
float estimationsult;
}Student;
```

Основні операції з таблицями: додавання й видалення рядка, модифікація рядка, пошук рядка за ключем.

Реалізація операцій над таблицями залежить від способу подання таблиць. Розглянемо ці способи.

4.4.1. ТАБЛИЦІ ЯК МАСИВИ СТРУКТУР

Найпростішим є подання таблиць у вигляді динамічних масивів їхніх рядків:

```
Student tableStArray[n];
```

Приклад 4.26. Таблиця як масив структур:

Лістинг.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>

#int countLine=50;/*задає максимальну кількість рядків у таблиці*/

typedef struct student {
    char name[10];
    int rate;
    float estimationsult;
}Student;
```

```
   typedef Student *StudentPtr;

   Student tableStArray[countLine];

   Student emptyCellArray;

   /*ініціалізація інформації про студента*/
   Student setEmtySt(Student st)
   {
       st.estimationsult=0;
       strcpy(st.name,"         ");
       st.rate=0;
       return st;
   }

   /*друкування таблиці*/
   void printArrayTable(void)
   {
       for(int i=0; i<countLine; i++)
             printf("\n%s\t%f\t%d", tableStArray[i].name,
tableStArray[i].estimationsult, tableStArray[i].rate);
   }

   /*ініціалізація таблиці*/
   void ArrayTableInit(void)
   {
       emptyCellArray=setEmtySt(emptyCellArray);
       for(int i=0; i<countLine; i++)
             tableStArray[i]=emptyCellArray;
   }

   /*пошук рядка в таблиці за ключовим полем name; результат – ін-
декс рядка, якщо його знайдено, або countLine, якщо відповідного
рядка немає*/
   int ArrayTableFind(char key[10])
   {
       int i=0;
       while((i<countLine)&&(strcmp(tableStArray[i].name, key))!=0)
             i++;
       return i;
   }

   /*перевірка на рівність двох рядків*/
   int StructEqual(Student st1, Student st2)
   {
       if((st1.rate==st2.rate)&&(st1.estimationsult==st2.estimation
sult)&&(strcmp(st1.name, st2.name)))
             return 1;
```

```
      else return 0;
   }

   /*додавання рядка на першу вільну позицію в таблиці*/
   void ArrayTableInsert(Student st)
   {
      int pos=ArrayTableFind(emptyCellArray.name);
      if(pos==countLine)
            printf("\nТаблиця  переповнена.  Запис  не  відбув-
ся!!!\n");
      else tableStArray[pos]=st;
   }

   /*видалення рядка за ключем*/
   void ArrayTableDelete(char key[10])
   {
      int pos=ArrayTableFind(key);
      if(pos==countLine)
            printf("Елемент для видалення не знайдено!");
      else tableStArray[pos]=emptyCellArray;
   }
   ■
```

4.4.2. ТАБЛИЦІ У ВИГЛЯДІ ЗВ'ЯЗАНИХ СПИСКІВ

Гнучкішим із погляду модифікації таблиць є їхнє подання у вигляді зв'язаних списків (див. підрозд. 4.3.1). Розглянемо реалізацію основних табличних операцій на основі однозв'язного списку.

Приклад 4.27. Таблиця як зв'язаний лінійний список:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <conio.h>

typedef struct item {
  char name[10];
  int  rate;
  float estimationsult;
  struct item *next; /*покажчик на наступний елемент списку*/
} Item;

typedef Item *Itemptr;

/*друкування значень елементів списку, start – голова списку*/
void display(Itemptr start)
```

```
{
  while(start) {
    printf("\n%s\t%f\t%d", start->name, start->estimationsult,
start->rate);
    start=start->next;
  }
printf("\n");
}

/* пошук попередника елемента elem, start – голова списку*/
Itemptr getPrev(Itemptr start, Itemptr elem)
{
  if(elem)
  {
        Itemptr prev=NULL;
        Itemptr p=start;
        while((p)&&(p!=elem)) {
              prev=p;
              p=p->next;
        }
        return prev;
  } else
  return NULL;   /*елемент не знайдено*/
}

/* пошук попередника елемента за значенням name.
Результат NULL, якщо елемент не знайдено, або він перший у списку
(не має попередників), start – голова списку*/
Itemptr getPrevElem(Itemptr start, char name[10])
{
 Itemptr prev=NULL;
  while(start) {
        if((strcmp(start->name, name))==0) return prev;
    prev=start;
      start=start->next;
  }
  return NULL; /*елемент не знайдено*/
}

/* пошук елемента за його значенням name, start – голова списку*/
Itemptr search(Itemptr start, char name[10])
{
  while(start) {
    if((strcmp(start->name, name))!=0) return start;
    start=start->next;
  }
  return NULL;  /*елемент не знайдено*/
```

```
}

/* пошук останнього елемента), start – голова списку*/
Itemptr getLast(Itemptr start)
{
      while(start->next)
      start=start->next;
      return start;
}

/*додавання нового елемента itm у кінець списку,start – голова
списку, результат – нова голова списку*/
Itemptr addItem (Itemptr start, Item itm)
{
      Itemptr p, p1;

      p=(Itemptr)malloc(sizeof(Item));
      p->estimationsult=itm.estimationsult;
      p->rate=itm.rate;
      strcpy(p->name, itm.name);
      p->next=NULL;
      if(!start) start=p;
      else{
            p1=getLast(start);
            p1->next=p;
      }
      return start;
}

/*наповнення списку, start – голова списку, результат – нова голо-
ва списку*/
Itemptr inputElem(Itemptr start)
{
      Item itm;
      int num=1;
      Itemptr itemList;;
      while(num)
      {
            Item st;

            printf("\nInput name: ");
            scanf("%s", &st.name);
            printf("\nInput estimationsult: ");
            scanf("%f", &st.estimationsult);
            printf("\nInput rate: ");
            scanf("%d", &st.rate);
            itemList=addItem(start, st);
            if(!start){ start=itemList;}
```

```
            printf("Insert else? 1/0");
            scanf("%d", &num);
      }
      return start;
}

/*видалення елемента за елементом на позиції  prevItem, start -
голова списку, результат - нова голова списку*/
Itemptr deleteItem(Itemptr start, Itemptr prevItem)
{     /*список не порожній*/
      if(start) {
            Itemptr p;
      /*видаляється перший елемент (якому ніщо не передує)*/

            if(!prevItem)   {
                  p = start;
                  start=start->next;
                  free(p);
                  return start;
            }
            else{
                  p=prevItem->next;
                  /*попередник не останній*/
                  if(p){
                        prevItem->next=p->next;
                        free(p);
                        return start;
                  } else return start;
            }
      }
      else return start;
}

/*видалення елемента зі введеним ключем, start - голова списку,
результат - нова голова списку*/
Itemptr deleteElem(Itemptr start)
{
      int num=1;
      Itemptr p;
      printf("\nDeleting:");
      char name[10];
      while(num)
      {
            printf("\nInput name: ");
            scanf("%s", &name);
            p=getPrevElem(start, name);
            start=deleteItem(start, p);
```

```
            display(start);
            printf("Delete else? 1/0");
            scanf("%d", &num);
        }
        return start;
}
```

■

4.4.3. ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ПРИ ПОШУКУ В ТАБЛИЦЯХ

Для підвищення ефективності обробки таблиць їх подають у вигляді так званих хеш-таблиць. Середній час пошуку елемента в такій таблиці становить $O(1)$, а в найгіршому випадку – $O(n)$.

Якщо взяти звичайну таблицю у вигляді масиву чи списку рядків, то останні розташовані в них або випадково, або в певному порядку. У будь-якому разі, щоб знайти рядок за даним ключем, доводиться переглянути кілька або всі рядки таблиці. Ідея хешування таблиць полягає в тому, щоб за допомогою спеціальної хеш-функції пов'язати розташування рядка в таблиці зі значенням його ключа. Тоді для отримання доступу до рядка достатньо вирахувати значення хеш-функції на його ключі. Кажуть, що хеш-функція *розставляє*[1] рядки в таблиці. Нехай K – сукупність усіх можливих ключів, m – фіксоване натуральне число. Хеш-функція h має вигляд $h : K \to 0..m-1$, а її значення називаються *хеш-числами*. *Хеш-таблицею* називається масив (*хеш-масив*) довжиною m, i-й елемент якого містить або рядок із ключем k таким, що $h(k) = i$, або покажчик на такий рядок чи зв'язаний список таких рядків. Якщо в таблиці немає двох рядків з однаковим хеш-числом, то пошук рядка в ній здійснюється за час $O(1)$. У протилежному випадку цей час залежить від довжини списків, що містять рядки з однаковими хеш-числами.

Для роботи з таблицею необхідні: хеш-функція, функція пошуку рядка, функція для занесення в таблицю нового рядка або модифікації вже існуючого й видалення рядка.

Хешування корисне, коли широкий діапазон можливих значень має зберігатися в невеликому обсязі пам'яті й потрібен швидкий доступ до даних. Хеш-таблиці часто застосовуються в базах даних, особливо в мовних процесорах типу компіляторів і асемблерів для організації таблиць ідентифікаторів.

[1] Іноді хеш-функції називають функціями розташування.

4.4.4. ПРЯМА АДРЕСАЦІЯ

Пряма адресація є найпростішим способом організації хеш-таблиць. Вона застосовується, якщо кількість n можливих ключів порівняно невелика й у таблиці немає двох рядків з однаковим хеш-числом. Хеш-таблиця у цьому випадку є масивом $T[n]$ довжиною n із компонентами – рядками таблиці або покажчиками на них – і називається *таблицею з прямою адресацією* (рис. 4.3). У таблиці з прямою адресацією кожному ключу відповідає елемент $T[i]$, де i – хеш-число ключа, пов'язане з одним і тільки одним рядком. Щоб отримати доступ до рядка такої таблиці, достатньо знайти хеш-число його ключа.

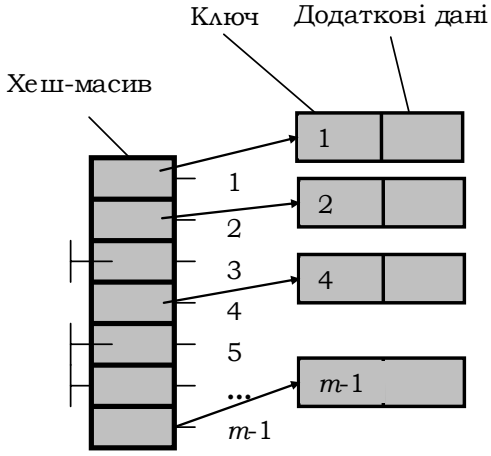


Рис. 4.3

Реалізуємо основні операції для таблиць із прямою адресацією. Кожна з них вимагає $O(1)$ часу.

Приклад 4.28. Хеш-таблиця з прямою адресацією й елементами-покажчиками на рядки таблиці:

Лістинг.

```
#define countKey 2100
/*додаткові дані*/
typedef struct information{
    int iYear;
    char *text;
}Info;
```

```
typedef Info *InfoPtr;

/*Опис структури (елемента)*/
typedef struct st{
    int key;
    InfoPtr info;
} St;

typedef St *StPtr;

StPtr AddressT[countKey];
/*пошук інформації за ключем*/
InfoPtr addressSearch(int k)
{
    InfoPtr IPtr=NULL;
    if((k>0)&&(k<countKey)) IPtr=AddressT[k]->info;
    return IPtr;
}

/*додавання інформації*/
void addressInsert(InfoPtr info)
{
    int k=addressFunction(info);
    StPtr PTR=(StPtr)malloc(sizeof(St));
    PTR->info=info;
    PTR->key=k;
    AddressT[k]=PTR;
}

/*пошук інформації за ключем*/
void addressDelete(InfoPtr info)
{
    int k=addressFunction(info);
    StPtr PTR=AddressT[k];
    free(info);
    free(PTR);
    AddressT[k]=NULL;
}

/*Розглянемо тривіальний варіант хеш-функції. Тут ключ рівний
полю iYear. Результат (-1) у випадку помилки*/
int addressFunction(InfoPtr info)
{
    if(info)
          return info->iYear;
    else return -1;
}
```

■

4.4.5. ХЕШ-ТАБЛИЦІ З КОЛІЗІЯМИ

Якщо множина K усіх ключів достатньо велика, то не завжди доцільно чи взагалі можливо зберігати хеш-масив $T[n]$, де $n = |K|$. До того ж якщо в зазначеному масиві буде багато порожніх елементів (тих, з якими не асоціюються рядки), то багато пам'яті витрачатиметься даремно.

Якщо кількість потенційно можливих рядків у таблиці суттєво менша, ніж кількість усіх ключів, то хеш-таблиця займає менше місця порівняно з прямою адресацією, при цьому на практиці пошук у хеш-таблиці в середньостатистичному випадку залишається $O(1)$ (рис. 4.4).

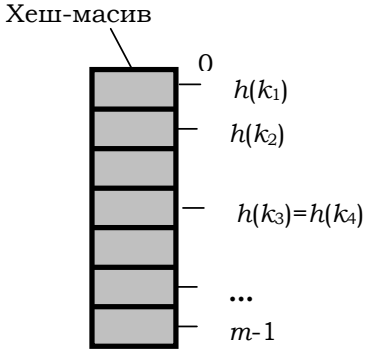


Рис. 4.4

Однак виникає *колізія* при збігу хеш-чисел для різних ключів: два або більше рядків претендують на одне місце в хеш-таблиці. Можна спробувати задати хеш-функцію так, щоб колізій не було, але при $|K| > m$ це неможливо. Розглянемо два способи виходу із цієї ситуації:

1) відкрите, або зовнішнє, хешування (дозволяє зберігати множини в потенційно нескінченному просторі);

2) закрите, або внутрішнє, хешування (використовує обмежений простір для збереження таблиць).

4.4.6. ВІДКРИТЕ ХЕШУВАННЯ

При відкритому хешуванні рядки таблиці з однаковими хеш-числами об'єднуються у зв'язані списки і хеш-масив містить покажчики на їхні голови (рис. 4.5). Щоб отримати доступ до певного рядка таблиці, потрібно знайти його хеш-число й зайти у відповідний список.

Бажано вибрати хеш-функцію h так, щоб вона приблизно рівномірно розподіляла рядки таблиці по списках. Зазвичай хеш-функція

обчислює певне натуральне число k, яке факторизується за модулем m (тобто береться залишок від ділення k на m).

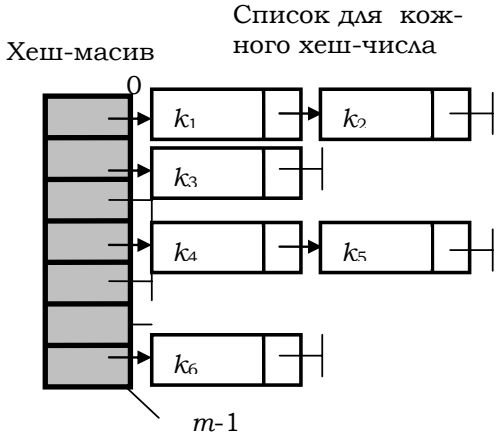


Рис. 4.5

Приклад 4.29. Хеш-функція на рядках.

Розглянемо хеш-функцію, визначену на ключах – символьних рядках, і реалізацію простого словника на основі відкритого хешування:

Лістинг.

```
#define countKey 701 /*довжина хеш-таблиці*/

typedef struct celltype {
    char *element;
    struct celltype *next;
}Celltype;

typedef Celltype *CelltypePtr;

CelltypePtr Dictionary[countKey];

int hashFunction(char *x)
{
    int h=0, sum=0;
    int len=strlen(x);
    for(int i=0; i<len; i++){
          sum+=x[i];
          h=sum%countKey;
    }
    return h;
}
```

```
void makenull(void) /*початкова ініціалізація*/
{
    for(int i=0; i<countKey; i++){
          Dictionary[i]=NULL;
    };
}

int hashMember(char *x) /*пошук слова x у словнику*/
{
    CelltypePtr current=Dictionary[hashFunction(x)];
    /*початкове значення current дорівнює заголовку сегмента,
    якому належить слово x*/
    while(current)
          if(current->element==x) return 1;
          else current=current->next;
    return 0; /*слово не знайдено*/
}

void hashInsert(char *x) /*занесення слова у словник*/
{
    int bucket; /*для номера сегмента*/
    CelltypePtr oldheader;

    if(!hashMember(x)){
          bucket=hashFunction(x);
          oldheader=Dictionary[bucket];

    Dictionary[bucket]=(CelltypePtr)malloc(sizeof(Celltype));
          Dictionary[bucket]->element=x;
          Dictionary[bucket]->next=oldheader;
    }
}

void hashDelete(char *x) /*видалення слова зі словника*/
{
    int bucket=hashFunction(x); /*для хеш-числа слова*/
    CelltypePtr current; /*покажчик на попередню комірку*/

    if (Dictionary[bucket]){
          if(Dictionary[bucket]->element==x) {
/*x у першій комірці*/
                CelltypePtr delEl=Dictionary[bucket];
                Dictionary[bucket]=Dictionary[bucket]->next;
/*видалення зі списку*/
    free(x);
                free(delEl);
          }
```

```
          else {
/*x не в першій комірці*/
                current=Dictionary[bucket];
/*покажчик на попередню комірку*/
                while(current->next){
                      if(current->next->element==x){
                            CelltypePtr delEl=current->next;
                            current->next=current->next->next;
/*видалення зі списку*/
free(x);
                            free(delEl);
                            return;
                      }
                      else /*x поки не знайдено*/
                            current=current->next;
                }
          }
    }
}
```

■

4.4.7. ЗАКРИТЕ ХЕШУВАННЯ

При закритому хешуванні немає списків із рядками – усі рядки зберігаються безпосередньо в хеш-масиві. Кожний елемент масиву порожній або містить лише один рядок. Якщо розташувати рядок із ключем x у елементі з індексом $h(x)$ неможливо (він уже зайнятий іншим рядком (*колізія*)), то за методикою повторного хешування будується послідовність інших індексів $h^{(1)}(x)$, $h^{(2)}(x)$, ..., що задають позиції для його можливого розміщення. Кожна із цих позицій послідовно перевіряється, і якщо вільних елементів немає, то таблиця заповнена.

При такому хешуванні покажчики взагалі не використовуються. За рахунок економії пам'яті можна збільшити кількість позицій у таблиці, що зменшує кількість колізій і пришвидшує пошук (рис. 4.6).

Лінійне хешування. Для хеш-функції $h: U \to \{0,1,\ldots m-1\}$ функція, що задає *лінійну послідовність* спроб, зображується формулою $h^{(i)}(k) = (h(k)+i)\%m$. Роботу з ключем k починають із комірки $T(h(k))$, а потім переходять до $T(h(k)+1)$, $T(h(k)+2)$, ...(після $T(m-1)$ – до $T(0)$).

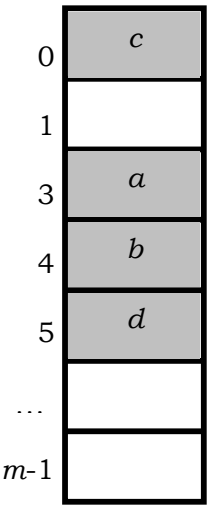


Рис. 4.6

Нехай $m = 8$ і ключі елементів a, b мають хеш-значення $h(a) = 3$, $h(b) = 3$, відповідно. Тоді, якщо ми хочемо вставити елемент b, а сегмент 3 уже зайнятий, то можна перевіряти на зайнятість сегменти 4, 5, 6, 7, 0, 1, 2 (саме в такому порядку).

Перевагою методу є простота реалізації, а недоліком те, що можуть утворитися *кластери* (довгі послідовності зайнятих комірок, які йдуть поспіль). Це подовжує пошук.

Одним із найкращих методів є *подвійне хешування*. Тут функція h має вигляд $h^{(i)}(k) = \left(h_1(k) + ih_2(k)\right)\%m$, h_1 та h_2 – звичайні хеш-функції.

Щоб послідовність випробуваних місць покривала всю таблицю, значення h'' має бути взаємно простим із m. Для цього потрібно вибрати за m степінь двійки, а функцію h'' узяти таку, щоб набувала непарних значень.

Приклад 4.30. Реалізація простого словника на основі закритого хешування з використанням лінійного методу:

Лістинг.

```
#define countKey 701

char *Dictionary[countKey];

int hashFunction(char *x)
{
```

```
      int h=0, sum=0;
      int len=strlen(x);
      for(int i=0; i<len; i++){
            sum+=x[i];
            h=sum%countKey;
      }
      return h;
  }

  void makenull(void)
  {
      for(int i=0; i<countKey; i++){
            strcpy(Dictionary[i], "\0");
      };
  }

  /*переглядає словник, починаючи від елемента hashFunction(x),
  поки не зустріне слово x, порожній
  елемент або не досягне кінця таблиці (в останніх випадках
  вважається, що таблиця не містить слово x).
  Результат - позиція, де зупинився пошук*/
  int locate(char *x)
  {
      int i=0, initial=hashFunction(x);

      while((i<countKey)&&(Dictionary[(initial+i)%countKey]!=x)&&
            (Dictionary[(initial+i)%countKey]))
            i++;
      return (initial+i)%countKey;
  }

  /*те саме, що й locate, але при досягненні значення
  deleted зупиняється*/
  int locate1(char *x)
  {
      int i=0, initial=hashFunction(x);

      while((i<countKey)&&(!strcmp(Dictionary[(initial+i)%countKey
], x))&&
            (!strcmp(Dictionary[(initial+i)%countKey],"\0"))&&
  (!strcmp(Dictionary[(initial+i)%countKey], "*********\0")))
            i++;
      return (initial+i)%countKey;
  }

  int hashMember(char *x)
  {
```

```
      return (Dictionary[locate(x)]==x);
   }

   void hashInsert(char *x)
   {
      if(Dictionary[locate(x)]==x) return; /*слово x уже є у слов-
нику*/

      int bucket=locate1(x);
      if((strcmp(Dictionary[bucket], "\0"))||
   (strcmp(Dictionary[bucket], "*********\n")))
            strcpy(Dictionary[bucket],x);
      else
            printf("ERROR");
   }
   void hashDelete(char *x)
   {
      int bucket=locate(x);
      if(strcmp(Dictionary[bucket], x)) strcpy(Dictionary[bucket],
"*********\n");
   }
   ■
```

4.4.8. ТАБЛИЦІ У ВИГЛЯДІ ДЕРЕВ ПОШУКУ

У підрозд. 3.7.2 ми розглянули програму для підрахунку частоти входжень службових слів у текст С-програми, що вводиться з клавіатури (прикл. 3.63). Для розв'язання задачі було побудовано таблицю з двома стовпчиками: у першому – усі службові слова мови С у лексикографічному порядку, у другому – частота їхніх входжень у С-програму. Така таблиця має фіксовану структуру, тому для її подання було обрано звичайний упорядкований масив структур. Тепер узагальнимо задачу і спробуємо порахувати частоту входжень усіх ідентифікаторів у С- програму. Така постановка потребує вже побудови динамічної таблиці з тією самою структурою рядків. Для цього скористаємося деревом пошуку, яке дозволяє обмежити час пошуку елемента висотою дерева (див. підрозд. 1.4.4). Порядок на ідентифікаторах, як і раніше, лексикографічний. У програму включено заголовний файл `"def.h"`, що містить описи функцій: `int getword (char *, int)` для читання з клавіатури чергового ідентифікатора (див. прикл. 3.63), `void treeprint (Itemptr3)` для прямого виведення дерева,

Itemptr3 talloc(void) для відведення місця під вузол дерева, **char *strdub(char *)** для створення копії рядка.

Лістинг.

```
typedef struct item { /*вузол дерева*/
 char *key; /*покажчик на слово*/
 int count; /*кількість входжень*/
 struct item *left; /*ліве піддерево*/
 struct item *right; /*праве піддерево*/
 }*Itemptr3;

#include "def.h"
#include <stdio.h>
#include <ctypeh.>
#include <string.h>
#define MAXWORD 100

Itemptr3 search2 (Itemptr3, char*);

/*частотний словник тексту програми*/
main()
{
 Itemptr3 root;
 char word [MAXWORD];
 root=NULL;
 while (getword (word, MAXWORD)!=EOF)
 if (isalpha (word [0])) root= search2 (root, word);
 treeprint (root);
 return 0;
}

/*search2: шукає й обробляє вузол зі словом word у піддереві p або
нижче нього. Якщо такий вузол відсутній - то вставляє його в дере-
во. На відміну від аналогічної функції search із прикл. 4.25,
search2 повертає покажчик на знайдений або побудований вузол*/

Itemptr3 search2(Itemptr3 p, char *word)
{
int cond;
if(p==NULL) { /*ідентифікатор зустрічається вперше*/
 if (p=talloc())==NULL){puts("No mamory"); exit(0);} /*створюється
новий вузол*/
if((p->key=strdub(word))==NULL) { puts("No mamory"); exit(0);};
p->count=1;
```

```
p->left=p->right=NULL;
 }
 else if ((cond=strcmp(word, p->key))==0)
 p->count++; /*ідентифікатор уже є в дереві*/
 else if (cond<0) /*менше кореня лівого піддерева*/
 p->left=addtree(p->left, word);
 else /*більше кореня правого піддерева*/
 p->right=addtree(p->right, word);
return p;
}
```

■

Ми розглянули таблиці в мові С. Для їхнього подання засобами С++ зручно користуватися контейнерними типами, що входять до складу стандартної бібліотеки С++ (див. вправу 32, підрозд. 4.3).

***Література для СР:**. подання таблиць і методи роботи з ними – [8, 55, 65].

Контрольні запитання та вправи

1. Що таке таблиця?
2. Що таке рядок таблиці?
3. Що таке таблиця як: а) масив; б) динамічний масив?
4. Які основні операції над таблицями?
5. Що таке таблиця як зв'язаний список рядків?
6. У чому полягає ідея хешування таблиць?
7. Що таке хеш-функція?
8. Що таке хеш-таблиця з прямою адресацією?
9. Що таке колізія рядків у хеш-таблиці?
10. Що таке відкрите й закрите хешування?
11. У чому полягає різниця між відкритим і закритим хешуванням?
12. Написати програму для обробки таблиць у вигляді динамічних масивів:

    а) каталог книг у бібліотеці: містить інформацію про автора, назву книжки, рік видання, видавництво й кількість сторінок;

    б) список студентів у групі: містить інформацію про ПІБ студента, рік вступу, рік народження, адресу й телефон;

    в) список товарів у магазині: містить інформацію про назву товару, виробника, кількість одиниць товару в наявності;

    г) список клієнтів банку: містить інформацію про ПІБ клієнта, номер рахунку, кількість грошей на рахунку;

д) телефонний довідник: містить інформацію про ПІБ, номер телефону, адресу.

13. Те саме, що й у вправі 12, але таблиці подати однозв'язним списком.
14. Те саме, що й у вправі 12, але таблиці подати двозв'язним списком.
15. Те саме, що й у вправі 12, але таблиці подати у вигляді закритих хеш-таблиць.
16. Те саме, що й у вправі 12, але таблиці подати у вигляді відкритих хеш-таблиць.
17. Створити заголовний файл **"def.h"** для реалізації таблиці у вигляді дерева пошуку з підрозд. 4.4.8.
18. Побудувати частотний словник у вигляді дерева пошуку для тексту, який задається: а) у файлі; б) символьним рядком.

4.5. Синтаксичний аналіз і обчислення виразів

- Лексичний аналіз виразів
- Синтаксичний аналіз
- Обчислення виразів

Ключові слова: *синтаксичний аналізатор* (*парсер*), *синтаксичний аналіз, лексичний аналіз, синтаксичне дерево, таблиця ідентифікаторів, метод рекурсивного спуску з поверненням, утиліти* Lex *та* Yacc.

Як транслятори, розроблені для компіляції програм, написаних мовами програмування високого рівня, перетворюють їхні тексти на команди, виконувані комп'ютером? Як компілятор відшукує помилки в тексті програми? Спробуємо розв'язати ці й подібні питання, відповівши спочатку на простіше: як написати програму, що одержує на вході символьний рядок, який містить числовий вираз, наприклад 7 – 2 * 3, а на виході видає результат? Процедура, що передує виконанню цієї дії, називається *синтаксичним аналізом виразів* (англ. – expression parsing) і є основою всіх компіляторів та інтерпретаторів мов, електронних таблиць та інших програм, в яких потрібно перетворювати числові вирази на програмний код. Зрозумівши механізм роботи синтаксичного аналізу виразів, ми навчимося створювати складніші синтаксичні аналізатори, а також зрозуміємо механізм роботи програм-трансляторів.

Синтаксичний аналізатор (*парсер*) – це програма, що виконує синтаксичний аналіз тексту програми, на першому кроці якого виконується *лексичний аналіз*, потім будується *синтаксичне дерево* програми – дерево її виведення в граматиці мови програмування. Парсер входить до структури транслятора.

Розглянемо принципи побудови парсерів на прикладі аналізу складних арифметичних виразів – звичайних арифметичних виразів і виразів присвоювання, а також процес обчислення значення таких виразів [136]. Незважаючи на те, що складні вирази можуть утворюватися з даних довільних типів, обмежимося лише числовими виразами. Вважатимемо, що вони складаються з таких елементів: числа, змінні, операції `+, -, /, *, ^` (піднесення до степеня), `%, =` і дужки.

Перелічені елементи можна комбінувати у виразах відповідно до правил побудови інфіксних термів (див. підрозд. 1.1.1) з урахуванням пріоритету операцій, наприклад: `(11+a)/2*6; 7+8; a+b-c; a=9-b; 10^3`.

Нехай операції мають пріоритет у порядку спадання:

(Вищий) (унарні `+, -`) ⇒ (`^`) ⇒ (`*, /, %`) ⇒ (`+, -`) ⇒ (присвоювання `=`) (Нижчий).

Операції з однаковим пріоритетом виконуються зліва направо.

Для спрощення вважатимемо, що імена змінних не залежать від регістра та є однолітерними (допускається 26 змінних, від `A` до `Z`). Наприклад, `b` та `B` позначають ту саму змінну. Кожне числове значення має тип `double`. Контроль за помилками – мінімальний.

Мову складних арифметичних виразів можна задати такою граматикою G:

<*складний-вираз*> → <*присвоювання*> | <*вираз*>

<*присвоювання*> → <*змінна*>=<*вираз*>

<*вираз*> → <*доданок*> [<*операція*1> <*доданок*>]

<*доданок*> → <*множник* >[<*операція*2> <*множник*>]

<*множник*> → <*степінь*>[<*операція*3> <*множник*>]

<*степінь*> → <*операція*1>(<*вираз*>) | (<*вираз*>)| <*змінна*> | <*число*>

<*операція*1> → + | -

<*операція*2> → * | / | %

<*операція*3> → ^

<*число*> → (<*цифра*>)*

<*цифра*> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<*змінна*> → *a* | *b* | … | *z* | *A* | *B* | … | *Z*

З теоретичними відомостями стосовно граматик можна ознайомитися в підрозд. 1.4.4.

Зрозуміло, що не становить принципової складності розширити такий синтаксичний аналізатор на випадок загального іменування, чутливого до регістра.

Спробуємо обчислити вираз 7 – 2 * 3. Зрозуміло, що значення дорівнює 1. Спочатку може спасти на думку такий алгоритм:

```
a=одержати перший операнд
while(є операнд){
 op=одержати операцію
 b=одержати другий операнд
 a=a op b
}
```

Однак при обчисленні виразу 7 – 2 * 3 у результаті виходить 15 замість 1, оскільки описана процедура не враховує пріоритет операцій. Не можна просто вибирати операнди й операції ліворуч і праворуч, оскільки за пріоритетом операцій множення застосовується раніше ніж віднімання.

4.5.1. ЛЕКСИЧНИЙ АНАЛІЗ ВИРАЗІВ

Для обчислення виразів необхідно вміти розбивати їх на лексеми (див. підрозд. 3.1.2). Які об'єкти вважати лексемами, залежить від мови програмування. З кожною лексемою пов'язується структура, що складається з пари: тип лексеми й покажчик на поле, де зберігається інформація про конкретну лексему. Кількість типів лексем залежить від конкретної мови, що аналізується (у даному випадку – мова арифметичних виразів).

Лексичний аналізатор – це програма, входом якої є низка символів, що зображує вхідний текст програми, а виходом – низка лексем, яка, у свою чергу, подається на вхід синтаксичного аналізатора.

Перехід до лексем робить зображення програми зручнішим для подальшої обробки. У процесі лексичного аналізу вилучаються з тексту несуттєві елементи (зайві пробіли, коментарі тощо).

Конструкції, що розпізнаються лексичним аналізатором, описуються автоматною граматикою й регулярними виразами (стосовно скінченних автоматів і регулярних виразів див. підрозд. 1.4.4.)

На першому етапі лексичного аналізу будується детермінований скінченний автомат за регулярними виразами, потім – таблиця переходів для автомата, а далі вхідний потік символів інтерпретується аналізатором у вихідний потік лексем.

В основі лексичного аналізатора лежить діаграма переходів відповідного скінченного автомата.

Типами лексем для лексичного аналізу описаного вище арифметичного виразу є: **<змінна>**, **<число>**, **<операція_1>**, **<операція_2>**, **<операція_3>**. Останні три типи зручно об'єднати, додавши до них дужки. Таким чином, для розроблюваного аналізатора використовуються лише три типи лексем: змінна, число й операція або роздільник. У реалізації їм відповідають константи **VARIABLE**, **NUMBER** і **DELIMITER** (**DELIMITER** використовується як для операцій, так і для дужок).

Лексичний аналізатор може бути як самостійною фазою трансляції (препроцесорна обробка), так і підпрограмою, що працює за принципом "дай лексему" (**getToken**). У першому випадку виходом аналізатора є таблиця лексем, у другому – лексема видається при кожному зверненні до аналізатора. При цьому зазвичай ознака класу лексеми повертається як результат функції, що реалізує лексичний аналізатор (**getToken**), а значення лексеми передається через глобальну змінну. У процесі обробки лексем аналізатор може або просто видавати значення кожної лексеми (при цьому побудова таблиць ідентифікаторів переноситься на пізніші фази), або самостійно будувати таблиці ідентифікаторів.

У будь-якому разі потрібен метод, що повертає один за одним усі лексеми виразу. Метод, що розбиває вираз на лексеми, має виконувати такі дії: 1) ігнорувати роздільники (пробіли, табуляції й переходи на новий рядок); 2) виділяти лексеми з тексту; 3) визначати тип лексеми. Кожен елемент виразу називається *лексемою* (**token**), тому метод, що повертає чергову лексему, часто називається **getToken()**. У ньому використовується покажчик на рядок із виразом, що розбирається. У розглядуваній версії функції **getToken()** такий покажчик називається **prog**. Атрибут **prog** має зберігати своє значення між викликами функції **getToken()** і бути доступним іншим методам. Крім значення лексеми, необхідно знати її тип (**VARIABLE**, **NUMBER** та **DELIMITER**). Нижче наведено фрагмент опису класу, відповідального за синтаксичний і лексичний аналіз і текст методу **getToken()** разом із необхідними деклараціями, константами й допоміжним методом.

Приклад 4.31. Лексичний аналіз:

Лістинг.

```
#define DELIMITER 1
#define VARIABLE  2
#define NUMBER    3
```

```
using namespace std;

class CSimpleParser {
public:

    CSimpleParser()
    {
          prog=new (char[100]);
          if(!prog) {
                printf("Помилка при виділенні пам'яті\n");
                exit(1);
          };
          for(int i=0; i<26; i++)
                vars[i]=0.0;
    };
    void evalExp(double *answer);
    char *prog;   /*указує на вираз, що аналізується*/

private:

char token[80];
    char tokType;

    double vars[26];  /*26 змінних, A-Z*/

    void setProg(char *pr)
    {
          prog=pr;
    };

    void setToken(char tok[80])
    {
          strcpy(token,tok);
    };

    void getToken();
    int isdelim(char c);
… … …

};

/*Повернення чергової лексеми*/
void CSimpleParser::getToken()
{
register char *temp='\0';

temp=token;
tokType=0;
```

```
if(!*prog) return; /*кінець виразу*/
while(isspace(*prog)) ++prog; /*пропустити пробіли,
          символи табуляції й порожнього рядка*/

if(strchr("+-*/%^=()", *prog)){
    tokType=DELIMITER;
    /*перейти до наступного символу*/
    *temp++ = *prog++;
}
else if(isalpha(*prog)) {
    while(!isdelim(*prog)) *temp++ = *prog++;
    tokType=VARIABLE;
}
else if(isdigit(*prog)) {
    while(!isdelim(*prog)) *temp++ = *prog++;
    tokType=NUMBER;
}

*temp='\0';
}
/*Повернення значення ІСТИНА, якщо символ c є роздільником*/
int CSimpleParser::isdelim(char c)
{

if(strchr("+-/*%^=()", c)||c==9||c=='\r'||c==0)
    return 1;
return 0;
}
```

Розглянемо наведені вище методи докладніше. Після кількох ініціалізацій метод **getToken()** перевіряє, чи не досягнутий символ кінця рядка (**'\0'**), який завершує вираз. Якщо у виразі ще є нерозібрана частина, то метод **getToken()** спочатку пропускає пробіли, якщо вони є. Після цього **prog** указує на число, змінну, операцію або, якщо вираз завершується пробілами, – то на символ кінця рядка (**'\0'**). Якщо черговий символ є операцією, то він повертається у вигляді рядка, збереженого в глобальній змінній **token**, а змінній **tokType**, що містить тип отриманої лексеми, присвоюється значення **DELIMITER**. Якщо ж наступний символ є літерою, то він вважається іменем змінної й повертається в рядковій змінній **token**. При цьому **tokType** одержує значення **VARIABLE**. У випадку, коли черговий символ є цифрою, зчитується все число, причому воно міститься в змінній **token**, а його типом є **NUMBER**. Нарешті, якщо наступний символ не є жодним із перерахованих вище, то вва-

жається, що досягнутий кінець виразу. У цьому випадку **token** містить порожній рядок, повернення якого означає кінець виразу.

Як уже було сказано раніше, щоб не ускладнювати код цього методу, були опущені деякі засоби контролю за помилками та зроблені деякі припущення. Наприклад, будь-який нерозпізнаний символ завершує вираз. Крім того, у даній версії програми імена змінних можуть мати будь-яку довжину, але значущою є тільки перша літера. Відповідно до вимог конкретного завдання можна ускладнити засоби контролю за помилками й додати інші подробиці.

Для кращого розуміння принципу дії методу **getToken()** нижче наведені лексеми, що повертаються ним, і типи лексем для вхідного виразу **A * B-(W+10):**

Лексема	Тип лексеми
A	VARIABLE
*	DELIMITER
B	VARIABLE
-	DELIMITER
(	DELIMITER
W	VARIABLE
+	DELIMITER
10	NUMBER
)	DELIMITER
0 (кінець рядка)	0(нуль)

Варто пам'ятати, що атрибут **token** завжди містить рядок, який завершується символом кінця рядка (**'\0'**), навіть якщо цей рядок складається тільки з одного символу ■

Після того, як у результаті роботи лексичного аналізу лексеми розпізнано, інформація про них збирається й запам'ятовується в одній із кількох таблиць (таблиці ідентифікаторів). Зміст цієї інформації залежить від мови (у даному випадку – мови арифметичних виразів).

Таблиця ідентифікаторів (*змінних*) забезпечує можливість швидкого додавання нових ідентифікаторів, нових відомостей про них і швидкого пошуку інформації для даного ідентифікатора.

У нашому випадку для спрощення вважатимемо, що застосовуються лише однолітерні змінні латинського алфавіту без урахування регістра. Тому нам потрібна така таблиця ідентифікаторів, де кожна змінна зберігається в одному елементі масиву з 26 компонентами типу **double** (у складнішому випадку для реалізації таблиці застосовують інші методи – зазвичай це хеш-таблиці). Отже, у вихідний текст аналізатора додамо такий фрагмент:

```
double vars[26]; /*26 змінних, A-Z*/
```

Ці змінні первісно ініціалізуються `0.0` у конструкторі класу `CSimpleParser`.

Крім цього, знадобиться метод для одержання значення заданої змінної. Оскільки імена змінних є літерами від `A` до `Z`, то їх можна використати для індексації масиву **vars**, віднімаючи код ASCII літери `A` від імені змінної. Нижче описано метод `findVar()`, що повертає значення змінної:

```
/*Одержання значення змінної*/
double CSimpleParser::findVar(char *s)
{
 if(!isalpha(*s)){
          serror(1);
          return 0.0;
    }
    return vars[toupper(*token)-'A'];
}
```

Цей метод приймає імена будь-якої довжини, але тільки перший символ є значущим. Дане обмеження за потреби можна змінити.

Розглянемо метод `atom()`, що обробляє числа та змінні:

```
/*Одержання значення числа або змінної*/
void CSimpleParser::atom(double *answer)
{
 switch(tokType) {
 case VARIABLE:
 *answer=findVar(token);
 getToken();
 return;
 case NUMBER:
 *answer=atof(token);
 getToken();
 return;
 default:
 serror(0);
 }
}
```

4.5.2. СИНТАКСИЧНИЙ АНАЛІЗ

Синтаксичний аналіз – це процес, в якому досліджується низка лексем і встановлюється, чи відповідає вона граматичним правилам, що описують синтаксис мови. На виході парсер видає синтаксичне дерево й таблиці (ідентифікаторів, типів).

Зазвичай фази лексичного й синтаксичного аналізу взаємодіють між собою в межах одного проходження програми. При цьому основною є фаза синтаксичного аналізу, який, у свою чергу, звертається до лексичного аналізу за черговим термінальним символом.

Теоретичні відомості про КВ- та LL(k)-граматики, побудову множини $FIRST_k$ (ω) можна знайти в підрозд. 1.4.4.

Більшість відомих методів аналізу належать до одного з двох класів, один з яких об'єднує *низхідні*, а інший – *висхідні* алгоритми. Походження цих термінів пов'язане з тим, як утворюються вузли синтаксичного дерева: від аксіом граматики до термінальних символів (*низхідні парсери*) чи від термінальних символів до аксіом граматики (*висхідні парсери*).

З низхідними парсерами пов'язані LL-граматики, які можуть бути проаналізовані достатньо простим для реалізації методом рекурсивного спуску.

З висхідними парсерами пов'язані LR(1)-граматики [7], що задають ширший клас мов (за їхньою допомогою визначається синтаксис більшості мов програмування). Для них розроблені спеціальні програми автоматичної побудови парсерів. Найвідоміші з них – утиліти Lex та Yacc [144].

Lex – програма для генерації лексичних аналізаторів. Вона є стандартним генератором лексичних аналізаторів у ОС Unix. Lex зчитує вхідний потік, що описує лексичний аналізатор, і повертає на виході код аналізатора мовою програмування С.

Yacc (від англ. Yet Another Compiler Compiler – ще один компілятор компіляторів) – програма для генерації парсерів у Unix-системах. Розроблена С. Джонсоном у компанії AT&T. Вона отримує на вхід LR(1)-граматику, описану в нотації БНФ, і генерує код парсера мовою програмування С. Пізніше програма була переписана іншими мовами, такими як Ada, Java, C# тощо. Існує Стандарт IEEE POSIX P1003.2, що визначає як функціональність, так і вимоги до програм типу Lex і Yacc.

4.5.3. ОБЧИСЛЕННЯ ВИРАЗІВ

Розглянемо складні арифметичні вирази, що описуються КВ-граматикою G (див. підрозд. 1.4.4), і на їхньому прикладі покажемо, як будується парсер і обчислюються значення виразів. Вираз потрібно спочатку проаналізувати й отримати синтаксичне дерево, потім шляхом рекурсивного обходу цього дерева обчислюється значення виразу.

Спочатку звернемося до граматики G_1, що є частиною граматики G і задає в ній вирази без присвоювань. Граматика G_1 є LL(1)-

граматикою. Для синтаксичного аналізу за такою граматикою може бути застосований метод рекурсивного спуску (без повернень).

Побудуємо для граматики G_1 парсер і програму для обчислення значень виразів. Розглянемо спочатку загальний алгоритм і окремі методи, застосовані в ньому.

Щоб зрозуміти, як саме програма обчислює вираз, розглянемо вираз 7 – 2 * 3. Припустимо, що покажчик **prog** указує на початок виразу.

При виклику методу **evalExp()** – вхідної точки програми – зі вхідного рядка вибирається лексема. Якщо лексема є порожнім рядком, то метод друкує повідомлення "Немає виразу" і завершується. Однак у нашому випадку лексемою є число 7. Оскільки це непорожній рядок, то викликається метод **evalExp1()** (аналіз: <*вираз*> → <*доданок*> <*операція*1> <*доданок*>). Потім метод **evalExp1()** викликає **evalExp2()** (аналіз: <*доданок*> → <*множник*> <*операція*2> <*множник*>); **evalExp2()** викликає **evalExp3()** (аналіз: <*множник*> → <*степінь*> <*операція*3> <*множник*>); той, у свою чергу, – **evalExp4()** (аналіз: <*степінь*> → <*операція*1> (<*вираз*>)). Потім метод **evalExp4()** перевіряє, чи не є лексема унарним плюсом або мінусом. У розглядуваному випадку це не так, тому викликається метод **evalExp5()** (аналіз: <*степінь*> → (<*вираз*>)| <*змінна*> | <*число*>). У цей момент **evalExp5()** може рекурсивно викликати або **evalExp1()** (у випадку виразу, вкладеного в дужки), або **atom()**, щоб визначити значення числа. Оскільки лексема не є відкриваючою дужкою, то виконується метод **atom()** і змінній ***answer** присвоюється значення 7. Потім відбувається перехід до наступної лексеми й повернення з ланцюжка викликів методів. Лексемою стає операція **–**, а керування повертається методу **evalExp1()**.

Те, що відбувається далі, дуже важливо. Оскільки поточною лексемою є символ -, то він зберігається у змінній **op**. Потім аналізатор вибирає наступну лексему, і спуск по ланцюжку починається знов. Як і раніше, викликається метод **atom()**. Отримане значення 2 повертається змінною ***answer**, і зчитується лексема *. Це викликає повернення по ланцюжку до **evalExp2()**, де зчитується остання лексема 3. У цей момент відбувається перша арифметична операція – множення 2 на 3. Отриманий результат повертається методу **evalExp1()**, де відбувається віднімання. У результаті віднімання отримуємо 1.

Повернемося тепер до граматики *G*, що задає складні арифметичні вирази.

Побудуємо значення функції FIRST_1 для правих частин правил цієї граматики:

$FIRST_1(<присвоювання>) = \{a, \ldots, z, A, \ldots, Z\}$

$FIRST_1(<вираз>) = FIRST_1(<доданок>\ [<операція1>\ <доданок>]) = \{+, -, (, a, \ldots, z, A, \ldots, Z, 0, \ldots 9\}$

$FIRST_1(<множник>\ [<операція2>\ <множник>]) = \{+, -, (, a, \ldots, z, A, \ldots, Z, 0, \ldots 9\}$

$FIRST_1((<вираз>)) = \{ (\}$

$FIRST_1(<змінна>) = \{a, \ldots, z, A, \ldots, Z\}$

$FIRST_1(<число>) = \{0, \ldots, 9\}$.

Очевидно, що граматика *G* не є LL(1)-граматикою. Це випливає з того, що перетин множин

$FIRST_1(<присвоювання>) \cap FIRST_1(<вираз>) = \{a, \ldots, z, A, \ldots, Z\}$

непорожній, отже, для неї метод рекурсивного спуску не буде працювати.

Існують алгоритми, що перетворюють неLL(1)- на LL(1)-граматики. Проте вони достатньо складні, тому для синтаксичного аналізу за неLL(1)-граматикою зручніше скористатися *методом рекурсивного спуску з поверненням*. При цьому лексичний аналізатор матиме додатковий метод, який у всіх неоднозначних ситуаціях перед початком аналізу запам'ятовує поточний стан лексичного аналізатора й намагається продовжити виведення за першою з можливих альтернатив. Якщо альтернатива невірна, то повертається до запам'ятованого стану й переглядає наступну. Якщо всі варіанти продовження виведення завершилися невдачею, то повертається повідомлення про помилку.

З технічного погляду це все, що потрібно аналізатору для коректної обробки змінних. Однак поки немає способу присвоїти змінним значення. Часто це робиться за межами аналізатора, але в аналізаторі можна розглядати знак рівності як знак операції присвоювання й обробляти його частиною аналізатора. Цього можна досягти кількома способами. Один із них – додати в аналізатор метод **`evalExp6()`** (аналіз: $<присвоювання> \rightarrow <змінна>=<вираз>$).

Приклад 4.32. Синтаксичний аналіз:

Лістинг.

```
/*Обробка присвоювання*/
void CSimpleParser::evalExp6(double *answer)
{
    int slot;
    char ttokType;
    char tempToken[80];

 if(tokType==VARIABLE) {
         /*зберегти стару лексему*/
```

```
          strcpy(tempToken, token);
          ttokType=tokType;
          /*обчислити індекс змінної*/
          slot=toupper(*token) - 'A';
getToken();
          if(*token != '=') {
 putback(); /*повернути поточну лексему*/
                /*відновити стару лексему - це не присвоювання*/
                strcpy(token, tempToken);
 tokType=ttokType;
          }
          else {
                getToken(); /*одержати наступну частину виразу*/
                evalExp1(answer);
                vars[slot]=*answer;
                return;
          }
    }
    evalExp1(answer);
}
```

У цьому методі доводиться заглядати вперед, щоб визначити, чи є насправді присвоювання. Це пов'язано з тим, що ім'я змінної завжди перебуває перед оператором присвоювання, але наявність імені змінної не гарантує, що за нею піде присвоювання. Інакше кажучи, аналізатор сприйме вираз A=100 як присвоювання, причому він може визначити, що A/10 ним не є. Для цього метод **evalExp6()** зчитує зі вхідного потоку наступну лексему. Якщо ця лексема не є знаком рівності, то вона за допомогою методу **putback()** повертається у вхідний потік для наступного використання:

```
/*Повернення лексеми у вхідний потік*/
void CSimpleParser::putback(void)
{
 char *t;

 t=token;
 for(; *t; t++) prog-i;
}
```

■

Приклад 4.33. Повний текст програми обчислення складних виразів:

Лістинг.

```
/*Даний модуль містить рекурсивний спадний
 парсер, що розпізнає змінні*/
#include <iostream>
#include <fstream>
#include <string>

#define DELIMITER 1
#define VARIABLE  2
#define NUMBER    3

using namespace std;

class CSimpleParser {
public:

    CSimpleParser()
    {
          prog=new (char[100]);//[100];
          if(!prog) {
                printf("Помилка при виділенні пам'яті.\n");
                exit(1);
          };
          for(int i=0; i<26; i++)
                vars[i]=0.0;
    };
    void evalExp(double *answer);
    char *prog;   /*містить вираз, що аналізується*/

private:
    void setProg(char *pr)
    {
          prog=pr;
    };

    void setToken(char tok[80])
    {
          strcpy(token,tok);
    };

    void evalExp1(double *answer);
    void evalExp2(double *answer);
    void evalExp3(double *answer);
    void evalExp4(double *answer);
    void evalExp5(double *answer);
    void evalExp6(double *result);
    void atom(double *answer);
    void getToken();
```

```
    void putback();
    void serror(int error);
    int isdelim(char c);
    /*Одержання значення змінної*/
    double findVar(char *s);
    char token[80];
    char tokType;

    double vars[26];  /*26 змінних, A-Z*/

};

/*Обробка присвоювання*/
void CSimpleParser::evalExp6(double *answer)
{
 int slot;
 char ttokType;
 char tempToken[80];

 if(tokType==VARIABLE) {
 /*зберегти стару лексему*/
 strcpy(tempToken, token);
 ttokType=tokType;
 /*обчислити індекс змінної*/
 slot=toupper(*token) - 'A';
getToken();
 if(*token != '=') {
 putback(); /*повернути поточну лексему*/
 /*відновити стару лексему - це не присвоювання*/
 strcpy(token, tempToken);
 tokType=ttokType;
 }
 else {
 getToken(); /*одержати наступну частину виразу*/
 evalExp1(answer);
 vars[slot]=*answer;
 return;
 }
 }
 evalExp1(answer);
}

/*Одержання значення змінної*/
double CSimpleParser::findVar(char *s)
{
 if(!isalpha(*s)){
 serror(1);
```

```
 return 0.0;
 }
 return vars[toupper(*token)-'A'];
}

/*Точка входження аналізатора*/
void CSimpleParser::evalExp(double *answer)
/*точка входження аналізатора*/
{
 getToken();
 if(!*token) {
 serror(2);
 return;
 }
 evalExp6(answer);
 if(*token) serror(0); /*остання лексема має бути нулем*/
}


/*Додавання або віднімання двох доданків*/
void CSimpleParser::evalExp1(double *answer)
{
double temp;
register char  op;

evalExp2(answer);
while((op=*token)=='-'||op=='+') {
    getToken();
    CSimpleParser(token);
    evalExp2(&temp);
    switch(op) {
          case '+':
                *answer=*answer+temp;
                break;
          case '-':
                *answer=*answer - temp;
                break;
    }
}
}

/*Множення або ділення двох множників*/
void CSimpleParser::evalExp2(double *answer)
{
double temp;
register char op;

evalExp3(answer);
```

```
while((op=*token)=='*'||op=='/'||op=='%') {
    getToken();
    evalExp3(&temp);
    switch(op) {
          case '*':
                *answer=*answer*temp;
                break;
          case'/':
                if(temp==0.0) {
                serror(3); /*ділення на нуль*/
                *answer=0.0;
                } else *answer=*answer/temp;
                break;
          case'%':
                *answer=(int) *answer % static_cast<int> (temp);
                break;
    }
}
}
/*Піднесення до степеня*/
void CSimpleParser::evalExp3(double *answer)
{
double temp, ex;
register int t;

evalExp4(answer);
if(*token=='^') {
    getToken();
    evalExp3(&temp);
    ex=*answer;
    if(temp==0.0) {
    *answer=1.0;
    return;
    }
for(t=temp-1; t>0; --t) *answer=(*answer)* static_cast<double>(ex);
}
}

/*Множення унарних операцій + та -*/
void CSimpleParser::evalExp4(double *answer)
{
register char op=0;

if((tokType==DELIMITER) && *token=='+'||*token=='-') {
    op=*token;
    getToken();
}
evalExp5(answer);
```

```
if(op=='-') *answer=-(*answer);
}

/*Обчислення виразів у дужках*/
void CSimpleParser::evalExp5(double *answer)
{
if((*token=='(')) {
    getToken();
    evalExp1(answer);
    if(*token!=')')
    serror(1);
    getToken();
}
else
    atom(answer);
}

/*Одержання значення числа або змінної*/
void CSimpleParser::atom(double *answer)
{
 switch(tokType) {
 case VARIABLE:
 *answer=findVar(token);
 getToken();
 return;
 case NUMBER:
 *answer=atof(token);
 getToken();
 return;
 default:
 serror(0);
 }
}

void CSimpleParser::putback()
{
char *t;

t=token;
for(; *t; t++) prog--;
}

/*Відображення повідомлення про помилку*/
void CSimpleParser::serror(int error)
{
static char *e[]={
    "Синтаксична помилка",
    "Незбалансовані дужки",
```

```
    "Немає виразу",
    "Ділення на нуль"
};
    printf("%s\n", e[error]);
    }

/*Повернення чергової лексеми*/
void CSimpleParser::getToken()
{
register char *temp='\0';

temp=token;
tokType=0;

if(!*prog) return; /*кінець виразу*/
while(isspace(*prog)) ++prog; /*пропустити пробіли,
                              символи табуляції й порожнього
                              рядка*/
if(strchr("+-*/%^=()", *prog)){
    tokType=DELIMITER;
    /*перейти до наступного символу*/
    *temp++=*prog++;
}
else if(isalpha(*prog)) {
    while(!isdelim(*prog)) *temp++=*prog++;
    tokType=VARIABLE;
}
else if(isdigit(*prog)) {
    while(!isdelim(*prog)) *temp++=*prog++;
    tokType=NUMBER;
}

*temp='\0';
}

/*Повернення значення ІСТИНА, якщо с є роздільником*/
int CSimpleParser::isdelim(char c)
{

if(strchr("+-/*%^=()", c)||c==9||c=='\r'||c==0)
    return 1;
return 0;
}
```

У наведеному вигляді програма підтримує такі операції: `+`, `-`, `*`, `/`, `%`. Крім того, вона вміє підносити до цілочислового степеня `(^)` і обчислювати унарний мінус, а також коректно розпізнавати дужки.

Програма складається з шести рівнів, а також методу **atom**, що повертає значення числа. Як обговорювалося раніше, у змінній **token** повертається чергова лексема з рядка, що містить вираз, а в **tokType** – тип лексеми. Атрибут **prog** указує на рядок, що містить вираз ■

Приклад 4.34. Застосування програми:

Лістинг.

```
int main()
{
 double answer;

 CSimpleParser *csp=new CSimpleParser;

 /*Обробка виразів до введення порожнього рядка*/
 int fl=1;
 do {
     cout<<"Введіть вираз:";
     gets(csp->prog);

 if(!*csp->prog) fl=0;
 else
 {
     csp->evalExp(&answer);
     cout<<"Результат:"<<answer<<endl;
 };
 } while(fl==1);
 delete csp;
 return 0;
}
```

Парсер дозволяє вводити вирази, подібні таким:

A = 10/4;
A – B;
C = A * (F – 21) ■

При вивченні коду парсерів розглядався метод **serror()**, що викликається в певних ситуаціях і повідомляє про помилки. На відміну від багатьох інших типів парсерів, рекурсивний спуск полегшує перевірку синтаксису, оскільки в більшості випадків вона відбувається в методах **atom()**, **findVar()** та **evalExp5()**, де перевіряється правильність розміщення дужок. Єдина проблема, пов'язана із синтаксичними помилками, полягає в тому, що при виявленні помилки аналіз виразу не припиняється. Це може привести до виведення кількох повідомлень про помилки.

Якщо залишити код програми без змін, то можуть виводитися відразу кілька повідомлень про синтаксичні помилки. У деяких ситуаціях це заважає, але в інших може бути дуже корисним, оскільки з'являється можливість виявити відразу кілька помилок. Даний парсер добре застосовний для настільного калькулятора, що було продемонстровано попередньою програмою, або для невеликої бази даних.

***Література для СР:** загальні методи синтаксичного аналізу – [7, 11, 65]; синтаксичний аналіз без повернення – [7, 11, 19, 65, 104]; висхідний синтаксичний аналіз – [7, 11, 65]; утиліти Lex та Yacc – [143, 144].

Контрольні запитання та вправи

1. Що таке лексичний аналіз?
2. Що таке синтаксичний аналізатор?
3. Що таке синтаксичне дерево?
4. Що таке таблиця ідентифікаторів?
5. Що таке метод рекурсивного спуску з поверненням?
6. Що таке утиліти Lex та Yacc?
7. Нехай задано автомат: $Q = \{q_0, q_1, q^*\}$; $F = \{q_1, q^*\}$; $T = \{0,1,2,3,4,5,6,7,8,9\}$; $q_0 1 \to q_1$; $q_0 2 \to q_1$; $q_0 3 \to q_1$;...; $q_0 9 \to q_1$; $q_1 1 \to q_1$; $q_1 2 \to q_1$; $q_1 3 \to q_1$; ...; $q_1 9 \to q_1$. Побудувати за ним програму лексичного й синтаксичного аналізаторів.
8. Переписати програму з прикл. 4.33 так, щоб вона аналізувала й обчислювала арифметичний вираз з дійсними однолітерними змінними й однолітерними змінними-константами обох регістрів.
9. Те саме, що й у вправі 8, але іменами змінних і констант є довільні ідентифікатори. Передбачити розширений контроль над помилками.
10*. Розробити стратегію і провести модульне та інтеграційне тестування аналізаторів із вправ 8 та 9 – спочатку для окремих функцій, потім – усієї програми.
11. Реалізувати алгоритм усунення з правил КВ-граматики недосяжних символів (див. вправу 49 із підрозд. 1.4).
12. Реалізувати алгоритм пошуку множини $\mathrm{First}_k(A)$ для нетерміналів граматики. Вивести множини $\mathrm{First}_k(A)$ для кожного нетермінала.
13. Реалізувати алгоритм перевірки, чи буде задана КВ-граматика $LL(k)$ – граматикою для даного k.
14. Побудувати скінченний автомат і реалізувати лексичний аналізатор для фрагмента мови С. Вивести вміст таблиці лексем піс-

ля обробки тексту програми або повідомлення про лексичні помилки. Типи лексем визначити самостійно. Це можуть бути:

- зарезервовані слова (**int**, **if**, **else**, **while**, **switch**, **case** тощо);
- ідентифікатори;
- числові константи (цілі й дійсні числа);
- літерні константи;
- коди операцій (**+**, **–**, *****, **=**, **==**, **<**,**>** тощо);
- роздільники (**{**, **,**, **;**, **}** тощо).

Підмножина мови С містить:

а) дані типу **int**, описи змінних, оператори присвоювання в довільній послідовності; операції **+, –, *, = =, !=, <, >**;

б) те саме, що й у а), але з доданими операторами **if** та **if-else** довільної вкладеності й у довільній послідовності;

в) те саме, що й у б), тільки оператори **if** та **if-else** замінити на оператори **while**;

г) те саме, що й у б), тільки оператори **if** та **if-else** замінити на оператори **for**;

д) те саме, що й у б), тільки оператори **if** та **if-else** замінити на оператори **do-while**;

е) те саме, що й у а), але з доданими типами **float** і масивами зазначених типів.

15. Реалізувати синтаксичний аналізатор для описаних у вправі 14 фрагментів мови С методом рекурсивного спуску.
16. Побудувати синтаксичний аналізатор для даного поняття й там, де це можливо, обчислити значення заданого виразу. Вхідний потік вводиться з клавіатури. Поняття задано в лінійній СД (жирні літери-дужки є метасимволами, див. підрозд. 1.4.4):

1) <*список-списків*> **::=** <*список*> **{;** <*список*>**}**
<*список*> **::=** <*літера*> **{,** <*літера*>**}**

2) <*дійсне-число*> **::=** <*ціле-число*> **.** <*ціле-без-знака*> |
<*ціле-число*> **[.** <*ціле-без-знака*>**]** *E*<*ціле-число*>
<*ціле-без-знака*> **::=** <*цифра*>{<*цифра*>}
<*ціле-число*> **::= [**- | +**]** <*ціле-без-знака*>

3) <*сума*> **::=** <*ціле*> {<*знак-операції*> <*ціле*>}
<*знак-операції*> **::=** - | + | *
<*ціле*> **::=** <*цифра*>{ <*цифра*>}

4) <*дужки*> **::=** <*квадратні*> | <*круглі*>
<*квадратні*> **::=** *B* | [<*круглі*> <*круглі*>]
<*круглі*> **::=** *A* | (<*квадратні*> <*квадратні*>)

5) <*простий-вираз*> **::=** <*простий-ідентифікатор*> |
(<*простий-вираз*> <*знак-операції*> <*простий-вираз*>)
<*знак-операції*> **::=** - | + | *
<*простий-ідентифікатор*> **::=** <*літера*>

6) <*список-параметрів*> ::= <*параметр*> {, <*параметр*>}
<*параметр*> ::= <*ім'я*> = <*цифра*> <*цифра*> |
<*ім'я*> = (<*список-параметрів*>))
<*ім'я*> ::= <*літера*> <*літера*> <*літера*>
7) <*дужки*> ::= <*квадратні*> | <*круглі*>
<*квадратні*> ::= *B* | [[<*квадратні*>] (<*круглі*>)]
<*круглі*> ::= *A* | ((<*круглі*>) [<*квадратні*>])
8) <*константний-вираз*> ::= <*цифра*> {<*цифра*>} |
(<*константний-вираз*> <*знак-операції*> <*константний-вираз*>)
<*знак-операції*> ::= - | + | *.
9) <*простий-логічний*> ::= <*простий-ідентифікатор*> |
TRUE | *FALSE* | (<*простий-логічний*> <*знак-операції*>
<*простий-логічний*>)
<*знак-операції*> ::= <*AND*> | *OR*
<*простий-ідентифікатор*> ::= <*літера*>

10) Визначити, чи еквівалентний заданий простий логічний вираз виразу *FALSE.*
11) Визначити, чи зберігає заданий простий логічний вираз свої значення при довільній перестановці значень аргументів.
12) Визначити, чи еквівалентні два прості логічні вирази.

4.6. Пошук у графах

- Зображення графів
- Пошук шляхів у графах
- Пошук у ширину
- Пошук у глибину
- Алгоритм Дейкстри
- Застосування алгоритмів пошуку
- Жадібні алгоритми

Ключові слова: *комбінаторний вибух, зображення графа у вигляді списку суміжних вершин і матриці суміжності, пошук у глибину, пошук у ширину, алгоритм Дейкстри.*

Багато комбінаторних задач прямо чи опосередковано використовують графи. Більшість із них зводиться до задач пошуку в графах. Типовою є така задача. Нехай необхідно доїхати від Києва до Сімферополя автобусом певної компанії АВС за умови, що між цими містами немає прямого рейсу.

Цілком природно зобразити всі рейси компанії у вигляді зваженого графа, вершини якого відповідають містам, а дуги – відстані між ними (рис. 4.7). Задача зводиться до пошуку відповідного шляху в цьому графі (бажано якомого коротшого).

Перше, що спадає на думку, – використати метод повного перебору всіх шляхів у графі. Проте слід узяти до уваги таке: при доданні нового проміжного міста в маршрут з'являється n варіантів для продовження шляху (n – кількість дуг, що виходять із відповідної вершини), тобто кількість потенційних ланцюжків, що ведуть до розв'язку, збільшується значно швидше, ніж кількість проміжних міст. Якщо між містами є, наприклад, k проміжних міст, то кількість усіх можливих варіантів маршрутів має порядок $O(n^k)$.

Простим прикладом, що описує дану проблему, є швидкість збільшення кількості перестановок із n елементів при зростанні n.

Як відомо, кількість перестановок із n елементів дорівнює $n!$. Звідси кількість перестановок чотирьох елементів дорівнює $4!$, тобто 24, п'яти – 120, а шести – уже 720. Кількість перестановок 1000 елементів становить

$$1000! = 4{,}023872600770937735437024339 23\mathrm{e}+2567 .$$

Це дає уявлення про *комбінаторний вибух*. Як тільки кількість об'єктів перевищить якесь порівняно невелике число, зростання кількості комбінаторних об'єктів, що перебирають при розв'язанні (тих самих маршрутів), стає нестримним. Труднощі можуть виникнути навіть не при перевірці такої величезної кількості об'єктів, а набагато раніше – при перерахуванні.

Саме через те, що кількість можливостей зростає дуже швидко, лише в найпростіших завданнях можна застосовувати повний перебір варіантів. У зв'язку із цим були розроблені численні методи скорочення повного перебору, які забезпечують вичерпність пошуку й гарантують знаходження результату, якщо він існує.

Усі ці методи пов'язані з різними стратегіями *обходу* графів. Однак зупинимося спочатку на основних способах зображення графів.

4.6.1. ЗОБРАЖЕННЯ ГРАФІВ

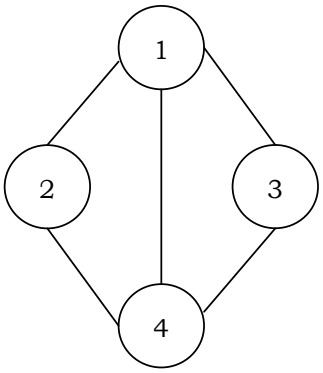
Теоретичні відомості про графи були розглянуті в підрозд. 1.4.4.

Існують два стандартні способи зображення графа $G=(V,E)$ у програмі. Перший спосіб подає граф у вигляді списку суміжних вершин. Для зображення графа $G=(V,E)$ у вигляді списку суміжних вершин використовують масив Adj із $|V|$ списків – по одному на вершину. Для

кожної вершини $u \in V$ список суміжних вершин $\mathrm{Adj}[u]$ містить у довільному порядку покажчики на всі суміжні з нею вершини (усі вершини v, для яких $(u,v) \in E$). Цей спосіб дає компактне зображення, особливо для розріджених графів. Його недолік: якщо треба дізнатися, чи є у графі ребро з u у v, то доводиться переглядати весь список. Ситуацію можна поліпшити, упорядкувавши множину пар лексикографічно.

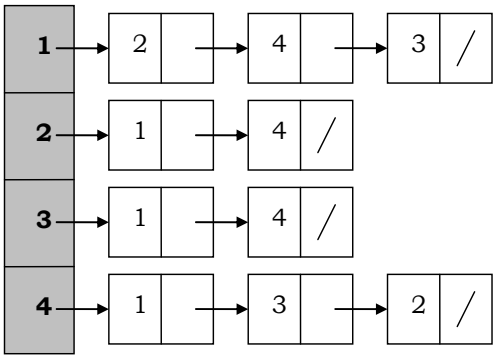
Нехай граф G має n вершин. Другий спосіб зображує граф як *матрицю суміжності* $A = (a_{ij})$ розміром $n \times n$. Усі вершини графа нумерують числами від 1 до n. Тоді $a_{ij} = 1$, якщо між i-ю та j-ю вершинами графа є ребро, і $a_{ij} = 0$, якщо такого ребра немає. Цей спосіб дозволяє швидко визначити, чи з'єднані дві вершини ребром, але потребує більше пам'яті.

Зображення неорієнтованого графа обома способами:



Граф 1

Зображення графа 1 у вигляді списку суміжних вершин:

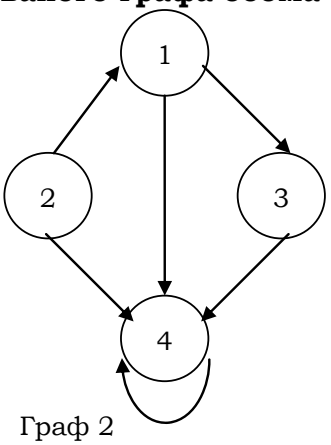


і за допомогою матриці суміжності:

	1	2	3	4
1	0	1	1	1
2	1	0	0	1
3	1	0	0	1
4	1	1	1	0

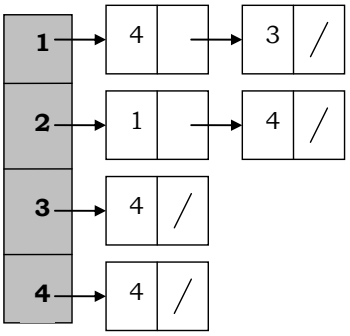
Неважко помітити, що для неорієнтованого графа сума довжин усіх списків суміжних вершин дорівнює подвоєній кількості ребер, оскільки ребру (u,v) відповідає по одному елементу зі списків $\mathrm{Adj}[u]$ та $\mathrm{Adj}[v]$. До того ж матриця суміжності для неорієнтованого графа симетрична відносно головної діагоналі, тобто збігається зі своєю транспонованою матрицею, оскільки (u,v) та (v,u) – одне й те саме ребро. Завдяки симетричності достатньо зберігати лише числа на головній діагоналі й вище неї – при цьому обсяг задіяної пам'яті зменшується майже вдвічі.

Зображення орієнтованого графа обома способами:



Граф 2

Зображення графа 2 у вигляді списку суміжних вершин:

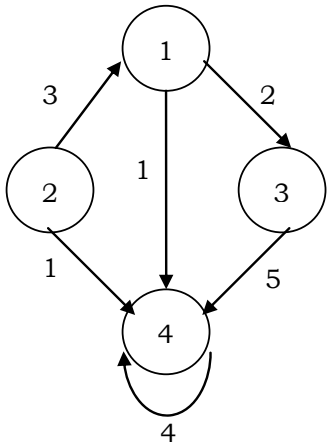


і за допомогою матриці суміжності:

	1	2	3	4
1	0	0	1	1
2	1	0	0	1
3	0	0	0	1
4	0	0	0	1

Для орієнтованого графа сума довжин усіх списків суміжних вершин дорівнює загальній кількості ребер: ребру (u, v) відповідає елемент v списку $\mathrm{Adj}[u]$.

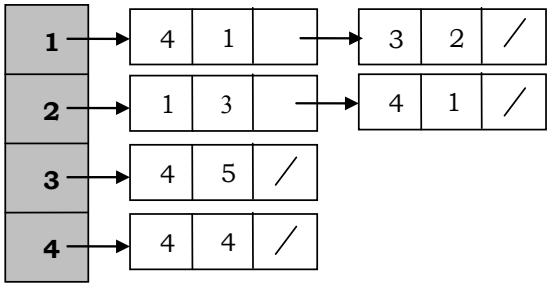
Зображення орієнтованого зваженого графа обома способами:



Граф 3

Списки суміжних вершин зручні для подання зважених графів, в яких кожному ребру відповідає деяка вага (напр., число).

Зображення графа 3 у вигляді списку суміжних вершин:



За допомогою матриці суміжності також зручно зображувати орієнтовані графи, при цьому вагу w ребра (u,v) можна зберігати на перетині рядка u та стовпчика v.

Зображення графа 3 за допомогою матриці суміжності:

	1	2	3	4
1	Nul	nul	2	1
2	3	nul	nul	1
3	Nul	nul	nul	5
4	Nul	nul	nul	4

Для відсутніх ребер можна записати значення nul (іноді пишуть 0 або ∞).

Отже, для невеликих графів, коли місця в пам'яті вистачає, матриця суміжності досить зручна. Якщо не потрібно зберігати вагу, то її елементи є бітами та їх можна розміщувати по кілька в одному машинному слові, що дає помітну економію пам'яті. Якщо ж потрібно зберігати великий зважений граф, то краще застосовувати списки суміжних вершин.

4.6.2. ПОШУК ШЛЯХІВ У ГРАФАХ

Для пошуку тих чи інших шляхів у графах застосовуються різні алгоритми. Розглянемо лише деякі з них:

- пошук у глибину;
- пошук у ширину;
- алгоритм Дейкстри для пошуку найкоротшого шляху від даної вершини до інших.

Іноді дуже складно оцінити ефективність методу пошуку. Для нас найважливішими є два критерії: 1) як швидко при пошуку знайдено розв'язок (шлях у графі); 2) наскільки задовільним є знайдений розв'язок.

При розв'язанні деяких задач головним є те, щоб це було зроблено з мінімальними зусиллями, але в інших ситуаціях важливо отримати оптимальний результат.

Швидкість пошуку визначається як довжиною шляху розв'язку, так і кількістю вершин, через які фактично доводиться пройти в процесі пошуку шляху. Варто пам'ятати, що при поверненні з тупика зусилля виявляються витраченими даремно. Тому необхідно виробити таку стратегію пошуку, завдяки якій до мінімуму зводиться можливість тупикових ситуацій.

Необхідно розуміти різницю між гарним і оптимальним розв'язками. Щоб знайти оптимальний розв'язок, може знадобитися вичерпний пошук для підтвердження того, що розв'язок – найкращий. Для

гарного розв'язку достатньо знайти шлях, що задовольняє набір обмежень; при цьому неважливо, чи є ще кращий розв'язок.

Методи пошуку, описані далі, не в усіх ситуаціях працюють однаково ефективно. Тому важко сказати, чи завжди один метод кращий за інший. Проте в середньому деякі методи будуть прийнятнішими ніж інші. Іноді сама постановка задачі підказує кращий метод.

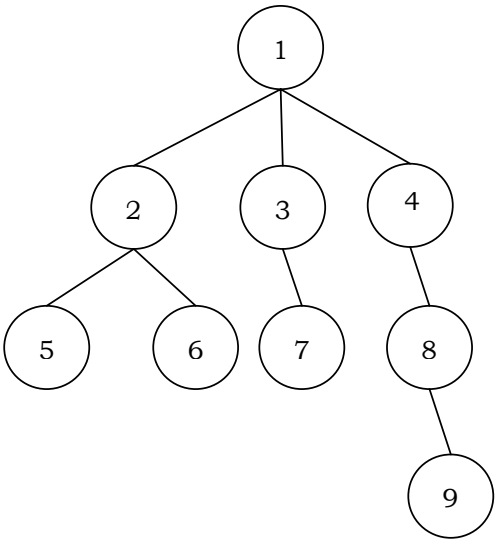
4.6.3. ПОШУК У ШИРИНУ

Є одним із базових алгоритмів пошуку. Нехай задано граф $G = (V, E)$ і фіксовану *початкову* вершину s. Алгоритм пошуку в ширину перераховує всі досяжні із s вершини в порядку зростання відстані від s. Відстанню вважається довжина найкоротшого шляху (кількість ребер чи сума вагових коефіцієнтів для зваженого графа). Під час пошуку шляху в графі виділяється *дерево пошуку в ширину* з коренем s. Воно містить усі досяжні із s вершини й тільки їх. Алгоритм може застосовуватись як для орієнтованих, так і для неорієнтованих графів.

Назва пояснюється тим, що в процесі пошуку ми йдемо в ширину, а не в глибину (спочатку переглядаємо сусідні вершини, потім їхніх сусідів і т. д.). Для наочності поділятимемо вершини графа на білі, сірі й чорні. Спочатку вони всі білі, а в процесі роботи алгоритму біла вершина може стати сірою, сіра – чорною, але не навпаки. Зустрівши нову вершину, алгоритм пошуку фарбує її в сірий чи чорний відповідно до правила: якщо $(u,v) \in E$ та u – чорна, то v – чорна або сіра. Різниця між сірими й чорними вершинами використовується алгоритмом для керування порядком обходу: сірі вершини – це ті, по яких ще потрібно рухатися, а всі сусіди чорних уже переглянуті. Отже, тільки сірі вершини можуть мати суміжні невиявлені вершини. Спочатку дерево пошуку складається тільки з початкової вершини s. При виявленні алгоритмом нової вершини v, суміжної з раніше знайденою вершиною u, вершина v разом із ребром (u,v) додається до дерева пошуку, стаючи *нащадком* вершини u, а u стає *батьком* v. Дерево пошуку в ширину зручно зберігати в черзі (див. вправу 16, б). Кожна вершина знаходиться тільки один раз, тому двох батьків у неї бути не може. Оцінюючи складність цього алгоритму, зауважимо, що вершини тільки темніють. Час обходу суміжних вершин до даної вершини становить $O(|E|)$, де $|E|$ – кількість ребер, а обчислювальна

складність алгоритму пошуку в ширину – $O(|V|+|E|)$, де $|V|$ – кількість вершин у графі.

Проілюструємо порядок обходу дерева в ширину, згідно з яким пронумеровано вершини:



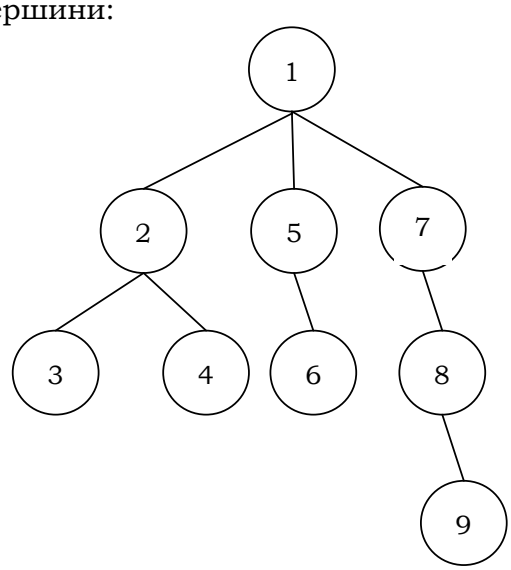
4.6.4. ПОШУК У ГЛИБИНУ

Стратегія пошуку в глибину: іти вглиб, поки є вихідні ребра, що не пройдені; повертатися й шукати інший шлях, коли таких ребер немає. Якщо після цього залишаються незнайдені вершини, то вибрати одну з них і повторювати процес доти, доки вони є.

Знайшовши вперше вершину v, суміжну з u, записуємо в поле $\pi[v]$ значення u. Утворюється дерево чи кілька дерев, якщо пошук відбувається з кількох вершин. Як і при пошуку в ширину, алгоритм пошуку в глибину використовує кольори вершин. Спочатку всі вершини – білі. При виявленні нової вершини вона фарбується в сірий колір, а після обробки – у чорний. Крім цього, пошук у глибину ставить на вершинах мітки часу. Кожна вершина має дві мітки: $d[v]$ показує, коли вершина виявлена, $f[v]$ – коли оброблена. Мітки використовуються в багатьох алгоритмах на графах. Під час виконання алгоритму обробка кожної вершини відбувається рівно один раз. Складність роботи алгоритму становить $O(|V|+|E|)$, де $|V|$ – кількість

вершин у графі, а $|E|$ – кількість ребер. Дерево пошуку в глибину зручно зберігати в стеку (див. вправу 16, а).

Проілюструємо порядок обходу дерева в глибину, згідно з яким пронумеровано вершини:



4.6.5. АЛГОРИТМ ДЕЙКСТРИ

Даний алгоритм запропонував Е. Дейкстра для знаходження найкоротшої відстані від заданої вершини до зваженого орієнтованого графа $G=(V,E)$ з початковою вершиною $s\in V$, в якому вага всіх ребер невід'ємна: $\forall(u,\ v)\in E:\ \omega(u,\ v)\geq 0$.

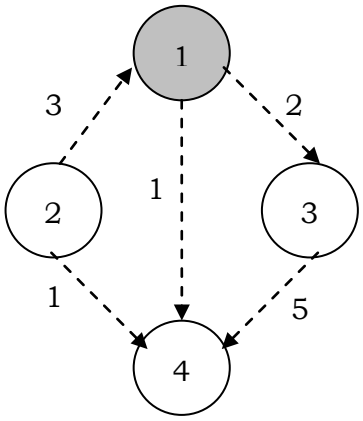
Кожній вершині з V зіставимо мітку – мінімальну відому відстань від вершини до s. На початку роботи алгоритму мітка вершини s ініціалізується значенням 0, мітки інших вершин – нескінченності. Це означає, що відстані від s до інших вершин поки невідомі. Усі вершини графа позначаються як невідвідані.

Якщо всі вершини переглянуті, то алгоритм завершується. Інакше зі ще не переглянутих вершин вибирається вершина u, яка має мінімальну мітку. Розглянемо всі можливі маршрути, в яких u є передостаннім пунктом, а для кожної сусідньої з u вершини – нову довжину шляху, що дорівнює сумі поточної мітки u й довжини ребра, яке з'єднує u із сусідом. Якщо отримана довжина менша за мітку сусіда, то замінимо останню на цю довжину. Розглянувши всіх сусідів, позна-

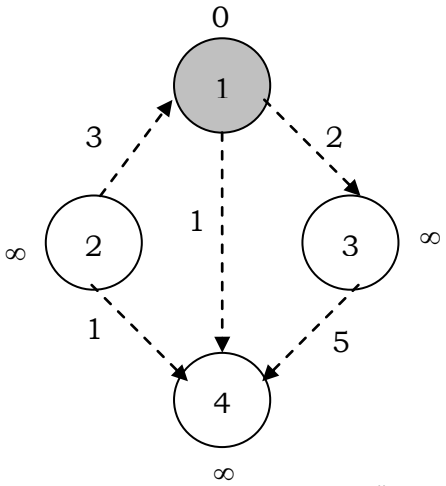
чимо вершину u як переглянуту й повторимо крок. Робота алгоритму завершується, коли всі вершини переглянуті.

Приклад 4.35. Алгоритм Дейкстри.

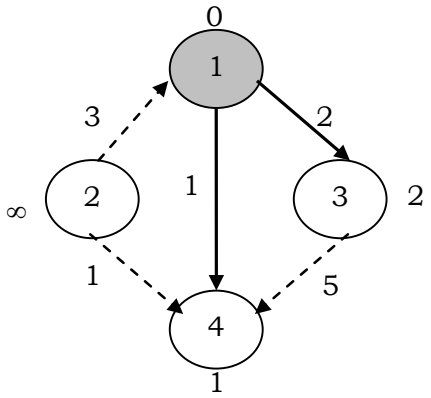
Нехай потрібно знайти відстань від вершини 1 до всіх інших для графа:



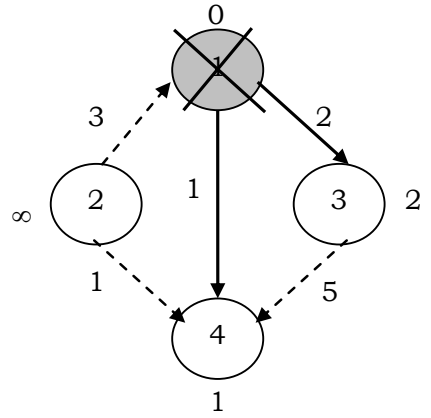
Початкова ініціалізація:



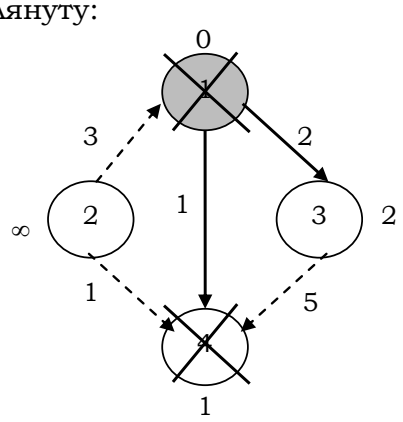
Крок 1. Мінімальну мітку має вершина 1. Її сусідами є вершини 3 та 4. Позначимо довжини шляхів до вершин через першу. Довжина шляху у вершини 3 чи 4 через вершину 1 дорівнює сумі найкоротшої відстані до вершини 1 і довжини ребра, що йде з 1 у 3 чи 4, відповідно, тобто 0 + 2 = 2 для вершини 3 (0 + 1 = 1 – для 4). Це менше поточних міток вершин 3, 4, тому нова мітка третьої вершини дорівнює 2, а четвертої – 1:



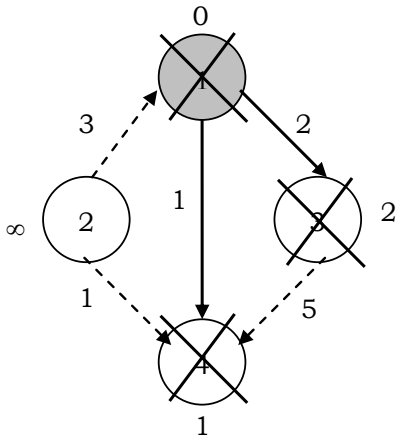
Усі сусіди вершини 1 перевірені. Поточна мінімальна відстань до вершини 1 вважається остаточною (те, що це дійсно так, уперше довів Дейкстра). Позначимо її як переглянуту:



Крок повторюється – беремо вершину з мінімальною міткою (4) і переглядаємо шляхи з неї. Таких шляхів у даному випадку немає. Викреслюємо її як переглянуту:



Знову беремо вершину з мінімальною міткою (3) і переглядаємо шляхи з неї. Є шлях у вершину 4, але ця вершина вже викреслена зі списку. Отже, з вершини 3 шляхів немає. Викреслюємо її як переглянуту:



На наступному кроці знову шукаємо вершину з мінімальною міткою (2), але її мітка ∞, тобто вона недосяжна з вершини 1, тому не може зменшити мітки інших вершин. Викреслюємо її з міткою ∞. Алгоритм завершений, оскільки всі вершини розглянуті ■

Складність алгоритму Дейкстри залежить від способів знаходження вершини v, збереження множини невідвіданих вершин і відновлення міток. Позначимо через n кількість вершин, а через m – ребер у графі G.

У найпростішому випадку, коли для пошуку вершини з мінімальним $d[v]$ проглядається вся множина вершин, а для збереження величин d – масив, час роботи алгоритму становить $O\left(|V|^2+|E|\right)$. Основний цикл виконується близько $|V|$ разів, кожного разу на знаходження мінімуму витрачається приблизно $|E|$ операцій, плюс кількість змін міток, що не перевищує кількості ребер у вихідному графі.

Для розріджених графів невідвідані вершини можна зберігати у двійковій купі, а як ключ використовувати значення $d[i]$. Швидкість роботи такої реалізації $O\left(|V|\log|V|+|E|\log|V|\right)$.

Якщо для зберігання невідвіданих вершин використати фібоначчієві купи, в яких час додавання елемента $O(1)$, то час роботи алгоритму становитиме $O\left(|V|\log|V|+|E|\right)$.

4.6.6. ЗАСТОСУВАННЯ АЛГОРИТМІВ ПОШУКУ

Повернемося до задачі, описаної на початку підрозділу. Нехай автобуси компанії АВС за розкладом здійснюють такі рейси:

Рейс	Відстань, км
Київ – Львів	544
Львів – Чернігів	701
Київ – Чернігів	151
Київ – Вінниця	266
Вінниця – Донецьк	812
Вінниця – Сімферополь	801
Вінниця – Львів	369
Чернігів – Житомир	297
Чернігів – Одеса	634
Одеса – Сімферополь	473
Донецьк – Сімферополь	571

Завдання полягає в тому, щоб написати кілька варіантів програм, які будуть вибирати маршрути поїздок і порівнювати їх.

Інформацію про рейси компанії АВС природно подати у вигляді зваженого орієнтованого графа, де вагою є відстань (рис. 4.7). Для спрощення зображення не будемо прописувати вагу на ребрах.

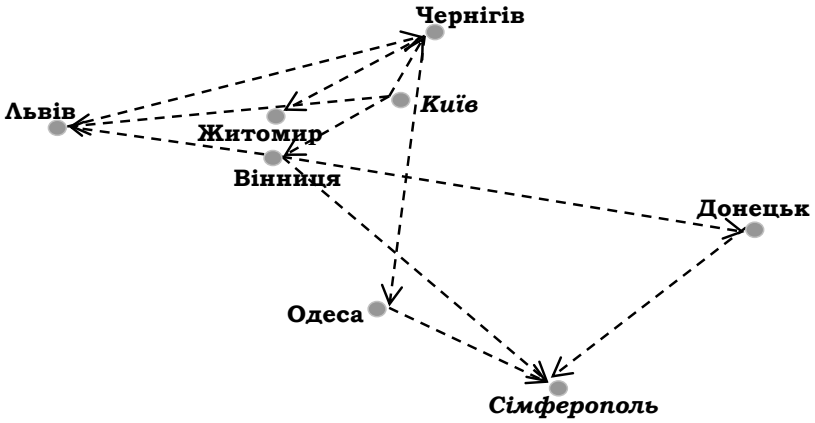
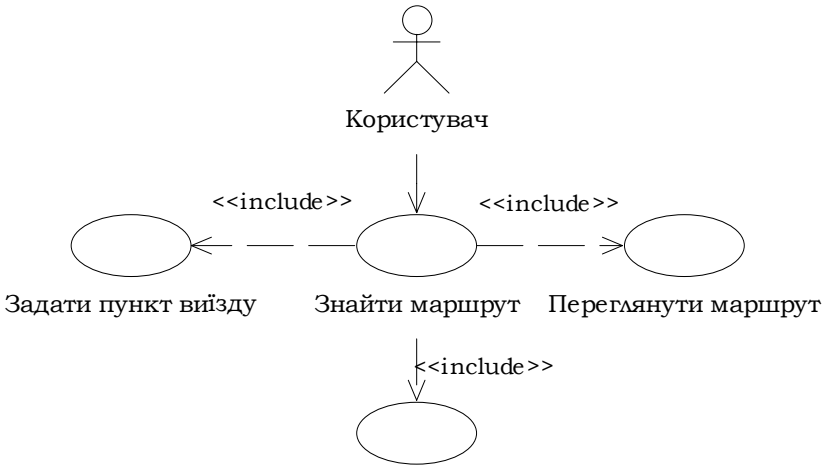


Рис. 4.7

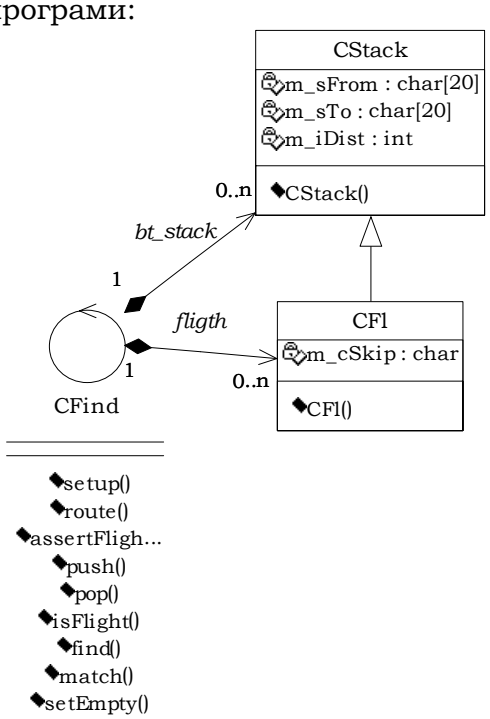
Для побудови програм застосуємо технологію UML-проектування, описану в підрозд. 2.7.3. Програма, призначена для вибору маршруту, має надавати користувачу можливість введення пунктів виїзду й прибуття, а також відображати знайдений маршрут.

Діаграма прецедентів для розроблюваної програми:

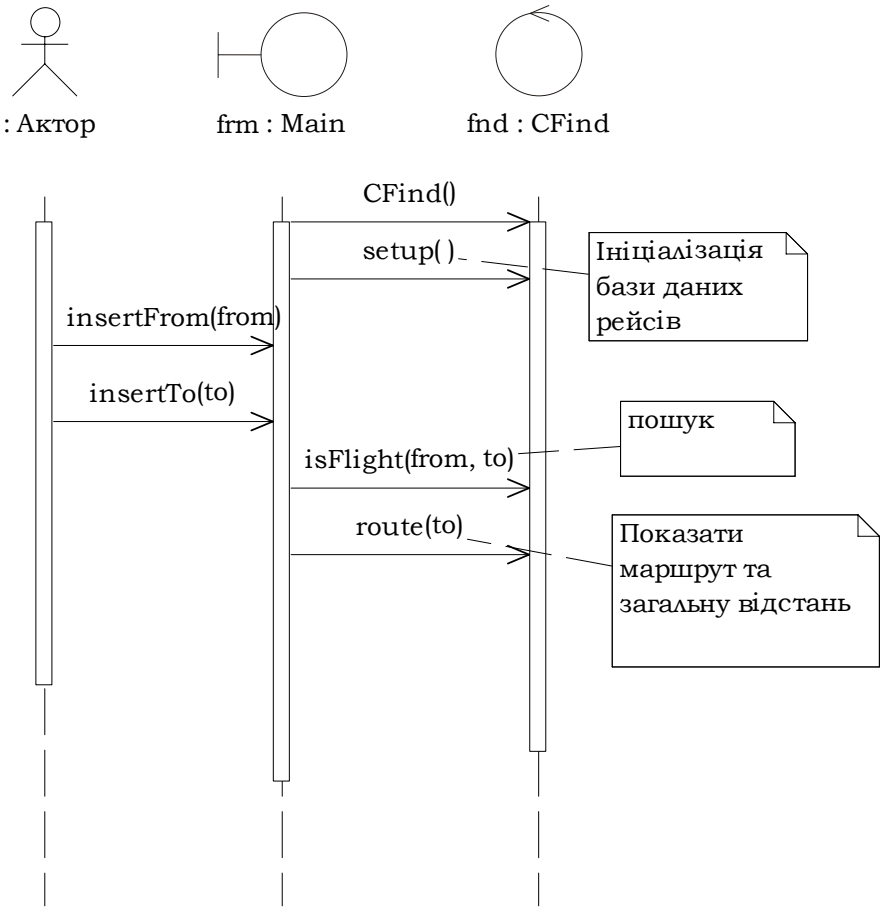


У програмі, призначеній для вибору маршруту, використовується база даних з інформацією про рейси компанії АВС. Кожен запис бази містить відомості про місто – пункт відправлення й місто – пункт прибуття, про відстань між ними, а також прапорець, що допомагає при пошуку з поверненням.

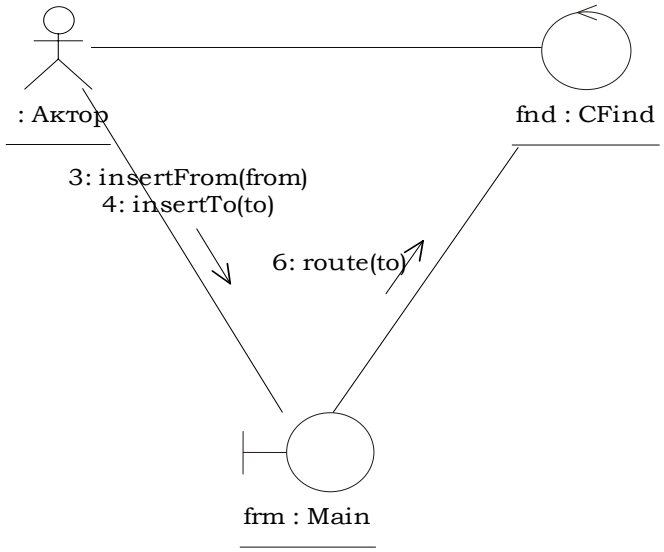
Діаграма класів програми:



Одночасно з розробкою діаграми класів розглянемо сценарії використання програми. Кожен сценарій реалізується сукупністю об'єктів (екземплярів класів), що взаємодіють між собою за допомогою відповідних методів. За різноманітними сценаріями взаємодії об'єктів класи наповнюються методами, що знаходить своє відображення на діаграмі класів. Розглянуті сценарії взаємодії об'єктів відображаються в діаграмах послідовності а) й кооперації б):

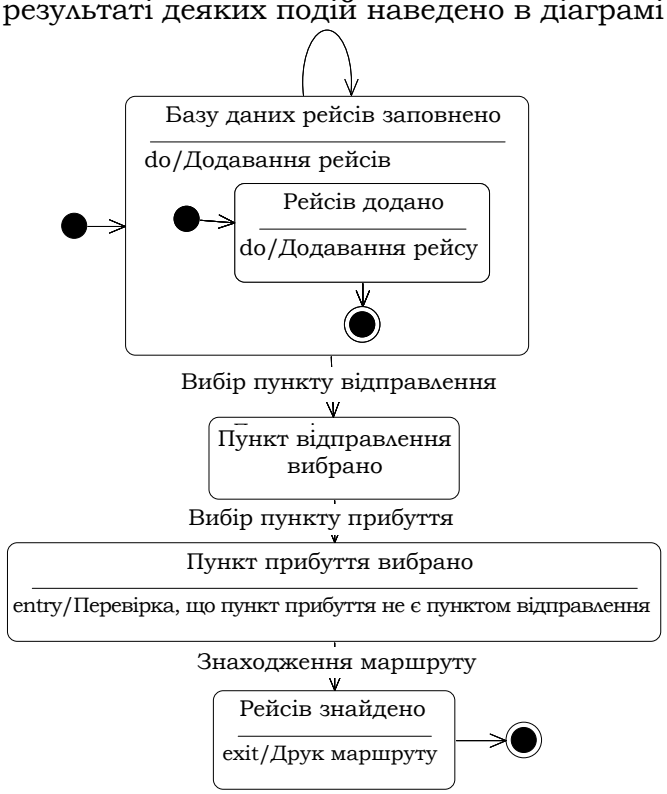


а)

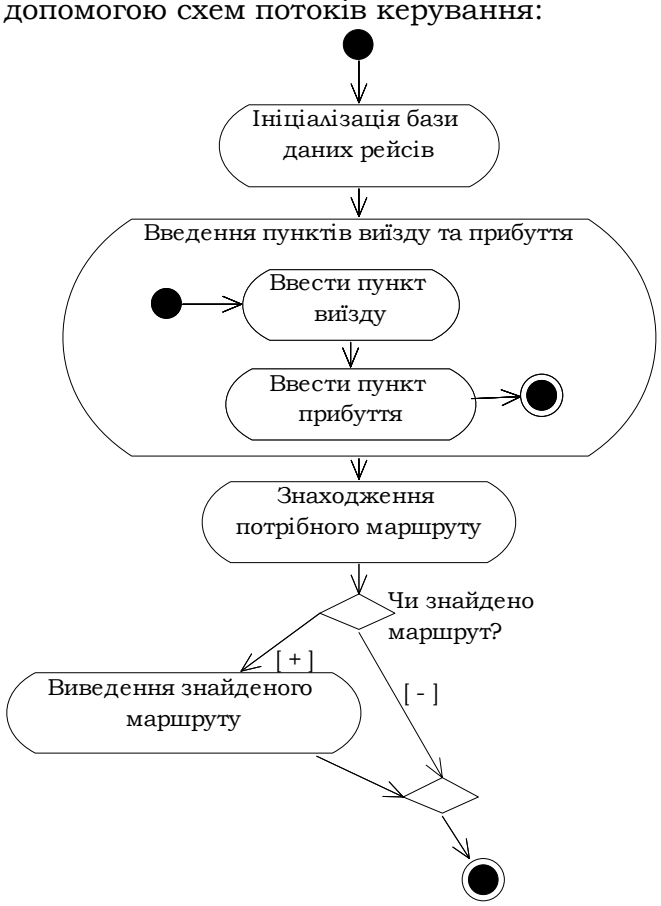


б)

Стани, в яких може перебувати конкретний об'єкт, і процес зміни його станів у результаті деяких подій наведено в діаграмі станів:



У нашій задачі необхідно проініціалізувати базу даних рейсів, ввести пункти виїзду та прибуття, знайти й надрукувати потрібний маршрут. Послідовність дій у програмі наведена на діаграмі діяльності за допомогою схем потоків керування:



У процесі проектування програм визначають, які класи, атрибути, методи та їхні параметри будуть використані для розв'язання задачі. Діаграми показують послідовність взаємодії об'єктів. Тепер можна розпочинати написання тексту програми. Надалі побудовані діаграми прецедентів, станів і діяльності можуть бути застосовані для її тестування.

Застосуємо спочатку алгоритм пошуку в глибину для знаходження маршруту.

Приклад 4.36. Пошук у глибину.

Лістинг.

```
#include <iostream>
```

```
#include <vector>
#include <stdio.h>
#include <string.h>

class CStack {
public:
    CStack(char *from, char *to, int dist)
    {
          strcpy(m_sFrom, from);
          strcpy(m_sTo, to);
          m_iDist=dist;
    };
  char m_sFrom[20];
  char m_sTo[20];
  int m_iDist;
};
/*база даних рейсів*/
class CFl:public CStack {
public:
    CFl(char *from, char *to, int dist):CStack(from, to, dist){
          m_cSkip=0;
    };

  char m_cSkip; /*використовується при пошуку з поверненням*/

};

class CFind {
private:
vector<CFl*>flight;/*вектор рейсів*/

/*стек, що використовується для пошуку
    з поверненням*/
vector<CStack*>bt_stack;
public:
/*методи, що використовуються при пошуку*/
void setup(void);
void route(char *to);
void assertFlight(char *from, char *to, int dist);
void push(char *from, char *to, int dist);
void pop(char *from, char *to, int *dist);
void isFlight(char *from, char *to);
int find(char *from, char *anywhere);
int match(char *from, char *to);
void setEmpty();
};
```

Окремі записи вводяться в базу даних за допомогою методу **assert_flight()**, а всю інформацію про рейси ініціалізує метод **setup()**. Код потрібних методів:

```
/*Ініціалізація бази даних рейсів*/
void CFind::setup(void)
{
  assertFlight("Kyev", "Lviv", 544);
  assertFlight("Lviv", " Tchernigiv", 701);
  assertFlight("Kyev", "Tchernigiv", 151);
  assertFlight("Kyev", "Vinnitsa", 266);
  assertFlight("Vinnitsa", "Donetsk", 812);
  assertFlight("Vinnitsa", "Simferopol", 801);
  assertFlight("Vinnitsa", "Lviv", 369);
  assertFlight("Tchernigiv", "Gitomir", 297);
  assertFlight("Tchernigiv", "Odessa", 634);
  assertFlight("Odessa", "Simferopol", 473);
  assertFlight("Donetsk", "Simferopol", 571);
}

/*Запам'ятати дані в базі*/
void CFind::assertFlight(char *from, char *to, int dist)
{
    CFl *fl=new CFl(from, to, dist);
    flight.push_back(fl);
}
```

Для написання коду, що виконує пошук маршруту з Києва в Сімферополь, потрібні кілька допоміжних методів. По-перше, метод для визначення наявності рейсу між двома містами. Цей метод назвемо **match()**; він повертає відстань між двома містами, якщо такий рейс є, і нуль, якщо такого рейсу немає:

```
/*Якщо між двома містами є рейс, то повертається
   відстань між ними, інакше повертається 0.*/
int CFind::match(char *from, char *to)
{
  for          (vector<CFl*>::iterator          t=flight.end()-1;
t!=flight.begin(); t--)
      if(!strcmp((*t)->m_sFrom, from) &&
            !strcmp((*t)->m_sTo, to)) return (*t)->m_iDist;
  return 0;  /*не знайдено*/
}
```

Іншим необхідним методом є **find()**. Він шукає в базі даних будь-який рейс із зазначеного міста. Якщо рейс з яким-небудь іншим міс-

том знайдений, то повертаються назва цього міста й відстань до нього від першого міста, інакше – нуль. Текст методу **find()**:

```
  /*Знаючи пункт відправлення (параметр from), знайти пункт при-
буття (параметр anywhere)*/
  int CFind::find(char *from, char *anywhere)
  {
      vector<CFl*>::iterator find_pos=flight.begin();
      while(find_pos<flight.end()) {
       if(!strcmp((*find_pos)->m_sFrom, from) &&
         !(*find_pos)->m_cSkip) {
           strcpy(anywhere, (*find_pos)->m_sTo);
           (*find_pos)->m_cSkip=1;//активізувати
           return (*find_pos)->m_iDist;
       }
       find_pos++;
     }
     return 0;
  }
```

Якщо поле **m_cSkip** (пропустити) має значення 1, то рейс між містами не обирається. Коли рейс знайдено, то його поле **m_cSkip** позначається як активне – таким чином реалізується пошук із поверненням з тупиків ■

Пошук із поверненням (*бектрекінг*) – важлива частина багатьох методів пошуку. У загальному вигляді він реалізується за допомогою рекурсивних підпрограм і спеціального стеку. При проходженні графа всі вершини, що зустрічаються, поміщаються у стек. При досягненні листка відбувається зупинка без можливості повторного спуску (у глибину), зі стеку забирається остання розташована там вершина й досліджується новий шлях, що починається з неї. Цей процес продовжується, поки не буде знайдена мета чи досліджені всі шляхи. Методи **push()** та **pop()** (означають відповідно "помістити у стек" і "виштовхнути зі стеку"), що керують стеком, використовуються для пошуку з поверненням. Значення, що поміщаються у стек, зберігаються в контейнері **bt_stack**:

```
  /*Підпрограми звернення до стеку*/
  void CFind::push(char *from, char *to, int dist)
  {
      CStack *st=new CStack(from, to, dist);
      bt_stack.push_back(st);
  }

  void CFind::pop(char *from, char *to, int *dist)
  {
```

```
    vector<CStack*>::iterator tos=bt_stack.end()-1;
    if(!bt_stack.empty()) {
     strcpy(from, (*tos)->m_sFrom);
     strcpy(to, (*tos)->m_sTo);
     *dist=(*tos)->m_iDist;
    bt_stack.pop_back();
  }
  else cout<<"EMPTY Стек порожній."<<endl;
}
```

Метод **isFlight()** – головна підпрограма для пошуку маршруту:

```
//пошук у глибину
void CFind::isFlight(char *from, char *to)
{
  int d, dist;
  char anywhere[20];

  //подивитися, чи є місто пунктом прибуття
  if(d=match(from, to)) {
    push(from, to, d);
    return;
  }
  //перевірити інший рейс
  if(dist=find(from, anywhere)) {
    push(from, to, dist);
    isFlight(anywhere, to);
  }
  else {
    //пошук із поверненням
    pop(from, to, &dist);
    isFlight(from, to);
  }
}
```

Метод **match()** перевіряє базу даних на наявність рейсу між містами, на які вказують параметри **m_sFrom** та **m_sTo**. Якщо такий рейс є, то мета пошуку досягнута. Тоді дані про рейси поміщають у стек і метод повертає керування. Інакше метод **find()** перевіряє, чи є рейс між містом, на яке вказує **from**, і будь-яким іншим містом. Якщо є, то відповідні дані поміщаються у стек і рекурсивно викликається **isFlight()**. Інакше виконується пошук із поверненням. Попередня вершина видаляється зі стеку, і рекурсивно викликається **isFlight()**. Цей процес повторюється доти, доки не буде знайдена ціль. Поле **m_cSkip** використовується при пошуку з поверненням, щоб не перевірялися повторно ті самі рейси:

```
  /*Показати маршрут і загальну відстань*/
  void CFind::route(char *to)
  {
    int dist;
    vector<CStack*>::iterator t=bt_stack.begin();
    dist=0;
    while(t<bt_stack.end()) {
        cout<<(*t)->m_sFrom<<"to";
      dist+=(*t)->m_iDist;
      t++;
    }
    cout<<to<<endl;
    cout<<"Відстань у кілометрах:"<<dist<<endl;
  }

  /*Якщо між двома містами є рейс, то повертається
     відстань між ними, інакше повертається 0*/
  int CFind::match(char *from, char *to)
  {
    for          (vector<CFl*>::iterator          t=flight.end()-1;
t!=flight.begin(); t--)
        if(!strcmp((*t)->m_sFrom, from) &&
              !strcmp((*t)->m_sTo, to)) return (*t)->m_iDist;
    return 0;  /*не знайдено*/
  }
```

Пошук у глибину маршруту від Києва до Сімферополя (повний текст програми):

Лістинг.

```
  /*Пошук у глибину*/
  #include <iostream>
  #include <vector>
  #include <stdio.h>
  #include <string.h>

  using namespace std;

  class CStack {
  public:
      CStack(char *from, char *to, int dist)
      {
            strcpy(m_sFrom, from);
            strcpy(m_sTo, to);
            m_iDist=dist;
      };
```

```
  char m_sFrom[20];
  char m_sTo[20];
  int m_iDist;
};

/*структура бази даних рейсів*/
class CFl:public CStack {
public:
    CFl(char *from, char *to, int dist):CStack(from, to, dist){
          m_cSkip=0;
    };

  char m_cSkip; /*використовується при пошуку з поверненням*/

};

class CFind {
private:
vector<CFl*>flight;/*вектор структур БД*/

/*стек, що використовується для пошуку
    з поверненням*/
vector<CStack*>bt_stack;
public:
void setup(void);
void route(char *to);
void assertFlight(char *from, char *to, int dist);
void push(char *from, char *to, int dist);
void pop(char *from, char *to, int *dist);
void isFlight(char *from, char *to);
int find(char *from, char *anywhere);
int match(char *from, char *to);
void setEmpty();
};

/*Ініціалізація бази даних рейсів*/
void CFind::setup(void)
{
  assertFlight("Kyev", "Lviv", 544);
  assertFlight("Lviv", " Tchernigiv", 701);
  assertFlight("Kyev", "Tchernigiv", 151);
  assertFlight("Kyev", "Vinnitsa", 266);
  assertFlight("Vinnitsa", "Donetsk", 812);
  assertFlight("Vinnitsa", "Simferopol", 801);
  assertFlight("Vinnitsa", "Lviv", 369);
  assertFlight("Tchernigiv", "Gitomir", 297);
  assertFlight("Tchernigiv", "Odessa", 634);
```

```
    assertFlight("Odessa", "Simferopol", 473);
    assertFlight("Donetsk", "Simferopol", 571);
  }

  /*Запам'ятати дані в базі*/
  void CFind::assertFlight(char *from, char *to, int dist)
  {
      CFl *fl=new CFl(from, to, dist);
      flight.push_back(fl);
  }

  /*Показати маршрут і загальну відстань*/
  void CFind::route(char *to)
  {
    int dist;
    vector<CStack*>::iterator t=bt_stack.begin();
    dist=0;
    while(t<bt_stack.end()) {
        cout<<(*t)->m_sFrom<<"to";
      dist+=(*t)->m_iDist;
      t++;
    }
    cout<<to<<endl;
    cout<<"Відстань у кілометрах:"<<dist<<endl;
  }

  /*Якщо між двома містами є рейс, то повертається
     відстань між ними, інакше повертається 0*/
  int CFind::match(char *from, char *to)
  {
    for (vector<CFl*>::iterator t=flight.end()-1; t!=flight.begin();
                                                          t--)
        if(!strcmp((*t)->m_sFrom, from) &&
              !strcmp((*t)->m_sTo, to)) return (*t)->m_iDist;
    return 0;  /*не знайдено*/
  }

  /*Знаючи пункт відправлення (параметр from), знайти пункт при-
буття (параметр anywhere)*/
  int CFind::find(char *from, char *anywhere)
  {
      vector<CFl*>::iterator find_pos=flight.begin();
      while(find_pos<flight.end()) {
       if(!strcmp((*find_pos)->m_sFrom, from) &&
         !(*find_pos)->m_cSkip) {
           strcpy(anywhere, (*find_pos)->m_sTo);
           (*find_pos)->m_cSkip=1;//активізувати
```

```
        return (*find_pos)->m_iDist;
      }
      find_pos++;
    }
    return 0;
}

//пошук у глибину
void CFind::isFlight(char *from, char *to)
{
  int d, dist;
  char anywhere[20];

  //подивитися, чи є місто пунктом прибуття
  if(d=match(from, to)) {
    push(from, to, d);
    return;
  }
  //перевірити інший рейс
  if(dist=find(from, anywhere)) {
    push(from, to, dist);
    isFlight(anywhere, to);
  }
  else {
    //пошук із поверненням
    pop(from, to, &dist);
    isFlight(from, to);
  }
}

/*Підпрограми звернення до стеку*/
void CFind::push(char *from, char *to, int dist)
{
    CStack *st=new CStack(from, to, dist);
    bt_stack.push_back(st);
}

void CFind::pop(char *from, char *to, int *dist)
{
    vector<CStack*>::iterator tos=bt_stack.end()-1;
    if(!bt_stack.empty()) {
     strcpy(from, (*tos)->m_sFrom);
     strcpy(to, (*tos)->m_sTo);
     *dist=(*tos)->m_iDist;
    bt_stack.pop_back();
  }
  else cout<<"EMPTY Стек порожній."<<endl;
}
```

Основна програма:

```
int main()
{
  char from[20], to[20];
  CFind *fnd=new CFind();
  fnd->setup();

  cout<<"From Пункт відправлення:";
  cin>>from;
  cout<<"To Пункт прибуття:";
  cin>>to;

  fnd->isFlight(from, to);
  fnd->route(to);
  return 0;
}
```

Зверніть увагу, що **main()** просить ввести пункти відправлення та прибуття. Це означає, що програму можна використовувати для визначення маршрутів між будь-якими двома містами.

Якщо обрано Київ і Сімферополь, то розв'язком буде: Київ – Чернігів – Одеса – Сімферополь (див. рис. 4.7). Відстань – 1258 км.

Пошуком у глибину знайдено один із розв'язків. Він не оптимальний (оптимальний у даному випадку – Київ – Вінниця – Сімферополь із відстанню 1067 км), але поганим його теж назвати не можна ■

У цьому випадку було з першої спроби знайдено маршрут без повернення – це перевага методу. Недоліком є те, що для відшукання оптимального розв'язку при пошуку в глибину доводиться пройти майже всі вершини.

Тепер звернемося до алгоритму пошуку в ширину для знаходження маршруту.

Приклад 4.37. Пошук у ширину.

Щоб програма виконувала пошук у ширину, необхідно змінити лише підпрограму **isFlight()**:

Лістинг.

```
  /*Визначити, чи є маршрут із міста from до міста to, пошук у
ширину*/
  void void CFind::isFlight(char *from, char *to)
  {
    int d, dist;
    char anywhere[20];

    while(dist=find(from, anywhere)) {
      //модифікація: пошук у ширину
```

```
      if(d=match(anywhere, to)) {
        push(from, to, dist);
        push(anywhere, to, d);
        return;
      }
    }
    //перевірити довільний рейс
    if(dist=find(from, anywhere)) {
      push(from, to, dist);
      isFlight(anywhere, to);
    }
    else {
      pop(from, to, &dist);
      isFlight(from, to);
    }
  }
```

Якщо обрано Київ і Сімферополь, то розв'язком буде: Київ – Вінниця – Сімферополь. Відстань – 1067 км.

Пошуком у ширину знайдено оптимальний розв'язок, але завдяки не методу, а способу організації даних. У цілому всі зауваження щодо швидкості роботи алгоритму пошуку в глибину діють і тут ■

Застосуємо для пошуку маршруту алгоритм Дейкстри.

Щоб знайти найкоротший шлях, перепишемо метод `find()`.

Приклад 4.38. Алгоритм Дейкстри.

Лістинг.

```
int CFind::find(char *from, char *anywhere)
{
  int dist=32000;//більше довжини найдовшого рейсу

  vector<CFl*>::iterator find_pos=flight.begin();
  vector<CFl*>::iterator pos=flight.begin();

  while(find_pos<flight.end()) {
    if(!strcmp((*find_pos)->m_sFrom, from) &&
      !(*find_pos)->m_cSkip) {
        if((*find_pos)->m_iDist<dist) {
        pos=find_pos;
        dist=(*find_pos)->m_iDist;
      }
    }
    find_pos++;
  }
  if(pos<flight.end()) {
    strcpy(anywhere, (*pos)->m_sTo);
```

```
      (*pos)->m_cSkip=1;
      return (*pos)->m_iDist;
    }
    return 0;
}
```

За допомогою цієї версії **find()** для маршруту Київ – Сімферополь отримуємо розв'язок: Київ – Вінниця – Сімферополь. Відстань – 1067 км. У даному випадку цей метод пошуку дозволяє знайти найкоротший маршрут ■

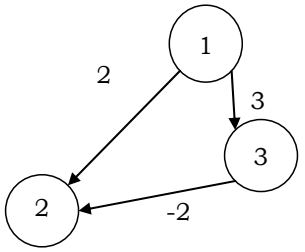
4.6.7. ЖАДІБНІ АЛГОРИТМИ

Жадібний алгоритм – це алгоритм, який на кожному окремому кроці обирає локально-оптимальний варіант.

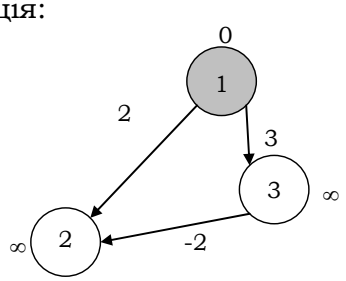
Жадібним є алгоритм Дейкстри для пошуку найкоротшого шляху, оскільки він завжди обирає вершину, найближчу до вихідної, серед тих, найкоротший шлях до яких ще невідомий.

Слід зазначити, що не кожен жадібний алгоритм дозволяє отримати оптимальний результат у цілому, проте для багатьох задач такі алгоритми дійсно дають оптимальний розв'язок. Наприклад, алгоритм Дейкстри за умови відсутності ребер із від'ємною вагою дає оптимальний розв'язок, але якщо припустити наявність таких ребер, то цей алгоритм у деяких випадках не дозволить знайти найкоротший шлях.

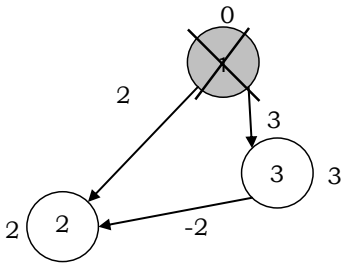
Застосуємо алгоритм Дейкстри для пошуку найкоротшого шляху з вершини 1 через вершину 2 до наступного графа, що містить ребро з від'ємною вагою:



Початкова ініціалізація:



Крок 1. Мінімальну мітку має вершина 1. Її сусідами є вершини 2 та 3. Позначимо довжини шляхів у ці вершини через першу. Довжина шляху у вершину 2 становить 2, у вершину 3 – 3. Це менше поточних міток вершин 2, 3, тому призначимо їм нові мітки. Усі сусіди вершини 1 перевірені – позначимо її як переглянуту:



Ми знайшли шлях із вершини 1 у 2. Згідно з алгоритмом Дейкстри пошук припиняється. Однак, оскільки існує шлях із вершини 3 у 2 довжиною –2, то оптимальним є незнайдений шлях 1–3–2 довжиною 1.

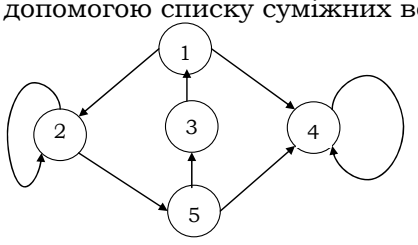
Якщо задача така, що єдиним способом знаходження оптимального розв'язку є повний перебір, то застосування жадібного алгоритму чи іншого евристичного методу для знаходження хорошого розв'язку може виявитися єдиною реальною можливістю досягнення результату.

У комбінаториці є розділ, пов'язаний із жадібними алгоритмами – теорія матроїдів. За допомогою цієї теорії можна розв'язувати задачі про застосовність конкретних жадібних алгоритмів.

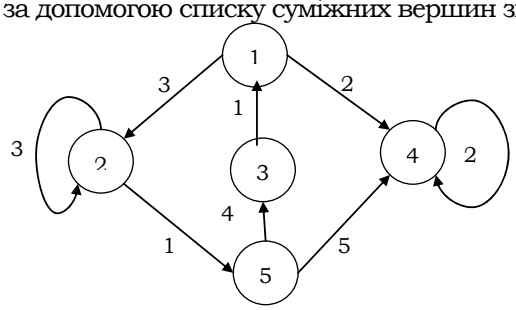
***Література для СР:** перебірні задачі – [8, 65, 95, 100, 125]; теорія матроїдів – [8, 65]; бектрекінг – [8, 65].

Контрольні запитання та вправи

1. Що таке комбінаторний вибух?
2. Що таке список суміжних вершин?
3. Як виглядає зображення графа у вигляді матриці суміжності?
4. Що таке пошук у глибину?
5. Що таке пошук у ширину?
6. Що таке алгоритм Дейкстри?
7. Дати визначення жадібного алгоритму.
8. Зобразити за допомогою списку суміжних вершин граф:



9. Зобразити за допомогою матриці суміжності граф із вправи 8.
10. Продемонструвати роботу алгоритму пошуку в глибину на заданому графі із вправи 8.
11. Продемонструвати роботу алгоритму пошуку в ширину на заданому графі із вправи 8.
12. Зобразити за допомогою списку суміжних вершин зважений граф:



13. Зобразити за допомогою матриці суміжності зважений граф із вправи 12.
14. Продемонструвати роботу алгоритму Дейкстри на заданому графі із вправи 12.
15. Чи для всіх можливих перестановок рядків розкладу рейсів компанії XYZ результат роботи запропонованих програм залишиться незмінним? Чому?
16. Елімінувати рекурсію в методі **`isFlight()`** для пошуку: а) у глибину (прикл. 4.36); б) у ширину (прикл. 4.37). Для подання дерева пошуку в глибину (ширину) використати стек (чергу) на базі масиву [100, 125].
17. Відредагувати запропоновані програми пошуку маршруту різними методами так, щоб рейси, пункт відправлення й прибуття вводилися з клавіатури.
18. Лабіринт може бути заданий матрицею суміжності, в якій для кожної пари кімнат указано, чи з'єднані вони коридором. Дано матрицю суміжності лабіринту з n кімнат і номери двох кімнат. Знайти шлях, що сполучає дані кімнати.
19. Перевірити зв'язність графа.
20. З'ясувати, чи є граф циклічним.
21. Побудувати транзитивно-рефлексивне замикання графа.
22. Побудувати матрицю відстаней між вершинами графа.
23. Для даного скінченного автомата реалізувати алгоритм пошуку: а) множини недосяжних станів; б) тупикових станів; в)* множин еквівалентних станів; г) відповіді на запитання, чи допускає скінченний автомат задане слово. Автомат задається з: 1) клавіатури; 2) файла.
24. Для даного скінченного недетермінованого автомата реалізувати алгоритм його детермінізації.

ВІДПОВІДІ ТА ВКАЗІВКИ

РОЗДІЛ 1

1.1. **8.** а) 8; б) 14. **21.** $(\forall y(\exists z(\forall x(P_1(x) \supset (P_2(y) \& (\neg P_1(z)))))))$. **23.** Нехай $n \mid m$ означає подільність n на m без залишку. а) $\forall x \forall y \exists z(x \mid z \& y \mid z \supset \forall d(x \mid d \& y \mid d \supset z \geq d))$. **26.** Перейти від формул ПЧП до пропозиційних форм шляхом заміни предикатних підтермів на пропозиційні змінні й використати формули із вправи 20 і закони булевої алгебри (див. вправу 24, підрозд. 1.2).

1.2. **6.** а) $\bar{A} \cup B$; б) C; в) $A \cap B \cap X$. **22.** Скористатися математичною індукцією по n.

1.3. **10.** Урахувати співвідношення $\max(x,y) = (x + y + | x - y |)/2$. Знайти залежність між функціями $\mathrm{sign}(x)$ та $|x|$. **11.** Написати спочатку допоміжні предикати: 1) Біла(x,y), який є істинним, якщо клітинка (x,y) біла (якщо горизонталь парна, то білими в ній є клітинки непарних стовпчиків, якщо непарна – то навпаки); 2) Гор. Вер. (x,y,u,v), який є істинним, якщо клітинки (x,y) та (u,v) розташовані на одній горизонталі або вертикалі; 3) Діагональ (x,y,u,v), який є істинним, якщо клітинки (x,y) та (u,v) розташовані на одній діагоналі. **12–13.** Застосувати композицію ітерації. **14.** б) Застосувати метод від супротивного. **27.** Застосувати алгоритм Евкліда через віднімання. **31.** У цьому прикладі функція керування не залежить від стану. а) $\delta(0) = x2 \leftarrow a * a$, $\delta(1) = x4 \leftarrow x2 * x2$, $\delta(2) = x8 \leftarrow x4 * x4$, $\delta(3) = x10 \leftarrow x8 * x2$. **32.** Див. [42].

1.4. **39.** Застосувати індукцію за кількістю станів автоматата. **40.** а) Найменші розв'язки є регулярними відносно коефіцієнтів a та b. Скористатися теоремою 1.1 про нерухому точку. (Див. [52]). **45.** Застосувати метод від супротивного. Припустити, що існує скінченний автомат, який допускає цю мову, і отримати протиріччя. **58.** Застосувати метод від супротивного.

РОЗДІЛ 2

2.1. 27. Варіант Ейлера для обходу конем:

20	45	26	39	18	43	24	33
27	38	19	44	25	32	17	42
46	21	36	29	40	23	34	31
37	28	47	22	35	30	41	16
48	9	62	3	54	15	60	5
63	2	55	8	61	4	53	14
10	49	64	57	12	51	6	59
1	56	11	50	7	58	13	52

28. а) 1. Kh5+ T:h5 2. Tg6+ Kp:g6 3.Te6×; б) 1. Tc6! Kp b7 2. Kd6+ Kp b8 3.Tc8×; в) 1. e8C! Kp:f6 2. g8T Kp e6 3. Tg6×; г) 1. Cf6! gf 2. 2. Kp f8 f5 3.Kf7×.

2.3. 16. $15=(1111)_2$, $255=(11111111)_2$, $256=(100000000)_2$, $32767=(111111111111111)_2$. **17.** $(1111)_{10}=15$, $(1111)_{16}=F$, $(11111111)_{10}=255$, $(1111110)_{16}=FE$. **18.** $(FF2)_2=111111110010$, $(ABCD)_2=1010101111001101$, $(FF2)_{10}=4082$. **19.** а) 11100111; б) 10011010. **20.** а) 36; б) 47989. **21.** а) 0.11111111111. **43, 45.** Алгоритм аналогічний алгоритму швидкого піднесення до степеня (див. прикл. 2.31).

2.5. 15. а) Див. побудову послідовності всіх десяткових цифр натурального числа у прикл. 2.31. **17.** б) $p_1=\sqrt{2}$, $p_i=\sqrt{2+p_{i-1}}$, фільтр: $i<n$. **20.** а) Покласти $a_i=(-1)^{i(i+1)/2}/2$ і знайти частку a_i/a_{i-1}; б) побудувати графік функції $a(k)=[\sqrt{k}], k\in N$ і звернути увагу на точки, в яких відбуваються стрибки значення функції; в) те саме, що й у б), тільки для функції $c(k)$; г)-д) див. прикл. 2.29. **17.** Див. прикл. 2.31. Узяти до уваги, що алгебра $(R,+)$ є півгрупою.

2.6. **13.** 1) (**Б**) $\varepsilon \in L_2$ (**І**) для $u \in L_2$: $aub \in L_2$; 2) (**Б**) $\varepsilon \in D_{2n}$ (**І**) для $u,v \in D_{2n}$, $i = \overline{1,n}$: $a_i u b_i \in D_{2n}$, $uv \in D_{2n}$. **16.** а) (**Б**) $\mathrm{ne}(\varepsilon,\varepsilon) = 0$, $\mathrm{ne}(\varepsilon,v) = 1, v \in \Sigma^+$ (**І**) для $a,b \in \Sigma$: $\mathrm{ne}(\mathrm{pre}(a,u),\ \mathrm{pre}(b,v) = (a \neq b \to 1 \mid \mathrm{ne}(u,v))$; д) розв'язок дає система індуктивних функцій $\mathrm{prefix}(u,n)$, $\mathrm{tail}(u,n)$ та $\mathrm{tail3}(u,i,n)$ таких, що $\mathrm{delsym}(u,k) = \mathrm{prefix}(u,k-1) \circ \mathrm{tail}(u,k+1)$, $\mathrm{tail}(u,n) = \mathrm{tail3}(u,1,n)$ (**БФС**) $\mathrm{prefix}(u,0) = \mathrm{prefix}(\varepsilon,n) = \varepsilon$, $\mathrm{tail3}(\varepsilon,i,n) = \varepsilon$ (**ІФС**) $\mathrm{prefix}(au,n) = a(\mathrm{prefix}(u,n-1))$, $\mathrm{tail3}(au,i,n) = (i \le n \to \mathrm{tail3}(u,i+1,n) \mid u)$. **17.** а) (**Б**) $\mathrm{count}(0) = 0$ (**І**) для $n > 0$ $\mathrm{count}(n) = \mathrm{count}(n/10) + 1$; в)-г) див. прикл. 2.43. **19.** (**Б**) $f(\varnothing) = 0, f(a(\varnothing,\varnothing)) = 1$, (**І**) для кореня a та піддерев d_1, d_2, з яких принаймні одне непорожнє: $f(a(d_1,d_2)) = f(d_1) + f(d_2)$. **20.** (**Б**) $f(\varnothing) = 0$, (**І**) для кореня a та піддерев d_1, d_2: $f(a(d_1,d_2)) = \max\{f(d_1), f(d_2)\} + 1$.

РОЗДІЛ 3

3.1. **4.** а) **<string >,<FFF>,<Null>**; б) **<"FDEL?>, <Null>**; з) **<x>,<==>,<(>, <a>, <++>, <+>,<b>,<*>,<2>,<)>**. **6.** 1) **< >,<*>,<*>,<*>,< >,</>,<*>**.

30. а) пари чисел 1,6; 2,19 та 3,492.

3.2. **20.** 1) 2057; 3) 8; 7) 13. **26.** 1) так. **31.** б) $a\%10$ – залишок від ділення a на 10. **32.** а) $(a == b)\,\&\&(a == c)\,\&\&(c == b)$. **36.** а) $swap(x,z)$, $swap(y,z)$, де $swap(u,v)$ обмінює значеннями змінні u та v. За умовою змінні x,y,z мають отримати значення відповідно змінних z,x,y. **41.** а) Скористатися функціями із вправи 40; є) слону на полі xy доступні всі поля, що розташовані на двох діагоналях (чотирьох променях), які проходять через поле. Перший промінь – уверх і праворуч із полями $xy,(x+1)(y+1),(x+2)(y+2),\ldots$, другий – уверх і ліворуч із полями $xy,(x-1)(y+1),(x-2)(y+2),\ldots$, третій – униз і ліворуч із полями $xy,(x-1)(y-1),(x-2)(y-2),\ldots$, четвертий – униз і праворуч із полями $xy,(x+1)(y-1),(x+2)(y-2),\ldots$, де $x+i(x-i)$ – i-та вертикаль за (перед) вертикаллю x. У функції врахувати, що координати полів не можуть виходити за межі діапазону $a..h \times 1..8$.

3.3. **16.** а) Буде надруковано: `while count i` у стовпчик: `1 2 3 4 5 6 7 8 9 10`. **18.** Рекурентні послідовності для `n!`: $a_0 = i_0 = 1, i_k = i_{k-1} + 1, a_i = a_{i-1} \times i_k$, $n! = a_n$. **22.** Урахувати, що при знаходженні чергового дільника d числа n до суми можна додати ще один дільник – n/d. При цьому на подільність достатньо перевірити лише числа до $[\sqrt{n}]$. Чому? **23–24.** Особливості машинної арифметики див. у підрозд. 2.3.3. **25.** Для знаходження рекурентної залежності між членами ряду поділити k-й його член на попередній $k-1$-й. **31.** Див. прикл. 2.28 про числа Фібоначчі.

3.4. **16.** За означенням вектор $\bar{a} = (a_1, \ldots, a_n)$ є рекурентною послідовністю без конструкторів. Опорною послідовністю для першого місцезнаходження c є модифікований лічильник $i_1 = 1$, $i_k = (i_{k-1} \le n \,\&\, \&a_{i_{k-1}} \ne c) \to (i_{k-1} = n \to -1 \mid i_{k-1} + 1) \mid (a_{i_{k-1}} = c \to i_{k-1} \mid -1)))$, $k > 1$, а фільтром – умова $(i_k \le n \,\&\, \&a_{i_k} \ne c)$. **18.** За означенням вектор $\bar{a} = (a_1, \ldots, a_n)$ є рекурентною послідовністю без конструкторів. Візьмемо $b_i = \max\{a_1, \ldots, a_i\}$, $i = \overline{1,n}$, як опорну послідовність ($b_n = \max\{a_1, \ldots, a_n\}$). Вона рекурентна: $b_1 = a_1$, $b_i = (a_i > b_{i-1} \to a_i \mid b_{i-1})$. Фільтром є умова $i \le n$, лічильник i ініціалізується з 1. **22.** У багаточлені $P(x) = a_n x^n + \ldots + a_1 x + a_0$ винести x за дужки: $P(x) = (\ldots((a_n x + a_{n-1})x \ldots + a_1)x + a_0$. У схемі Горнера обчислення значення $P(x)$ розпочинається з найбільш внутрішніх дужок. **23.** Знайдені прості числа зберігаються в масиві $p(n)$ у порядку зростання, $p(0) = 2, p(1) = 3$. Наступні кандидати x на просте число – непарні: 5, 7, …. Урахувати, що для перевірки числа x на простоту достатньо перевірити, чи ділиться воно на одне з простих чисел $p(i)$, $i = 0,1,\ldots$ таких, що $p(i)^2 < x$. Для подальшого вдосконалення алгоритму див. [19, 33]. **24.** Ініціалізувати цілий масив $p(n)$ за формулою $p(i) = i, i = \overline{0,n}$. Вилучення кратних чисел з масиву означає їхнє обнулення. Наступний цикл вилучає всі парні числа, які більші 2: `for (k=2;k<=n;k++) p[2*k]=0;`, потім вилучаються всі числа, кратні 3 і більші ніж 3 і т. д., відшукується перше $p[i] \ne 0$ і викреслюються всі числа $2i$, $3i$,... тощо для всіх $p[i] \le [\sqrt{n}]$. **25.** а) Двійкових цифр у числі $k = 2^{100}$ – 101. Реалізувати в циклі множення 100 разів довгого

двійкового числа a на 2 з результатом знов у a. Число a – масив $a(101)$ з нулів і одиниць, початкове значення його $a(101)=1$, усі решта компонентів – 0. У кінці перетворити двійкове число a на десяткове. Скористатися схемою Горнера (див. вправу 22). б) Кількість десяткових цифр у числі 100! обмежена величиною $[\lg 100!]+1=[\sum_{i=1}^{100}\lg i]+1<10+180=190$ (насправді вона значно менша). Реалізувати множення довгих чисел, цифри яких зберігаються в масивах вигляду `char a(190)`. **27.** б) Для подання підмасивів масиву $a(n)$ використати булевий масив $b(n)$ такий, що $\forall i=\overline{0,n-1}(b[i]=1\Leftrightarrow a[i]$ входить до підмасиву. Ідея нерекурсивного перебору всіх 2^n підмасивів полягає в побудові в масиві b двійкових подань кожного зі значень $i=0,1,2,\ldots,2^n-1$. Усі такі подання можна отримати, якщо по черзі додавати 1 до двійкового числа, яке зберігається в масиві b з нульовим початковим значенням. Процес завершити, коли b буде складатися лише з одиниць.

3.5. **17.** 1.

3.6. **16.** Спочатку за алгоритмом Евкліда написати рекурсивну функцію для обчислення $\text{НСД}(m,n)$.

3.8. **37.** Застосувати обмежену чергу у вигляді масиву. **40.** б) Реалізувати пошук входження слова у файлі за допомогою скінченних автоматів [8, 65]. **44.** Числа ввести й попередньо впорядкувати.

СПИСОК ЛІТЕРАТУРИ

1. Абрамов, С. А. Математичекие построения и программирование / С. А. Абрамов. – М.: Наука, 1978.
2. Агафонов, В. Н. Спецификация программ: понятийные средства и их организация / В. Н. Агафонов. – Новосибирск: Наука, 1987.
3. Айрапетян, Р. А. Отладчик SoftICE. Подробный справочник / Р. А. Айрапетян. –СОЛОН-Пресс, 2003.
4. Андерсон, Р. Доказательство правильности программ / Р. Андерсон; пер. с англ. – М.: Мир, 1982.
5. Анисимов, А. В. Информатика. Творчество. Рекурсия / А. В. Анисимов. – К.: Наук. думка, 1988.
6. Анисимов, А. В. Кодирование данных линейными формами числовых последовательностей / А. В. Анисимов // Кибернетика и системный анализ. – 2003. – № 5. – С. 16-36.
7. Ахо, А. Теория синтаксического анализа, перевода и компиляции / А. Ахо, Дж. Ульман. – М.: Мир, 1978. – Т. 1: Синтаксический анализ.
8. Ахо, А. Структуры данных и алгоритмы / А. Ахо, Дж. Хопкрофт, Дж. Ульман. – М.: Вильямс, 2000.
9. Бауер, Ф. Л. Информатика / Ф. Л. Бауер, Г. Гооз. Т. 1-2. – М.: Мир, 1990.
10. Бежанова, М. М. Современные понятия и методы программирования / М. М. Бежанова, И. В. Поттосин. – М.: Научный мир, 2000.
11. Белов, Ю. А. Інструментальні засоби програмування / Ю. А. Белов, В. С. Проценко, П. Й. Чаленко. – К.: Либідь, 1993.
12. Боггс, У. UML и Rational Rose 2002 / У. Боггс, М. Боггс. – М.: ЛОРИ, 2004.
13. Ботвинник, М. М. Алгоритм игры в шахматы / М. М. Ботвинник. – М.: Наука, 1968.
14. Буй, Д. Б. Метод нерухомої точки в програмології: загальна теорія / Д. Б. Буй // Вибрані питання програмології. Праці наукового семінару "Програмологія та її застосування". – К.: Науковий світ, 2007. – С. 98-111.
15. Буй, Д. Б. Примитивные программные алгебры вычислимых функций / Д. Б. Буй, В. Н. Редько // Кибернетика. – 1987. – № 3. – С. 68-74.
16. Буч, Г. Объектно-ориентированный анализ и проектирование. С примерами приложений на C++ / Г. Буч; пер. с англ. – 2-е изд. – СПб.: Невский диалект, 2001.
17. Бьюзен, Т. Супермышление / Т. Бьюзен. – Минск: ООО "Попурри", 2003.
18. Винниченко, И. В. Автоматизация процессов тестирования / И. В. Винниченко. – СПб.: Питер, 2005.
19. Вирт, Н. Систематическое программирование. Введение / Н. Вирт. – М.: Мир, 1987.
20. Гинзбург, С. Математическая теория контекстно-свободных языков / С. Гинзбург. – М.: Мир, 1970.

21. Глибовець, М. М. Мова програмування Сі : навч. посіб. з лабораторного практикуму / М. М. Глибовець, І. В. Кравченко, Є. С. Бубенщиков. – К.: НаУКМА, 1999.

22. Глибовець, М. М. Мова програмування Сі : навч.-метод. посіб. / М. М. Глибовець, В. І. Ляшко, В. С. Проценко. – К.: ВД "КМ Академія", 2002.

23. Глушков, В. М. Алгебра. Языки. Программирование / В. М. Глушков, В. М. Цейтин, Е. Л. Ющенко. – К.: Наук. думка, 1974.

24. Глушков, В. М. Теория автоматов и формальные преобразования микропрограмм / В. М. Глушков // Кибернетика. – 1965. – № 5.

25. Гнеденко, Б. В. Элементы программирования / Б. В. Гнеденко, В. С. Королюк, Е. Л. Ющенко. – М.: Физматгиз, 1963.

26. Готт, В. С. Методологические проблемы информатики / В. С. Готт, Э. П. Семенюк, А. Д. Урсул // Закономерности развития современной математики. – М.: Наука, 1987.

27. Грис, Д. Наука программирования / Д. Грис. – М.: Мир, 1984.

28. Грэхэм, Р. Конкретная математика. Основание информатики / Р. Грэхэм, Д. Кнут, О. Паташник. – М.: Мир, 1998.

29. Гэри, М. Вычислительные машины и труднорешаемые задачи / М. Гэри, Д. Джонсон; пер. с англ. – М.: Мир, 1982.

30. Дастин, Э. Автоматизированное тестирование программного обеспечения. Внедрение, управление и эксплуатация / Э. Дастин, Дж. Рэшка, Дж. Пол. – М.: ЛОРИ, 2003.

31. Дейстра, Э. Дисциплина программирования / Э. Дейстра. – М.: Мир, 1978.

32. Дорошенко, А. Ю. Алгеброалгоритмічні основи програмування / А. Ю. Дорошенко, Г. С. Фінін, Г. О. Цейтлін. – К.: Наук. думка, 2004.

33. Дрейфус, М. Практика программирования на Фортране. Упражнения с комментариями / М. Дрейфус, К. Ганглоф. – М.: Мир, 1978.

34. Евстигнеев, В. А. Толковый словарь по теории графов в информатике и программировании / В. А. Евстигнеев, В. Н. Касьянов. – Новосибирск: Наука, 1999.

35. Ематин, В. Автоматизированное тестирование при разработке ПО / В. Ематин, Б. Позин // Сетевой журн. – 2003. – № 3.

36. Енциклопедія кібернетики; т. 1-2. – К.: Наук. думка, 1973.

37. Ершов, А. П. Информатика: предмет и понятие / А. П. Ершов // Кибернетика. Становление информатики. – М.: Наука, 1986. – С. 28-31.

38. Задірака, В. К. Методи захисту фінансової інформації : навч. посіб. / В. К. Задірака, О. С. Олексюк. – К.: Вища школа, 2000.

39. Зубенко, В. В. Элементарные программные алгебры: автореф. дис. канд. физ.-мат. наук / В. В. Зубенко. – КГУ им. Т. Г. Шевченко, 1981.

40. Зубенко, В. В. Алгебраїчні специфікації інформаційних систем / В. В. Зубенко // Наукові записки НаУКМА. – Т. 19-20 : Комп'ютерні науки. – К. 2002. – С. 5-17.

41. Зубенко, В. В. Про становлення інформатики як наукової та учбової дисципліни / В. В. Зубенко // Проблеми програмування. – 2008. – № 2-3. – С. 459-466.

42. Зубенко, В. В. Про темпоральні процедури та алгоритми / В. В. Зубенко // Вісн. Київ. ун-ту. Серія: Кібернетика. – 2005. – № 6. – С. 20-25.

43. Зубенко, В. В. Програмування: Мова Сі: методична розробка для студентів першого курсу спеціальностей "Інформатика" та "Соціальна інформатика" / В. В. Зубенко. – К.: ВПЦ "Київ. ун-т", 2006.

44. Зубенко, В. В. Темпоральні процедури та алгоритми в контексті дескриптивних систем / В. В. Зубенко // Вибрані питання програмології. Праці наук. семінару "Програмологія та її застосування". – К.: Наук. світ, 2007. – С. 98-111.

45. Зубенко, В. В. Темпоральні процедури та алгоритми / В. В. Зубенко // Проблеми програмування. – 2006. – № 2-3. – С. 53-59.

46. Зубенко, В. В. Методичні рекомендації до виконання практичних та лабораторних робіт з дисципліни "Програмування" для студентів першого курсу факультету кібернетики / В. В. Зубенко, Г. А. Кияшко – К.: ВПЦ "Київ. ун-т", 2006.

47. Зубенко, В. В. Методичні рекомендації до оформлення звітних матеріалів з дисципліни "Програмування" для студентів першого курсу факультету кібернетики / В. В. Зубенко, Г. А. Кияшко. – К.: ВПЦ "Київ. ун-т", 2006.

48. Зубенко, В. В. Информатика как научная дисциплина / В. В. Зубенко, Ю. В. Сидоренко // Искусственный интеллект. – Донецк, 2009. – № 1.

49. Йодан, Э. Структурное проектирование и конструирование программ / Э. Йодан. – М.: Мир, 1979.

50. Калужнин, Л. А. Об алгоритмизации математических задач / Л. А. Калужнин // Проблемы кибернетики. – М.: Наука, 1959. – Вып. 2.

51. Калужнін, Л. А. Алгоритми і математичні машини / Л. А. Калужнін, В. С. Королюк. – К.: Радянська шк., 1964.

52. Капитонова, Ю. В. Математическая теория проектирования вычислительных систем / Ю. В. Капитонова, А. А. Летичевский. – М.: Наука; гл. ред. физ.-мат. лит, 1988.

53. Керниган, Б. Практика программирования / Б. Керниган, Р. Пайк. – СПб.: Невск. Диалект, 2001.

54. Керниган, Б. Элементы стиля программирования / Б. Керниган, Ф. Плоджер. – М.: Радио и связь, 1984.

55. Керниган, Б. Язык программирования Си / Б. Керниган, Д. Ричи; 3-е изд., испр. – СПб.: Невск. Диалект, 2001.

56. Клини, С. Введение в метаматематику / С. Клини. – М.: Наука, 1957.

57. Клюшин, Д. А. Полный курс C++. Профессиональная работа / Д. А. Клюшин. – М.: Вильямс, 2005.

58. Кнут, Д. Искусство программирования / Д. Кнут. – Т. 1. – Вып. 1: MMIX-RISC-компьютер нового тысячелетия. – М.: Вильямс, 2007.

59. Кнут, Д. Искусство программирования / Д. Кнут. Т. 1: Основные алгоритмы; 3-е изд. – М.: Вильямс, 2000.

60. Кнут, Д. Искусство программирования. / Д. Кнут. – Т. 3: Сортировка и поиск. – М.: Вильямс, 2007.

61. Кнут, Д. Информатика и её связь с математикой / Д. Кнут // Современные проблемы математики. – М.: Знание, 1977. – С. 4-32.

62. Кон, П. Универсальная алгебра / П. Кон. – М.: Мир, 1968.

63. Конверський, А. Є. Логіка : підручн. для ВНЗ / А. Є. Конверський. – К.: Укр. центр духовної культури, 1999.

64. Коплиен, Дж. Программирование на С++ / Дж. Коплиен; пер. с англ. – СбП: Питер, 2004.

65. Кормен, Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест. – М.: МЦНМО, 2000.

66. Кук, Д. Компьютерная математика / Д. Кук, Г. Бейз. – М.: Наука, 1990.

67. Лавріщева, К. М. Визначення предмету – програмна інженерія / К. М. Лавріщева // Проблеми програмування. – 2008. – № 2-3: спец. випуск. – С. 191-204.

68. Лавріщева, К. М. Програмна інженерія / К. М. Лавріщева. – К.: BHV, 2007.

69. Лавров, С. С. Программирование. Математические основы, средства, теория / С. С.Лавров. – СПб.: БХВ-Петербург, 2001.

70. Лаптев, В. В. С++. Объектно-ориентированное программирование. Задачи и упражнения / В. В. Лаптев, А. В. Морозов, А. В. Бокова. – СПб.: Питер, 2007.

71. Лебедев, С. А. Малая электронная счётная машина / С. А. Лебедев, Л. Н. Дашевский, Е. Л. Шкабара. – М.: Из-во АН СССР, 1952.

72. Леффингуєлл, Д. Принципы работы с требованиями к программному обеспечению. Унифицированный подход / Д. Леффингуєлл, Д. Уидриг. – М.: Вильямс, 2002.

73. Липпман, С. Язык программирования С++. Вводный курс / С. Липпман, Ж. Лажойе; 3-е изд.; пер. с англ. – СПб.: Невск. Диалект; М.: ДМК Пресс, 2001.

74. Лисовик, Л. П. Теория трансдьюсеров / Л. П. Лисовик. Т. 1-3. – К.: Феникс, 2005-2007.

75. Майер, Б. Методы программирования / Б. Майер, К. Бодуен. Т. 1-2. – М.: Мир, 1982.

76. Майерс, Г. Надёжность программного обеспечения / Г. Майерс. – М.: Мир, 1980.

77. Макконел, Дж. Основы современных алгоритмов / Дж. Макконел. – 2-е изд., доп. – М.: Техносфера, 2004.

78. Маліновський, Б. М. Відоме і невідоме в історії інформаційних технологій в Україні / Б. М. Маліновський. – К.: Академперіодика, 2001.

79. Мальцев, А. И. Алгебраические системы / А. И. Мальцев. – М.: Наука, 1970.

80. Мальцев, А. И. Алгоритмы и рекурсивные функции / А. И. Мальцев. – М.: Наука, 1965.

81. Манин, Ю. И. Доказуемое и недоказуемое / Ю. И. Манин. – М.: Сов. радио, 1979.

82. Манна, З. Теория неподвижной точки программ / З. Манна // Кибернетический сборник. Новая серия. – М.: Мир, 1978. – Вып. 15. – С. 38-100.

83. Машина играет в шахматы / Г. М. Адельсон-Вельский, В. Л. Арлазаров, А. Р. Битман, М. В. Донской. – М.: Наука, 1983.

84. Мендельсон, Э. Введение в математическую логику / Э. Мендельсон. – М.: Наука, 1971.

85. Месарович, М. Общая теория систем: математические основы / М. Месарович, Я. Такахара. – М.: Мир, 1978.

86. Михалевич, В. С. Информатика – новая область науки / В. С. Михалевич, Ю. М. Каныгин, В. И. Гриценко // Кибернетика. Становление информатики. – М.: Наука, 1986.

87. Мороз, А. И. Кибернетика в системе современного научного знания / А. И. Мороз // АН УССР. Ин-т философии. – К.: Наук. думка, 1988.

88. На пути к языку широкого спектра для поддержки спецификации и разработки программ / Ф. Бауэр, М. Брой, Р. Гнац и др. // Требования и спецификации в разработке программ; пер. с англ. – М.: Мир, 1984. – С. 28-46.

89. Непейвода, Н. Н. Прикладная логика / Н. Н. Непейвода. – 2-е изд. – Новосибирск: Изд-во Новосиб. ун-та, 2000.

90. Непейвода, Н. Н. Стили и методы программирования. Курс лекций : учеб. пособие / Н. Н. Непейвода. – Интернет-ун-т информ. технологий, 2005. – Режим доступа: http://ulm.uni.udm.ru/~nnn.

91. Непейвода, Н. Н. Основания программирования : учеб. пособие / Н. Н. Непейвода, И. Н. Скопин. – М.; Ижевск, 2003. – Режим доступа: http://ulm.uni.udm.ru/~nnn.

92. Непомнящий, В.А. Прикладные методы верификации программ / В. А. Непомнящий, О. М. Рякин. – М.: Радио и связь, 1988.

93. Нікітченко, М. С. Інтенсіонально-орієнтований підхід до теорії програмування / М. С. Нікітченко // Вибрані питання програмології. Праці наукового семінару "Програмологія та її застосування". – К., 2007. – С. 22-54.

94. Нікітченко, М. С. Математична логіка та теорія алгоритмів / М. С. Нікітченко, С. С. Шкільняк. – К.: ВПЦ "Київ. ун-т", 2008.

95. Новиков, Ф. А. Дискретная математика для программистов / Ф. А. Новиков. – СПб.: Питер, 2000.

96. Объектно-ориентированный анализ и проектирование. С примерами приложений / Г. Буч, Р. А. Максимчук, М. У. Энгел и др. ; пер. с англ. – 3-е изд. – М.: Вильямс, 2008.

97. Основи дискретної математики / Ю. В. Капітонова, С. Л. Кривий, О. А. Летичевський та ін. – К.: Наук. думка, 2002

98. Павловская, Т. А. С++. Объектно-ориентированное программирование : практикум / Т. А. Павловская, Ю. А. Щупак. – СПб.: Питер, 2008.

99. Петрук, В. І. Факультету кібернетики – 40. Нарис історії (1969-2009) / В. І. Петрук. – К., 2009.

100. Порублев, И. Н. Алгоритмы и программы. Решение олимпиадных задач / И. Н. Порублев, А. Б. Ставровский. – М.: Вильямс, 2007.

101. Поспелов, Д. А. Становление информатики в России / Д. А. Поспелов // Очерки истории информатики в России; под ред. Д. А. Поспелова, Я. И. Фета. – Новосибирск: Научно-издательский центр ИГТМ СО РАН, 1998.

102. Прата, С. Язык программирования С++. Лекции и упражнения / С. Прата; пер. с англ. – М.: ООО "ДиаСофтЮП", 2005.

103. Пратт, Т. Языки программирования: разработка и реализация / Т. Пратт, М. Зелковиц; пер. с англ.; 4-е изд. – СПб: Питер, 2002.

104. Проценко, В. С. Техніка програмування мовою Сі / В. С. Проценко, П. Й. Чаленко, А. Б. Ставровський. – К.: Либідь, 1993.

105. Рассева, Е. Математика метаматематики / Е. Рассева, Р. Сикорский. – М.: Наука, 1972.

106. Редько, В. Н. Дескриптологические основания программирования / В. Н. Редько // Кибернетика и системный анализ. – 2002. – № 1. – С. 3-19.

107. Редько, В. Н. Основания дескриптологии / В. Н. Редько // Кибернетика и системный анализ. – К., 2003. – № 5. – С. 16-36.

108. Редько,В. Н. Основания композиционного программирования / В. Н. Редько // Программирование. – М., 1979. – № 3.

109. Редько, В. Н. Основания программологии / В. Н. Редько // Кибернетика и системный анализ. – К., 2000. – № 1. – С. 33-57.

110. Редько, В. Н. Введение в операционные системы СМ ЭВМ / В. Н. Редько, А. П. Жежерун, В. В. Зубенко. – К.: УМК ВО, 1989.

111. Рекомендации по преподаванию программной инженерии и информатики в университетах; пер. с англ. – М.: ИНТУИТ.РУ "Интернет-университет Информационных технологий", 2007.

112. Рефакторинг. Улучшение существующего кода / М. Фаулер, К. Бек, Дж. Брант и др. – Символ-Плюс, 2003.

113. Решение математических задач на автоматических цифровых машинах. Программирование для быстродействующих электронных счётных машин / Л. А. Люстерник, А. А. Абрамов, В. И. Шестаков, М. Р. Шура-Бура. – М.: Изд-во АН СССР, 1952.

114. Себеста, Р. У. Основные концепции языков программирования / Р. У. Себеста. – М.: Вильямс, 2001.

115. Сергієнко, І. В. Інформатика та комп'ютерні технології / І. В. Сергієнко. – К.: Наук. думка, 2004.

116. Сергієнко, І. В. Виклик часу в кібернетичному вимірі / І. В. Сергієнко. – К.: Академперіодика, 2007.

117. Симович, С.В. Общая информатика. Новое издание / С. В. Симович. – СПб.: Питер, 2008.

118. Скотт, Д. Набросок математической теории вычислений / Д. Скотт // Кибернетический сборник. – М.: Мир, 1977. – Вып. 24. – С. 107-121.

119. Скотт, Д. Теория решеток, типы данных и семантика / Д. Скотт // Данные в языках программирования. – М.: Мир, 1982. – С. 25-53.

120. Соммервилл, И. Инженерия программного обеспечения / И. Соммервилл – 6-е изд. – М.: Вильямс, 2002.

121. Справочная книга по математической логике : в 4 т. / под ред. Дж. Барвайса. – М.: Наука, 1982-83.

122. Ставровський, А. Б. Програмування. Перші кроки / А. Б. Ставровський, Т. О. Карнаух. – М.: Вільямс, 2005.

123. Стол, Р. Р. Множества. Логика. Аксиоматические теории / Р. Р. Стол. – М.: Просвещение, 1968.

124. Страуструп, Б. Язык программирования С++. Специальное издание / Б. Страуструп. – СПб.: Невск. Диалект, 2006.

125. Сэджвик, Р. Фундаментальные алгоритмы на С++ / Р. Сэджвик; пер. с англ. – М.: ООО "ДиаСофтЮП", 2002.

126. Тарасов, С.М. Игры для всех: азартные и неазартные / С. М. Тарасов, С. П. Попов. – М.: Профиздат, 1991.

127. Тассел, Д. Стиль, разработка, эффективность, отладка и испытание программ / Д. Тассел; пер. с англ. – М.: Мир, 1985.

128. Тестирование на основе моделей / А. Петренко, Е. Бритвина, С. Грошев и др. // Открытые системы. – 2003. – № 9.

129. Успенский, В. А. Теория алгоритмов: основные открытия и приложения / В. А. Успенский, А. Л. Семенов. – М.: Наука; гл. ред. физ.-мат. лит., 1987.

130. Факультету кібернетики – 30 / за ред. проф. О. К. Закусила. – К., 1999.

131. Фреге, Г. Основи арифметики / Г. Фреге. – К.: Port-Royal, 2001.

132. Фьюэр, А. Задачи по языку Си / А. Фьюэр. – М.: Финансы и статистика, 1985.

133. Харбисон, С. П. Язык программирования С / С. П. Харбисон, Г. Л. Стил. – 5-е изд. – М.: БИНОМ, 2004.

134. Хоменко, Л. Г. История отечественной кибернетики и информатики / Л. Г. Хоменко. – К.: Ин-т кибернетики им. В. М. Глушкова НАН Украины, 1998.

135. Шахматный словарь. – М.: Физкультура и спорт.

136. Шилдт, Г. Полный справочник по С / Г. Шилдт. – М.: Вильямс, 2006.

137. Ющенко, Е. Л. Адресное программирование и особенности решения задач на машине "Урал" / Е. Л. Ющенко. – К.: КВИРТУ, 1960.

138. Apt, K. R. Verification of Sequential and Concurrent Programs / K. R. Apt, E. V. Olderog. – Berlin: Springer-Verlag, 1991.

139. Bauer, F. L. Algorithmic Language and Program Development / F. L. Bauer, H. Woussner. – NY: Springer-Verlag, Inc., 1982.

140. Fejer, P. A. Mathematical foundation of computer science / P. A. Fejer, D. A. Simovici. – Vol. 1: Sets, Relations and Induction. – NY: Springer-Verlag, Inc., 1991.

141. Hoffman, U. Stack Based Dynamic Languages for Intelligent Systems / U. Hoffman, A. Protasov, V. Zubenko // УСиМ. – 1997. – № 3. – С. 59-65.

142. Kalmar, L. An Argument Against the Plausibility of Church's Thesis / L. Kalmar // Constructivity in Mathematics. Proceed. of Colloq. held at Amsterdam, 1957. – Amsterdam: North-Holland, 1963.

143. Levine, J. Flex & Bison / J. Levine. – O'Reilly Media, August 2009.

144. Levine, J. LEX & YACC / J. Levine, T. Mason, D. Brown.– 2 ed. – O'Reilly, 1992.

145. Manna, Z. The Logical Basis for Computer Programming / Z. Manna, R. Waldinger. – Vol. 1: Deductive Reasoning. – Addison-Wesley Publishing Company, Inc., 1985.

146. McCarthy, J. A Basis for a Mathematical Theory of Computation / J. McCarthy // Computer Programming and Formal Systems. – Amsterdam: North-Holland, 1963. – P. 33-70.

147. Newel, A. Computer Science / A. Newel, A. Perlis, H. Simon // Science. – 1967. – № 157. – P. 1373-1374.

148. Stoy, J. E. The Scott-Sterachey Aproach to Programming Language Theory / J. E. Stoy. – Cambridge, Massachusetts: The MIT Press, 1979.

149. The Munich Project CIP. – Vol. 1: The Wide Spectrum Language CIP-L // Lecture Notes in Computer Science. – № 183.

150. Zubenko, V. Applied Program Algebras / V. Zubenko. – Kiel: CAU Kiel, Bericht. – № 8504.

151. http://alglib.sources.ru/articles/zeromatr.php

152. http://comsys.ntu-kpi.kiev.ua/science/first_department.shtml

153. www.icfcst.kiev.ua/museum

ПРЕДМЕТНИЙ ПОКАЖЧИК

ЗМІСТ