МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ

ЧЕРНІГІВСЬКИЙ ДЕРЖАВНИЙ ІНСТИТУТ ЕКОНОМІКИ І УПРАВЛІННЯ

Кафедра «Економічної кібернетики та інформатики»

ЗАЦЕРКОВНИЙ В.І., ГУР'ЄВ В.І., СІМАКІН Ю.С., ФІРСОВА І.В.

«Ухвалено»
на засіданні кафедри
«Економічної кібернетики
та інформатики»
Протокол № 6
від «20» грудня 2011 р.

АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ

КОНСПЕКТ ЛЕКЦІЙ

для студентів спеціальності
«Економічна кібернетика»

Чернігів 2012 р.

УДК. 004.42, 519.85

Рецензенти: Казимир В.В., д.т.н., професор;
Сахно Є.Ю., д.т.н., професор.

Авторський колектив:

Зацерковний В.І., к.т.н., доцент;
Гур'єв В.І., к.т.н., доцент;
Сімакін Ю.С., викладач;
Фірсова І.В., ст.викладач

Алгоритмізація та програмування. Конспект лекцій – Чернігів,: ЧДЕІУ, 2012. – 184 с.

Конспект лекцій «Алгоритмізація та програмування» має на меті сформувати в студентів науково обґрунтовані погляди на сучасні технологічні процеси в інформаційній галузі. Конспект лекцій допоможе ознайомитися з основними поняттями теорії алгоритмів, з найбільш поширеними алгоритмічними системами; засвоїти принципи організації алгоритмічних процесів та форми їхньої реалізації; ознайомитися з основними поняттями проектування автоматизованих комплексів задач; навчитися будувати базові алгоритми процесів пошуку, передавання, обробки інформації в різних інформаційних технологіях; навчитися здійснювати аналіз та контроль алгоритму на різних етапах життєвого циклу програмного продукту; ознайомитися з основними мовами опису алгоритмів, особливостями програмування, принципами організації та реалізації програм; складати супровідну проектну документацію.

ВСТУП

„У кожній дисципліні є стільки науки, скільки у ній математики”
Д.І. Менделєєв

ХХ століття в галузі науки і техніки принесло людству багато великих досягнень: радіо, звукове кіно, телебачення, атомна енергія, космічні польоти, електронно-обчислювальні машини – ось тільки найголовніші віхи, відомі кожному. Напевно, не менш відомі кібернетика, вірусологія, генетика.

Однак не всім відомо, що найбільшим досягненням науки ХХ ст. є теорія алгоритмів – нова математична дисципліна. Теорія електронних обчислювальних машин, теорія і практика програмування *не* можуть обійтися без неї. Математична логіка і кібернетика пред'являють на неї свої права. Однак вона є самостійною наукою, яка готова служити усім наукам, і має своє обличчя, свій предмет.

Сама назва – теорія алгоритмів – говорить про те, що її предмет – алгоритми. Що це таке? Поняття алгоритму одночасно є надзвичайно простим і дуже складним. Його простота – у величезній кількості алгоритмів, з якими ми маємо справу в їх повсякденності. Але ці ж обставини роблять його мрячним, розпливчастим, який важко піддається строгому науковому визначенню.

Кожна людина щодня зустрічається з множиною задач від найпростіших і добре відомих до надзвичайно складних. Для багатьох задач існують певні правила (інструкції, приписи), які пояснюють виконавцю, як розв'язувати дану задачу. Ці правила людина може вивчити заздалегідь або сформулювати в процесі розв'язання задачі. Чим точніше і зрозуміліше будуть описані правила розв'язку задач, тим швидше людина оволодіє ними і буде ефективніше їх застосовувати. Джерелами виникнення алгоритмів слугують: спостереження і експеримент, наукова теорія, минулий досвід тощо.

Вирішення багатьох задач людина може передавати технічним пристроям – персональним комп’ютерам (ПК), автоматам, роботам тощо.

Інформаційні технології, основним елементом яких є комп’ютер, використовуються практично у всіх сферах людської діяльності. Комп'ютер потрібний людині для розв’язання різноманітних практичних задач. Як відомо, ПК – це пристрої для збереження та обробки інформації. На відміну від телевізора або магнітофона, які здійснюють тільки заздалегідь закладені у них функції, ПК можуть розв’язувати будь-які задачі[1]. Проте, для цього комп’ютеру необхідно попередньо скласти, на зрозумілій йому мові, точну та докладну послідовну інструкцію (програму[2], алгоритм), яка буде вказувати йому, як і що саме треба робити з інформацією.

Розв’язання задачі за допомогою ПК передбачає використання двох термінів програма і алгоритм, які доволі близькі за змістом.

Під програмою розуміють інформацію кінцевого обсягу, яка будучи введенною в комп’ютер, призводить до розв’язання задачі. Комп’ютер сприймає цю інформацію, якщо вона закодована (представлена) машинною мовою або якщо в комп’ютері є інша програма (транслятор або інтерпретатор), яка перетворює інформацію в машинний код. Різні типи комп’ютерів можуть характеризуватись різними машинними мовами, які вони використовують. Для комп’ютерів класичної, традиційної архітектури програма представляє собою послідовність команд або рядком тексту на мові програмування високого рівня.

Якщо є програма і математична модель реального комп’ютера, то можна провести повний аналіз характеристик обчислювального процесу – часу роботи програми для різних вхідних даних, потрібний обсяг пам’яті, довести, що програма дійсно розв’язує поставлену задачу, а не якусь іншу (правильність програми), дослідити ті або інші проблеми.

Однак аналіз на основі програм має й свої недоліки. По-перше, його результати пов’язані як з самою задачею, так і з типом комп’ютера. По-друге, припущення про наявність програми знімає питання про принципову можливість розв’язку задачі на певному комп’ютері, тобто далеко не всі питання, які нас цікавлять, можна сформулювати. По-третє, аналіз на основі програми найчастіше є надлишковим: іноді цікавить відповідь на просте питання або відповідь з невисокою точністю, а в результаті отримуємо ретельний аналіз, який потребує тривалого часу.

Прикладна математика постійно зіштовхується з низкою проблемам при дослідженні реальних об’єктів. З цією метою був розроблений метод їх розв’язання – абстракція. Необхідно абстрагуватись від несуттєвих деталей і тим самим істотно спростити задачу. Навряд чи кому прийде в голову враховувати в математичній моделі комп’ютера, створеній для розв’язання

1 Задача (problem, task) – проблема, що підлягає розв’язанню.

2 Комп’ютерна програма – запис алгоритму розв’язання задачі у вигляді послідовності команд або операторів мовою, яку розуміє комп’ютер.

поставлених нами задач, його колір або масу. Однак часто можна вважати однаковою тривалість різних елементарних операцій, наприклад додавання і множення, вважати що пам'ять має практично необмежений об'єм тощо. Йдучи цим шляхом, можна виділити основні, найбільш істотні властивості програм і комп'ютерів, що їх виконують, ввести відповідні математичні об'єкти і побудувати математичну теорію комп'ютерної обробки інформації.

Алгоритмізація це розділ інформатики, який вивчає методи і прийоми побудов алгоритму, а також їх властивості.

Алгоритм є математичною абстракцією програми.

Застосування технічних пристроїв пред'являє дуже строгі вимоги до точності опису правил і послідовності виконання дій. Тому розробляються спеціальні мови для чіткого і строгого опису різних правил.

Будь-яка діяльність завжди має мету. Мета діяльності – це майбутній бажаний ідеальний результат діяльності. Людина, яка виконує певну роботу, зазвичай розуміє, для чого вона це робить і що хоче отримати в результаті цієї діяльності. При цьому, діяльність може мати одну мету (ціль), або певну їх сукупність. Цілі можуть створювати ієрархію, у якій існують як проміжні, так і цілі більш високого рівня.

Алгоритм можна розглядати як певний універсальний засіб для розв'язання цілого класу задач. Але існують такі класи задач, для розв'язку яких немає загального універсального алгоритму. Проблеми розв'язку такого роду задач називаються алгоритмічно нерозв'язними проблемами. Однак алгоритмічна нерозв'язність проблеми розв'язання задач того або іншого класу не означає неможливість розв'язання певної окремої задачі з цього класу. Перехід від інтуїтивного поняття алгоритму до формального визначення алгоритму (рекурсивні функції, машини Тюрінга) дозволяють довести алгоритмічну нерозв'язність низки проблем.

Поняття, алгоритми і методи теорії формальних мов, граматик і автоматів є теоретичною основою сучасної теорії програмування, побудови алгоритмічних мов, проектування мовних процесорів, зокрема, компіляторів, ассемблерів, макрогенераторів тощо.

Врешті-решт, теорія алгоритмів прийшла до того, що об'єктами обробки стали самі алгоритми. Основними завданнями є автоматична верифікація і оптимізація програм й системи з розпаралелюванням виконання програм на багатопроцесорних обчислювальних системах. Наступним напрямом є лінгвістичні алгоритми: перевірка орфографії, автоматичний переклад, програми, що „розмовляють", робота з граматикою.

Формування наукового поняття алгоритму, яке стало важливою науковою проблемою, не закінчене сьогодні. І хоча теорія алгоритмів є математичною дисципліною, вона ще не дуже схожа на такі широко відомі науки, як геометрія або теорія чисел. Вона ще тільки зароджується, причому тим вихідним матеріалом, на підставі якого повинне бути побудоване широке наукове поняття алгоритму, є інтуїтивне поняття, також дуже широке, але недостатньо ясне.

В реальному житті виконання будь-яких дій пов'язане з витрачанням різних ресурсів: матеріалів, енергії і часу. Навіть здійснюючи які-небудь записи, ми витрачаємо ресурси (наприклад, папір, чорнила і час). Ще недавно деякі задачі не можна було розв'язати через занадто велику кількість необхідних для цього операцій і занадто малу швидкість їх виконання. Поява електронних обчислювальних машин зробило такі задачі розв'язними. Це значить, що „математизуючи" поняття алгоритму, потрібно абстрагуватись, відволіктись від обмеженості ресурсів, вимагаючи тільки їх кінцевості, інакше теорія алгоритмів застаріє, як тільки розвиток науки і техніки дозволить переступити через існуючі межі ресурсів.

Цей конспект лекцій допоможе починаючому користувачу розібратися в тому, як будуються алгоритми. З цією метою в розгляд вводяться так звані алгоритмічні операції, які складаються з ряду дій, що реалізують деяку алгоритмічну функцію – закінчену чинність. Одні з цих операцій можна вважати "простими", інші же для своєї реалізації вимагають використання деяких інших алгоритмічних операцій. Алгоритмічні операції представляються в деякому ідеалізованому вигляді. На практиці звичайно ці операції в алгоритмі переплітаються, тобто в записи алгоритму однієї операції можуть бути присутніми складники інших операцій. Така ідеалізація на початкових етапах навчання представляється виправданою.

Конспект лекцій складається з двох розділів.

В першому розділі наведено основні поняття і визначення дисципліни: алгоритм і його властивості, класифікація даних, правила запису алгоритмів, елементи блок-схем, псевдокод, поняття типів та кодування даних, викладені основні алгоритмічні операції. Як правило, на початку розробки кожної алгоритмічної операції розглядаються необхідні приклади. По ходу розв'язання прикладів увага акцентується на важливих для наступних розробок алгоритму моментах. Після цього дається узагальнений алгоритм розглядуваної операції. Часто після розробки алгоритмічної операції приводиться розв'язання прикладної задачі, що базується на знаннях про раніше розроблені операції. Другий розділ є вступом в технологію розробки програм на мові програмування Turbo Pascal.

РОЗДІЛ 1.

ОСНОВИ АЛГОРИТМІЗАЦІЇ

1. ВВЕДЕННЯ В ТЕОРІЮ АЛГОРИТМІВ

„Хто як програмує, так і їсть”.
Народна мудрість

Як відомо, персональні комп'ютери – це пристрої для збереження та обробки інформації. На відміну від пристроїв побутової техніки, які здійснюють тільки заздалегідь закладені у них функції, персональні комп'ютери спроможні розв’язувати будь-які задачі. Проте, для цього комп’ютеру необхідно попередньо скласти на зрозумілій йому мові, точну та докладну послідовну інструкцію (алгоритм, програму), яка буде вказувати йому, як, коли і що саме треба робити з інформацією.

Сам по собі комп’ютер не має знань у жодній галузі свого використання, а всі ці знання зосереджені у програмах, які виконує комп’ютер. Працюючі з комп’ютером, більшість людей не займається створенням програм, а лише використовують їх. Саме тому їх й називають користувачами.

Змінюючи програми для комп’ютера, можна перетворити його в робоче місце геодезиста або конструктора, економіста або землевпорядника, бухгалтера або менеджера тощо, редагувати на ньому документи або розслабившись, пограти у цікаву гру.

Таким чином, для ефективного використання комп’ютера обов’язково потрібно знати призначення та властивості програм, які необхідні для роботи з ним та способи їх створення.

1.1. Поняття алгоритму та програми

У своєму повсякденному житті людина використає велику кількість правил та сукупностей послідовностей дій, завдяки яким вона розв’язує задачі різної складності, що виникають перед нею. Важко навіть перерахувати всі задачі, при розв’язанні яких ми використовуємо певні алгоритми. Одна шкільна математика містить величезну кількість алгоритмів, починаючи від алгоритмів виконання арифметичних дій над натуральними числами, дробами і закінчуючи алгоритмами розв'язання усякого роду рівнянь, нерівностей, систем рівнянь, правил визначення значень функцій за таблицями Брадіса тощо. Однак алгоритмізація не є прерогативою математики. У живій природі і в повсякденному житті будь-якій масовій цілеспрямованій діяльності відповідає заздалегідь створений алгоритм. Наприклад, за визначеним алгоритмом здійснюється масове виготовлення взуття й пошив одягу, збирання автомобілів на конвеєрі, випічка хліба й виплавка металу. В побуті це стосується, правил настанови дзвінка будильника на необхідну годину, правил переходу вулиці, й багатьох інших задач. Причому порушення подібних правил або послідовності їх виконання може призвести як до невірного, так навіть і до трагічного результату.

При цьому, ми дуже рідко застосовуємо термін „алгоритм”. Навіть у математиці, де виникло поняття алгоритму і багато століть розроблялися алгоритми для розв'язання все нових і нових класів задач, математики тривалий час не замислювалися над тим, що таке алгоритм.

Дійсно, якщо знайдений загальний метод розв'язку певної задачі з нового класу задач, нам немає потреби замислюватися над питанням, чи є цей метод алгоритмом. Є метод, і ми його застосовуємо до будь-якої задачі даного класу задач, для якого він призначений. Так формулу коренів квадратного рівняння ми використовуємо при розв'язанні будь-якого квадратного рівняння. Інтуїтивно зрозуміло, що дана формула задає загальний метод (алгоритм) розв'язку будь-якого квадратного рівняння. З арифметики відомі алгоритми визначення найбільшого загального дільника і найменшого загального знаменника, з шкільної алгебри – формули Вієтта розв’язання квадратних рівнянь (формула – це компактний запис послідовності виконання арифметичних дій), з лінійної алгебри – правило Крамера і спосіб Гауса розв’язання систем лінійних алгебраїчних рівнянь тощо. Або згадаємо, як ми у початкових класах розв’язували задачі: спочатку задавали собі питання „скільки?”, а після цього на нього відповідали. Вже тоді ми почали навчатися будувати алгоритми, не підозрюючи про таке хитре слово.

Чому ж тоді протягом багатьох століть в математиці, де все строго визначається і доводиться, обходилися лише інтуїтивним поняттям алгоритму?

Один з творців математичних теорій алгоритмів радянський математик А. А. Марков (1903 – 1979 рр.) вважав, що це було припустиме, поки термін „алгоритм” зустрічався в математиці лише у висловленнях наступного типу: „Для розв'язку таких-то задач є алгоритм, і ось в чому він полягає”.

За тією ж причиною інтуїтивне поняття алгоритму достатнє й при постановці питання про переклад вже наявного алгоритму в програму для розв'язання задачі на комп’ютері.

<u>Комп’ютерна програма</u> – запис алгоритму розв’язання задачі у вигляді послідовності команд або операторів мовою, яку розуміє комп’ютер.

Якщо виникає необхідність доручити іншій людині певну роботу, спочатку їй необхідно роз’яснити порядок та технологію її виконання, тобто скласти алгоритм. Ступінь деталізації при цьому може бути різною. Для фахівця з великим досвідом, який вже неодноразово виконував

таку роботу, достатньо сформулювати тільки мету, а доручене завдання він виконає за наявним у нього досвідом і схемою дій. Якщо у виконавця досвід роботи не дуже великий або взагалі відсутній, то необхідно сформулювати основні етапи (кроки) процесу виконання роботи та докладно визначити послідовність виконання найбільш відповідальних з них.

Найретельніше треба розписувати послідовність виконання окремих операцій, якщо виконавцем буде комп'ютер. Він може лише точно виконувати те, що йому доручено, тобто виконувати певний алгоритм. Саме тому, одним з базових понять інформатики є поняття алгоритму, як певного правила перетворення інформації.

Слово „алгоритм", за суттю є синонімом слів спосіб, рецепт, інструкція. Він містить вказівки про те, які операції у ході обробки даних і у якій послідовності необхідно виконати, щоб розв'язати задачу. Як міг би сказати Козьма Прутков[1], *„добрий алгоритм подібний гострому ножу – і перший і другий досягають мети легко і просто*". Інше порівняння: *людина, яка використовує поганий алгоритм, подібна повару, який відбиває м'ясо викруткою: ледь-ледь їстівний і малопривабливий результат досягається ціною великих зусиль*.

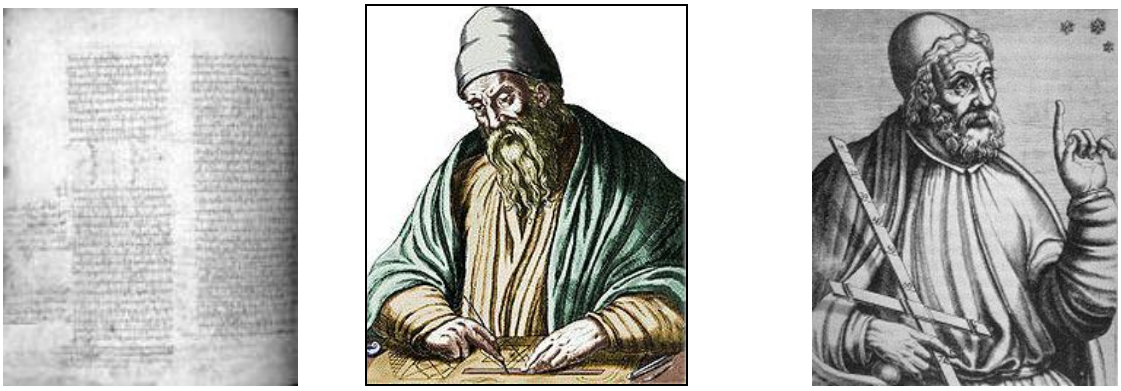
Послідовності вищеназваних правил виконання дій інколи називають *алгоритмами*. В повсякденному житті цей термін використовується рідко, значно частіше він застосовується у математиці. Навчаючись арифметиці у початковій школі, ми познайомилися з додаванням і множенням двох чисел. Нам у явній формі не говорили, що у будь-якої пари чисел існують добуток і сума, а вказали способи і правила їхнього знаходження. Такі способи і правила є прикладами алгоритмів і обчислювальних (ефективних) процедур. Їхнє застосування не вимагає винахідливості чи кмітливості, учню необхідно тільки слідувати інструкціям вчителя.

Таким чином, алгоритм можна трактувати, як технологічну інструкцію з окремих приписів, виконання яких у заданій послідовності призводить до заздалегідь передбачуваного результату. Навіть життя на Землі зародилось саме тоді, коли на молекулярному рівні був записаний алгоритм життєвого циклу білкової клітинки від народження до смерті. Усі живі організми здійснюють свій життєвий цикл за таким створеним природою алгоритмом з деякими відхиленнями, викликаними впливом навколишнього середовища. За суттю алгоритм – це один із видів інформації, яка є внутрішньою для діючого об'єкта і яка використовується для обробки зовнішньої інформації, що приймається каналами зв'язку. Ми будемо займатися алгоритмами штучного типу, які створюються з участю людини, переважно у галузі математики і програмування.

Давайте розберемося, а як же так вийшло, що алгоритми з галузі математики перекочували у галузь програмування.

Пошуки різноманітних алгоритмів входили до переліку найважливіших задач в усі часи існування науки. Звернемося до історії. Зародки алгоритмів можна знайти ще у глибокій давності.

Вважається, що найстародавнішим нетривіальним алгоритмом є спосіб знаходження найбільшого загального дільника двох цілих чисел. Він був викладений в працях давньогрецького математика Евкліда – сьомій книзі „Начал" близько 2300 років потому (рис.1.1). Однак, є припущення, що алгоритм Евкліда лише інтерпретація алгоритму, запропонованого Евдоксом Кіндським за 75 років до цього.



а) б) в)

Рис. 1.1 Алгоритм знаходження найбільшого загального дільника двох чисел а) і його ймовірні винахідники: б) Евклід; в) Евдокс Кіндський

[1] **Козьма́ Петро́вич Пруткóв** – літературна маска, під якою в журналах «Современник», «Искра» та інших виступали в 50–60 рр. XIX ст. поети Олексій Толстой (найбільший в кількісному обчисленні вклад), брати Олексій, Володимир і Олександр Жемчужнікови (фактично – їх колективний псевдонім), а також Петро Єршов.

Алгоритм Евкліда знаходження найбільшого загального дільника двох цілих позитивних чисел. Нехай *A*, *C* – два даних додатних цілих числа. Потрібно знайти їх найбільший загальний дільник. Якщо *C* ділить *A*, то *C* є загальним дільником чисел *C* і *A*, оскільки ділить і саме себе. Очевидно, що *C* й буде найбільшим дільником, оскільки немає числа, більшого за *C*, яке б ділило *C*. Якщо ж *C* не ділить *A*, то починаємо безперервно віднімати менше з чисел *A*, *C* з більшого до тих пір, поки не отримаємо певне число, яке націло ділить попередній від'ємник. Рано або пізно таке число утвориться, оскільки якщо різниця буде дорівнювати одиниці, то одиниця буде ділити попередній від'ємник. Отже, нехай *E* – додатний залишок від ділення *A* на *C*, *F* – додатний залишок від ділення *C* на *E*, *F* ділить *E*. Оскільки *F* ділить *E*, а *E* ділить *C* – *F*, то *F* також ділить *C* – *F*, але *F* ділить само себе, отже, *F* ділить *C*. Але *C* ділить *A* – *E*; тому *F* також ділить *A* – *E*. Але воно також ділить *E*, тому воно ділить й *A*. Отже, *F* – загальний дільник чисел *A* і *C*. Тепер можна стверджувати, що він є найбільшим таким дільником. Дійсно, якщо *F* – не найбільший загальний дільник чисел *A* і *C*, то певне більше число буде ділити їх обидва. Нехай таким числом буде *G*. Оскільки *G* ділить *C*, яке, в свою чергу, ділить *A* – *E*, то *G* ділить *A* – *E*; *G* ділить також всі *A*, отже, воно ділить й залишок *E*. Але *E* ділить *C* – *F*, тому *G* ділить *C* – *F*. Але *G* ділить також усі *C*, отже, воно ділить й залишок *F*; таким чином, більше число ділить менше, а це неможливо. Тому не існує такого числа, більшого, чим *F*, яке б ділило числа *A* і *C*, це означає, що *F* є найбільшим загальним дільником.

Крім цього відомого алгоритму, у стародавній Греції були отримані способи знаходження площ і об'ємів геометричних фігур і тіл за їх розмірами.

В давньоєгипетських папірусах можна знайти ще більш стародавні приклади обчислювальних процедур. Так, єгиптянам був відомий метод множення, заснований на подвоєнні і додаванні (до речі, на цій же ідеї базуються деякі широко застосовувані сучасні алгоритми).

Приблизно у 1800 р. до н.е. стародавній вавилонянин, ім'я якого на жаль не збереглося для історії, виклав на глиняній таблиці алгоритм розрахунку часу, який потрібний для збільшення вдвічі певного об'єму зерна, при 20% щорічному прирості (рис.1.2).

Рис. 1.2. Алгоритм розрахунку часу

На відміну від Стародавнього Єгипту у Вавилоні було прийнято крім розв'язку прикладу давати пояснення процесу обчислень, які були б придатні для розв'язання задачі у загальному випадку. Наприклад, якщо один з співмножників дорівнював одиниці, множення все-таки виконувалося. І хоч вавилонянці використовували алгоритми за півтори тисячі років до греків, історики науки одностайно визнають пріоритет Евкліда. Справа в тому, що алгоритм Евкліда має достатньо складну структуру: він є ітераційним (так називають алгоритми, що містять багаторазове повторення певних дій). Вавилонські алгоритми представляють виключно історичний інтерес, в той час як алгоритм Евкліда не втратив свого практичного значення до сьогодні.

Алгоритм аль-Каші обчислення значення x^n, де n – додатне число є ще одним стародавнім алгоритмом, який широко використовувався на практиці.

Крок 1. Введемо три величини – *N: = n*; *y: = 1*; *z: = x*. В цей момент справедливо співвідношення $x^n = y \times z^n$.

Крок 2. Ділимо N націло на 2, *N: = N div 2*; одночасно визначаємо, чи було до того *N* парним. Якщо *N* було парним, то переходимо до кроку 5.

Крок 3. Множимо *y* на *z*, *y : = y* x *z*.

Крок 4. Якщо N дорівнює нулю, то відповідь дорівнює *y*.

Крок 5. Множимо *z* на себе, *z* := *z* x *z*. Повертаємося до кроку 2.

Цей алгоритм був запропонований на самому початку XV ст. (втім, близько 200 р. до н. е. в індійському математичному трактаті «Чанда-сутра» наводиться опис аналогічного методу). Саме таку процедуру ще у XVIII ст. до н. е. єгипетські математики використовували для множення.

Крок 1. Вводимо три величини – *N: = n*; *y:= 0*; *z:= x*.

Крок 2. Ділимо N націло на 2, *N: = N div 2*; одночасно визначаємо, чи було *N* до цього парним. Якщо N було парним, то переходимо до кроку 5.

Крок 3. Збільшуємо *y* на *z*, *y: = y + z*.

Крок 4. Якщо *N* дорівнює нулю, то відповідь дорівнює *y*.

Крок 5. Додаємо *z* з собою, *z: = z + z*.

Повертаємося до кроку 2. В кроках 3 і 5 множення замінене додаванням, і в кроці 1 *y* прирівнюється не одиниці, а нулю. В результаті виконання такого модифікованого алгоритму отримуємо добуток двох чисел *n* і *x*: *y = n x x*. Це доволі зручний для практичних обчислень «вручну» спосіб множення, який зводиться до більш простих операцій подвоєння, ділення навпіл і додавання. Саме такий прийом застосовується при обчисленнях на рахівницях – в Європі він традиційно носить назву «російський селянський метод».

Математики Стародавнього Сходу винайшли десяткову систему і дали правила обчислення у цій системі.

У 820 р. арабський вчений-математик Абу Абдуллах Мухаммед ібн Муса Аль-Хорезмі (в перекладі з арабської означає «Мухамед син Муси з Хорезма»), скорочено Аль-Хорезмі, уродженець Хіви) написав твір, де вперше навів опис індійської позиційної десяткової системи числення. Від назви книги, яка арабською мовою звучала як „Аль-Джабр Ва-Аль-Мукабала" („Книга про додавання і віднімання"), виникло слово „алгебра". Ця книга відіграла велику роль у розвитку математики.

Рис. 1.3. Абу Абдуллах Мохаммед ібн Муса Аль-Хорезмі

Творча діяльність Аль-Хорезмі протікала у IX ст. головним чином у Багдаді, де на той час правив халіф Ал-Мамун, який був покровителем математики і який зібрав у створеному ним «Будинку мудрості», свого роду академії наук, багато великих учених того часу.

В одній зі своїх праць Аль-Хорезмі описав десяткову систему числення і вперше сформулював правила виконання арифметичних дій над цілими числами і простими дробами.

Арабський оригінал цієї книги Аль-Хорезмі втрачений, але є латинський переклад XII ст., за яким Західна Європа ознайомилася з десятковою позиційною системою числення і правилами виконання в ній арифметичних дій.

Аль-Хорезмі прагнув до того, щоб сформульовані ним правила були зрозумілими для усіх грамотних людей. Досягатися цього у IX ст., коли ще не була розроблена математична символіка (знаки операцій, скобки, буквені позначення тощо), було надзвичайно важко. Однак Аль-Хорезмі вдалося виробити в своїх працях такий стиль чіткого, строгого словесного припису, який не давав читачу ніякої можливості ухилитися від запропонованого викладу або пропустити певні дії.

В латинському перекладі арифметичної праці Ал-Хорезмі правила починалися словами Dixit Algorizmi (Алгорізмі сказав). В інших латинських перекладах автор йменувався Algorithmus (Алгорітмус). Поступово люди забули, що Алгорізмі – це автор правил, і стали самі ці правила називати алгоритмами. Таким чином «Алгорізмі сказав» поступово перетворилося в «алгоритм каже».

Отже, слово «алгоритм» – латинізоване ім'я Аль-Хорезмі. Як науковий термін це слово первісно позначало лише правила десяткової системи числення. Потім, протягом століть цей термін поступово набуває все більш широкого значення, позначаючи вже не тільки правила десяткової системи числення, але й будь-які точні правила дій. Врешті-решт, слово «алгоритм» стало науковим терміном, який визначає одне з фундаментальних понять сучасної математики і інформатики.

Аль-Хорезмі сформулював правила обчислень у новій системі і, імовірно, вперше використав цифру 0 для позначення пропущеної позиції у запису числа (її індійську назву араби переклали як *as-sifr* або просто *sifr*, звідси такі слова, як цифра і шифр). Приблизно у цей же час, індійські цифри почали застосовувати й інші арабські вчені.

У першій половині XII ст. книга Аль-Хорезмі у латинському перекладі потрапила до Європи. Перекладач, ім'я якого на жаль не збереглось для історії, дав їй назву „Algoritmi de numero Indorum" („Індійське мистецтво лічби, створене Аль-Хорезмі"), а звідси з'явився термін „алгоритм" від імені автора книги Аль-Хорезмі, яке у латинських перекладах книги перетворилося у *algorismi*.

Рис. 1.4. Сторінка Алгебри Аль-Хорезмі

У своїй праці „Індійське мистецтво лічби" аль-Хорезмі сформулював правила запису натуральних чисел за допомогою арабських цифр і правила дій над ними стовпчиком. Крім того, сформулював правила виконання чотирьох арифметичних дій над багатозначними числами. Так наприклад, дія додавання двох багатозначних чисел розкладається на ланцюжок елементарних операцій, при здійсненні кожної з яких, виконавець бачить тільки дві відповідні цифри доданків (одна з яких може бути з поміткою, яка нагадує про перенос одиниці).

Ці операції бувають двох типів:

1) запис відповідної цифри суми;
2) позначка про перенос над сусідньою зліва цифрою.

Правило або алгоритм приписує належний порядок виконання цих операцій – справа наліво. Формальний характер цих елементарних операцій полягає у тому, що вони можуть бути виконані автоматично зараз і назавжди заданій таблиці додавання цифр, при повному відволіканні від їх змістовного змісту.

Аналогічно обстоїть справа з іншими трьома арифметичними діями, а також діями з добування квадратного кореня.

Алгоритми, у яких основну роль відіграють чотири арифметичних дії, називають числовими алгоритмами. Вони відіграють важливу роль у різноманітних галузях як елементарної, так і вищої математики і задаються зазвичай, у вигляді словесних приписів або різноманітних формул й схем.

Низка важливих для обчислювальної техніки алгоритмів були розроблені у XVIII – XIX ст. І. Ньютоном, Л. Ейлером, К.Ф. Гаусом – видатними математиками свого часу. Ці алгоритми не втратили свого значення до сьогодні.

Якщо проаналізувати усі названі або відомі алгоритми, то можна встановити, що вони поділяються на низку більш простих дій, які, у свою чергу, можуть бути представлені ще більш простими, упритул до елементарних операцій – арифметичних або логічних, тобто таких, які можуть використовуватись комп'ютером.

На сучасному етапі розвитку обчислювальної техніки інтерес до алгоритмів істотно виріс, що зумовлено передусім можливостями використання комп'ютерів у техніці, економіці, наукових дослідженнях. Справа у тому, що комп'ютер під час роботи виконує задану програму, а програма є певним алгоритмом, записаним у спеціальних позначеннях, зрозумілих комп'ютеру. Як відомо, програма – це логічно упорядкована послідовність команд, необхідних для управління комп'ютером (виконання ним конкретних операцій), тому процес програмування зводиться до створення послідовності команд, необхідних для розв'язання певної задачі. Відповідну систему позначень називають мовою програмування. Комп'ютерні програми створюють програмісти – люди, які навчились процесу їх складання (програмуванню). Але треба пам'ятати, що програміст – це не той, хто пише програми, а той, чиї програми працюють.

***Мова програмування** – це сукупність засобів і правил подання алгоритму у вигляді, придатному для виконання на комп'ютері*.

До 30-х років XX ст., поняття алгоритму залишалося інтуїтивно зрозумілим, що мало скоріше методологічне, а не математичне значення. Задача точного визначення алгоритму була розв'язана у 30-х роках XX ст. у працях Гілберта, Черча, Кліні, Поста, Тюрінга на основі поняття рекурсивної функції і на основі опису алгоритмічного процесу.

В подальшому, ***алгоритмом почали називати точний припис, який визначав послідовність дій і забезпечував при цьому потрібний результат з вихідних даних.*** Алгоритм може бути призначений для виконання його людиною або автоматичним пристроєм. Побудова алгоритму, навіть найпростішого є творчим процесом. Він доступний виключно живим істотам, при чому, як вважали досить тривалий час, тільки людині. Інша річ – реалізація алгоритму. Її можна доручити суб'єкту або об'єкту, яким не потрібно вникати у суть справи, і які взагалі, часом принципово неспроможні зрозуміти сутність задачі. Такий суб'єкт або об'єкт прийнято називати *формальним виконавцем*.

Формальний виконавець – пристрій, який зчитує програму, змінює її у відповідності з нею свій внутрішній стан, і виконує команди, здійснюючи покрокову обробку інформації. Щоб задати формального виконавця достатньо визначити:

а) систему команд;

б) форму подання вхідної і вихідної інформації;

в) множину внутрішніх станів; г) мову програмування.

Прикладом формального виконавця може слугувати пральна машина-автомат, яка неухильно виконує запропоновані їй дії, якщо ви навіть забули засипати пральний порошок. Людина також може виступати у ролі формального виконавця, але значно частіше формальними виконавцями виступають різноманітні автоматичні пристрої, у тому числі й комп'ютер.

Кожний алгоритм будується у розрахунку на конкретного виконавця. Ті дії, які може здійснювати виконавець, називаються його припустимими діями. Алгоритм повинен містити тільки ті дії, які припустимі для даного виконавця.

***Алгоритм** – це точно визначена послідовність дій для певного виконавця, які виконуються за певними правилами і які приводять через певну кількість кроків до розв'язання задачі*.

Виконавець алгоритмів призначає елементарні дії, з яких формується алгоритм. ***Окремі дії, які складають алгоритм, називаються операціями.*** При цьому під операцією розуміється як і одинична дія, наприклад, додавання, так і група взаємозалежних дій.

Головними особливостями будь-якого алгоритму є можливість розв'язання задачі в узагальненому вигляді, а також здатність виконувати дії з вирішення задач для конкретних значень (не тільки людиною, а й різноманітними технічними пристроями (виконавцями)).

В математиці часто виникають так звані алгоритмічні проблеми, коли потрібно знайти єдиний метод (алгоритм) для розв'язку будь-якої задачі з даного класу однотипних задач. Алгоритмічні проблеми виникали і вирішувалися у різних галузях математики протягом усієї її історії. Однак у перше десятиліття XX ст. накопичилось багато класів задач, для розв'язку яких алгоритми знайти не вдавалося. Зусилля, що докладались до знаходження алгоритмів розв'язання цих класів задач, навели на думку: а раптом для того або іншого класу просто неможливо знайти алгоритм, тобто шуканий алгоритм взагалі не існує?

Якщо це дійсно так, то необхідно припинити пошук, тому що безперспективно шукати те, чого немає. Але тоді виникає необхідність доказу неіснування шуканого алгоритму. А щоб довести неіснування алгоритму для розв'язання певного класу задач, потрібно вже точно знати, що таке алгоритм, тобто потрібно перевести інтуїтивне поняття алгоритмів в розряд точних, математичних понять.

Саме тому в рамках математичної логіки в 30-х рр. XX ст. було розроблене точне визначення поняття алгоритму, причому були побудовані різні варіанти математичного уточнення цього поняття, іншими словами, різні математичні теорії алгоритмів. Завдяки цьому вдалося виявити нерозв'язні алгоритмічні проблеми, тобто неіснування відповідних алгоритмів. Тепер, коли не вдається знайти алгоритм для певного класу задач, можна спробувати довести, що шуканий алгоритм не існує.

При розв'язанні складних задач виконавцем яких буде ЕОМ, складання алгоритму розв'язку задачі є необхідним етапом, який деталізує метод розв'язання задачі для подальшого програмування. Програма здійснює ще більш всебічну деталізацію розв'язання задачі і його візуалізацію.

Поняття алгоритму близьке до інших понять, таких, як *метод* (метод Гауса розв'язання систем лінійних рівнянь), *спосіб* (спосіб побудови трикутника за трьома сторонами за допомогою циркуля і лінійки) або навіть *рецепт*. Інші поняття, такі, як *процедура* (судова процедура), *програма* (програма дій уряду з виводу економіки з кризи), *процес* (мирний процес на Близькому Сході), також мають чимало спільного з алгоритмом. Тому необхідно чітко сформулювати відмітні особливості саме алгоритмів.

1.2. Визначення та властивості алгоритму

ДГСТ 19781-74 визначає, що *„алгоритм – це точний припис, який визначає обчислювальний процес, що йде від варійованих початкових даних до кінцевого результату”.*

Алгоритм передбачає наявність вихідних даних і певного результату.

Алгоритм – це точно визначена інструкція, послідовно застосовуючи яку до вихідних даних, можна отримати розв'язок задачі.

Багато алгоритмів володіють тією властивістю, що залишаються правильними для різних наборів вихідних даних (приготування супу однаково для різних використовуваних овочів; обчислення площі прямокутника однаково для будь-яких значень *a* і *b*). Таку властивість алгоритмів називають *масовістю.* Вона дозволяє багаторазово застосовувати один і той же алгоритм для розв'язку задачі. При цьому необхідно мати на увазі, що для кожного алгоритму є певна множина об'єктів, припустимих вихідних даних. Наприклад, *a* і *b* можуть бути тільки дійсними позитивними числами; в алгоритмі ділення дійсних чисел ділене може бути будь-яким, а дільник не повинен дорівнювати нулю.

А чи завжди вдається отримати результат, застосовуючи алгоритм до певного набору з множини припустимих вихідних даних? Ні, не завжди: так, при діленні 7 на 3 обидва цілих числа є припустимими, але результат 2,333... є нескінченним десятковим дробом. Про алгоритми такого роду кажуть, що вони є *потенційно виконуваними*, але складаються з нескінченної кількості кроків. У зв'язку з цим потрібно ввести поняття *скінченності алгоритму*. Зміст його полягає у тому, що виконання алгоритму повинне обов'язково приводити до його завершення. Одночасно можна навести приклади формально нескінченних алгоритмів, які широко застосовуються на практиці. Наприклад, алгоритм роботи системи збору метеорологічних даних полягає у безперервному повторенні послідовності дій „Виміряти температуру повітря” і „Визначити атмосферний тиск”, які виконуються з певною частотою (щохвилини, щогодини або щодня) під час існування даної системи.

На практиці зазвичай цікавить результат, який видається алгоритмом через кінцеву кількість кроків і за кінцевий час (дійсно, яка користь від алгоритму розрахунку прогнозу погоди на завтра, якщо результат обчислюється тиждень). Властивість скінченності має особливо важливе значення, оскільки часто потрібна не потенційна, а саме реальна виконуваність алгоритму. Повернемося до прикладу з діленням. Якщо обірвати процес на певному кроці ділення 7 на 3, то отримане число, взагалі кажучи, не буде часткою від ділення, проте його можна прийняти за наближений результат, обчислений з задовільною для конкретної задачі точністю.

Алгоритм відноситься до основних понять математики, і тому не має чіткого визначення. Найчастіше це поняття формулюють наступним чином: ***„точний припис про порядок виконання дій, із заданої фіксованої множини, для розв’язання будь-яких задач заданого класу".***

Розглянемо докладніше ключові слова у цьому визначенні:

- ***„із заданої фіксованої множини”*** означає, що множина дій, які використовуються у припису, оговорена заздалегідь і не може змінюватись в процесі виконання алгоритму;
- ***„точний припис”*** означає, що припис однозначний й однаково розуміється усіма виконавцями алгоритму і при однакових вихідних (початкових) даних, будь-який виконавець завжди отримає один і той же результат;
- ***„розв’язання будь-яких задач заданого класу”*** означає, що цей припис призначений для розв’язку класу задач, а не однієї окремої задачі. Пізніше ми докладніше розглянемо зміст виразу *„клас задач”.*

Для багатьох задач можна побудувати послідовність операцій їх розв’язання. Якщо цю послідовність можна використати для розв’язання подібних задач, то вона наближається до алгоритму. Можна стверджувати, що в процесі формального розв’язання задачі, рішення спочатку описується мовою математики у вигляді формул, а потім на мові алгоритмів у вигляді певного процесу, у якому використовуються попередньо визначені математичні формули й умови їх виконання.

Властивості алгоритму:

❶. ***Дискретність – розчленованість алгоритму на окремі елементарні операції (дії, директиви, кроки), можливість виконання яких не викликає сумніву у виконавця і які здійснюються за певними командами.***

Алгоритм повинен представляти процес розв’язання задачі як послідовне виконання простих або заздалегідь визначених закінчених дій (кроків). Тобто, кожна дія, яка передбачена алгоритмом, виконується тільки після того, як завершилось виконання попередньої. Виконавець не може перейти до виконання наступної операції не закінчивши повністю виконання попередньої. Кожен крок – це одна дія або підпорядкований алгоритм (метод покрокової деталізації). Припис алгоритму потрібно виконувати послідовно, один за одним, у відповідності із зазначеним порядком їх запису. Виконання усіх приписів гарантує правильне розв’язання задачі.

***Команда* – вказівка виконавцю на здійснення певної дії.** Команди виконуються одна за одною. Після кожного кроку виконання алгоритму точно відомо, яка команда буде наступною. Алгоритм представляє послідовність команд (інструкцій, директив), які визначають дії виконавця (суб'єкта або керованого об'єкта).

Таким чином, виконуючи алгоритм виконавець може не вникати у зміст того, що він виконує і разом з тим отримувати потрібний результат. У цьому випадку ведуть мову про формальні дії виконавця, тобто він не відволікається від змісту поставленої задачі і тільки строго виконує певні правила (інструкції). Це дуже важлива особливість алгоритмів. Наявність алгоритму формалізує процес, виключає необхідність будь-яких міркувань при його виконанні.

Отже, створення алгоритму дає можливість розв'язувати задачу формально, механічно виконуючи команди алгоритму у зазначеній послідовності.

Побудова алгоритму для вирішення задачі з якої-небудь предметної галузі вимагає від людини глибоких знань у цій галузі, буває пов'язане з ретельним аналізом поставленої задачі, складними, іноді з дуже громіздкими міркуваннями. На пошуки алгоритму розв'язання деяких задач вчені витрачають багато років. Але коли алгоритм створений, розв'язання задачі за готовим алгоритмом вже не вимагає жодних міркувань і зводиться до строгого виконання команд алгоритму.

❷. ***Зрозумілість* – представлення алгоритму у вигляді, зрозумілому для виконання.** Усі дії, символи операцій в алгоритмі повинні бути або загальноприйнятими, або попередньо чітко й однозначно визначеними. Не допускаються двозначності, неоднозначності. Наприклад, послідовність дій: *y=a#b; z=y@* – це не алгоритм, оскільки операції зі знаками # і @ не визначені.

❸. ***Детермінованість* – однозначна визначеність результату кожної операції при заданих початкових даних, незалежно від виконавця.** Кожне правило виконання алгоритму повинно бути чітким, однозначним й не залишати місця для свавілля. Завдяки цій властивості, виконання алгоритму носить механічний характер і не потребує жодних додаткових вказівок або відомостей про розв'язувану задачу. Алгоритм повинен бути зрозумілим не тільки автору але й виконавцю. Якщо запропонувати певному студенту приготувати торт, то зазвичай у нього нічого не вийде, оскільки що він цього не вміє робити. Але якщо скласти йому детальний алгоритм процесу виготовлення, розбити його на такі елементарні кроки, щоб він зрозумів і зміг виконати кожний крок, то у цьому випадку є висока вірогідність успішного приготування цього делікатесу.

Таким чином, *правильно побудований обчислювальний процес повинен залежати тільки від початкових (вхідних) даних. При однакових початкових даних результат за заданим алгоритмом повинен бути завжди однаковим*.

❹. ***Масовість (універсальність) – можливість застосування алгоритму до будь-яких вхідних даних задач виділеного класу.*** Практична цінність алгоритму не дуже велика, якщо його можна використати тільки один раз.

❺. ***Скінченність* – алгоритм повинен приводити до вирішення задачі за певну кількість кроків.**

❻. ***Результативність* – виконання дій певного алгоритму має завершитися результатом**. Нескінченний процес в інформатиці називається *зацикленням.* Правильно побудований процес виконання дій комп'ютером повинен привести до результату через скінченну кількість кроків.

Приклад властивостей алгоритму представлені в табл.1.1.

Таблиця 1.1.

Властивості алгоритму

Властивості	**Приклад виконання властивості**	**Приклад невиконання властивості**
дискретність	Стрічувати не можна, помилувати !	Стрічувати не можна помилувати!
зрозумілість	Інструкція українською або російською мовами з малюнками	Інструкція японською мовою без малюнків
детермінованість	На дубі скринька, а у скриньці качка, а у качці яйце, а в яйці голка, а в голці смерть Кащея	Піди туди, не знаю куди, принеси то, не знаю що
масовість	Кожній дочці батько привіз подарунок	Депутат турбувався про добробут виборців
результативність	Мама зварила відмінну кашу у горщику	Каша вже заповнила усі вулиці, а горщик все варив кашу

Поняття алгоритму визначене переліком властивостей 1–6 (табл..1.1), не можна вважати строгим, оскільки при формулюванні властивостей використані терміни *„величина", „спосіб", „простий", „локальний"*, точний зміст яких не встановлений. Тому, дане визначення будемо називати *нестрогим* (або *інтуїтивним*) поняттям алгоритму.

Одна з задач алгоритмізації полягає у тому, щоб прослідкувати перетворення інформації за ланцюжком: ***задача ⇨ алгоритм ⇨ програма ⇨ комп'ютер ⇨ результат розв'язання.***

Задача ..... формулюється на звичній природній людській мові у вигляді формул, співвідношень, залежностей. Це етап постановки й формалізації задачі.

Алгоритм .....розроблюється і записується одним із способів запису на одній з формальних мов. Етап називають етапом розробки і запису алгоритму.

Програма .....це той же алгоритм, але записаний на мові, зрозумілій комп'ютеру – мові програмування.

Комп'ютер.....перекладає введену програму з мови програмування на внутрішню мову комп'ютера – мову машинних команд й розв'язує задачу. Це етап розв'язання задачі.

Результат.....видається комп'ютером у передбачуваному програмою вигляді і аналізується користувачем.

Структурний взаємозв'язок основних понять та елементів, що складають суть алгоритму представлено на рис.1.5.

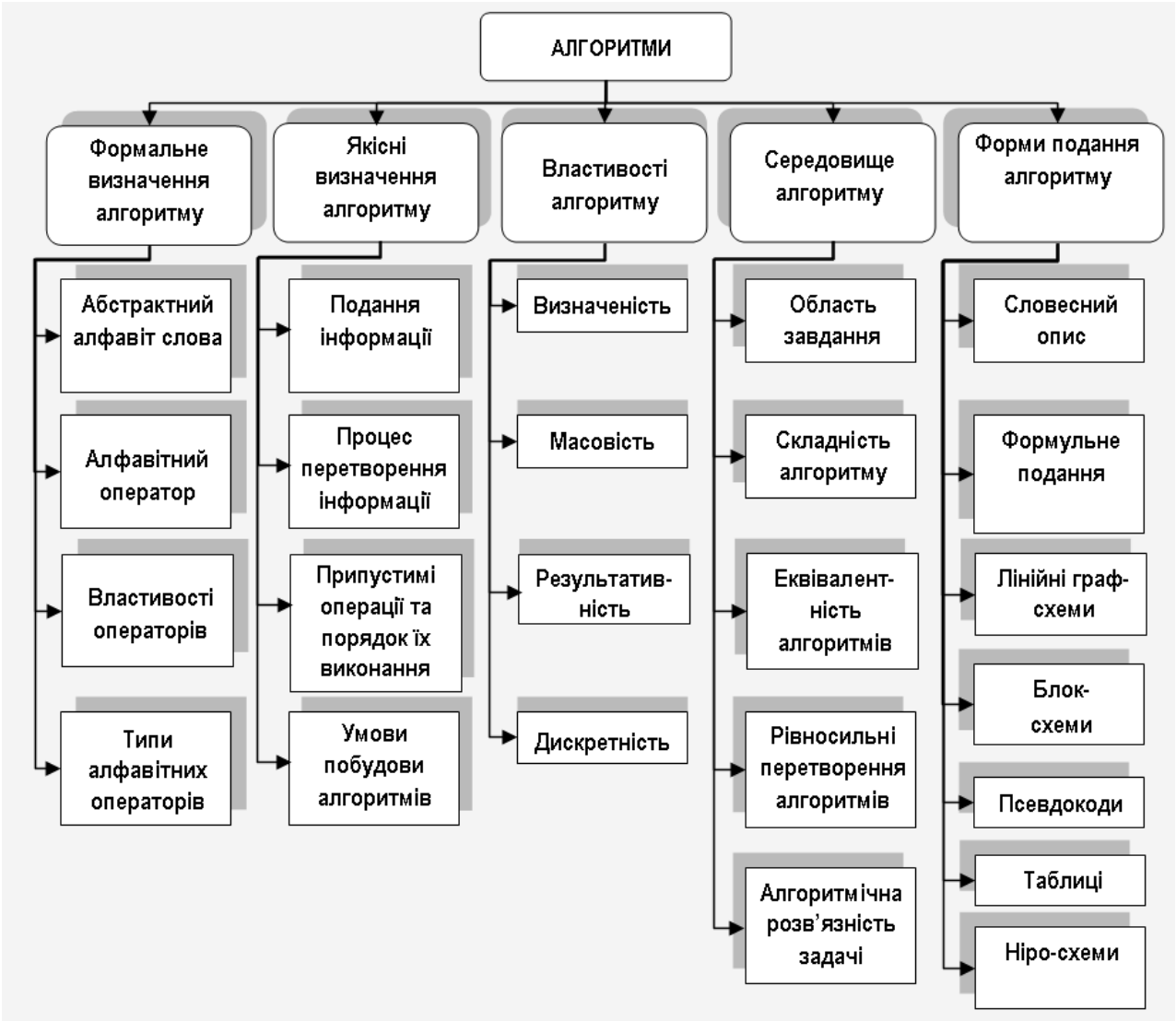


Рис. 1.5. Структурний взаємозв'язок основних понять та елементів, що становлять суть алгоритму

1.3. Характеристики алгоритму

З поняттям алгоритму пов'язані такі поняття, як область його визначення, складність, еквівалентність, алгоритмічна розв'язаність тощо.

Область визначення алгоритму *– це множина даних, до яких застосовується алгоритм.* Якщо алгоритм розрахунку завершується без результату або продовжується необмежено довго, то кажуть про незастосовність алгоритму до цих вхідних даних.

Складність алгоритму *– це величина, яка характеризує довжину опису алгоритму*. А для оцінки складності обчислень, які виконуються у даному алгоритмі, використовується так звана „сигнальна функція".

Алгоритмічна розв'язність *масової проблеми – можливість побудови алгоритму розв'язання всіх задач даного класу.*

Існують класи задач, для розв'язання яких взагалі не існує єдиного універсального способу. Це алгоритмічно нерозв'язні проблеми. Однак це не означає, що неможливо розв'язати навіть окремі задачі даного класу. На практиці часто буває так, що для певного класу задач, для якого доведена неможливість існування алгоритму, який розв'язує усі задачі цього класу, *завжди можна побудувати алгоритм, який розв'язує майже усі такі задачі.* Термін „майже усі" використано у розумінні, що ймовірність зустріти на практиці конкретну задачу цього класу, яка не розв'язується за допомогою побудованого алгоритму є достатньо малою.

Для визначення алгоритмічного розв'язання задач певного класу необхідно або побудувати алгоритм розв'язання, або довести неможливість побудови такого алгоритму, тобто довести, що проблема є алгоритмічно нерозв'язною.

Наприклад, алгоритмічно розв'язувана проблема – доведення тотожностей у алгебрі (відомі правила перетворення алгебраїчних виразів), у той самий час розв'язання диференційних рівнянь – проблема алгоритмічно нерозв'язна. Є проблеми, про які невідомо, чи є вони алгоритмічно розв'язні або нерозв'язні. Це свідчить про те, що на даний час, учені не в змозі побудувати алгоритм або довести неможливість його побудови, бо то є задачі одного рівня складності.

1.4. Принципи побудови алгоритму

Щоб побудувати алгоритм, необхідно дотримуватись певних умов:

- вхідні та вихідні дані задати у вигляді послідовності слів;
- процес розв'язання задачі це є процес перетворення вхідних даних у вихідні. Процес перетворення складається із сукупності елементарних припустимих операцій формального характеру.

Припустима елементарна операція ***– це проста, чисто механічна дія, результат якої не залежить від виконавця*** (машини чи людини);

- послідовність припустимих операцій не залежить від конкретних вхідних даних;
- порядок виконання припустимих операцій визначається однозначно;
- сукупність припустимих операцій визначається класом задач та типом даних.

Розв'язання будь-якої задачі є творчим процесом, який складається з декількох послідовних етапів. До них відносяться :

I. Аналіз постановки задачі та її предметної області

1. Розуміння постановки і вимог початкової задачі, визначення предметної області, для якої поставлена задача.
2. Аналіз предметної області, виявлення даних, які фіксують вхідну і вихідну інформацію (визначення їх структури і властивостей), визначення відношень між даними, умов та обмежень, які накладаються на ці відношення.

II. Формальне моделювання розв'язання задачі

3. Вибір і застосування формальної системи для опису моделі предметної області і розв'язання задачі.
4. Формування основної ідеї, вибір методів розв'язання задачі.
5. Визначення технологій, засобів і виконавця розв'язання задачі, побудова алгоритмів, що реалізують обрані методи.

III. Практичне розв'язання

6. Застосування обраних методів і засобів для розв'язання задачі.
7. Аналіз отриманих результатів.

Вищеперераховані етапи орієнтовані для отримання рішення не окремо взятої конкретної задачі, а певного класу задач. Етап побудови алгоритмів, що реалізує обрані методи розв'язання задачі, деталізує й візуалізує процес їх розв'язку. Алгоритмізація дозволяє вже на цьому етапі оцінити ефективність розв'язання задачі, уточнити методи розв'язання для різноманітних потоків вхідних даних і виявити можливі помилки.

У цій послідовності найбільш трудомістким й рутинним є етап застосування обраних методів і засобів для розв'язання задачі. На даний час найбільше поширеним засобом для розв'язання задач є електронна обчислювальна машина (ЕОМ, комп'ютер). Застосування обраних методів і алгоритмів для розв'язання на ЕОМ включає подальшу деталізацію її розв'язку за рахунок опису послідовності застосовуваних операцій у вигляді програми для ЕОМ. Це додає процесу розв'язання не тільки візуальність, але й інтерактивність.

Однак не всі задачі, що розв'язуються за допомогою ЕОМ, вимагають складання складних програм. Зокрема, задачі обчислень в електронних таблицях або задачі пошуку і вибірки даних у базах даних завдяки впровадженню новітніх інформаційних технологій взагалі не потребують програмування у класичному значенні цього слова, що істотно розширює сферу використання комп'ютерів. Однак при розв'язанні цих задач необхідне виконання вищенаведених етапів.

1.5. Аналіз постановки задачі та її предметної області

На першому етапі уточнюється постановка задачі, після чого виявляються окремі явища, об'єкти, процеси, їх зв'язки і залежності предметної області. Тут визначаються такі поняття як початкові (вхідні) і вихідні дані, які абстрактно представляють інформацію про процеси предметної області реального світу, а також пояснюються яким чином з початкових даних можуть бути отримані результати.

Початкові дані повинні бути повними – тобто необхідну і достатню їх кількість для розв'язання задачі. Якщо дані неповні, необхідно прикласти додаткові зусилля для збору додаткових даних; ця ситуація може також виникнути на подальших етапах при виборі методу розв'язання.

Розрізняють початкові дані трьох видів: *постійні*, *умовно-постійні* і *змінні*.

***Постійні початкові дані** – це дані, які зберігають свої значення у процесі розв'язання задачі (математичні константи, координати нерухомих об'єктів) і не залежать від зовнішніх факторів.*

***Умовно-постійні дані** – це дані, які іноді можуть змінювати свої значення; причому ці зміни не залежать від процесу розв'язання задачі, а зумовлені зовнішніми факторами (величина прибуткового податку, курс валют, кількість днів у поточному році тощо).*

***Змінні дані** – це дані, які змінюють свої значення у процесі розв'язку задачі.*

На цьому етапі важливо не тільки класифікувати дані за відношенням до процесу розв'язання, але визначити їх найменування, тип, структуру і обмеження, які накладаються на значення. Бажано також визначити припустимі й неприпустимі операції за відношенням до різноманітних типів початкових даних.

На рис. 1.6 представлена класифікація даних за структурною ознакою.

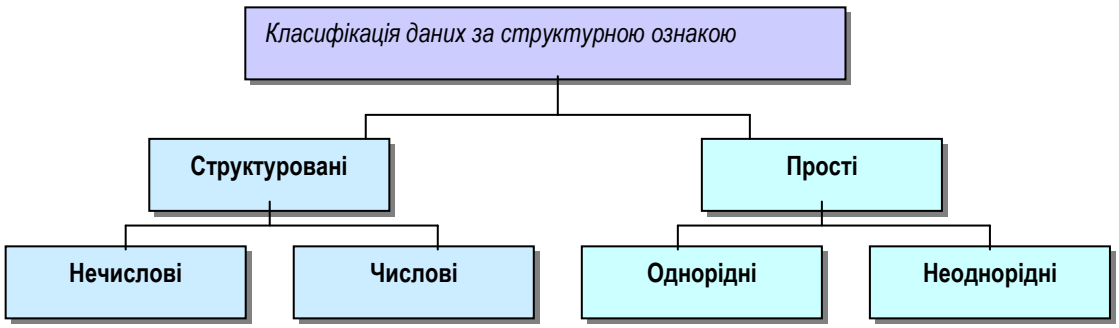


Рис. 1.6. Класифікація даних

Дані відносять до простого типу, якщо у кожний момент часу вони визначають одне й тільки одне значення. Діапазон зміни можливих значень, визначається типом даних. Наприклад, потрібно обчислити площу поверхні певного тіла. Очевидно, що для отримання інформації про площу обчислення поверхні будь-якого тіла, достатньо використовувати дані простого числового типу. Прості дані визначають таке відношення: „одне ім'я – одне значення".

Структуровані дані відрізняються від простих тим, що до них застосовано інше відношення: „одне ім'я – багато значень". Якщо всі елементи, що входять у таку структуру є однотипними, то така структура називається однорідною, у протилежному випадку – неоднорідною. Класичним прикладом однорідної структури є масив, який є послідовністю однотипних значень, наприклад, [2,3,7,51]. Неоднорідна структура на відміну від однорідної, містить значення різноманітних типів, які відносяться до одного поняття або об'єкту. Це означає, що такі структуровані дані несуть в собі більше інформації. Для представлення неоднорідних структур використовують запис.

***Запис** – це структура, яка призначена для подання різноманітних даних.* Запис складається з іменованих полів, кожне з яких повинно містити значення певного типу. Розглянемо простий приклад. Задача полягає у визначенні у певній країні міста з максимальною кількістю жителів. Дані, які необхідно проаналізувати, це атрибутивні (нечислові), які містять інформацію про назву міста, і числові дані, які містять інформацію про чисельність населення у цьому місті. Для вибору структури, яка містить дані про назву міста і кількість жителів у ньому, потрібно вибрати неоднорідну структуру – запис, зразок якого представлений в табл. 1.2.

Таблиця 1.2.

Ім'я поля: *Місто*	Ім'я поля: *Кількість жителів*
Тип поля: *Рядок символів*	Тип поля: *Число*
Значення: Київ	Значення: 2 *578 676*

Для структури, яка містить інформацію про множину досліджуваних міст країни, також можна вибрати однорідну структуру типа масив, що складається з записів таблиці 1.2.

Визначення відношень між даними, умов і обмежень, які накладаються на значення даних і ці відношення, залежать від конкретної постановки задачі і вимог користувача.

У результаті аналізу, постановка і вимоги задачі можуть бути представлені в узагальненому вигляді.

1.6. Формальне розв'язання задачі

Після проведення аналізу постановки задачі, виявлення даних, їх структури і відношень між ними є можливість приступити до побудови формальної моделі. Це найбільш важливий етап у процесі розв'язання задачі.

***Модель** – це спрощене подання реального об'єкту, процесу або явища.*

***Моделювання** – це побудова моделі для дослідження об'єкта (процесу, явища), що моделюється з метою отримання нової інформації про цей об'єкт (процес, явище)*.

Складовими такої моделі повинні бути набір елементарних кроків, засоби визначення наступних кроків тощо.

Для опису моделі предметної області розв'язуваної задачі необхідно обрати певну формальну систему. Виходячи з постановки задачі можна відразу визначити один чи декілька видів моделей, придатних для опису і моделювання розв'язання вашої задачі: математичні, геометричні, структурні, логічні тощо.

Найбільш поширеними і добре вивченими є математичні моделі, які описують залежності між даними числового типу. Наприклад, для математичної моделі зірки, можна використовувати систему рівнянь, які описують процеси, що відбуваються у надрах зірки. Математичною моделлю іншого роду є математичні співвідношення, які дозволяють розрахувати оптимальний план роботи підприємства.

До основних переваг математичних моделей безумовно відносяться добре вивчені і широко використовувані математичні методи розв'язання досить великого класу задач, що значно полегшує формування основної ідеї і вибір методів розв'язання задачі.

Приступаючи до розробки моделі, потрібно спробувати вирішити задачу для конкретних вхідних даних, а потім узагальнити отриманий розв'язок на основі його аналізу для будь-яких значень вхідних даних. Перед тем як визначити розв'язання задачі для конкретних вхідних даних доцільно знайти відповіді на наступні питання:

1) Чи існують розв'язки аналогічних задач?

2) Яка математична модель найбільш підходить для розв'язання цієї задачі?

Приклад.

Постановка задачі. Потрібно визначити придатність даної аудиторії для проведення навчальних занять.

Розв'язання.

Етап 1. Аналіз постановки задачі і її предметної області.

В результаті аналізу предметної області, виявляємо, що ця предметна область пов'язана з освітнім процесом. Постановка задачі може бути сформульована наступним чином. Необхідно визначити, чи підходить чи ні певна аудиторія для проведення занять групи студентів, при забезпеченні необхідної норми площі для кожного студента.

Введемо позначення для вхідних і вихідних даних. Початкові дані повинні бути представлені простими змінними значеннями числового типа: *A* – ширина аудиторії, *B* – її довжина, *K* – кількість студентів в групі, *N* – припустима мінімальна кількість квадратних метрів для одного студента (норма), *M* – кількість столів в аудиторії.

Вихідними даними будуть виступати повідомлення: „Аудиторія може бути використана для проведення навчальних занять" і „Аудиторія не може бути використана для проведення навчальних занять".

Етап 2. Формальне рішення.

Визначимо відношення між вхідними і вихідними даними. Для цього уведемо проміжні дані числового типу: *S* – площа аудиторії, *C* – необхідна за нормами площа для проведення занять для групи з *K* студентів, *D* – необхідна кількість столів для навчання групи, що складається з *K* студентів. Опишемо співвідношення між вхідними і вихідними даними, використовуючи математичні залежності.

Математична модель:

$$S = A*B,$$

$$C=N*K, \quad S>=C, \quad K<=D.$$

Таким чином, побудована математична модель задачі, що розв'язується.

Питання для самоконтролю

1. *Звідки з'явилась назва терміну „алгоритм"?*
2. *Що розуміють під терміном „чисельні алгоритми"?*
3. *Що розуміють під терміном „мова програмування"?*
4. *Що розуміють під терміном „алгоритм"?*
5. *Що розуміють під терміном „операція"?*
6. *Дайте характеристику властивостей алгоритму.*
7. *Які характеристики алгоритму Вам відомі?*
8. *Назвіть принципи побудови алгоритму.*
9. *В чому полягає аналіз постановки задачі та її предметної області?*
10. *Що розуміють під терміном „модель"?*
11. *Що розуміють під терміном „моделювання"?*

2. ФОРМИ ТА ЗАСОБИ ПОДАННЯ АЛГОРИТМІВ

„Зло помилки не в тому, що вона буває, а в тому,
що люди, які зробили помилку, не намагаються її виправити ”.
Конфуцій

Алгоритм – це система правил, яка сформульована на мові, зрозумілій виконавцеві алгоритму, визначає процес переходу від припустимих вихідних даних до певного результату і володіє властивостями масовості, скінченності, визначеності, детермінованості. Це визначення алгоритму не є строгим (хоча б тому, що в ньому використовуються не точно визначені терміни, наприклад „правило”). Протягом багатьох століть поняття алгоритму пов'язувалося з числами і відносно простими діями над ними. Так й сама математика була більшою частиною наукою про обчислення, наукою прикладною. Найчастіше усього алгоритми представлялися у вигляді математичних формул. Порядок елементарних кроків алгоритму задавався розстановкою скобок, а самі кроки полягали у виконанні арифметичних операцій і операцій відношення (перевірки рівності, нерівності тощо). І хоч ці формули могли бути достатньо громіздкими, а обчислення вручну – вкрай трудомісткими (французький астроном Жан Жозеф Леверьє провів десятки років в розрахунках орбіт планет Сонячної системи, в результаті чого йому поталанило виявити невідому раніше планету Нептун), суть самого обчислювального процесу залишалася цілком очевидною. У математиків не виникала потреба в усвідомленні і строгому визначенні поняття алгоритму, в його узагальненні. Але з розвитком математики з'являлися нові об'єкти, якими доводилося оперувати ученим: вектори, матриці, графи, множини тощо. Як визначити для них однозначність або як встановити скінченність алгоритму, які кроки можуть вважатися елементарними (наприклад, чи є таким обернення матриці або знаходження перетинання двох множин)? Виникла ідея про існування алгоритмічно нерозв'язних проблем, таких, для яких неможливо знайти процедуру розв'язку. Отже, потрібно було навчитися математично строго доводити факт відсутності відповідного алгоритму. Це можна зробити тільки у випадку створення строгого визначення алгоритму. Спроби виробити таке визначення й привели на рубежі 20–30-х рр. XX ст. до виникнення теорії алгоритмів.

2.1. Загальні поняття про алгоритмізацію

Як ми вже попередньо вияснили, алгоритм описує розв’язання задачі у вигляді точно визначеної послідовності дій для певного виконавця, з перетворення початкових даних у вихідні (результуючі). Оскільки однієї цілі можна досягти різними способами (за рахунок різних зусиль й витрат), то відповідно й алгоритмів досягнення цієї мети можна побудувати багато.

Процес складання алгоритмів називають алгоритмізацією. Алгоритм повинен містити необхідну і достатню інформацію про хід здійснення процесу розв’язання конкретної задачі (проблеми). Один і той же процес можна планувати і розписувати з різним ступенем деталізації. Так наприклад, директора заводу може цікавити, як здійснюється виробництво продукції на рівні взаємодії цехів, бригадира ж конкретного цеху більше цікавить, чим займається кожен робітник його бригади. Зайва деталізація алгоритму для директора буде заважати у керівництві заводом, як і нестача інформації про робітників, не дозволить бригадиру організувати ефективну роботу бригади.

Очевидно також, що певний виконавець може сприймати алгоритм, виконувати його, тільки у тому випадку, коли алгоритм представлений у зрозумілому йому вигляді. Це означає, що будь-яке подання алгоритму є певним інформаційним блоком, тобто подання алгоритму є інформацією, і вона розташована на певному носії інформації.

Таким чином, сам алгоритм є певна абстракція але реальна його реалізація можлива тільки у вигляді його подання. Очевидно також, що у будь-якого алгоритму можуть бути різні подання. Вони можуть бути схожими, однак можуть і не мати нічого спільного, крім реалізації одного алгоритму.

В інформатиці склалися цілком визначені традиції у поданні алгоритмів розрахованих на різних виконавців. Засоби, які використовуються для запису алгоритмів у значному ступені визначаються тим, для якого виконавця призначається алгоритм. Але будь-яка форма запису (подання) алгоритму повинна забезпечувати властивості алгоритму: дискретність, зрозумілість, детермінованість, масовість, скінченність та зрозумілість.

Вибір способу запису алгоритму залежить й від характеру задачі. Алгоритм обчислювального процесу можна записати формулою (послідовністю формул) або вербально (словами, природною людською мовою). Зрозуміло, що алгоритм заварювання чаю мабуть зручніше записати вербально, у пронумерованих пунктах, а ось алгоритм розв’язання квадратного рівняння буде більш зрозумілим при його запису комбінацією слів і формул.

Якщо з алгоритмом працює людина, то це може бути й традиційна людська мова (англійська, японська або українська), мова малюнків, мова символів тощо. Однак, досить часто використання цих природних засобів спілкування, веде до непорозуміння. Іноді причина криється в неоднозначності використовуваної термінології. Наприклад, речення: „*Відвідування онуків – це велике навантаження на нервову систему*" може мати подвійний зміст: або приїзд онуків викликає дуже багато турбот, або поїздка до них є серйозним іспитом для людини похилого віку. Друге джерело проблем – це неправильне розуміння алгоритму, яке викликане недостатньою деталізацією його опису.

Для запису алгоритмів, призначених для виконавців – автоматів, необхідна повна формалізація, тому у цих випадках застосовують спеціальні формальні мови.

Розглянемо основні засоби, які використовуються для запису алгоритмів.

У комп'ютерних науках ці проблеми вирішують шляхом створення чітко визначеного набору складових блоків, з яких можуть конструюватись відображення алгоритму. Такі блоки називаються *примітивами*. Те що примітивам дають точні визначення, усуває багато проблем неоднозначності й одночасно потребує однакового рівня деталізації для всіх описуваних за їх допомогою алгоритмів.

Набір примітивів разом з набором правил, які встановлюють, як ці примітиви можуть комбінуватись для представлення більш складних конструкцій, складають мову програмування.

Алгоритм, що реалізує рішення задачі, можна представити різноманітними способами. Існує декілька способів та форм подання алгоритмів.

На практиці найбільше поширення отримали наступні способи подання алгоритмів:

- ***словесний.*** Алгоритм може бути поданий у вигляді записів літературної мови, наприклад, української. У цьому випадку послідовністю речень описується послідовність дій виконавця, яким у більшості випадків може бути тільки людина. Жодних спеціальних правил і вимог до таких записів алгоритмів не висувається. Головне, що б виконувалися вимоги, які висуваються до алгоритмів. В літературі присвяченій алгоритмам іноді використовується такий спосіб запису алгоритмів. Опис алгоритму Евкліда знаходження найменшого загального дільника двох цілих додатних чисел складається з трьох кроків і займає усього декілька рядків.

Крок 1. Розділити m на n, де $m>n$. Нехай p – залишок від ділення.

Крок 2. Якщо p дорівнює нулю, то n і є шуканий найменший загальний дільник.

Крок 3. Якщо p не дорівнює нулю, то зробимо прирівняємо m до n, а n прирівняємо до p. Повернутися до кроку 1.

В той же час навіть значно „спрощений" опис, даний Евклідом, вимагає біля сторінки тексту(!), і зрозуміти з його опису, що саме і для чого робиться, далеко не просто. Для запису алгоритмів потрібна певна мова. При цьому вкрай важливим є вибір цієї мови. Незручність української (як і будь-якої іншої природної) мови для цих цілей очевидна. Наприклад, фраза „хлопчик взяв з собою зошит в клітку" може повідомляти нам про школяра, який підготувався до уроку арифметики. Але може матись на увазі юний любитель птахів, який захопив клітку з папугою і засунув туди зошит для запису своїх спостережень і відчуттів... Слова можуть мати самі різноманітні змістовні відтінки, і зрозуміти їх без детального аналізу усього тексту часом просто неможливо;

- ***аналітичний (словесно-формульний)***;
- ***графічний*** (зображення за допомогою графічних символів). Графічний спосіб подання алгоритмів має низку переваг завдяки візуальності і явному відображенню процесу розв'язання задачі. У графічній формі алгоритм подається у вигляді геометричних фігур. Зазвичай вони поєднуються лініями, які показують напрямок передачі інформації при виконанні алгоритму. Існує декілька варіантів графічного подання алгоритмів, але найбільше поширення отримало (і стало фактичним стандартом графічного подання) подання у вигляді блок-схем. Метод блок-схем вперше був запропонований Фон Нейманом – одним з перших розробників обчислювальної техніки.

Алгоритми, які подаються графічними засобами, отримали назву <u>візуальних алгоритмів.</u>

Алгоритмічний мова – це система правил і позначень для точного і однозначного запису алгоритмів. Такий запис є формалізованим. Це означає, що запис підкоряється строгим вимогам синтаксису мови.

Мова програмування – це система позначень і правил для запису алгоритмів, призначена для використання на ЕОМ. На практиці мови програмування прив'язані до конкретних класів ЕОМ, операційних систем тощо. У мовах програмування істотними є технічні й технологічні аспекти, які не характерні для алгоритмічних мов, оскільки зазвичай є машинно-незалежними.

Спосіб подання алгоритмів на алгоритмічних мовах/мовах програмування відіграє провідну роль. Існує велика кількість мов програмування. Одні з них широко поширені: Basic, Pascal, C/C++, Modula, Fortran. Інші ж мають спеціальне призначення: Prolog, Forth, Lisp. Деякі мови відіграли помітну роль у програмування, але зараз не використовуються. Прикладом є мова Algol. Саме ця мова стала основою для розробки більш досконалих мов, таких як Паскаль,

Сі та інших. Алгол використовувався також як алгоритмічна мова для запису алгоритмів, у тому числі як автокод. Можна також відзначити таку важливу мову програмування для науково-технічних розрахунків: Фортран.

Існують мови декларативного (логічного) програмування, наприклад Пролог, де відсутні алгоритмічні інструкції, а є описи даних і зв'язків між ними. Виконуюча система провадить пошук найкращого способу розв'язання поставленої задачі. На декілька інших принципах побудовані функціональні мови, наприклад, Лісп. Основними керуючими структурами таких мов є послідовність викликів так званих рекурсивних функцій.

У 1985 р. фундатор шкільної інформатики академік О.П. Ершов запропонував для запису алгоритмів нову алгоритмічну мову, яку охрестили шкільною алгоритмічною мовою. Іноді цю мову називають Е-мовою, на честь його винахідника, але ця назва є неформальною;

- ***псевдокод*** (напівформалізовані описи алгоритмів умовною алгоритмічною мовою, яка включає в себе, як елементи мови програмування, так і речення природної мови, загальноприйняті математичні позначення тощо. При цьому немає жодних спеціальних вимог до оформлення таких записів, за винятком вимоги однозначності при реалізації записаних дій);
- ***таблиці значень***. Таблиці значень представляють алгоритм неявно, як певне перетворення конкретних початкових даних у вихідні. Табличний спосіб опису алгоритмів може бути з успіхом застосований для перевірки правильності функціонування розробленого алгоритму на конкретних тестових наборах вхідних даних, які разом з результатами виконання алгоритму фіксуються у „таблицях трасування";
- ***програмний*** (тексти на мовах програмування).

Таким чином, кожний спосіб подання алгоритмів має свої переваги і свої недоліки, усі способи подання алгоритмів можна вважати взаємодоповнюючими один одного. На етапі проектування алгоритмів найкращим способом є графічне представлення, на етапі перевірки – табличний опис, на етапі застосування – текстовий запис у вигляді програми.

2.2. Словесний опис

Словесний алгоритм – словесно сформульована послідовність правил виконання задачі. Алгоритм задається у довільному викладі природною людською мовою. Це найстаріша форма подання алгоритму. У ньому формальні правила перетворення інформації формулюються, нумеруються та вказується послідовність їх виконання. Така форма є прийнятною для досить простих або, навпаки, складних задач, розв'язання яких можна скласти з готових блоків (процедур обробки інформації), а за допомогою словесного опису вказати порядок їх виклику.

Наприклад, розглянемо алгоритм пошуку максимального числа з двох значень змінних:

1. визначимо формати змінних X, Y, M, де X, Y – значення для порівняння, M – змінна для збереження максимального значення;

2. отримуємо два значення чисел X, Y для порівняння;

3. порівняємо X та Y;

4. якщо X менше Y, то це означає, що більшим є число Y;

5. помістимо у змінну M значення Y;

6. якщо X не менше (більше) Y, значить більшим є число X;

7. помістимо у змінну M значення X.

Словесний спосіб не найшов широкого поширення за наступними причинами:

⇨ такі описи не є строго формалізованими;

⇨ страждають багатослівністю записів;

⇨ припускається неоднозначність тлумачення деяких приписів.

2.3. Аналітичний опис (словесно-формульна форма)

Словесно-формульна форма – це форма, яка використовує загальноприйняті математичні позначення, коментарі до них, які пояснюють дії та їх послідовність, що визначаються за допомогою міток.

Наприклад: *дано два вектори $A=(a_1, a_2, \ldots a_n)$ та $B= (b_1, b_2, \ldots b_n)$. Знайти вектор C, елементи якого обчислюються за формулою $c_i=a_i+b_i$.*

Розв'язання:

Для визначення порядку введемо мітки Q_1, Q_2, Q_3, якщо не вказано, до якої мітки переходити, то це перехід до наступної дії.

$i=1$

Q_1: $c_i=a_i + b_i$

Якщо $i=n$, то перейти до Q_3, інакше – до Q_2

Q_2: $i=i+1$. Перейти до Q_1.

Q_3: Закінчити обчислювання

2.4. Графічний спосіб

Графічний спосіб подання алгоритмів передбачає використання різних геометричних фігур, і дозволяє наочно зобразити послідовність різних етапів процесу алгоритмізації та їх взаємозв'язок.

2.4.1. Блок-схеми

***Блок-схеми** – це графічне зображення логічної структури алгоритму, у якому кожний етап процесу переробки даних подається у вигляді геометричних фігур (блоків), що мають певну конфігурацію залежно від характеру виконуваних операцій.*

Кожна людина знає, що будь-яка побутова річ призначена для конкретного використання. Наприклад, цукорниця призначена, для збереження у ній цукру, пляшка – для збереження рідини. Нам це зручно, ми ніколи не забудемо, що й де зберігається. Для сліпих людей зручно зберігати продукти у різному за формою посуду. Наприклад, сіль – у квадратній банці, а перець у – круглій. У такій ситуації у них не виникає потреби пробувати спеції на смак, щоб визначити що де знаходиться й не плутати їх. Аналогічно відбувається і з блок-схемами, де у кожному блоці виконуються тільки притаманні йому одні й ті ж дії.

Блок-схеми можуть бути традиційні й структуровані. Цікаво, що блок-схему, яку більшість з нас зазвичай представляє, коли чує слово „алгоритм", придумав математик *Джон фон Нейман*, який ілюстрував блок-схемою модель зіткнення ядерних частинок при розробці Манхетенського проекту (рис. 2.1).

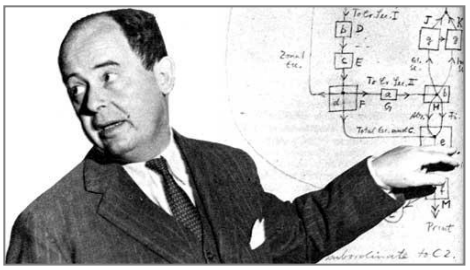


Рис. 2.1. Джон фон Нейман з розробленим алгоритмом

Переваги графічного способу подання алгоритмів були очевидні. Завдяки своїй компактності і наочності, подання алгоритму у вигляді пов'язаних між собою блоків, що відповідають певним діям, отримало велике поширення.

Спочатку кожний з великих виробників комп'ютерів розробляв свою систему блоків, які відтворювали його підхід до обробки інформації. Намагаючись перевершити конкурентів, компанії навіть випускали власні трафарети для креслення блок-схем алгоритмів, які на той час були більш популярними, чим спеціальне програмне забезпечення, включаючи спеціально розроблену для креслення блок-схем мову **SFL** (**S**ystems **F**lowchart **L**anguage). Свої блок-схеми мали й окремі організації, наприклад, військово-повітряні сили США.

У 1961 р. *Міжнародна організація стандартів (ISO – **I**nternational **S**tandards **O**rganization)* заснувала комітет „Комп'ютери і обробка інформації", учасниками якого були представники комп'ютерних компаній і компаній-користувачів, на який був покладений обов'язок розробки стандартів для блок-схем. Зі своїм завданням комітет упорався у 1963 р., коли був випущений перший стандарт, де були визначені усі основні умовні знаки, що стосувались комп'ютерних алгоритмів, а також правила креслення блоків.

Але розвиток мов програмування призвів до необхідності впровадження нових блоків. Стандарт перетерпів низку значних змін у 1965 р. і невеликих корегувань у 1966 і 1968 рр. У 1970 р. американський стандарт був приведений у відповідність з ISO. Головне призначення стандарту – визначити зовнішній вид блоків та їх призначення. Тут представлені: символи даних, символи процесу, символи ліній і спеціальні символи. Для кожного з них вказується форма, але не розмір.

Схеми алгоритмів і програм входять у склад програмної документації і оформлюються у відповідності з ДГСТ 19.701 – 90 (ICO 5807 – 85) „Схеми алгоритмів, програм, даних і систем". При цьому використовуються умовні графічні позначення (УГП), які вписуються в прямокутник (рис. 2.2). Сторони прямокутника мають наступні розміри: a = 10, 15, 20 … через 5 мм, b = 1,5a або $b = 2a$.

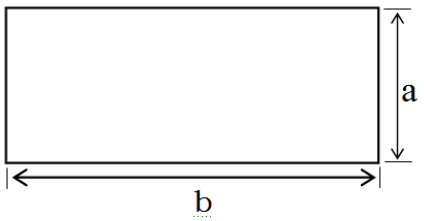


Рис. 2.2. Основний елемент схем алгоритмів

Найбільше часто використовувані блоки наведені на рис.2.3.

Цей спосіб виявився досить зручним для зображення алгоритмів й отримав широке поширення у науковій та навчальній літературі.

<u>Структурна (блок) схема алгоритму</u> – це графічне зображення алгоритму у вигляді схеми пов'язаних між собою за допомогою стрілок (ліній переходу) блоків – графічних символів, кожний з яких відповідає одному кроку алгоритму. Усередині блоку дається опис відповідної дії (табл.2.1).

Найменування	Позначення	Функція
Термінатор		Елемент відображає вхід із зовнішнього середовища або вихід з неї (найбільш часте застосування - початок і кінець програми). Всередині фігури записується відповідна дія.
Процес		Виконання однієї або кількох операцій, обробка даних будь-якого виду (зміна значення даних, форми подання, розташування). Всередині фігури записують безпосередньо самі операції.
Рішення		Показує рішення або функцію перемикального типу з одним входом і двома або більше альтернативними виходами, з яких тільки один може бути обраний після обчислення умов, визначених всередині цього елементу. Вхід в елемент позначається лінією, що входить зазвичай у верхню вершину елементу. Якщо виходів два чи три то зазвичай кожен вихід позначається лінією, що виходить з решти вершин (бічних і нижній). Якщо виходів більше трьох, то їх слід показувати однією лінією, що виходить з вершини (частіше нижній) елемента, яка потім розгалужується. Відповідні результати обчислень можуть записуватися поруч з лініями, що відображають ці шляхи.
Зумовлений процес		Символ відображає виконання процесу, що складається з однієї або кількох операцій, що визначений в іншому місці програми (у підпрограмі, модулі). Всередині символу записується назва процесу і передані в нього дані.
Дані		Перетворення даних у форму, придатну для обробки (введення) або відображення результатів обробки (вивід). Цей символ не визначає носія даних (для вказівки типу носія даних використовуються специфічні символи).
Межа циклу		Символ складається з двох частин - відповідно, початок і кінець циклу - операції, що виконуються всередині циклу, розміщуються між ними. Умови циклу і збільшення записуються всередині символу початку або кінця циклу - в залежності від типу організації циклу. Часто для зображення на блок-схемі циклу замість цього символу використовують символ рішення, вказуючи в ньому умову, а одну з ліній виходу замикають вище в блок-схемі (перед операціями циклу).
З'єднувач		Символ відображає вихід в частину схеми і вхід з іншої частини цієї схеми. Використовується для обриву лінії та продовження її в іншому місці (приклад: поділ блок-схеми, що не поміщається на листі). Відповідні сполучні символи повинні мати одне (при тому унікальне) позначення.
Коментар		Використовується для більш детальної інформації про кроки, процесу або групи процесів. Опис поміщається з боку квадратної дужки і охоплюється їй по всій висоті. Пунктирна лінія йде до описуваного елементу, або групі елементів (при цьому група виділяється замкнутої пунктирною лінією). Також символ коментаря слід використовувати в тих випадках, коли обсяг тексту в будь-якому іншому символі (наприклад, символ процесу, символ даних та ін) перевищує його обсяг.

Рис. 2.3. Складові блок-схеми

Графічне зображення алгоритму широко використовується перед програмуванням задачі внаслідок його наочності й тому, що зорове сприйняття зазвичай, значно полегшує процес написання програми. Крім того, значно полегшується корегування програми при можливих помилках, осмислюванні процесу обробки інформації. У кінці алгоритму повинні розташовуватися блоки виведення значень вихідних даних В алгоритмі повинний бути тільки один блок початку виконання і один блок закінчення виконання алгоритму. Зв'язки між блоками вказуються спрямованими або неспрямованими лініями.

Етап проектування алгоритму йде за етапом формального розв'язку задачі, на якому визначені вхідні і вихідні дані, а також залежності між ними.

При побудові алгоритмів для складної задачі використовують системний підхід з використанням декомпозиції (спадне проектування „зверху-вниз"). Як і при розробці будь-якої складної системи, при побудові алгоритму використовують дедуктивний і індуктивний методи. При дедуктивному методі, алгоритм розглядається як окремий випадок загальновідомих алгоритмів. Індуктивний метод застосовують у випадку, коли не існує загальних алгоритмічних розв'язків.

<u>Загальними правилами при проектуванні візуальних алгоритмів є наступні:</u>

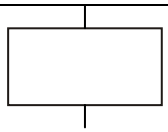
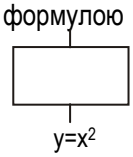
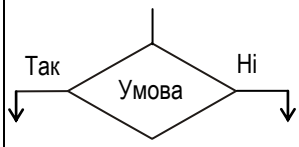
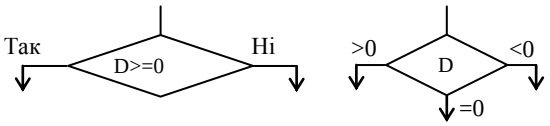
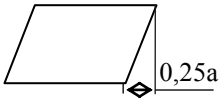
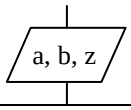
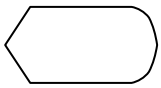
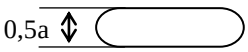
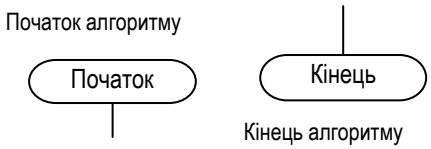
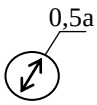
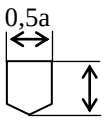
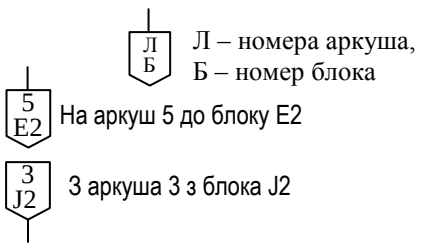
✓ на початку алгоритму повинні бути блоки введення значень вхідних даних.

✓ після введення значень вхідних даних можуть з'являтися блоки обробки і блоки умови.

Принцип програмування *„зверху – вниз"* вимагає, щоб блок-схема поетапно конкретизувалась і кожний блок „розписувався" до елементарних операцій – примітивів. Але такий підхід можна виконати при розв'язанні нескладних задач. При розв'язанні скільки-небудь складної задачі, блок схема „розповзеться" до таких розмірів, що її неможливо буде охопити одним поглядом.

Таблиця 2.1

Структурні блоки для опису алгоритмів

Найменування	Позначення	Дія
Процес		Виконання будь-яких операцій з обробки даних, наприклад обчислення за формулою 
Розв'язання		Вибір напрямку виконання алгоритму залежно від умови 
Введення – виведення		Введення – виведення даних. Усередині блока – імена змінних 
Дисплей		Подання даних на дисплеї при виведенні. Усередині записують імена виведених змінних
Документ		Виведення даних на паперовий носій. Усередині записують імена виведених змінних
Пуск – зупинка		
З'єднувач		Зв'язок між перерваними лініями схеми 
Міжсторінковий з'єднувач		Зв'язок між блоками на різних аркушах 

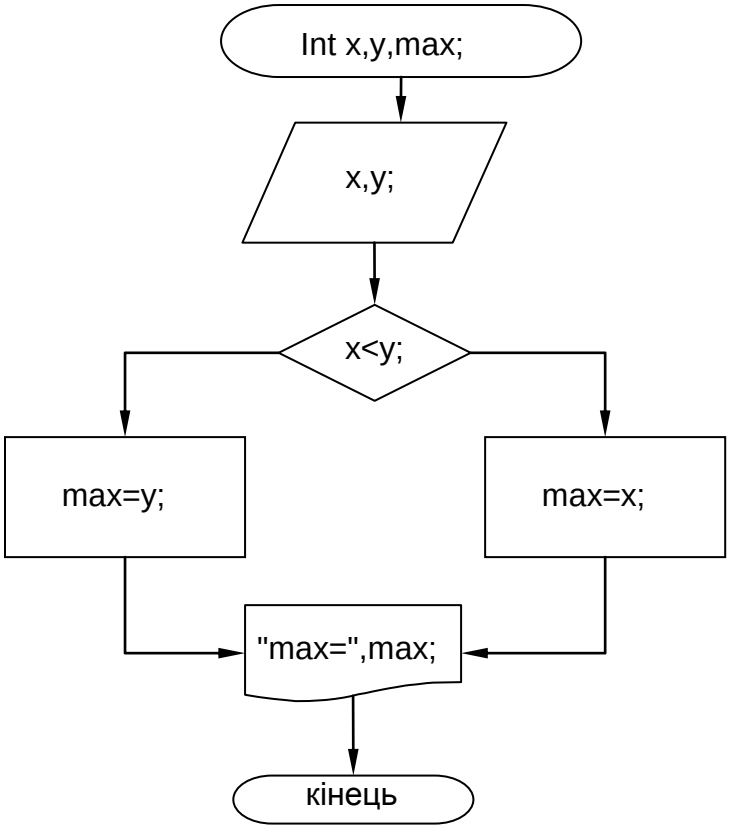


Рис. 2.4. Блок-схема алгоритму знаходження максимального числа з двох значень

2.4.2. Граф-схеми

<u>Граф-схеми алгоритмів (ГСА)</u> – це подання алгоритму у вигляді системи точок, кожна з яких визначає дію та стрілок, які вказують перехід від однієї дії до іншої.

ГСА знаходять широке застосування у практиці проектування пристроїв ЕОМ і зокрема, мікропрограмних автоматів, через їх гарну оглядовості, простоту конструкції мови, можливість перетворень і формального переходу до автоматичного відображення.

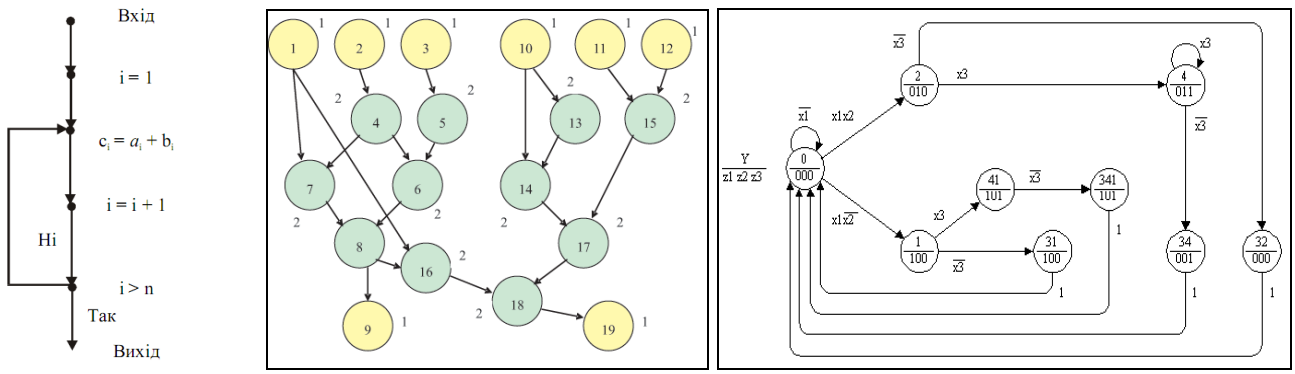


Рис. 2.5. Приклади граф-схем алгоритму

2.4.3. HIPO-схеми

Подання алгоритму у вигляді HIPO-схеми (Hierarchy Input Process Output) – це таблиця, у якій в трьох колонках розміщуються вказівки щодо вхідної інформації, дій над нею та вихідної інформації, тобто алгоритму з елементами технології обробки інформації.

Алгоритм розв'язання задачі у вигляді HIPO-схеми зображено на рис. 2.6.

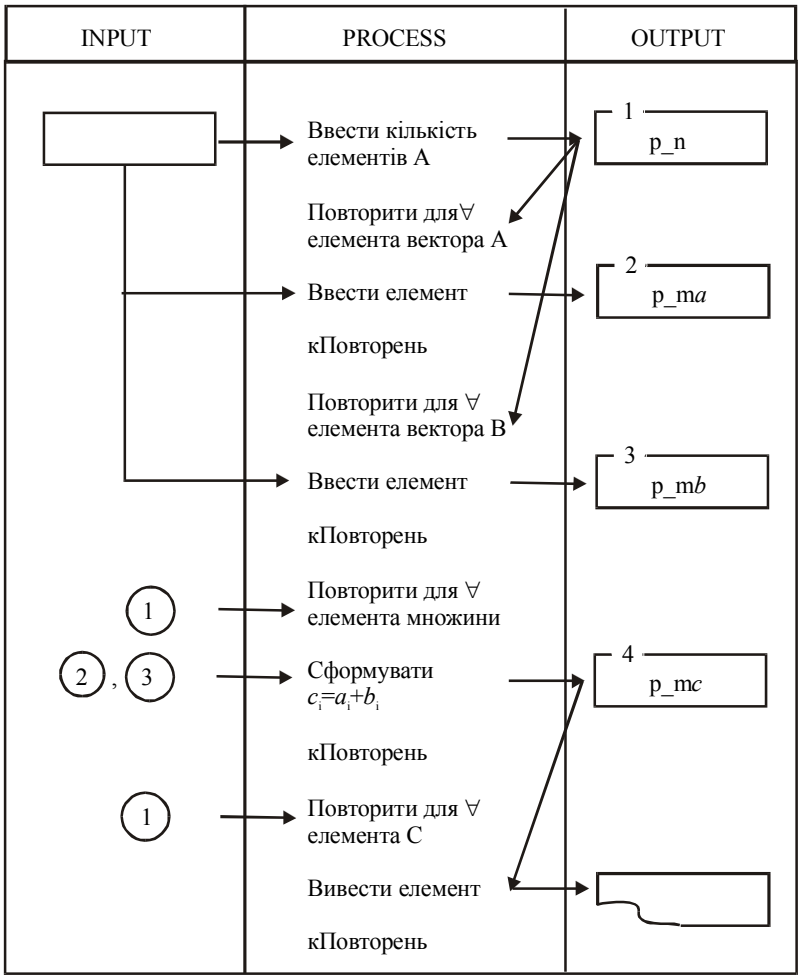
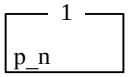
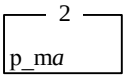


Рис. 2.6. HIPO-схема алгоритму задачі

На рис. 2.6  означає «поле 1, де міститься n»,  означає «поле 2 розміщення елементів масиву *a*» тощо.

У лівій колонці ❶ означає, що використовується значення з поля 1, тобто *n* тощо.

У середній колонці

Повторити для ∀ елемента А ввести елемент к Повторень

означає: «Повторити для кожного елемента масиву А дію вводу. Кінець повторень». Стрілка вказує, з якого поля треба взяти кількість елементів масиву. Взагалі, ∀ означає «для всіх» або «для кожного». Аналогічно можна пояснити всі послідовні дії алгоритму.

2.5. Операторні схеми

Подання алгоритмів за допомогою операторних схем наближається до програмного вигляду. Суть цього способу опису алгоритму полягає у тому, що кожний оператор позначається літерою (наприклад, *A* – арифметичний оператор, *P* – логічний оператор тощо). Тобто всі перетворення інформації подають у вигляді припустимих операцій. Тут використовуються логічні схеми алгоритмів, якими називають вираз, складений з операторів, що слідують один за одним. Оператори записуються зліва направо у послідовності їх виконання, причому, кожний оператор має індекс, який вказує порядковий номер оператора. Алгоритм записується в один рядок у вигляді послідовності операторів.

Розрізняють такі типи операторів:

1. Арифметичні оператори.
2. Оператори перевірки логічних умов.
3. Оператори переадресації.
4. Оператори переносу.
5. Оператори формування ↑, → , ↓.

Арифметичні оператори використовуються для запису арифметичних дій і позначаються першими великими літерами латинського алфавіту: *A*, *B*, *C*.

Оператори перевірки логічних умов визначають порядок дій алгоритму. Позначаються малими літерами, а умова записується поряд у дужках, наприклад: *p(a>0)*.

Оператори переадресації змінюють значення різних параметрів. Вони позначаються літерою *F*, а поряд у дужках вказують параметр. Величину, на яку змінюється параметр, задають у вигляді степеня. Наприклад:

$$F^1(i) = F(i),\ F^2(i),\ F^{-1}(i)\ldots$$

Тут параметр *i* змінюється відповідно на 1, 2, -1.

Оператори переносу змінюють значення одного параметра значенням іншого, наприклад, [*a, b*] параметр *b* замінюється на *a*.

Оператори формування присвоюють початкові значення параметрів, наприклад $\{_{5\to}i\}$

Оператори виконуються у тому порядку, у якому вони записані. Щоб замінити порядок їх виконання, використовуються пари стрілок, які мають свої номери $\uparrow^1$ – звідки переходити; $\downarrow_1$ – куди переходити за *i*-стрілкою.

Розв'язання: Нехай D_i – обчислює величину $c_i := a_i + b_i$. Тоді алгоритм визначається такою операторною схемою:

$$\{1 \to i\}\downarrow_1 DiF(i)\,p(i > n)\uparrow^1 \text{ зупинення.}$$

2.6. Псевдокоди

Псевдокод – це неформальний запис алгоритму, який використовує структуру поширених мов програмування але нехтує деталями коду, неістотними для розуміння алгоритму (опис типів, виклик підпрограм тощо). Мова програмування доповнюється природною мовою, компактними математичними позначеннями. Псевдокод є більш зрозумілою ніж програми формою запису алгоритмів.

Псевдокод займає проміжне місце між природною і формальною мовою. З одного боку, він наближений до звичайної природної мови, тому алгоритми можуть на ньому записуватися і зчитуватись як звичайний текст. З іншого боку, у псевдокоді використовуються деякі формальні конструкції і математична символіка, яка наближає запис алгоритм до загальноприйнятого математичного запису.

У псевдокоді не застосовуються строгі синтаксичні правила для запису команд, які властиві формальним мовам. Це полегшує запис алгоритму на стадії його проектування і дає можливість використовувати більш широкий набір команд, розрахований на абстрактного виконавця. Однак у псевдокоді, зазвичай є деякі конструкції, які властиві формальним мовам, які полегшують перехід від запису на псевдокоді до запису алгоритму на формальній мові. Зокрема, у псевдокоді, аналогічно формальним мовам, є службові слова, зміст яких визначений раз і назавжди. Вони виділяються у друкованому тексті жирним шрифтом, а в рукописному тексті підкреслюються. Загального або формального визначення псевдокоду не існує, тому можливі різні псевдокоди, що відрізняються набором службових слів і основних (базових) конструкцій. Для прикладу приведемо запис на одному з псевдокодів алгоритму:

алгоритм *алгоритм Евкліда;*
начало
пока перше число не дорівнює другому
повторять
начало
если числа рівні
то стоп **все;**
иначе визначити більше з двох чисел;
заменить більше число на різницю більшого і меншого чисел
конец;
взяти перше число як відповідь
конец

Цей алгоритм можна записати простіше, але для демонстрації основних можливих конструкцій псевдокоду приведено саме такий запис. В силу своїх особливостей псевдокоди, як і інші вищеописані засоби запису алгоритмів, орієнтовані на людину.

Псевдокод, володіючи усіма позитивними якостями схем, має ще й власні переваги. Псевдокод менше схем обмежений правилами зображення. Головне – використання тільки службових конструкцій і дотримання правил східчастого запису.

2.7. Таблиці розв'язків

Подання алгоритмів обробки інформації за допомогою таблиць рішень найчастіше використовується у тих випадках, коли на кожному кроці перетворення інформації потрібно проводити аналіз стану системи (зі скінченої множини станів) та вхідної інформації і кожна з комбінацій результатів аналізу спричиняє перехід у новий стан та певну вихідну інформацію. Побудувати такий процес з багатьма розгалуженнями у вигляді блок схеми неможливо. Щоб пояснити побудову таблиць, необхідно мати попередню уяву про абстрактні автомати, які буде розглянуто пізніше.

2.8. Програмний запис алгоритмів

Вище відзначалося, що при запису алгоритму у словесній формі, у вигляді схеми або на псевдокоді можливе певне свавілля при зображенні команд. Разом з тим такий запис настільки точний, що дозволяє людині зрозуміти сутність справи і виконати алгоритм.

Однак на практиці виконавцями алгоритмів виступають спеціальні автомати – електронні обчислювальні машини (ЕОМ) – комп'ютери. Тому алгоритм, призначений для виконання на комп'ютері, повинен бути записаний на мові, „зрозумілій" комп'ютеру. І тут на перший план висувається необхідність точного запису команд, який не залишає місця для довільного тлумачення їх виконавцем. Отже, мова для запису алгоритму повинна бути формалізована. Таку мову прийнято називати *мовою програмування,* а запис алгоритму на цій мові – *програмою для комп'ютера.*

Мова програмування – це формалізована мова, яка представляє собою сукупність алфавіту, правил написання конструкцій (синтаксис) і правил тлумачення конструкцій (семантика).

Сьогодні можна нарахувати декілька сотень мов програмування, які розраховані на різні галузі використання комп'ютерів, на різні класи задач. Ці мови класифікують за різними рівнями. Найбільш загальною є класифікація за ступенем залежності мови від комп'ютера (ЕОМ). За цією ознакою мови поділяються на дві великі групи:

- машинно-залежні;
- машинно-незалежні.

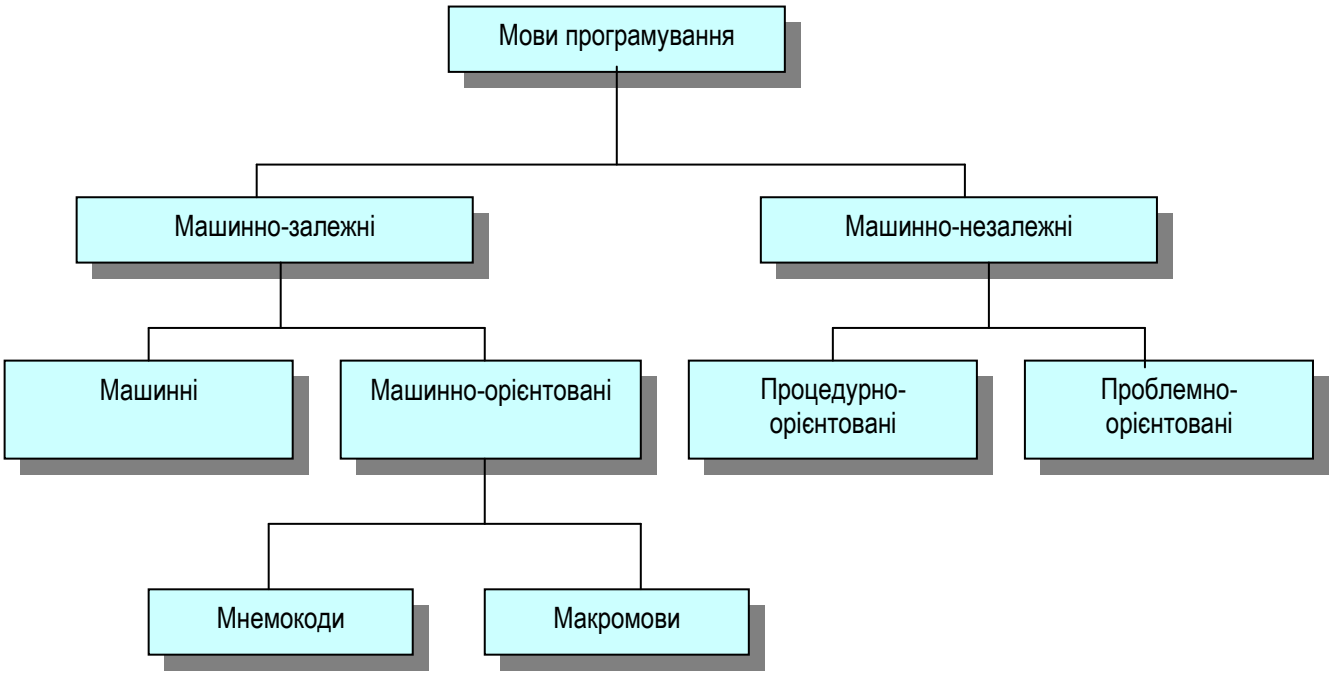


Рис. 2.7. Мови програмування

Машинно-залежні мови, у свою чергу, поділяють на *машинні* і *машинно-орієнтовані*. ***Машинно–орієнтовані мови іноді називають автокодами***. Розрізняють два рівня машинно-орієнтованих мов. До першого рівня відносяться мови символьного кодування, які інакше називають *мнемокодами*, а до другого – *макромови*.

Мнемокод відрізняється від машинної мови відповідного комп'ютера заміною цифрових кодів операцій буквеними (мнемонічними), а цифрових адрес операндів – буквеними або буквено-цифровими. При перекладі на мову комп'ютера кожна команда мнемокоду заміняється відповідною командою машинної мови (<< один в один >>).

Використання *мнемокоду* дозволяє автоматизувати роботу програміста з розподілу пам'яті, точніше за привласненням істинних адрес. Це особливо корисно при програмуванні для машин зі змінним форматом команд. Крім того, мнемокод істотно полегшує роботу зі складання великих програм, коли окремі сегменти (модулі) програми складаються різними програмістами і об'єднуються в одну програму на етапі завантаження.

Мова другого рівня – *макромова* – поряд з символічними аналогами машинних команд, з яких складається мнемокод, припускає також використання макрокоманд, які не мають прямих аналогів у машинній мові. При трансляції кожна макрокоманда замінюється групою команд машинної мови (<< один в декілька >>). Використання макрокоманд скорочує програму, підвищує продуктивність програміста. Програміст, який використовує машинно-орієнтовану мову повинен бути добре знайомим з особливостями устрою комп'ютера, для якого складається програма.

Однак безпосередній запис алгоритму на машинній мові вимагає від розробника надмірної деталізації алгоритму, у результаті чого запис (послідовність машинних команд), виходить не наочним і важким для розуміння. Писати програми на машинній мові досить складна і стомлююча праця, тому розробники алгоритмів використовують зазвичай, мови програмування більш високого рівня, у яких прийнята символічна форма запису, наближена до загальноприйнятої математичної.

Машинно-незалежні мови також поділяються на дві групи за ступенем деталізації програми. До першої групи відносяться процедурно-орієнтовані мови, а до другої – проблемно-орієнтовані.

Процедурно-орієнтовані мови призначені для опису алгоритмів (процедур) розв'язання задач, тому їх також називають алгоритмічними, хоча поняття алгоритмічної мови не збігається з поняттям мови програмування. Якщо запис здійснюється алгоритмічною мовою безпосередньо і придатний для введення у комп'ютер (ЕОМ) та перетворення у готову робочу програму, то така мова є одночасно мовою програмування. Процедурно-орієнтовані мови створювалися під конкретні класи задач, наприклад, Фортран для наукових розрахунків, Кобол – для економічних розрахунків, Ліспі і Пролог – для штучного інтелекту тощо. Але при цьому потрібен переклад програми, написаній на такій мові на машинну мову. Цей переклад виконує транслятор.

Будь-яку ***програму, яка перекладає довільний текст на певну комп'ютерну мову, називають транслятором***. Зазвичай, процеси трансляції й виконання програм розділені у часі. Спочатку вся програма транслюється, а потім виконується. Транслятори, що працюють у такому режимі, називають трансляторами компілюючого типу. Якщо вхідною мовою такого транслятора є процедурно-орієнтована мова високого рівня, то транслятор називають компілятором.

Програма-компілятор (від слова compile – складати, збирати) ***перекладає початковий текст програми у машинний код і записує його у пам'ять у формі виконуваного файла (завантажувального файла).*** Після цієї операції, програма вже виконується незалежно від початкового тексту. Саме тому, раніше програми-компілятори називали просто і точно – трансляторами (перекладачами).

Якщо трансляція і виконання програм зливаються у часі, то такі транслятори називають інтерпретаторами. ***Програма-інтерпретатор*** завжди працює спільно з початковим текстом. Вона розбирає кожну інструкцію початкового тексту (інтерпретує її) і негайно виконує (файл на машинній мові не створюється). Програма в режимі інтерпретації працює набагато повільніше, на відміну від програми у машинному коді. Це пов'язано з тим, що кожну інструкцію доводиться розбирати в момент виконання (а не заздалегідь, як при компіляції). Багато інструкцій у програмі виконуються багаторазово, і при кожному виконанні інтерпретуються заново. Тому усюди де можливо, прагнуть замінити режим інтерпретації режимом компіляції. Правда, інтерпретація має й свої переваги: за її допомогою простіше відлагоджувати програму. Іноді користуються режимом „псевдокомпіляції": прискорюють інтерпретацію за рахунок попереднього запам'ятовування тих чи інших елементів „розібраних" команд у пам'яті машини.

Найбільш часто зустрічається транслятор інтерпретуючого типу (транслятор з мови Basic, де команди зчитуються, перетворюються й відразу виконуються). Підсумком роботи такого транслятора є необхідні результати. Транслятор з мови Pascal – компілюючого типу. Текст з початкової мови перекладається у текст на машинній мові і утворює так званий об'єктний модуль. Потім об'єктний модуль опрацьовується програмою редактора міжпрограмних зв'язків і тільки після цього програма буде готова до виконання.

Інтерпретатор значно простіше компілятора, оскільки негайне виконання розпізнаних операторів вхідної мови робить непотрібним дії, пов'язані з компонуванням об'єктної програми, оформленням її в єдиний модуль завантаження або у вигляді декількох модулів, якщо вона досить велика.

Недолік інтерпретатора полягає у неефективному використанні машинного часу. Наприклад, при виконанні циклічних програм, один й той же оператор доводиться

інтерпретувати багаторазово. При повторному виконанні програми інтерпретацію приходиться виконувати знову, у той час, як транслятор компілюючого типу дозволяє виконати трансляцію тільки один раз, а потім зберігати програму у машинних кодах. За зазначеною причиною інтерпретатори застосовуються відносно рідко.

Програма отримана після обробки транслятором або безпосередньо виконується на ЕОМ, або піддається обробці іншим транслятором.

За рівнем вхідної мови транслятори прийнято поділяти на *ассемблери, макроассемблери, компілятори, генератори.*

Вхідною мовою асемблера є мнемокод, макроасемблера – макромова, компілятора – процедурно-орієнтована мова, а генератора – проблемно–орієнтована мова. У зв'язку з цим, вхідну мову називають за типом транслятора: мова асемблера, мова макроасемблера тощо.

Сучасне програмування істотно відрізняється від технології розробки програм для попереднього покоління ЕОМ. Серед відносно нових особливостей і напрямків цієї технології можна відзначити:

- застосування об'єктно-орієнтованих мов;
- візуальне програмування (складання екранної форми за допомогою миші з готових „напівфабрикатів" – об'єктів);
- швидка розробка додатків (**RAD – R**apid **A**pplications **D**evelopment);
- програмування з використанням функцій **API Windows (A**pplications **P**rogramming **I**nterface – інтерфейс прикладного програмування);
- бази даних і додатки для багатьох користувачів (додатки, з якими одночасно працює декілька користувачів) тощо.

Програма на процедурно-орієнтованій мові майже не залежить від конкретної ЕОМ, на якій буде розв'язуватись задача. Слово „майже" потрібно розуміти у тому сенсі, що у більшості випадків, програми розв'язання однієї й тієї ж задачі, для різних ЕОМ відрізняються лише деякими непринциповими деталями зовнішнього оформлення, які при переході від ЕОМ до ЕОМ заміняються механічно. Структура процедурно-орієнтованих мов ближче до природної мови, наприклад української або англійської, чим до мови ЕОМ. Тому переклад з процедурно-орієнтованої мови на машинну мову здійснюється за принципом „декілька в декілька". Іншими словами, у більшості випадків тут можна встановити відповідність лише між групою елементарних конструкцій мови і групою команд ЕОМ, подібно тому, як при перекладі з англійської мови на українську мову групи слів або навіть групи речень заміняють групою слів на іншій мові. Переклад за словами тут неможливий.

До проблемно-орієнтованих мов відносять *непроцедурні* мови, які не потребують детального запису алгоритму розв'язання задачі. Користувач повинен лише вказати формулювання задачі або назвати послідовність задач з раніш підготовленого набору, вказати початкові дані й необхідну форму видачі результатів. Ця інформація використовується спеціальною програмою – генератором, – для генерування робочої програми.

За відношенням до транслятора, усі вищеперераховані мови, крім машинних мов, є вхідними. У процесі трансляції, програма на вхідній мові перекладається на внутрішню мову, більш зручну для подальшої роботи транслятора, а потім послідовно відбувається декілька стадій обробки. На кожній стадії відтрансльована програма подається у проміжній мові. І, врешті-решт, після обробки транслятором утворюється програма на вихідній мові.

Командні мови баз даних призначені для розширення можливостей середовища управління базами даних, для створення власних функцій інтерфейсу – взаємодії з користувачем.

Наприклад, цікавою є еволюція мови програмування BASIC. Вона була задумана як універсальна мова для починаючих програмістів (за аналогією з BASIC ENGLISH, – підмножиною англійської мови, виділеної для навчання іноземців). Перші версії (або „діалекти") цієї мови включали невелику кількість найнеобхідніших команд і передбачали тільки режим інтерпретації. Однак сучасні варіанти мови BASIC не тільки не поступаються за можливостями багатьом „грандам" (типу мови С), але іноді й перевершують їх. Наприклад, Visual Basic використовується у суперсучасних системах, заснованих технології „клієнт-сервер". Одночасно BASIC став своєрідним „есперанто" у світі інформаційних технологій. На цій мові часто пишуться приклади програм або їх фрагментів у книгах, статтях, інструкціях до програмних продуктів тощо.

Фірма Microsoft використовує мову Visual Basic для розширення функцій своїх програмних продуктів. Уже в пакеті Microsoft Office для Windows 3-х користувачам і програмістам пропонувалися діалекти Word Basic і Access Basic, а нині у Microsoft Office передбачена універсальна мова Visual Basic for Applications (VBA – Visual Basic для додатків). Раніше, ця мова використовувалася тільки в електронних таблицях Microsoft Excel. За допомогою VBA можна створювати власні програмні модулі, власні інтерфейси для офісних додатків Microsoft Word, Microsoft Excel, Microsoft Access.

Питання для самоконтролю

1. *Що розуміють під терміном „алгоритмізація”?*
2. *Від чого залежить спосіб запису алгоритму?*
3. *Які способи представлення алгоритмів Вам відомі?*
4. *Поясніть сутність словесного опису алгоритмів.*
5. *Поясніть сутність аналітичного опису алгоритмів.*
6. *Поясніть сутність графічного опису алгоритмів.*
7. *Що розуміють під структурною схемою алгоритму?*
8. *Поясніть сутність опису алгоритмів за допомогою операторних схем.*
9. *Поясніть сутність опису алгоритмів за допомогою псевдокоду.*
10. *Поясніть сутність опису алгоритмів за допомогою таблиць рішень.*
11. *Поясніть сутність програмного опису алгоритмів.*
12. *Що розуміють під мовою програмування?*
13. *Що розуміють під терміном „автокод”?*
14. *Що розуміють під терміном „мнемокод”?*
15. *Що розуміють під терміном „макромова”?*
16. *Що розуміють під терміном „транслятор”?*
17. *Що розуміють під терміном „компілятор”?*
18. *Що розуміють під терміном „інтерпретатор”?*

3. ТИПИ АЛГОРИТМІЧНИХ ПРОЦЕСІВ

Швидкість розв'язку залежить від краси обмежень.
Георгій Александров

Алгоритм стосовно обчислювальної машини – точний припис дій або операцій та правил їх виконання, за допомогою яких, починаючи з певних початкових даних можливо вирішити задачу фіксованого типу.

Проблемами теорії алгоритмів є визначення основних властивостей алгоритмів, пошук ефективних алгоритмів і алгоритмічно нерозв'язних задач.

3.1. Класифікація алгоритмічних процесів

Сукупність задач, з якими стикається людина, поділяються на два класи: обчислення функції і розпізнавання образів (класифікація). Функція є обчислюваною якщо можлива побудови алгоритму для її обчислення. Множина є розрізненою, якщо існує алгоритм, який дозволяє визначити приналежність даного об'єкту до цієї множини чи ні.

Розрізняють наступні типи алгоритмічних процесів:

• ***механічні, або детерміновані*** – ***задають визначені дії, позначаючи їх у єдиній та достовірній послідовності, забезпечуючи тим самим однозначний результат, якщо виконуються умови, для яких розроблено алгоритм***. Наприклад, алгоритм роботи пральної машини, двигуна тощо;

• ***ймовірні (стохастичні)*** – ***дають програму розв'язання задачі декількома шляхами або способами, які приводять до ймовірного досягнення результату***;

• ***евристичні*** – ***це алгоритми, у яких досягнення кінцевого результату програми дій однозначно не визначені, оскільки не визначені усі послідовності дій і дій виконавця.*** У цих алгоритмах використовуються універсальні логічні процедури і способи прийняття рішень, які ґрунтуються на аналогіях, асоціаціях й минулому досвіді розв'язання подібних задач;

• ***лінійні – алгоритми, які виконуються послідовно у часі, один за одним***. Типовим прикладом лінійного обчислювального процесу є стандартна обчислювальна схема, що складається з таких етапів:

- введення початкових даних;
- обчислення за формулами;
- виведення результату;

• ***розгалужені – це алгоритми, які містять хоча б одну умову, за результатом перевірки якої, обчислювальна машина забезпечує перехід на один із можливих кроків***;

• ***циклічні – це алгоритми, які передбачають багатократне повторення однієї і тієї ж дії (одних і тих же операцій) над новими початковими даними***. До циклічних алгоритмів зводиться більшість методів обчислювань, перебору варіантів;

• ***комбіновані (складні)*** – ***алгоритми, які містить декілька структур одночасно***.

При побудові алгоритмів для складної задачі використовують системний підхід з використанням декомпозиції (спадне проектування зверху-вниз). При розробці алгоритму використовують дедуктивний і індуктивний методи. При дедуктивному методі розглядається окремий випадок загальновідомих алгоритмів. Індуктивний метод застосовують у випадку, коли не існує загальних алгоритмічних рішень.

Одним з системних методів розробки алгоритмів є метод структурної алгоритмізації. Цей метод заснований на візуальному поданні алгоритму у вигляді послідовності керуючих структурних фрагментів. Виділяють три базові фрагменти, які керують процесом обробки інформації структури:

- ✓ *композицію;*
- ✓ *альтернативу;*
- ✓ *ітерацію.*

За допомогою цих структур можна описати будь-які процеси обробки інформації.

Композиція (проходження) – це лінійна керуюча конструкція, яка не містить альтернативу і ітерацію. Вона призначена для опису єдиного процесу обробки інформації.

Альтернатива – це нелінійна керуюча конструкція, яка не містить ітерацію. Вона призначена для опису різних процесів обробки інформації, вибір яких залежить від значень вхідних даних.

Ітерація – це циклічна керуюча структура, яка містить композицію і розгалуження. Вона призначена для організації повторюваних процесів обробки послідовності значень даних.

У відповідності з наявністю в алгоритмах керуючих структур композиції, альтернативи і ітерації алгоритми класифікують на: *лінійні (рис.3.1)*, *розгалужені* (рис.3.2) й *циклічні* алгоритми (рис.3.3).

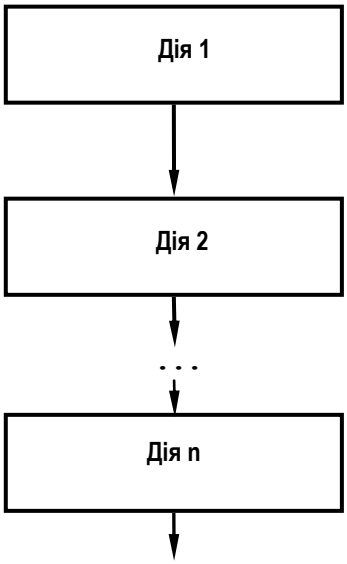


Рис.3.1. Лінійна алгоритмічна структура

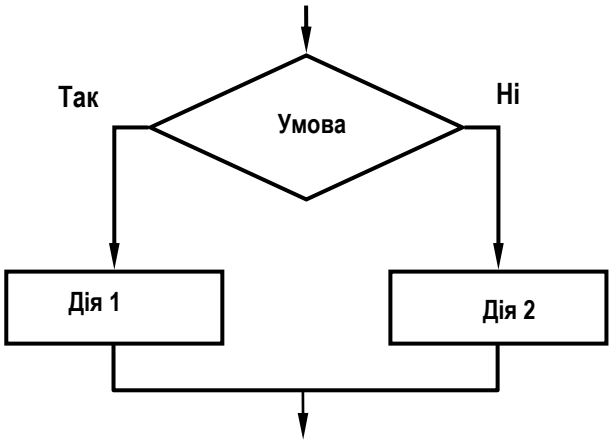


Рис.3.2. Розгалужена алгоритмічна структура

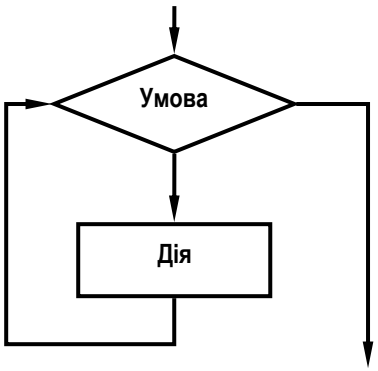


Рис.3.3. Циклічна алгоритмічна структура

3.2. Лінійні алгоритми

Найпростішими для алгоритмізації є задачі, у яких перетворення інформації відбувається послідовно за певними формулами, які розкладаються на елементарні операції. Кожна дія виконується одна за одною, послідовно у порядку розташування блоків і при цьому жодна з дій не пропускається й не повторюється. Лінійні алгоритми не містять перевірок умов. Потрібно тільки визначити раціональну послідовність цих операцій і виконати їх за схемою (рис. 3.1).

Приклад 1: $Z=\sin^2(x^2+y^2) + \cos^2 \ln(x^2+y^2)$

Оскільки аргументом функцій *sin* та *ln* є вираз x^2+y^2, його треба обчислити його у першу чергу. Отже, алгоритм визначення Z матиме вигляд представлений на рис. 3.4.

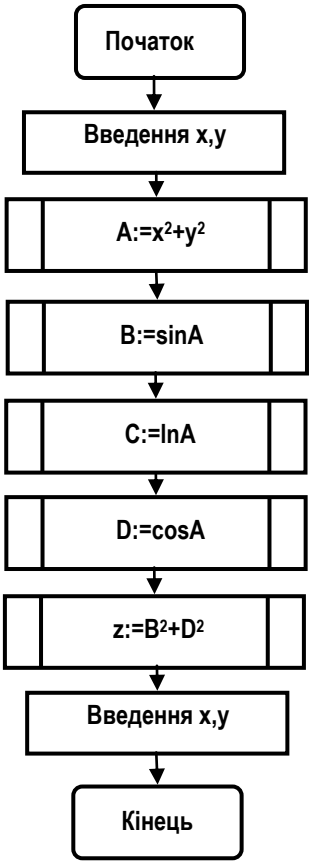


Рис. 3.4. Приклад лінійного алгоритму

У практичній діяльності програмісти зазвичай прагнуть до того, аби алгоритм мав лінійну структуру.

3.3. Розгалужені алгоритми

У тих випадках, коли перетворення інформації може здійснюватись за різними схемами, залежно від властивостей вхідних даних або проміжних результатів, використовуються розгалужені алгоритми.

Структура „розгалуження" передбачає виконання однієї з двох груп дій залежно від виконання умови у блоці розгалуження. На рис. 3.2 знаком „Так" показано виконання умови, а знаком „Ні" – його невиконання. Часто використовується неповна команда розгалуження, коли один з блоків дії відсутній.

В алгоритмі розгалуження передбачаються усі можливі варіанти обробки інформації, кожний з яких розробляється як окрема гілка алгоритму, а вибір однієї з них для виконання, здійснюється за допомогою перевірки певної умови, що відображує властивості інформації, використовуваної у процесі перетворення. При цьому деякі дії можуть взагалі не виконуватись (пропускатися). В алгоритмічних системах для цього є спеціальні блоки-розпізнавачі.

Залежно від того, на яку кількість гілок розгалужується алгоритм, він може бути простим або складним. Для простого розгалуженого процесу перевіряється одна умова (один розпізнавач), для складного – дві чи більше умов, кожна з яких виокремлює одну гілку.

Просте розгалуження відбувається за схемою, представленою на рис. 3.5.

Умова формулюється таким чином, щоб відповідь перевірки була „так" чи „ні".

Проста умова містить два вирази (значення), які поєднуються знаком операції відношення:

> більше за ...

< менше за ...

≥ більше або дорівнює...

≤ менше або дорівнює...

≠ не дорівнює...

Результатом перевірки умови є логічний вираз **ІСТИНА**, якщо умова виконується, або **ХИБНІСТЬ**, якщо умова не виконується.

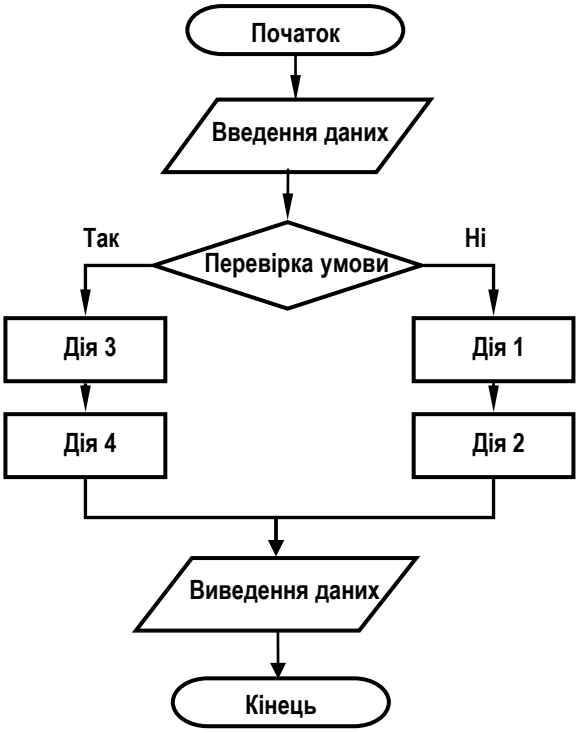


Рис. 3.5. Схема простого розгалуження

Приклад 2: **Дано дійсні числа x та y. Обчислити max(x, y).**
Алгоритм розв'язання представлений на рис. 3.6

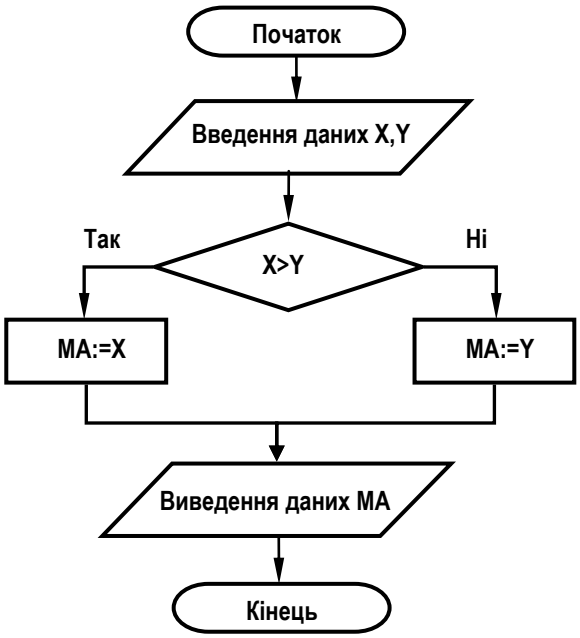


Рис. 3.6. Алгоритм обчислення максимального числа

Складна умова містить дві або більше простих умов, поєднаних знаками логічних операцій:

І – усі вказані умови мають виконуватися одночасно, тобто результатом усіх включених простих умов має бути **ІСТИНА**. При цьому операція **І** дасть результат **ІСТИНА,** а якщо хоча б одна з перелічених умов має результат **ХИБНІСТЬ**, операція **І** дасть результат **ХИБНІСТЬ**.

АБО – деякі з указаних умов можуть виконуватися, а деякі – ні. Якщо жодна з перелічених умов не виконується, то результатом операції **АБО** буде **ХИБНІСТЬ**, у всіх інших випадках – результат **ІСТИНА**.

НЕ – заперечення для умови.

Складне розгалуження відбувається послідовно, з відокремленням гілок за схемою, представленою на рис. 3.7.

Розгалужень може бути скільки завгодно. Збільшення кількості умов робить алгоритм більш складним і заплутаним, він втрачає наочність, перевірити його правильність досить складно. У таких випадках необхідно перехід до будь-якої гілки розгалуженого алгоритму пов'язати з деякою змінною, кожне значення якої відповідатиме одній з гілок розгалуження, тобто одному з варіантів обробки інформації. Це можуть бути не тільки окремі значення, а й проміжки значень, до яких належатиме конкретне значення змінної. Тоді всі логічні блоки алгоритму об'єднуються в один блок аналізу цієї змінної, який матиме не два входи, а стільки, скільки існує варіантів обробки.

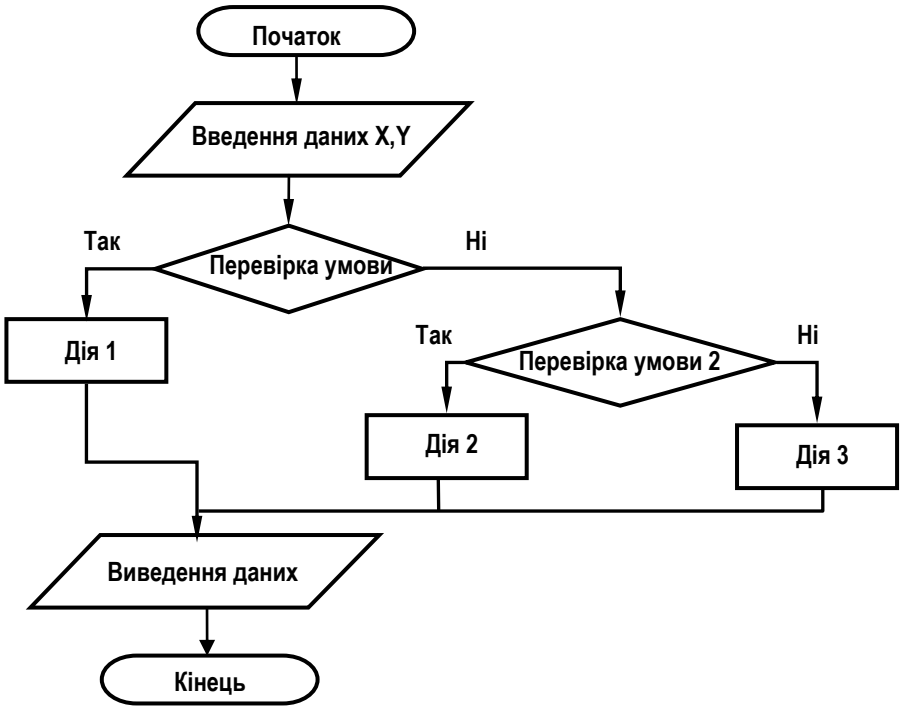


Рис. 3.7. Схема складного розгалуження

3.4. Циклічні алгоритми

Циклом називають повторення послідовної кількості кроків алгоритму. Обчислювальний процес, який містить цикл, називається циклічним. Керування повторенням циклу здійснюється за допомогою змінної, яка називається *параметром циклу*. Спочатку цьому параметру надають певне початкове значення. Потім цикл виконується зміною параметра. При кожному повторенні від початкового до кінцевого значення збільшується на величину, яка називається кроком циклу. Крок циклу може бути позитивним або негативним. Залежно від цього параметр циклу зростає або зменшується. Цикл припиняється, якщо значення параметра вийде за межі діапазону між початковими і кінцевим значенням.

Комбінуючи базові структури між собою, можна відтворювати алгоритм, який реалізує складний обчислювальний процес.

Структурна побудова алгоритму включає:

✓ використання методу покрокової деталізації;

✓ використання на кожному із зазначених кроків трьох перелічених базових структур;

✓ аналіз створеного алгоритму – методу ручної „прокрутки" – перевірка правильності функціонування створеного алгоритму шляхом підстановки вхідних значень і перегляду роботи алгоритму вручну.

Розрізняють такі види циклів:

– *цикл із передумовою* (спочатку умова, потім дія);

– *цикл із постумовою* (спочатку дія, потім умова);

– *цикл із параметром.*

Перші два види циклів використовуються у випадках, коли кількість повторень заздалегідь невідома.

При кожному черговому виконанні циклу перевіряється умова на продовження роботи, і якщо умова набуває результату **ІСТИНА**, цикл виконується, а якщо умова набуває результату **ХИБНІСТЬ** – цикл не виконується.

Цикл з передумовою характеризується тим, що спочатку перевіряється умова (звідси і назва – цикл з передумовою). Якщо умова виконується, то виконується дія. Потім знову перевіряється умова тощо. Виконання циклу припиняється, коли умова перестане виконуватися. Для цього необхідно, щоб дія в циклі впливала на зміну умови. У протилежному випадку відбудеться „зациклювання" – нескінченне виконання циклу.

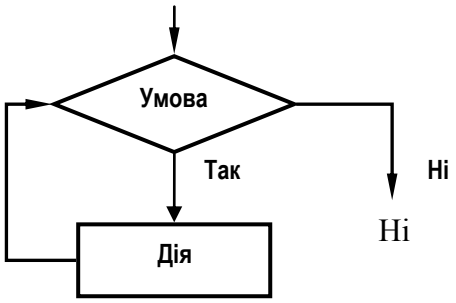


Рис. 3.8. Блок-схема циклу з передумовою

Цикл з постумовою характеризується тим, що спочатку виконується дія, а потім перевіряється умова. Повторення дії відбувається в тому випадку, якщо умова не виконується.

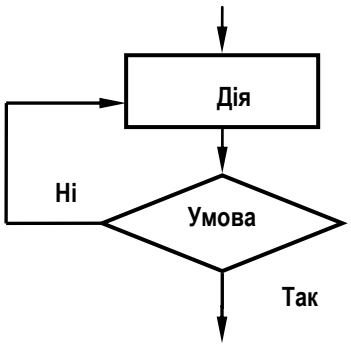


Рис. 3.9. Блок-схема циклу з постумовою

Дія в циклі з постумовою виконується завжди хоча б один раз, а з передумовою вона може не виконуватись жодного разу, якщо із самого початку умова не виконується.

Приклад. Задана спадна геометрична прогресія із першим членом a і знаменником b. Визначити всі члени цієї прогресії, значення яких більше від z. Блок-схема алгоритму обчислення членів спадної прогресії зображена на рис. 3.10.

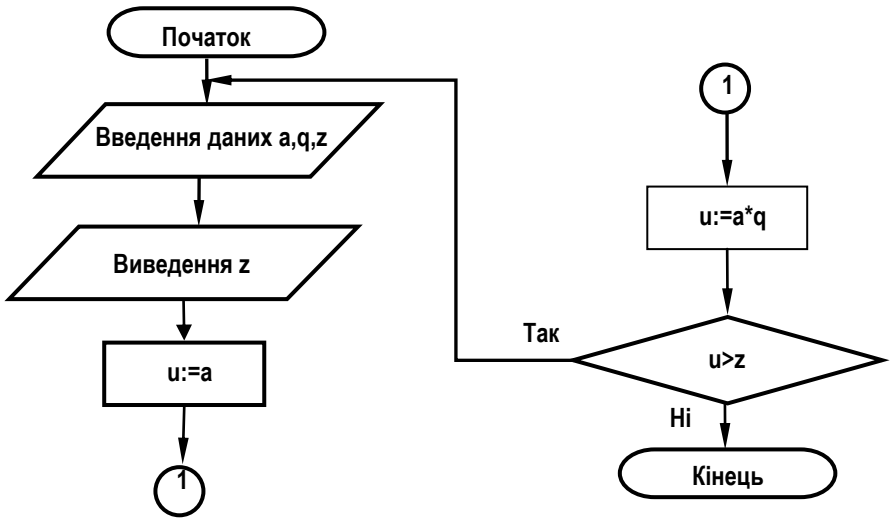


Рис. 3.10. Алгоритму обчислення членів спадної прогресії

Цикл з параметром. Відмітною особливістю цього циклу є те, що заздалегідь відома кількість повторень циклу. У зв'язку з цим, цикл буде виконуватися, доки значення змінної, яка називається параметром циклу, не перевищить зазначену кількість повторень.

Спочатку параметру циклу присвоюється певне початкове значення. Потім цикл виконується зі змінною параметра при кожному повторенні від початкового до кінцевого значень на величину, яка називається кроком циклу. Цикл припиняється, якщо параметр циклу здобуває значення, яке знаходиться поза межами діапазону між початковим і кінцевим значеннями.

Крок циклу може бути додатним або від'ємним. В залежності від цього параметр циклу зростає чи зменшується.

Цикл з параметром будується на основі одного з перших двох циклів. У більшості випадків він використовує цикл з передумовою (рис. 3.11)

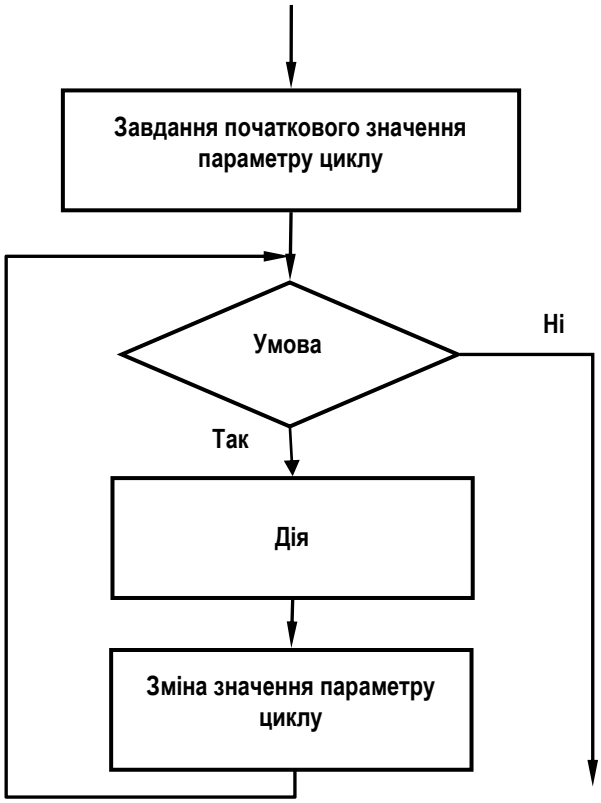


Рис. 3.11. Блок-схема циклу з параметром на базі циклу з передумовою

У зв'язку з широким використанням циклів для них використовується спеціальний символ „**Межа циклу**", який складається з двох частин: початку і кінця циклу.

У цих символах вказують ім'я циклу з параметром – початкове й кінцеве значення, а також крок циклу.

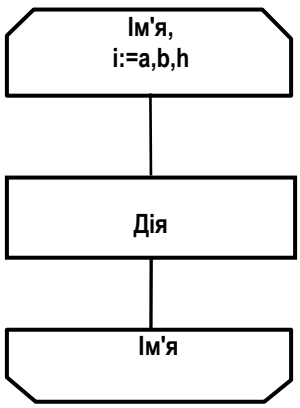


Рис. 3. 12. Цикл з параметром

тут Ім'я – назва циклу;

i – параметр циклу;

a – початкове значення параметра циклу;

b – кінцеве значення параметру циклу;

h – крок циклу.

Якщо *h = 1*, то його можна не вказувати.

При організації циклічних обчислювальних процесів часто виникає необхідність перебирати значення не однієї, а декількох змінних. У цьому випадку йде мова про зовнішній і внутрішні цикли. Для кожного значення параметра в зовнішньому циклі відбувається багаторазове виконання дій у внутрішньому циклі, який називається вкладеним.

Питання для самоконтролю

1. *Які типи алгоритмічних процесів Вам відомі?*
2. *Що розуміють під механічним алгоритмом?*
3. *Що розуміють під ймовірним алгоритмом?*
4. *Що розуміють під евристичним алгоритмом?*
5. *Що розуміють під лінійним алгоритмом?*
6. *Що розуміють під розгалуженим алгоритмом?*
7. *Що розуміють під циклічним алгоритмом?*
8. *Що розуміють під комбінованим алгоритмом?*
9. *В чому полягає сутність методу структурної алгоритмізації?*
10. *Що розуміють під терміном „параметр циклу”?*
11. *Які типи циклічних алгоритмів Вам відомі?*

4. АЛГОРИТМІЧНІ СИСТЕМИ

„Алгоритмічні концепції відіграють у процесі навчання і виховання сучасної людини фундаментальну роль, яку можна порівняти лише з роллю писемності”.

А. Тюрінг

4.1. Загальні відомості

Процес створення комп'ютерної програми для вирішення будь-якої практичної задачі, складається з декількох етапів:

- формалізації і створенні технічного завдання на вихідну задачу;
- розробки алгоритму вирішення задачі;
- написання, тестування, настроювання та документування програми;
- отримання рішення вихідної задачі шляхом виконання закінченої програми.

Можна вважати, що половина задачі виконана, якщо знати, що вихідна задача має роозв'язки. Зазвичай більшість задач, які зустрічаються на практиці, не мають чіткого і однозначного опису. Певні задачі такі, як розробка рецепту вічної молодості або збереження миру у всьому світі, взагалі неможливо сформулювати в термінах, які допускають комп'ютерне розв'язання. Навіть, якщо припустити, що задача може бути розв'язана на комп'ютері, як правило, для її опису потрібна достатньо велика кількість різноманітних параметрів.

Однак, якщо визначені аспекти задачі, що розв'язується можна представити в термінах певної формальної моделі, то взагалі кажучи, можна з'ясувати, чи має дана задача розв'язання, бо у цьому випадку в рамках формальної моделі можна визначити, існують чи ні методи і алгоритми розв'язання цієї задачі. Навіть якщо такі методи і алгоритми не існують на сучасному етапі, то залучення засобів і властивостей формальної моделі допоможе в побудові „більш-менш придатного” розв'язку вихідної задачі.

Алгоритми, які знаходять „придатне”, а не оптимальне рішення, називаються евристичними.

Коли побудована або підібрана „придатна” модель вихідної задачі, то природно шукати рішення в термінах цієї моделі. На цьому етапі основна мета полягає у побудові рішення в формі алгоритму, який складається з кінцевої послідовності інструкцій, кожна з яких має чіткий зміст і може бути виконана з кінцевими обчислювальними витратами за певний час.

Виникає питання: а наскільки важливо мати точне визначення алгоритму, якщо і без нього можливе складання і застосування алгоритмів (причому, навіть без вживання цього терміну)? Тим більше, що інтуїтивне поняття алгоритму хоча і не було строгим, але було настільки ясним, що практично до XX ст. не виникало ситуацій, коли математики розійшлися би у поглядах відносно того, чи є алгоритмом той чи інший конкретний процес. Це підтверджується багатьма яскравими прикладами, які надала алгебра і теорія чисел. Серед них алгоритм Евкліда, – знаходження найбільшого загального дільника двох натуральних чисел або двох цілочислених багаточленів, алгоритм Гауса для вирішення систем лінійних рівнянь, алгоритм знаходження раціональних коренів багаточлена однієї змінної з раціональними коефіцієнтами, алгоритм Штурма визначення дійсних коренів багаточлена з дійсними коефіцієнтами на певному відрізку дійсних чисел, алгоритм розкладання багаточлена однієї змінної над кінцевим полем на незвідні множники. Вказані алгоритмічні проблеми вирішені шляхом указівки конкретних вирішуваних процедур. Для отримання результатів такого типу достатньо інтуїтивного поняття алгоритму.

Становище істотно змінилося на початку XX ст., коли були сформульовані такі проблеми (задачі), алгоритмічне вирішення яких не було очевидним. Дійсно, для доказу існування алгоритму необхідно просто розв'язати дану задачу користуючись набором відомих прийомів, або при їх відсутності, запропонувати нові прийоми – в таких ситуаціях достатньо й інтуїтивного поняття алгоритму, щоб переконатися у тому, що описаний процес є алгоритмом.

Набагато складніше, довести факт неможливості побудови алгоритму розв'язання певної задачі (або класу задач). Без точного визначення алгоритму ця проблема втрачає сенс. Іншою підставою, що вимагала побудови точного визначення алгоритму, стала невизначеність поняття *„елементарність кроку”* при виконанні алгоритмічних дій. До того часу, доки математика вивчала числові об'єкти, дії з ними зводилися до певної послідовності обчислювальних операцій, і елементарними вважалися арифметичні операції, а також декілька логічних, пов'язаних з перевіркою відношень між величинами (рівність, нерівність, більше, менше тощо). Однак пізніше математика перейшла до вивчення властивостей і дій зі складними об'єктами – векторами, матрицями, множинами, функціями, і поняття елементарності кроку алгоритму перестало бути очевидним. Наприклад, чи можна вважати елементарним кроком взяття інтеграла або транспонування матриці? Зрозуміло що ні.

Нарешті, у тих ситуаціях, коли задача припускає побудову декількох алгоритмів розв'язання, то з теоретичної і з практичної точок зору, виявляється істотним питання їх зіставлення і вибору найбільш ефективного, що також неможливо без строгого визначення алгоритму.

Таким чином, виникла необхідність у точному визначенні поняття *„будь-який алгоритм”*, у максимально загальному значенні, під яке підходили б усі види алгоритмів.

Саме тому, у 20-х рр. XX ст., проблема визначення алгоритму стала однією з центральних математичних проблем. Це визначення, з одного боку, повинно було відповідати сутності інтуїтивного поняття алгоритму, а з іншого боку, бути формально строгим. Спроби формулювання такого поняття привели до появи у 30-х рр. XX ст. теорії алгоритмів як самостійної науки, яка разом з математичною логікою вивчає основні засоби математики – методи доказів, способи побудови аксіоматичних теорій, властивості математичних процедур і тощо. Коли ж у 40-50-х рр. XX ст. почався інтенсивний розвиток обчислювальної техніки і наук, пов'язаних з її функціонуванням і використанням, то з'ясувалося, що в основі цих наук також повинна лежати *теорія алгоритмів*, оскільки *комп'ютер може реалізувати тільки такі процедури, які можуть бути подані у вигляді алгоритмів*. Будь-яка програма є ні що інше, як запис алгоритму на мові, яку може „зрозуміти залізний виконавець” – комп'ютер.

Таким чином, з практичної точки зору також представляється необхідним уточнити поняття алгоритму.

Розглянемо, що саме в інтуїтивному понятті алгоритму потребує уточнення.

4.2. Основні вимоги до алгоритмів

1. Кожний алгоритм має справу з даними – вхідними, проміжними, вихідними. Для того, щоб уточнити поняття даних, фіксується кінцевий (обмежений) алфавіт вихідних символів (цифри, букви тощо) і вказуються правила побудови алгоритмічних об'єктів. Типовим використовуваним засобом є індуктивна побудова. Наприклад, визначення ідентифікатора у мові АЛГОЛ: ***ідентифікатор – це або літера, або ідентифікатор, до якого приписана праворуч або літера, або цифра***. Слова кінцевої довжини у кінцевих алфавітах – найбільш поширений тип алгоритмічних даних, а кількість символів у слові – природна міра об'єму даних. Іншим випадком алгоритмічних об'єктів є формули. Прикладом можуть слугувати формули алгебри предикатів[1] і алгебри висловлень. У цьому випадку не кожне слово в алфавіті буде формулою.

2. Алгоритм для розміщення даних потребує пам'яті. Пам'ять, зазвичай вважається однорідною і дискретною, тобто складається з однакових комірок, причому кожна комірка може містити один символ даних, що дозволяє узгодити одиниці виміру об'єму даних і пам'яті.

3. Алгоритм складається з окремих елементарних кроків, причому множина різних кроків, з яких складається алгоритм є кінцевою (обмеженою). Типовим прикладом множини елементарних кроків слугує система команд ЕОМ.

4. Послідовність кроків алгоритму детермінована, тобто після кожного кроку вказується, який крок потрібно виконувати далі або вказується, коли роботу алгоритму вважати закінченою.

5. Алгоритм повинний володіти результативністю, тобто зупинятися після кінцевого числа кроків (залежного від вихідних даних) з виведенням результату. Дану властивість іноді називають збіжністю алгоритму.

6. Алгоритм припускає наявність механізму реалізації, який за описом алгоритму викликає процес обчислення на підставі вихідних даних. Передбачається, що опис алгоритму і механізм його реалізації кінцеві.

Аналіз основних вимог до алгоритмів, показує, що вони аналогічні вимогам до обчислювальних машин. Зокрема, вимога 1 відповідає цифровій природі ЕОМ, вимога 2 – пам'яті ЕОМ, вимога 3 – програмі машини, вимога 4 – її логічній природі, вимоги 5, 6 – обчислювальному пристрою та його можливостям.

Є також деякі риси неформального поняття алгоритму, відносно яких не досягнуто остаточної угоди. Ці риси сформулюємо у вигляді питань і відповідей.

- Чи слід фіксувати кінцеву границю для розміру вхідних даних ?
- Чи слід фіксувати кінцеву границю для числа елементарних кроків ?
- Чи слід фіксувати кінцеву границю для розміру пам'яті ?
- Чи слід обмежити число кроків обчислення ?

[1] **Предикат** (лат. praedicatum – заявлене, згадане, сказане) – будь-який математичний вислів, у якому є щонайменше одна змінна. **Предикат** є основним об’єктом вивчення логіки першого порядку.

На всі ці питання далі приймається відповідь „**Ні**”, хоча можливі й інші варіанти відповідей, оскільки у фізично існуючих ЕОМ відповідні розміри обмежені. Однак теорія, що вивчає алгоритмічні обчислення, які можна здійснити принципово, не повинна врахувати такого роду обмеження, оскільки вони можуть бути подолані у принципі (наприклад, будь-який фіксований розмір пам'яті завжди можна збільшити на одну комірку).

Таким чином, уточнення поняття алгоритму пов'язано з уточненням алфавіту даних і форми їх подання, пам'яті і розміщення у ній даних, елементарних кроків алгоритму і механізму реалізації алгоритму. Однак ці поняття самі потребують уточнення. Зрозуміло, що їх словесні визначення потребують введення нових понять, для яких, у свою чергу, знову будуть потрібні уточнення тощо. Тому в теорії алгоритмів прийнятий інший підхід, заснований на конкретній алгоритмічній моделі, де усі сформульовані вимоги виконуються певним чином. При цьому алгоритмічні моделі, що використовуються є універсальними, тобто моделюють будь-які інші зрозумілі алгоритмічні моделі, які дозволяють зняти можливе заперечення проти такого підходу: чи не приводить жорстка фіксація алгоритмічної моделі до втрати спільності формалізації алгоритму? Тому дані алгоритмічні моделі ототожнюються з формальним поняттям алгоритму.

Уточнення поняття алгоритму провадиться в рамках *алгоритмічних моделей.*

Алгоритмічна модель визначає набір засобів, використання яких припустиме при розв'язанні задачі, тобто перелік елементарних кроків, способи визначення наступного кроку тощо.

Алгоритмічні моделі можуть бути *теоретичними* і *практичними.* З теоретичної точки зору, найбільший інтерес представляють моделі, які, з одного боку були б *універсальними*, тобто дозволяли б описати будь-який алгоритм, а з іншого боку – максимально *простими*, щоб використовували мінімум засобів для розв'язання задачі. Вимога простоти стає актуальною для виділення дійсно необхідних елементів і властивостей алгоритму і забезпечення тим самим доказу загальних стверджень про ці властивості. А в практичних і прикладних моделях більш значимим є зручність програмування і ефективність обчислень, тому їх засоби – набір елементарних кроків є набагато більшими і складнішими, що значно ускладнює теоретичний аналіз.

У подальшому, розглянемо основні типи алгоритмічних моделей, які розрізняються вихідними визначеннями, що таке алгоритм.

4.3. Підходи до побудови строгого визначення алгоритмів

Формальні визначення алгоритму з'явилися у 30-х - 40-х рр. XX ст. Можна виділити три основних типи універсальних алгоритмічних моделей, які різняться вихідними евристичними ідеями відносно того, що таке алгоритм.

Перший тип пов'язує поняття алгоритму з найбільш традиційними поняттями математики – обчисленнями і числовими функціями. Найбільш розвинена і вивчена модель цього типу – рекурсивні функції – є історично першою формалізацією поняття алгоритму.

Ця модель заснована на функціональному підході і розглядає поняття алгоритму з точки зору того, що можна обчислити за його допомогою. Такі функції отримали назву *обчислюваних.* Поняття обчислюваної функції не є строгим, як і поняття алгоритму. Однак, завдяки роботам А.Черча, К. Геделя, С. Кліні, була обґрунтована тотожність класу усюди визначених обчислюваних функцій з класом *частково рекурсивних* функцій, які визначаються строго. Це дозволило звести проблему алгоритмічної можливості розв'язання задач до доказу можливості (або неможливості) побудови рекурсивної функції, яка вирішує задачу. Була сформульована теза (теза Чёрча), яка стверджує, що даний клас функцій співпадає з множиною функцій, для яких є алгоритм обчислення значень за значенням аргументів. Саме цим шляхом А. Черчу вдалося довести нерозв'язність однієї з проблем математичної логіки – вирахування істинності предикатів.

Другий тип заснований на поданні алгоритму як певного детермінованого пристрою, спроможного виконувати у кожний окремий момент деякі примітивні операції або інструкції (строго фіксовану множину операцій). Таке подання не залишає сумнівів в однозначності алгоритму і елементарності його кроків. Основною теоретичною моделлю цього типу, створеною у 30-х рр. XX ст. є машина Тюрінга, яка представляє собою автоматну модель, в основі якої лежить аналіз процесу виконання алгоритму як сукупності набору інструкцій.

Основна ідея цього напрямку полягає у тому, що алгоритмічні процеси – це процеси, які може здійснювати відповідним чином улаштована *„машина”.* У відповідності з цією ідеєю, в точних математичних термінах були описані доволі вузькі класи машин, однак при цьому було доведено, що за допомогою цих машин можна здійснити або імітувати усі алгоритмічні процеси, які будь-коли описувалися математиками. Була сформульована теза (теза Тюрінга), яка стверджує, що будь-який алгоритм може бути реалізований на відповідній машині Тюрінга. Іншою теоретичною моделлю даного типу є машина довільного доступу (МДД) – введена відносно недавно (у 70-х рр. XX ст.) з метою моделювання реальних обчислювальних машин й отримання оцінок складності обчислень.

Даний підхід розвивався в працях Е. Поста і А. Тюрінга одночасно із згаданим вище підходом Черча. Доказ алгоритмічної можливості розв'язання задачі зводиться до доказу існування машини, яка її вирішує.

Третій тип алгоритмічних моделей – це перетворення слів у довільних алфавітах, у яких елементарними операціями є підстановки, тобто заміни частини слова (підслова) іншим словом. Перевага цього типу полягає у його максимальній абстрактності і можливості застосувати поняття алгоритму до об'єктів довільної природи. Моделі другого і третього типу доволі близькі і відрізняються головним чином евристичними підходами. Прикладами моделей цього типу є нормальні алгоритми Маркова і канонічні системи Поста.

Саме з метою визначення існування алгоритмічно розв'язних задач й існують алгоритмічні системи, кожна з яких призначена для побудови алгоритмів розв'язання задач певного класу. Це рекурсивні функції, нормальні алгоритми Маркова, машини Поста та Тюрінга, абстрактні автомати, формальні граматики тощо.

Пізніше була доведена теорема про можливість зведення однієї алгоритмічної моделі до іншої, наслідком якої стало ствердження: *„Будь-яку рекурсивну функцію можна обчислити за допомогою відповідної машини Тюрінга”* або *„для будь-якої задачі, що розв'язується за допомогою машини Тюрінга, існує нормальний алгоритм Маркова, який також дозволяє її вирішувати ”*.

Таким чином, всі алгоритмічні моделі виявляються *еквівалентними*, у чому проглядається глибокий зміст, оскільки результат обробки інформації визначається характером функції (алгоритмом) і вхідними даними, але не залежить від вигляду алгоритмічної моделі. Можна довести, що алгоритми Поста також зводяться до алгоритмів, які реалізуються за допомогою частково рекурсивних функцій. Справедливе й зворотне ствердження.

Отже, довівши неможливість побудови рекурсивної функції (машини Тюрінга, Поста, нормальних алгоритмів Маркова) можна стверджувати про відсутність для певної задачі алгоритму, який розв'язує цю задачу.

На сучасному етапі теорія алгоритмів утворює теоретичний фундамент обчислювальних наук. Використання комп'ютерів стало певним стимулом розвитку теорії алгоритмів і вивченню алгоритмічних моделей, до самостійного вивчення алгоритмів з метою порівняння їх робочих характеристик (кількості дій, витрат пам'яті) а також їх оптимізації. Виник важливий напрямок у теорії алгоритмів – складність алгоритмів й обчислень. Почала складатися метрична теорія алгоритмів, основним змістом якої є класифікація задач за класами складності. Самі алгоритми стали об'єктом точного дослідження, як і об'єкти, для яких вони призначені.

У цій галузі природно виділяються задачі отримання верхніх і нижніх оцінок складності алгоритмів, для яких методи розв'язання задач істотно різняться. Для отримання верхніх оцінок достатньо інтуїтивного поняття алгоритму. Для цього будується неформальний алгоритм розв'язання конкретної задачі і потім він формалізується для реалізації на придатній алгоритмічній моделі. Якщо спостерігається, що складність (час або пам'ять) обчислення для цього алгоритму не перевершує значення придатної функції при усіх значеннях аргументу, то ця функція оголошується верхньою оцінкою складності розв'язання розглянутої задачі. В галузі отримання верхніх оцінок отримано багато яскравих результатів для конкретних задач. Серед них розроблені швидкі алгоритми множення цілих чисел, багаточленів, матриць, розв'язання лінійних систем рівнянь, які вимагають значно менше ресурсів, ніж традиційні алгоритми.

Встановити нижню оцінку – означає довести, що жодний алгоритм обчислення не має складності меншої, чим задана межа оцінки. Для отримання результатів такого типу необхідна точна фіксація розглянутої алгоритмічної моделі, а такі результати можуть бути отримані тільки в дуже жорстких обчислювальних моделях. У зв'язку з цим, отримала розвиток проблематика отримання „відносних” нижніх оцінок, так звана теорія *NP*-повноти, яка пов'язана зі складністю вирішування переборних задач.

4.4. Визначення алгоритмічної системи

Абстрактний алфавіт та скінченна сукупність припустимих операцій складають алгоритмічну систему.

У різних алгоритмічних системах застосовують різні сукупності об'єктів (символів або станів), різні сукупності припустимих операцій та різні правила їх виконання. Одні й ті самі дані можуть бути представлені у різних алфавітах, а алгоритм розв'язання однієї й тієї самої задачі можна уявити послідовностями різних припустимих операцій. Отже, одна й та сама задача може бути розв'язана в межах різних алгоритмічних систем. Складність алгоритму та простота його реалізації залежатимуть від того, наскільки вдало було обрано алгоритмічну систему для розв'язання конкретної задачі.

Доведено можливість та правомірність переходу з однієї алгоритмічної системи до іншої у процесі побудови та реалізації алгоритму розв'язання задачі (тобто виконання еквівалентних перетворень алгоритму).

У будь-якій алгоритмічній системі розрізняють два типи припустимих операцій: операції дії та операції розпізнавання. За допомогою операції дії здійснюється перехід з одного стану інформації до іншого.

Операції розпізнавання використовуються для встановлення тих чи інших особливостей стану інформації для визначення операцій дій.

В теорії алгоритмів розроблені та використовуються різні алгоритмічні системи, кожну з яких призначено для побудови алгоритмів розв'язання задач певного класу. Це такі системи, як рекурсивні функції, нормальні алгоритми Маркова, машини Поста та Тюрінга, абстрактні автомати, формальні граматики та інші.

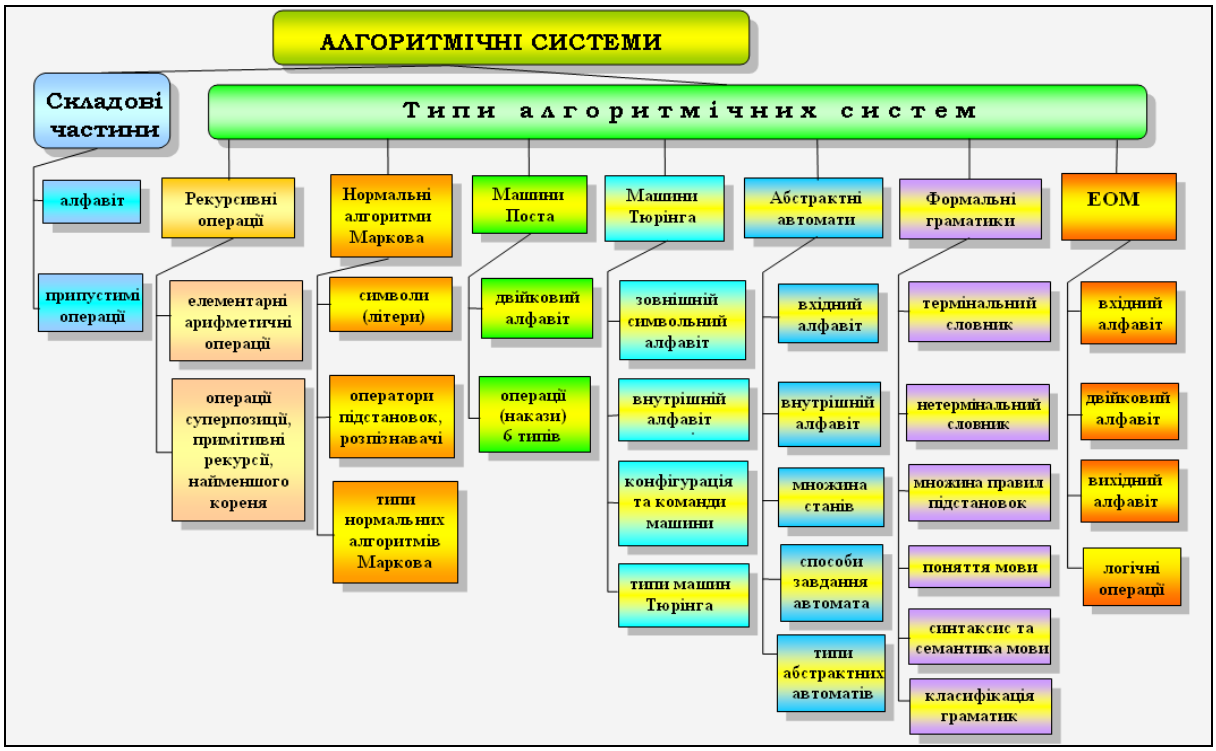


Рис. 4.1. Типи алгоритмічних систем

4.5. Виконання алгоритму

***Виконання алгоритму* – процес покрокового, точного виконання алгоритму виконавцем. *Виконавець алгоритму* – це суб'єкт або пристрій, здатний правильно інтерпретувати опис алгоритму і виконати перелік дій, які містяться у ньому.**

Вказівки щодо виконання дій для кожного виконавця, формулюються за допомогою певної *мови*, яка включає набір службових слів, що позначають дії (*команди*), а також синтаксичні правила їх об'єднання.

Сукупність припустимих команд утворює систему команд виконавця (СКВ).

Системи команд різних виконавців можуть різнитися детальністю опису дій. Як буде показано нижче, елементарною (найпростішою) дією при обробці дискретної інформації є заміна одного знака іншим. Однак можна побудувати абстрактний або реальний пристрій, який буде здатний виконувати цілу *низку* подібних елементарних дій за закладеним у нього правилом – певну *комплексну* (інтегральну) дію. При побудові алгоритму для такого виконавця, розробнику достатньо вказати послідовність інтегральних команд, а їх перетворення у низку істинно елементарних кроків буде здійснюватись самим виконавцем. Така „багатоступінчаста" алгоритмізація широко застосовується при управлінні комп'ютером.

Істинно елементарними, потрібно вважати дії процесора, хоча їх загальна кількість у сучасних процесорів досягає декількох сотень і навіть тисяч. Їх називають *машинними командами*, а їх позначення – *машинними кодами*. Першим (найнижчим) рівнем, на якому відбувається відхід від машинних кодів є *код асемблера* – внутрішня (тобто апаратно-залежна) мова програмування. Поєднання елементарних дій у складні команди на цьому рівні ще не відбувається і загальна кількість команд асемблера збігається з кількістю команд процесора, однак, використовується символьна форма подання машинних кодів і регістрів процесора – *мнемоніка*, яка вже є набагато зручнішою для користувача, на відміну від двійкового машинного коду. Переклад мнемонік у машинні команди здійснює програма – *асемблер*; саме з нею має справу програміст.

Команди, які поєднують елементарні дії, з'являються у мовах програмування високого рівня. Наприклад, у тексті програми достатньо написати *„Write”*, а транслятор мови перекладе її у послідовність елементарних кроків: переривань, пересилань тощо. Відносно до програміста, виконавцем у цьому випадку є транслятор мови програмування. Ще більшу ступінь інтеграції елементарних команд може забезпечити прикладна програма, яка є виконавцем відносно кінцевого користувача. Система команд такого виконавця включає всі команди управління, які подаються у вигляді меню, екранних кнопок, вікон настроювання й інших елементів інтерфейсу. Використання однієї команди може викликати ланцюжок складних дій, наприклад, вирівнювання багатьох рядків тексту.

Таким чином, при запису алгоритмів можливі ситуації, коли мова подання алгоритму є формальною, але у ній використовуються складні команди, які самим виконавцем перекладаються на рівень елементарних дій.

Якщо виконавцем є людина, то припустима нестрога формалізація подання алгоритмів. Людина володіє власним мисленням і знаннями, спираючись на які, вона може компенсувати неточності алгоритму, виконувати дії і домагатись результату. Подібні алгоритми слід вважати ще менш строгими, чим ті, що були розглянуті вище, оскільки вони, як правило, не володіють усіма перерахованими властивостями. Прикладами можуть слугувати кулінарні рецепти, інструкції із застосування побутових приладів, алгоритми розв’язання шкільних задач тощо.

За визначенням, алгоритм повинний бути обов'язково пов'язаний з обробкою дискретної інформації. Однак цей термін використовується і для позначення дій з управління виконавцем, який прямо не виконує перетворення інформації. Тому виникає необхідність розширення сфери застосовності поняття алгоритму на послідовність будь-яких дискретних дій, наприклад *інструкції з управління певними агрегатами і пристроями.*

Всі алгоритмічні задачі прийнято поділяти на два великих класи:

✓ задачі, пов'язані з обчисленням значення функції;

✓ задачі, пов'язані з розпізнаванням належності об'єкта заданій множині, що рівноцінно отриманню відповіді на питання: чи володіє об'єкт заданою властивістю ?

У першому випадку алгоритм *Q* починає працювати з початковими даними – словом *P,* складеним на основі алфавіту *A*, і за кінцеве число кроків (перетворень) він повинний зупинитися і видати результат $P_k = f_Q(P)$. Результат залежить (є функцією) від початкового слова, а також послідовності обробки, тобто залежить від самого алгоритму. При цьому *обчислення* розуміють у широкому змісті – як алфавітне перетворення.

В задачах, віднесених до другого класу, в результаті виконання алгоритму отримується відповідь на питання: *„Чи є істинним висловлення, що $x \in M$”* ? або, що аналогічно перевірці істинності предиката $x \in M$ і видається один з двох можливих результатів: ІСТИНА або ХИБНІСТЬ. Цей клас можна вважати варіацією першого, оскільки ***предикат – це функція, яка приймає два значення залежно від умови.*** Однак, розподіл цих класів задач корисний, оскільки приводить до двох важливих понять теорії алгоритмів – *обчисленої функції і розв'язної множини.*

Функція називається обчислюваною, якщо є алгоритм, який обчислює її значення.

Множина називається розв’язною, якщо є алгоритм, який для будь-якого об’єкту дозволяє визначити, чи належить він даній множині чи ні.

Алгоритм – це точний припис про виконання у певному порядку системи операцій, які визначають процес переходу від початкових (вихідних) до шуканого результату для вирішення задач певного типу.

Для розв’язання одного й того ж класу задач, може існувати декілька алгоритмів, які реалізують різні обчислювальні процеси. Ці обчислювальні процеси різняться набором дій та їх кількістю. Кількість дій в обчислювальному процесі – важлива характеристика алгоритму, оскільки вона визначає час і ресурси виконавця, які потрібні для виконання алгоритму.

Для розв’язання одного й того ж класу задач існують різні алгоритми різної складності.

Теорія алгоритмів зробила істотний вплив на розвиток ЕОМ і практику програмування. В теорії алгоритмів були передбачені основні концепції, закладені в апаратуру і мови програмування ЕОМ. Перераховані вище головні алгоритмічні моделі математично еквівалентні; але на практиці вони істотно різняться складними ефектами, які виникають при реалізації алгоритмів. Як наслідок – виникли різні напрямки у програмуванні. Так, мікропрограмування будується на ідеях машин Тюрінга, структурне програмування запозичало свої конструкції з теорії рекурсивних функцій, мови символьної обробки інформації (РЕФАЛ, ПРОЛОГ) беруть початок від нормальних алгоритмів Маркова і систем Поста.

4.6. Складність алгоритму

По мірі накопичення досвіду роботи в тій або іншій галузі знань людина починає інтуїтивно відчувати, що одна задача є більш складною, на відміну від іншої. Зрозуміло, такого суб'єктивного, не вираженого строгою кількісною мірою поняття, як „легше" або „важче", недостатньо. Тоді як же можна (і чи можна взагалі) визначити дійсну складність задачі? Для цього існують різні підходи. Найчастіше задачу називають складною через те, що вона володіє розгалуженою логічною структурою, містить велику кількість перевірок умов, переходів тощо. Програма, яка реалізує алгоритм її розв'язку, також здається нам складною, оскільки не важко зрозуміти, що ж вона робить. А ось для комп'ютера виконання цієї програми жодної складності не представляє! Тому що він просто виконує одну за іншою команди, які там написані. Для нього абсолютно неважливо, які вони саме – додавання або перевірки умови і переходу, цикли або команди вибору.

Більш складною може здатися програма з текстом, що складається з великої кількості операторів. Але й такий погляд на складність виявляється не цілком виправданим. Дійсно, для комп'ютера дві програми – що складаються з одного оператора циклу, який 100 разів повторює додавання двох чисел, і яка складається з 100 операторів додавання, виписаних підряд, – практично однакові.

В той час, коли починала складатися теорія алгоритмів, обчислювальні можливості людини всё ще залишалися вкрай скромними. Тому протягом тривалого часу розробка алгоритмів на практиці не представлялась можливою. Лише з появою перших комп'ютерів учені стали приділяти особливу увагу питанням створення алгоритмів. З початком їх практичного використання з'ясувалося, що величезне значення має ще їх одна властивість – ефективність. Не тільки весь алгоритм, але й кожний його крок повинні бути такими, щоб виконавець був здатний виконати їх за певний час, який лежить в певних припустимих межах. А якщо мова йде про обчислювальну машину, то в межах, зумовлених конкретним запасом пам'яті. З ефективністю алгоритму інтуїтивно пов'язане поняття вартості, під яким розуміється кількісна оцінка необхідних для його реалізації ресурсів. Більш точний зміст поняттю вартості надає поняття складності алгоритмів. Воно матеріалізує вартість у часі і в просторі, дозволяє оцінити термін виконання і об'єм необхідної при цьому пам'яті.

Часова складність алгоритму – це час T, необхідний для його виконання. Він дорівнює:

$$T=kt,$$

де k – кількість елементарних кроків алгоритму;

t – середній час виконання одного кроку.

Оскільки t залежить від виконавця, який реалізує алгоритм, то природно вважати, що складність алгоритму у першу чергу залежить саме від k.

Складністю алгоритму називається кількість дій в обчислювальному процесі цього алгоритму.

Зверніть увагу, саме в обчислювальному процесі, а не в самому алгоритмі. Для того, щоб порівнювати складність алгоритмів, треба щоб кількість дій підраховувалася в термінах однакових дій.

Просторова складність алгоритму виражається кількістю одиниць пам'яті, потрібних для його реалізації.

Просторова складність не може зростати швидше часової складності, адже на використання кожної додаткової одиниці пам'яті обов'язково знадобиться додатковий час. Тому для простоти часто обмежуються розглядом лише часової складності.

На практиці складність алгоритму зазвичай пов'язують з основним його параметром, який істотно впливає на кількість виконуваних дій. Зазвичай, це розмір масиву оброблюваних даних (розмір задачі). Наприклад, в задачі множення двох квадратних матриць це кількість їх рядків, в задачі упорядкування за алфавітом слів певного тексту – кількість слів тощо. Складність алгоритму (кількість операцій, необхідних для його реалізації) виражають у вигляді функції від об'єму вхідних даних. Наприклад, час роботи алгоритму, що виконує тільки операції зчитування і занесення n даних в оперативну пам'ять, визначається формулою:

$$T=an+b,$$

де a – час зчитування запису, b – сумарний час допоміжних операцій.

Коефіцієнти a і b невідомі, але вони залежать від характеристик комп'ютера, транслятора тощо. Однак більш важливим є характер залежності від основного параметра, оскільки саме він визначає складність. Тут має місце лінійна залежність від n, і складність алгоритму тому називається лінійною. Записується вона як $O(n)$, а читається «O більше від n». Можна дослідити процес упорядкування за зростанням n елементів масиву за допомогою так званого прямого вибору. Він виконується наступним чином: визначається найменший елемент у всьому масиві і міняється місцями з першим елементом; потім визначається найменший елемент у тій частині масиву, яка залишилась і перемінюється місцями з другим елементом тощо.

При першому разі при пошуку мінімального елемента виконуються ($n-1$) порівнянь, потім ($n-2$) порівнянь. Всього необхідно виконати $(n-1)+(n-2)+...+1=(n^2-n)/2$ порівнянь. Для великих n можна вважати, що складність визначається тільки старшим членом цього полінома, тому її записують як $O(n^2)$. Кажуть, що цей алгоритм має квадратичну складність. А ось складність алгоритму множення матриць (таблиць) розміру $n \times n$ буде уже кубічною, $O(n^3)$. Адже для обчислення кожного елемента результуючої матриці потрібно n множень і $(n-1)$ додавань, а усього цих елементів n^2. На практиці час виконання алгоритму може залежати від набору даних, до яких він застосовується. Наприклад, для деяких алгоритмів сортування він істотно скорочується, якщо первісно ці дані були частково упорядковані. Щоб ураховувати це, зберігаючи можливість аналізувати алгоритм незалежно від даних, розрізняють: максимальну складність $T_{max}(n)$ – час виконання алгоритму у тому випадку, коли обраний набір n даних найбільшим чином збільшує термін виконання алгоритму; середню складність $T_{ср.}(n)$ – середній час виконання алгоритму, який застосовується до n довільних даних.

Складність задачі – це складність найкращого відомого алгоритму для розв'язку даної задачі. Тому складність залежить від рівня розвитку методів розв'язання. Наприклад, відома задача перевірки планарності графа (тобто перевірки того, чи можна зобразити граф на площині так, щоб його ребра не перетинались). Перший алгоритм її розв'язку, запропонований був запропонований у 1930 р., мав складність $O(n^6)$. Потім його неодноразово удосконалювали, а у 1974 р. вдалося побудувати алгоритм складності $O(n)!$

Для розв'язку певної задачі можуть бути розроблені алгоритми різної складності. Логічно скористатися найкращим серед них, який володіє найменшою складністю але одночасно хочеться знати: чи не можна його ще додатково покращити? Для цього треба розділити задачі (і алгоритми) на групи – класи складності. Зрозуміло, що найкращими є лінійні алгоритми, які мають складність порядку $O(n)$, де n – розмірність вхідних даних. Такі алгоритми дійсно існують. Наприклад, додавання двох чисел стовпчиком у випадку, якщо одне з них складається з n, а інше – з m цифр, потребує не більше $max(n, m)$ додавань і не більше $max(n, m)$ запам'ятовувань, тобто даний алгоритм має складність порядку $O(n+m)$. Цей вираз показує тільки порядок величини – постійні фактори у ньому не враховуються. На жаль, кількість алгоритмів, які мають лінійну складність (а тим більше таких, складність яких не залежить від розмірності оброблюваних даних), вкрай мала.

Приклад. Нехай є алгоритм A. Майже завжди існує параметр n, який характеризує об'єм оброблюваних алгоритмом даних. Нехай функція $T(n)$ – час виконання A, а f – функція від n. Кажуть, що алгоритм A має теоретичну складність $O(f(n))$, якщо:

$$\frac{T(n)}{f(n)} \underset{n\to\infty}{\to} k$$

де k – константа. Якщо алгоритм виконується за фіксований час, який не залежить від розміру задачі, то стверджують, що його складність дорівнює $O(1)$.

4.7. NP – складні задачі

Розглянемо задачу комівояжера. Комівояжер (роз'їзний торговець) виїжджає з будинку, він повинен відвідати усіх своїх клієнтів, які проживають у різних населених пунктах. Відстані між ними відомі. Комівояжеру потрібно вибрати такий маршрут, щоб відвідати кожного клієнта рівно один раз і при цьому проїхати мінімальну відстань (або проїхати не більше, заданої відстані). Ці обмеження легко зрозуміти. Дійсно, комівояжер прагне максимально скоротити витрати на пальне (або не бажає витратити на нього більше певної суми).

Розв'язок можна шукати наступним чином. По черзі розглядаємо всі імовірні варіанти маршрутів, обираючи серед них той, який має мінімальну довжину. Кількість варіантів маршрутів легко підрахувати – воно дорівнює $n!$ Це означає, щоб знайти самий короткий маршрут, необхідно перевірити $n!$ варіантів розв'язання. Але $n!$ зростає швидше, чим 2^n . Так що, якщо комівояжеру потрібно відвідати багато клієнтів, розв'язок цієї задачі стає досить трудомістким. В той же час, перебираючи маршрути, можна виявити задовольняючий варіант дуже швидко – якщо пощастить, то практично відразу.

Алгоритм пошуку розв'язку даної задачі складається з двох етапів: пред'явлення певного маршруту (етап угадування); перевірка, чи є цей маршрут рішенням (етап перевірки); якщо є, то виконання алгоритму припиняється.

Етап перевірки має поліноміальну складність (він може полягати у виконанні усього лише одного порівняння).

Етап угадування полягає у знаходженні чергового маршруту і обчисленні його довжини; очевидно, що час його виконання пропорційний кількості населених пунктів n, тобто етап також має поліноміальну складність. Такі алгоритми називають недетермінованими. При певній частці вдачі можна розв'язати задачу комівояжера за поліноміальний час.

Задачі, які мають недетермінований розв'язок, який вдається отримати за поліноміальний час, були названі NP- складними (від nondeter-ministic – „недетерміновані" і polinomial – „поліноміальні"). Таких задач дуже багато, вони мають величезне практичне значення. Тому вкрай важливо знати відповідь на питання: а чи не існують детерміновані алгоритми їх розв'язання, які мають поліноміальну складність? Думки учених з цього приводу діаметрально протилежні. Більша частина з них упевнені, що знайти детерміновані алгоритми розв'язання цих задач ніколи не вдасться.

Існують алгоритми розв'язання багатьох NP-складних задач, які для значного діапазону вихідних даних забезпечують прийнятний час розв'язку. В основі таких алгоритмів часто лежить метод гілок і границь, який динамічно відсікає безперспективні напрямки перебору варіантів розв'язання. Тому головним моментом в практичному розв'язанні NP-складних задач є уміння будувати наближені або евристичні алгоритми, які ґрунтуються на деяких догадках (евристиках). При побудові наближених алгоритмів зазвичай вимагають, щоб вони мали поліноміальну складність. Відомо, що іноді вдається знайти розв'язок, який на 10–15% поступається оптимальному. Однак, незважаючи на таку гарну статистику, задачі як і раніше залишаються NP-складними і, на жаль, не мають гарантованих оцінок часу розв'язку. Навіть незначна зміна вихідних даних спроможна привести його до різкого стрибку. А в деяких випадках взагалі, час може стати неприпустимо великим.

4.8. Поліноміальні і експоненціальні алгоритми

Поліноміальним (або алгоритмом поліноміальної часової складності) називають алгоритм, у якого часова складність дорівнює $O(p(n))$, де $p(n)$ – полином від n. Задачі, що мають алгоритм розв'язання, складність якого визначається поліномом заданої постійної і не залежить від n степеня, відносять до класу *P*. Це фактично означає, що їх можна розв'язувати з використанням комп'ютера. Бо навіть для дуже великих значень n загальна кількість операцій, необхідних для реалізації поліноміального алгоритму, всё одно залишається такою, що сучасний комп'ютер без проблем зможе їх виконати за прийнятний час.

Існує велика кількість інших задач, складність яких становить не менше $O(x^n)$, де x – константа. Такі задачі відносяться до класу *E* і називаються експонентними. Серед них задача побудови усіх підмножин заданої множини або усіх піддерев заданого графа. Дійсно, якщо множина *M* містить n елементів, то загальна кількість підмножин, які з них можна утворювати, дорівнює 2^n. При великих значеннях n ця задача стає практично нерозв'язною, оскільки алгоритм її розв'язання повинен включати в себе 2^n кроків. Відповідно і алгоритми, в оцінку складності яких n входить в показник степеня, відносяться до експонентних.

Вважають, що задача не є добре розв'язною, поки для неї не отриманий поліноміальний алгоритм. Якщо ж для її розв'язку не існує подібного алгоритму, то задачу називають важко розв'язною. (Правда, при невеликих значеннях n експонентний алгоритм може бути навіть менш складним, чим поліноміальний. Незважаючи на це відмінності між алгоритмами цих типів досить велике і проявляється при великих значеннях n.)

Примітка. Відоме поняття поліноміального зведення задач. Кажуть, що задача *A* поліноміально зводиться до задачі *B*, якщо існує функція поліноміальної складності, яка перетворює розв'язок задачі *A* в розв'язок задачі *B*. Встановлено, що серед *NP*-складних задач існують такі, які можна поліноміально звести до будь-якої іншої задачі з класу *NP*. Їх назвали NP-повними задачами (до них відноситься зокрема, і задача комівояжера). Якщо б вдалося знайти детермінований алгоритм поліноміальної складності для розв'язку хоча б однієї NP-повної задачі, це б означало, що існують алгоритми поліноміальної складності розв'язку будь-якої задачі з класу *NP*.

Питання для самоконтролю

1. *З яких етапів складається процес створення комп'ютерної програми?*
2. *Які алгоритми називаються евристичними?*
3. *Які основні вимоги, які пред'являються до алгоритмів?*
4. *Навіщо створюються алгоритмічні моделі?*
5. *Які підходи до побудови строго визначення алгоритмів Вам відомі?*
6. *Які типи алгоритмічних систем Вам відомі?*
7. *Що розуміють під виконанням алгоритму?*
8. *Що розуміють під виконавцем алгоритму?*
9. *Що розуміють під системою команд виконавця?*
10. *Яка функція є обчислюваною?*
11. *Що розуміють під складністю алгоритму?*

5. РЕКУРСИВНІ ФУНКЦІЇ

Що я чую, я забуваю.
Що я бачу, я запам'ятовую.
Що я роблю, я знаю.
Конфуцій

Історично першою алгоритмічною системою була система, яка ґрунтувалась на використанні конструктивно визначених арифметичних (цілочисельних) функцій, які отримали назву рекурсивних.

5.1. Загальні поняття

Тривалий час вважалося, що будь-які математично строго і коректно сформульовані задачі можуть бути вирішені алгоритмічно, хіба що для них поки що не знайдено відповідних алгоритмів. Віра в унікальність алгоритмічних методів була підірвана у 1931 р. Куртом Геделем, який довів дві теореми.

Гедель Курт Фрідріх (28.4.1906, Брюнн, нині Брно–14.1.1978, Принстон,США) – видатний логік і математик. У ранньому дитинстві Гёдель переніс декілька хвороб, самою серйозною з яких був приступ ревматизму у шестирічному віці. Подих смерті, який Гёдель відчув у цьому віці, спричинив болісну іпохондрію, якою він страждав усе життя. У восьмирічному віці, читаючи медичний підручник, Гёдель переконався, що у нього слабке серце, хоча жоден з лікарів не знаходив тривожних симптомів. Пізніше, уже наприкінці життя, Гёдель помилково вирішив, що його хочуть отруїти, і, відмовившись від прийому їжі, вморив себе голодом. Ще в дитинстві Гёдель проявив надзвичайні здібності до природних наук і математики, і за свою допитливу натуру отримав сімейне прізвисько „пан Чому” (der Herr Warum").

Рис.5.1. Курт Фрідрих Гедель

К. Гедель вступив до Віденського університету, так і не зробивши вибір між математикою і фізикою, але натхненний запальними лекціями професора Ф. Фуртвенглера з теорії чисел, вирішив присвятити себе „числам”. Лекції були тим більш незвичними, що Фуртвенглер, паралізований від шиї і нижче, змушений був їх читати, сидячи в інвалідній колясці, без конспектів, а його асистент робив викладки на дошці.

К. Гедель отримав докторську ступінь у Віденському університеті (1930 р.), де він працював приват-доцентом математичного факультету (1933-1938 рр.).

З 1940 р К.Гедель працював у Принстонському інституті перспективних досліджень (США), який він відвідував у 1933-1935 рр. (з 1953 р. – професор Принстонського університету). Почесний доктор декількох американських університетів, член Американської академії мистецтв і наук, Американського філософського товариства, Національної АН США, Британської академії наук і королівського товариства.

К. Гедель визнаний гідним вищої нагороди США для учених – Ейнштейнівської премії (1951 р.) і медалі за науку (1975 р.). Основні напрямки досліджень – математична логіка і теорія множин. Займався також філософією математики. Ним доведена теорема про неможливість повної формалізації процесу логічного виводу. В теорії множин відома його робота про несуперечність гіпотези континууму. В класичному обчисленні предикатів сформулював і довів теорему про повноту (1930 р.) і теорему про неповноту (1931 р.).

У 1930 р. К. Гедель довів теорему про повноту обчислення предикатів першого порядку, згідно якої кожна істинна формула цього обчислення доказова. Цей результат вселяв надію на виконання програми Гільберта обґрунтування математики в рамках фінітизму, який є відгуком на виявлення в основах математики парадоксів. Намагаючись довести в контексті цієї програми несуперечність математичного аналізу, Гедель за допомогою особливої процедури (гёделевська нумерація), конструктивним чином довів теорему про неповноту будь-якої несуперечливої системи, достатньо великою для того, щоб містити арифметику натуральних чисел (1931 р.). Ця теорема встановлювала наявність в такого роду системах змістовно істинних, але недовідних і незаперечних (тобто нерозв'язних) її внутрішніми засобами пропозицій (*перша теорема Геделя*), що надає їй відтінок теореми чистого неіснування. Такою нерозв'язною пропозицією є й пропозиція, яка виражає несуперечність системи (*друга теорема Геделя*).

На мові рекурсивних функцій (визначення загальнорекурсивної функції було дано Гёделем у 1934 р.) це означало наявність множини, яку можна було рекурсивно перечислити, яка була нерекурсивна. Коли звістка про теорему Геделя досягла США, великий математик Джон фон Нейман негайно ж замінив частину свого курсу про програму Гільберта революційною роботою Геделя. Теореми Геделя можуть розглядатися і як теореми теоретичного програмування. Нерозв'язні пропозиції, спосіб побудови яких дається теоремами Геделя – метаматематичний, а не математичної природи. Лише наприкінці 70-х-початку 80-х рр. встановлена математична неповнота арифметики Пеано (Парис, Кірбі, Харрінгтон), що ще більше підсилює результат Геделя. Теореми Геделя можуть бути інтерпретовані в дусі свідоцтва недосяжності основних цілей програми Гільберта і непереборності „суб'єктивного” фактора і своєрідному „пріоритеті” змістовних відносно формальних компонентів у розвитку математики, що визначало характер переосмислення філософських поглядів на природу і механізми еволюції логіко-математичного знання після відкриття Геделя. Вважається, що теореми Геделя вказують на внутрішні границі формалізації і аксіоматизації.

Теореми Геделя пов'язані з обмежувальними тенденціями в основах математики, викликаними до прагненням уникнути парадоксів (протиріччя), яке досягається збідненням виразних засобів формалізованих мов (а звідси неминучим „відсіканням" низки важливих розділів класичної математики), або вибором систем, всередині яких бездоказова їх несуперечність. Між тим саме розуміння несуперечності складали лейтмотив обмежувальних тенденцій. В плані можливості оперувати з протиріччями без загрози стати тривіальними, великий інтерес представляються паранесуперечливі системи, в яких втрачають сенс традиційне прочитання і стандартні методи доказів теорем Гёделя.

К. Геделю належать великі результати в галузі інтуіціоністської логіки і математики (наприклад, розробка прийняття занурення інтуіціоністської арифметики в класичну). У 1939 р. К. Гедель довів відносну несуперечність узагальненої континуум-гіпотези і аксіоми вибору, для чого він запровадив поняття *„конструктивна множина"*.

В середині 1940-х рр. інтереси Гёделя переносяться у сферу філософії. Він ще у 1920-30-х рр. відвідував Віденський гурток позитивістів, але з останніми був згоден лише у питанні про значення логічних методів для аналізу наукового знання. В галузі філософії математики Гедель схилявся до платонізму (зокрема, Гедель заявляв, що володіє хистом екстрасенсорного сприйняття математичних сутностей). Увагу Геделя було приковано до Лейбниця, Канта, Гуссерля. Праці Гёделя із загальної теорії відносності, де ним був знайдений розв'язок рівнянь Ейнштейна у вигляді космологічної моделі з замкненими годиноподібними лініями (1949 р.) – були стимульовані саме захопленням філософією Канта.

Перша теорема Геделя стверджує, що у всякій несуперечливій теорії першого порядку (зокрема, у всякій формальній арифметиці) є така незаперечна формула, яка не може бути виведена в цій теорії.

Інакше кажучи, у будь-якій складній несуперечливій теорії існує таке твердження, яке засобами самої теорії неможливо а ні довести, а ні спростувати.

Друга теорема Геделя стверджує, що у всякій несуперечливій теорії першого порядку (зокрема у формальній арифметиці), є деяка формула, яка змістовно доводить її несуперечність, але не може бути виведена у цій теорії. Іншими словами, несуперечність теорії не може бути доведена засобами цієї теорії, однак існують докази несуперечливості формальної теорії, що використовують засоби, невимовні в ній.

Теорія, яка містить нерозв'язне, тобто невиведене й незаперечне, називається неповною і згідно з цим твердження отримали назву теорем Геделя про неповноту.

Виникле у зв'язку з теоремами Геделя про неповноту припущення про неможливість алгоритмічного вирішення багатьох математичних проблем викликало необхідність формалізації поняття алгоритму для доказу алгоритмічної нерозв'язності деяких задач.

Це пов'язане з тим, що, по-перше, не для усіх задач можливо створити алгоритми їх розв'язку. А, по-друге, щоб зробити математично строгий висновок про неможливість побудувати алгоритм, потрібно мати строге (формальне) визначення самого алгоритму. Але поняття алгоритму відноситься до фундаментальних невизначуваним поняттям. У питанні про алгоритм у нас собача позиція. Розуміти розуміємо, а сказати не можемо. Якщо десь і можна зустріти „визначення" алгоритму, то там, що ні слово – то алегорія... З цього тупика був знайдений нетривіальний вихід. Поняття алгоритму замінили строго формалізованими математичними моделями. Серед найвідоміших рекурсивні функції, машини Тюрінга і нормальні алгорифми Маркова. Ці математичні моделі виступають в ролі „конкретизацій поняття алгоритму". Тобто тривала практика підтверджує так звану тезу Чёрча, яку можна сформулювати наступним чином: *Для будь-якої алгоритмічно розв'язної задачі можна побудувати рекурсивну функцію (машину Тюрінга, нормальний алгорифм Маркова). І навпаки, для задач, для яких не можна побудувати перераховані конкретизації, не існує алгоритму розв'язання.*

Рекурсивні функції *(від латинського recursio – повернення)* назва, яка закріпилась за одним з найбільш популярних варіантів уточнення загального поняття арифметичного алгоритму (алгоритму, припустимі вихідні дані якого, представляють собою системи натуральних чисел, а можливі результати застосування є натуральними числами). Рекурсивні функції були введені науковий обіг у 30-х рр. XX ст. С. К. Кліні, які в свою чергу ґрунтувались на дослідженнях К. Геделя, Ж. Ербана та інших видатних математиків.

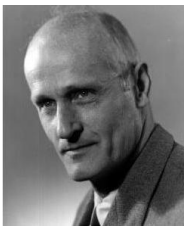


Рис.5.2. Кліні Стефан Коул

Кліні Стівен Коул (Kleene Stephen Cole) (5.1.1909, Хартфорд, штат Коннектикут – 25 .01. 1994, Мадісон, Вісконсін, США) – відомий американський логік і математик, член Національної Академії Наук США (1969 р.). Закінчив Принстонський університет (1930 р.). У 1934 р. отримав ступінь доктора філософії в Принстонському університеті. З 1935 р. працює у Вісконсинському університеті (з 1948 р. професор). Основні праці з теорії алгоритмів і рекурсивних функцій, а також проблем інтуіціоністської логіки і математики. Зокрема, ним доведена еквівалентність, введеного А.Черчем поняття λ-визначності функції з загальнорекурсивністю. Введене Кліні поняття (рекурсивної можливості реалізації формул лежить в основі інтуіціоністської інтерпретації арифметичних суджень. Кліні – автор широко відомих монографій з

математичної логіки, основам математики і теорії рекурсивних функцій. В книзі «Вступ до математики» (1952 р.) дав нарис стану основ математики і напрямків математичної логіки, які виникли в середині XX ст. у зв'язку з цим. У ній докладно розглянуті інтуіціоністські системи. Запровадив поняття рекурсивної реалізуємості формул. Відома класифікація Кліні-Мостовського для теоретико-числових предикатів (1943р.), введена незалежно С.Кліні і А.Мостовським.

Рекурсією називають таку конструкцію, при якій функція викликає саму себе. Іншими словами, ***рекурсія – часткове визначення об'єкта через себе, визначення об'єкта з використанням раніше визначених.*** Рекурсія використовується коли можна виділити самоподібність задачі. Рекурсивні функції – це обчислювані функції.

Як відомий герой Мольєра не підозрював, що він розмовляє прозою, так і ми, найчастіше не підозрюємо, що користуємося рекурсією. Як приклад, наведемо декілька відомих застосувань рекурсії в математиці та інших галузях науки, техніки, виробництва.

Рекурсія у математиці. Факторіал цілого невід'ємного числа n позначається $n!$ і визначається як рекурсія:

$$n!=n\times(n-1)!$$ при $n>0$ і $n!=1$ при $n=0$

Числа Фібоначчі визначаються як:

- перше і друге числа Фібоначчі дорівнюють 1;
- для $n>2$, n–i число Фібоначчі дорівнює сумі $(n-1)$-го і $(n-2)$-го чисел Фібоначчі.

Числовий ряд, що носить сьогодні ім'я Фібоначчі, виріс із проблеми з кроликами, яку Фібоначчі виклав у своїй книзі „Liber abaccі", написаної у 1202 р.

Фермер годує кроликів. Кожен кролик народжує одного кролика коли йому стає 2 місяці, а потім дає потомство в 1 кролик кожного наступного місяця. Скільки кроликів буде у фермера через n місяців, якщо спочатку у нього був лише один (вважаємо, що кролики не гинуть і кожен народжений дає потомство за вищеописаною схемою)?

Очевидно, що першого та другого місяця у фермера лишається один кролик, оскільки потомства ще немає. На третій місяць буде два кролика, оскільки перший через два місяці народить другого кролика. На четвертий місяць перший кролик дасть ще одного, а другий кролик потомства не дасть, оскільки йому ще тільки один місяць. Отже на четвертий місяць буде три кролики.

Можна помітити, що кількість кроликів після n – го місяця дорівнює кількості кроликів, які були у $n-1$ місяці плюс кількість народжених кроликів. Останніх буде стільки, скільки є кроликів що дають потомство, або дорівнює кількості кроликів, яким вже виповнилося 2 місяці (тобто кількості кроликів після $n-2$ місяця).

Якщо через F_n позначити кількість кроликів після n-го місяця, то має місце наступне рекурентне співвідношення:

$F_n = F_{n-1} + F_{n-2}$, $F_1 = F_2 = 1$

Покладемо $F_0 = 0$, при цьому співвідношення при $n=2$ залишиться істинним. Таким чином утворюється послідовність

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots ,$$

Рекурсія в геометрії. Практично всі геометричні фрактали задаються у формі нескінченної рекурсії. (наприклад, трикутник Серпінського).

Рис.5.3. Трикутник Серпінського

Рис.5.4. Вацлав Серпинський (14 березня 1882 р., Варшава, Польща – 21 жовтня 1969 р., Варшава) – видатний польський математик. Відомий своїми працями з теорії множин, аксіоми вибору, континуум-гіпотезі, теорії чисел, теорії функцій а також топології. Автор понад 700 статей і 50 книг.

Народився Вацлав Серпінський 14 березня 1882 р. у Варшаві. Батько його був лікарем. Читати і писати хлопчик навчився ще до школи. У гімназії Вацлав виявив особливий інтерес і здібності до вивчення математичних дисциплін.

У 1900 р. Серпінський успішно закінчив гімназію і вступив на фізико-математичний факультет Варшавського університету. На той час у Варшавському університеті бракувало викладачів-професорів.

Царський уряд Росії уникав призначати професорів-поляків в цей університет, побоюючись їх політичної неблагонадійності, а професори-росіяни йшли туди неохоче. У 1896 р. викладати математичні дисципліни у Варшавський університет був призначений молодий доктор математичних наук, геніальний український математик Георгій Феодосійович Вороний, який тільки-но закінчив Петербурзький університет. В цей час на фізико-математичному факультеті працювало лише два професори. Георгій Феодосійович добре розумів прагнення польського студентства до незалежності і свободи.

Молодий професор з усією енергією взявся налагоджувати нормальну роботу факультету — він читав лекції з кількох математичних дисциплін, писав для студентів підручники, видавав свої лекції і вів семінарські заняття. Вороний звернув увагу на здібного й активного студента, який дуже цікавився математикою і працював в його семінарі. Це був Вацлав Серпінський. Професор уважно стежив за успіхами Серпінського і уже в 1904 р. запропонував йому написати перший науковий твір з галузі теорії чисел. Студент Серпінський блискуче справився з цією темою і за написану працю був нагороджений Радою університету золотою медаллю.

Наступив буремний 1905 рік. По всій Росії почались революційні виступи. Ця хвиля захопила й Варшаву. Студенти, разом з революційними робітниками і населенням Варшави, вийшли на демонстрацію. Царський уряд за допомогою поліції і солдатів жорстоко придушив усі виступи. Університет майже на два роки був закритий.

Студенту В. Серпінському, який брав активну участь у демонстраціях, довелося залишити Варшаву і переїхати до Кракова, який в той час знаходився на території Австро-Угорської імперії. Там він вступив до Ягеллонського університету, заснованого ще в 1364 р. У цьому стародавньому університеті через два роки Серпінському було надано звання доктора філософії.

Ще через два роки (1908) В. Серпінського призначили доцентом Львівського університету, в якому він зайняв посаду професора. Серпінський першим у Львові викладав теорію множин, теорію функцій дійсної змінної та аналітичну теорію чисел, разом з Юзефом Пузиною керував математичним семінаром. Студенти розмножували лекції Серпінського літографським способом, пізніше багато з цих лекцій були надруковані.

В ці роки у Львівському університеті навчався Зарицький Мирон Онуфрійович, український математик, під впливом якого В. Серпінський визначив напрямок подальшої наукової роботи.

Поряд з читанням лекцій Серпінський продовжував власні дослідження і у 1907 р. видав перший великий науковий твір з теорії множин та її застосування до різних галузей математики (топології, теорії функцій дійсної змінної тощо). У 1911 р. Краківська Академія Наук нагородила його премією за праці, опубліковані в 1909–1910 рр., а у 1913 р. Серпінський був відзначений за твір «Нариси теорії множин».

Наближався 1914 р., рік початку першої світової війни. Царський уряд вживав заходів до «очищення» свого тилу від «підозрілих» елементів близьких до майбутнього фронту областей. Професор Серпінський був висланий у Вятку, а потім жив у Москві. Відірваний від рідної землі і улюбленої праці, В. Серпінський тяжко переживав, цю незаслужену кару. Проте в цей час відбувається його зближення з деякими видатними російськими математиками, зокрема з професором Лузіним.

Повернутися до Польщі В. Серпінський зміг лише після революції 1917 р. в Росії. Він прибув туди в 1918 р. і почав викладати математичні дисципліни у Варшавському університеті. Професор Серпінський розгорнув велику організаторську роботу, щоб нормалізувати й налагодити роботу всіх відділів університету. Насамперед він дбав про поповнення професорського складу свого факультету. Серед його студентів було багато обдарованих людей, і деякі з них пізніше стали відомими математиками (Куратовський, Заранкевич, Попруженко та інші).

Під час другої світової війни багато вчених Польщі загинули. Одних замучили фашисти в концентраційних таборах, інших знищили в газових камерах. Університетські приміщення були зруйновані, бібліотеки спалені. Серпінському пощастило врятуватись, переховуючись на підпільних квартирах. Він і там знаходив час і працював зі своїми студентами.

Після звільнення Польщі Радянською Армією з лютого 1945 р. Серпінський почав працювати у Краківському університеті, а восени того самого року переїхав до Варшави і став викладати математичні дисципліни у Варшавському університеті. Викладацьку і наукову роботу Серпінського у 1949 р. відзначив уряд ПНР, присудивши йому Державну премію.

У 1952 р. Серпінського обрано віце-президентом Польської Академії Наук, заснованої на базі Краківської Академії наук, яка існувала майже 80 років. У 1948 р. за ініціативою і активною участю В. Серпінського було організовано Державний математичний інститут, який мав проводити дослідження у галузі теорії множин, топології, функціонального аналізу, теорії функцій дійсної змінної і геометрії чисел.

Серпінського заслужено вважають батьком польської школи математиків. Помер він у 1969 р. Усе своє довге життя Вацлав Серпінський віддав благородному служінню науці.

Фракта́л (лат. *fractus* – дроблений, зламаний, розбитий) – складна геометрична фігура, яка володіє властивістю самоподоби, тобто складається з декількох частин, кожна з яких подібна всій фігурі цілком. У більш широкому значенні під фракталами розуміють множини точок в евклідовому просторі, які мають дробову метричну розмірність (у значенні Мінковського або Хаусдорфа), або метричну розмірність, відмінну від топологічної.

Рекурсія у логіці. Визначення у логіці, яке використовує рекурсію отримало назву індуктивного визначення.

Ханойська вежа. Легенда свідчить про те, що у Великому храмі міста Бенарас, який розташований під Ханоєм, під собором, що позначав середину світу, знаходиться бронзовий диск, на якому укріплені 3 алмазних стержні, висотою в один лікоть і товщиною з бджолу. За легендою, давним-давно монахи цього монастиря провинилися перед богом Брамою.

Розгніваний, Брама спорудив три високих стержні і на один з них поклав 64 диска з чистого золота, причому так, що кожний менший диск лежить на більшому. При перекладанні можна використовувати всі три стержні. Як тільки усі 64 диска будуть перекладені зі стержня, на який Бог Брама склав їх при створенні світу, на інший стержень, вежа разом із храмом перетворяться на пил і під громові розкати загине світ.

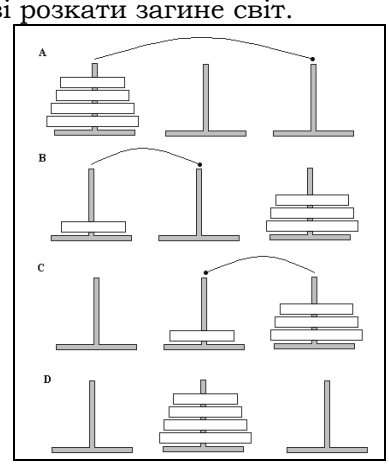


Рис.5.5. Рекурсивний розв'язок головоломки „Ханойська вежа"

Рекурсивний варіант розв'язку задачі:

Потрібно застосувати рекурсивний алгоритм, переклавши *n–1* кільце з першої піраміди на третю піраміду. Потім зробити очевидний хід, переклавши останнє найбільше кільце з першої піраміди на другу. Потім знову застосувати рекурсію, переклавши *n–1* кільце з третьої піраміди на другу піраміду.

Кількість перекладань залежно від кількості кілець обчислюється за формулою:

$$k=2^n-1.$$

Кількість переміщень дисків, які повинні зробити монахи, дорівнює 18 446 744 073 709 551 615. Якби монахи, працюючи день і ніч, виконували щосекундні переміщення диску, їх робота тривала б 580 мільярдів років.

В інформатиці задачі, засновані на легенді про Ханойську вежу, часто розглядають як приклад використання рекурсивних алгоритмів і перетворення їх до нерекурсивних.

Програма обчислення кількості переміщень кілець на мові Паскаль представлена нижче.

```
procedure perenos(a,c,b:char; n:integer);
begin
    if(n<1) then exit;
    perenos(a,b,c,n-1);
    writeln('z ',a,' na ',c,'-',n);
    readkey;
    perenos(b,c,a,n-1);
end;

begin clrscr;
perenos('a','b','c',5);
end.
```

Рекурсія у фізиці. Класичним прикладом нескінченної рекурсії у фізиці є два поставлені одне проти одного дзеркала: у них утворяться два коридори загасаючих відображень дзеркал.

Іншим прикладом нескінченної рекурсії є ефект самозбудження (додатного зворотного зв'язку в електронних схемах посилення, коли сигнал з виходу попадає на вхід, підсилюється, знову попадає на вхід схеми і знову підсилюється. Підсилювачі, для яких такий режим роботи є штатним, називаються автогенератори.

Рекурсія у виробництві. Приклад рекурсії у виробництві наведений на рис.5.6.

Рис.5.6. Приклад рекурсії при виготовленні сувенірів

Рекурсія в літературі. Російська народна казка-пісня „У попа була собака...” виявляє приклад рекурсії:

У попа була собака, він її любив
Вона з'їла шматок м'яса, він її убив
Убив і закопав,
А на могилі написав:
«У попа була собака, він її любив
Вона з'їла шматок м'яса, він її убив
Убив і закопав,
А на могилі написав:
«У попа була собака, він її любив
Вона з'їла шматок м'яса, він її убив
Убив і закопав,
А на могилі написав:
...
(лапки цитат ніколи не закриються).

Рекурсія у програмуванні. Рекурсія у програмуванні – виклик функції або процедури з неї самої (зазвичай з іншими значеннями вхідних параметрів), або безпосередньо через інші функції (наприклад, функція *A* викликає функцію *B*, а функція *B* – функцію *A*).

Якщо команда звернення алгоритму до самого себе знаходиться у самому алгоритмі, то таку рекурсію називають *прямою*. Можливі випадки, коли рекурсивний виклик даного алгоритму відбувається із допоміжного алгоритму, до якого у даному алгоритмі є звернення. Така рекурсія називається *непрямою*.

Кількість вкладених викликів функції або процедури називається глибиною рекурсії.

Сутність рекурсивного визначення об'єкта полягає в тому, що таке визначення здатне описувати нескінченно велику кількість об'єктів. За допомогою рекурсивної програми можна описати нескінченне обчислення, причому без явних повторень частин програми.

Корисність, важність і необхідність рекурсії, як одного з концептуальних методів розв'язання практичних задач підкреслювалася багатьма метрами інформатики. Пошлемося лише на двох лауреатів премії Тюрінга: американського фахівця з системного програмування Д. Кнута та англійського теоретика інформатики Ч. Хоара.

Рис.5.7. Дональд Ервін Кнут (англ. *Donald Ervin Knuth*, 10 січня 1938) – американський інформатик, ідеолог програмування та почесний професор Стенфордського університету.

Дональд Кнут народився 10 січня 1938 р. у США, автор фундаментальної праці „Мистецтво програмування”; вважається одним з батьків аналізу складності алгоритмів. Розробник типографічної системи TeX та пов'язаної мови визначення шрифтів і системи їх рендерингу METAFONT.

На юного Дональда Кнута сильно вплинули праці Андрія Петровича Єршова, який згодом став його товаришем. Ось як описує Д. Кнут перше знайомство з працями А. Єршова. *„Це почалося ще коли я був студентом останнього курсу. Тоді тільки з'явилася книжка Андрія „Програмування для БЕСМ”, і ми, група студентів, отримали змогу переконати викладача російської мови включити її до курсу як одну з двох збірок текстів для вивчення наукової лексики*”.

Професор Д. Кнут удостоєний багатьох премій та нагород у галузі програмування та обчислювальної математики, серед яких премія Тюрінга (1974 р.), Національна наукова медаль США (1979 р.) та AMS Steele Prize за серію науково-популярних статей, премія Харві (1955 р.), премія Кіото (1996 р.) за досягнення в галузі передових технологій, премія імені Грейс Мюррей Хоппер (1971 р.) тощо.

У кінці січня 2009 р. Кнут займав 20-те місце у списку найцитованіших авторів у проекті CiteSeer.

„На світовому ринку комп'ютерної літератури існує багато книг, призначених для навчання основним алгоритмам, які використовуються при програмуванні. Їх доволі багато, і вони в значному ступені конкурують між собою. Однак серед них є особлива книга. Це тритомник „Мистецтво програмування” Д.Е. Кнута, який стоїть поза усякої конкуренції і входить до золотого фонду світової літератури з інформатики...”

Рис.5.8. Сер Чарльз Ентоні Ричард Хóар (англ. *Charles Antony Richard Hoare* або *Tony Hoare* або *C.A.R. Hoare*, р. 11 січня 1934 р., Коломбо, Цейлон, Британська імперія, нині Шрі-Ланка) – видатний англійський учений у галузі інформатики та обчислювальної техніки.

Чарльз Ентоні Ричард Хóар став популярним, як розробник алгоритму „швидкого сортування” (1960 р.), який сьогодні є одним з найвідоміших алгоритмів сортування.

Іншими цікавими результатами його робіт є мова Z специфікацій і паралельна модель взаємодії послідовних процесів (CSP, Communicating Sequential Process). До числа його заслуг відносять розробку логіки Хоара (англ. *Hoare Logic*), наукової основи для конструювання коректних програм, яка використовується для визначення і розробки мов програмування. Хоар написав низку праць зі створення специфікацій, проектування, реалізації і супроводу програм, які показують важливість наукових результатів для збільшення продуктивності комп'ютерів і підвищення надійності програмного забезпечення.

Ч. Хоара народився в Коломбо, столиці Шрі-Ланки. Отримав ступінь бакалавра з класичних мов в Університеті Оксфорда (Merton College)) у 1956 р. Проходив службу у ВМС Великобританії у 1956–1958 рр. Вивчивши російську мову, навчався комп'ютерному перекладу під керівництвом Андрія Миколайовича Колмогорова у Московському державному університеті.

В 1960 р. через політичну кризу, пов'язану зі знищенням розвідувального літака U-2,залишив Радянський Союз і почав працювати у невеличкій компанії з виробництва комп'ютерів Elliott Brothers, де займався реалізацією мови ALGOL60. Там же він почав займатися розробкою алгоритмів.

У 1968 р. став професором інформатики і обчислювальної техніки Королівського Університету Белфаста (Queen's University, Belfast).

У 1977 р. повернувся до Оксфорду, як професор обчислювальної техніки, очоливши дослідницьку групу Programming Research Group, до задач якої входило зміцнення зв'язків промислових, академічних і державних структур, які працювали в галузі IT-індустрії. Тематика його досліджень в Оксфорді – коректність програмних специфікацій, проектування і розробка критичних і некритичних систем.

У 1999 р. вийшов на пенсію у званні почесного професора і перейшов на посаду провідного дослідника в Microsoft Research у Кембриджі.

У 1980 р. отримав Премію Тюрінга (премія ACM) за „видатні досягнення у визначенні і дизайні мов програмування".

У 2000 р. був визнаний гідним лицарського титулу за заслуги в галузі освіти і комп'ютерних наук, Премії Кіото тощо.

Д. Кнут широко використовував рекурсію при викладі матеріалу у своїй класичній праці „Мистецтво програмування для ЕОМ". Ч. Хоару належать наступні слова „треба віддати належне генію розробників Алгола-60 за те, що вони включили до своєї мови рекурсію і дали мені тим самим можливість доволі елегантно описати мій винахід (мова йде про швидке сортування – Quick Sort). До цього лише треба додати, що сьогодні, практично усі діючі мови програмування підтримують рекурсію.

Рекурсія – метод визначення класу чи об'єктів методів попереднім завданням одного чи декількох (звичайно простих) його *базових* випадків чи методів, а потім завданням на їхній основі правила побудови класу, який визначається.

Під рекурсією розуміють такий спосіб організації обчислювальні системи, при якому вона в окремі моменти свого розвитку, зумовленими її правилами, може створювати (викликати) власні змінені копії, взаємодіяти з ними і включати їх в свою структуру.

Рекурсія – найважливіше фундаментальне поняття теорії алгоритмів. Цією теорією було доведено, що використовуючи рекурсію, можна з обмеженої кількості функціональних одиниць отримати усе розмаїття обчислюваних функцій.

Термін рекурсивна функція в теорії алгоритмів використовується для позначення трьох класів функцій:

– примітивно рекурсивні функції;

– загальнорекурсивні функції;

– частково рекурсивні функції.

Останні збігаються з класом обчислюваних за Тюрінгом функцій. Визначення цих трьох класів сильно пов'язані.

Множина частково рекурсивних функцій включає в себе множину загальнорекурсивних функцій, а загальнорекурсивні функції включають в себе примітивно рекурсивні функції. Частково рекурсивні функції іноді називають просто рекурсивними функціями.

Поняття рекурсії є достатньо простим для розуміння і не пов'язане зі знанням будь-якого певного формалізму або спеціальної нотації. У загальному випадку на рекурсію треба дивитися як на введення у визначення об'єкта посилання на сам об'єкт, або як на прийом зведення розв'язання певної задачі до розв'язання „більш простої" задачі такого ж класу.

У програмуванні це проявляється побудовою програм (процедур і функцій), які при виконанні звертаються безпосередньо самі до себе або через ланцюжок інших програм. Гадана при цих самовикликах або послідовних циклічних викликах видимість порочного кола (circulus vitiosus – лат.), не більш чим ілюзія. У багатьох конкретних випадках простими міркуваннями шляхом відстеження значень однієї або декількох керуючих величин вдається здійснити доказ завершуваності обчислень за кінцеву кількість кроків.

Історично першою алгоритмічною системою яка почала використовувати рекурсію, була система, яка ґрунтувалась на використанні конструктивно визначених арифметичних (цілочисельних) функцій, які назвали рекурсивними. Значення такої функції y для будь-якого довільного значення аргументу x (з області визначення функції) знаходиться через значення цієї функції від аргументу $x–1$. Тобто можна побудувати рекурентні співвідношення, які визначають, як саме залежить $f(x)$ від $f(x–1)$.

Для подальшого розгляду нам знадобиться низка визначень. Нехай є дві множини X і Y.

Якщо певним елементам множини X поставлені у відповідність однозначно визначені елементи множини Y, то стверджують, що задана часткова функція з X в Y.

Сукупність тих елементів множини X, у яких є відповідні елементи в Y, називається <u>областю визначення функції</u>, а сукупність тих елементів Y, які відповідають елементам X, називаються <u>сукупністю значень функції</u>.

Якщо область визначення функції з X в Y співпадає з множиною X, то функція називається визначеною всюди.

Відправна ідея побудови точного визначення алгоритму, що ґрунтується на понятті рекурсивної функції, полягає в тому, що будь-які дані (безумовно дискретні) можна закодувати натуральними числами у певній системі числення, і тоді будь-яке їх перетворення, зведеться до послідовності обчислювальних операцій, а результат обробки також буде представляти собою ціле число. При даному підході будь-який алгоритм, єдиний для даної числової функції, обчислює її значення, а його елементарними кроками виявляються звичайні арифметичні і логічні операції. Такі функції одержали назву *обчислюваних*.

Нехай є клас функцій типу $y(x_1, x_2, \ldots, x_n)$, особливостями яких є те, що всі аргументи функції $x_1,\ldots, x_n$ цілочисельні, і значення функції y також виражається цілим числом. Іншими словами, розглядаються функції, аргументи і значення яких – дискретні.

Функція $y(x_1, x_2, \ldots, x_n)$ називається <u>ефективно обчислюваною</u>, якщо існує алгоритм, який дозволяє обчислити її значення за відомим значенням аргументу.

Оскільки поняття алгоритму у цьому визначенні береться в інтуїтивному змісті, то і поняття ефективно обчисленої функції виявляється інтуїтивним. Проте, при переході від алгоритмів до обчислюваних функцій, виникає одна досить суттєва обставина. Сукупність процесів, які задовольняють умовам дискретності, зрозумілості, детермінованості, масовості, скінченності і результативності і, отже таких, що підпадають під інтуїтивне поняття алгоритму, досить значна і недоступна для перегляду. Навпаки, сукупність обчислених функцій для самих найрізноманітніших процесів, які задовольняють перерахованим умовам, виявилася одними й тими ж і притому такими, які доволі легко описуються простими математичними термінами. Ця точно описувана сукупність числових функцій, яка співпадає з сукупністю усіх обчислених функцій носить назву сукупності *рекурсивних функцій*.

Будь-які алгоритмічні моделі і в тому числі рекурсивні функції, повинні передбачати визначення елементарних кроків алгоритму і способів побудови з них певної послідовності перетворень, які забезпечують необхідну послідовність обробки даних. У рекурсивній моделі такими „елементарними кроками" є так звані *найпростіші* числові функції S^1, 0^n і I_m^n комбінацією яких будуються більш складні функції і які *визначаються* наступним чином:

- $S^1(x) = x+1$ – *це одномісна* (яка має один аргумент) *функція безпосереднього проходження;*
- $0^n(x_1,x_2,\ldots,x_n) = 0$ – це n-місна функція, яка *тотожно дорівнює нулю;*
- $I_m^n(x_1,\ldots,x_n)=x_m(1\le m \le n; n=1,2,\ldots)$ – n-місна функція *тотожного повторення значень одного з своїх аргументів.*

Перераховані найпростіші функції усюди визначені і інтуїтивно обчислені. Над ними визначаються операції, які володіють тією властивістю, що їх застосування до функцій, обчислених в інтуїтивному змісті, породжує нові функції, також свідомо обчислені в інтуїтивному змісті. Часткові функції, які можна одержати за допомогою цих операторів з найпростіших функцій називаються *<u>частково рекурсивними</u>*.

У 1936 р. математиком А. Черчем була сформульована гіпотеза про те, що <u>клас рекурсивних функцій тотожний класу усюди визначених обчислюваних функцій</u>. Поняття обчислюваної функції точно не визначається, тому тезу Чёрча довести неможливо.

Рис.5.9. Чёрч Алонзо

Черч Алонзо **–** американський логік, математик. Професор Принстонського університету (1947–1967 рр.). З 1967 р. – професор математики і філософії Каліфорнійського університету (Лос-Анджелес). Наукові праці відносяться до різних галузей логіки.

Черч розвинув думку про відділення поняття функції від поняття множини. У 1936 р. висунув основну гіпотезу теорії обчислюваних функцій (теза Черча): кожна ефективно обчислювана функція є загальнорекурсивною. У 1935 р. навів приклад невирішуваної масової проблеми, а у 1966 р. довів, що вирішувана проблема для обчислення предикатів є невирішуваною. Ці результати істотно вплинули на розвиток математичної логіки. Зробив істотний вклад у розвиток комбінаторної логіки; йому належать дослідження в галузі логічної семантики і модальної логіки.

Гіпотеза А. Черча полягає в тому, що клас побудованих таким чином частково рекурсивних функцій співпадає з класом функцій, які припускають алгоритмічне обчислення.

Якщо певним елементам масиву *X* поставити у відповідність однозначно визначені елементи множини *Y* то кажуть, що задана часткова функція з *X* в *Y*.

Сукупність тих елементів множини *X* у яких є відповідні елементи в *Y*, називається областю визначеності функції, а сукупність цих елементів множини *Y* – сукупністю значень функції.

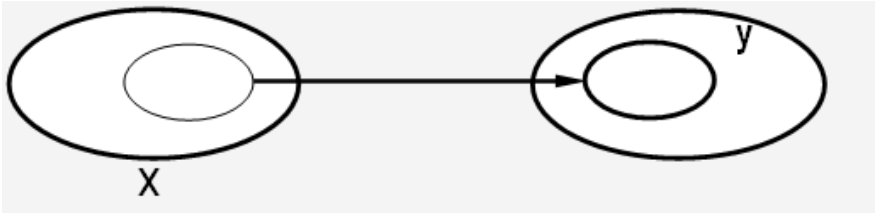


Рис.5.10. Область визначення елементів множини *Y*

Якщо область визначеності функції з X в *Y* співпадає з множиною *X*, то функція називається всюди визначеною.

Рис.5.11. Приклад усюди визначеної функції

Кліні ввів поняття частково рекурсивної функції і висунув гіпотезу, що всі часткові функції, які обчислюються за допомогою алгоритмів, є частково рекурсивними.

Цю гіпотезу також неможливо довести, як і гіпотезу Черча, однак дослідження останніх років виявили повну доцільність вважати поняття частково рекурсивної функції науковим еквівалентом інтуїтивного поняття обчислюваної часткової функції.

Теза Черча виявилась достатньою, щоб придати необхідну точність формулюванню алгоритмічних проблем і у низці випадків зробити можливим доказ їх нерозв'язності.

Питання про можливість обчислення функції рівносильний питанню про її рекурсивність. Поняття рекурсивної функції строге, тому звичайна математична техніка дозволяє іноді безпосередньо довести, що функція, яка дозволяє вирішити задачу, не може бути рекурсивною, то задача є невирішуваною.

Застосування рекурсивних функцій в теорії алгоритмів ґрунтується на ідеї нумерації слів у довільному алфавіті послідовними натуральними числами. Найпростіше таку нумерацію можна здійснити, розташовуючи слова у порядку зростання їх довжин, а слова, які мають однакову довжину – у довільному (лексикографічному) порядку.

Перейдемо до розгляду операторів, які забезпечують перетворення найпростіших функцій.

5.2. Суперпозиція часткових функцій

Нехай *m*-місні функції

$$f_1(x_1,\ldots, x_m),\ f_2(x_1,\ldots, x_m),\ldots, f_n(x_1,\ldots, x_m)$$

підставляються в *n*-місну функцію $g(x_1,\ldots, x_n)$. В результаті формується *n*-місна функція

$$h(x_1,\ldots, x_n)=g(f_1(x_1,\ldots, x_m),\ldots, f_n(x_1,\ldots, x_m))$$

Кажуть, що функція *h* отримана з функцій $g, f_1, \ldots, f_n$ *суперпозицією* (або підстановкою). Символічно така підстановка позначається наступним чином:

$$S^{n+1}(g,f_1,\ldots,f_n),$$

де індекс зверху позначає кількість функцій, які підставляються як аргументи.

Якщо ми вміємо обчислювати функції $g, f_1, \ldots, f_n$, то функція *h* також може бути обчислена. Зрозуміло також, що якщо всі функції $g, f_1, \ldots, f_n$ усюди визначені, то і функція *h* також усюди визначена. Таким чином, якщо функції $g, f_1, \ldots, f_n$ інтуїтивно обчислена, то буде інтуїтивно обчисленою і функція *h*.

Приклад 5.1. Знайти значення $S^2(S^1,0^1)$.

Для цього значення найпростішої функції 0^1 повинно бути підставлено в $S^1(x)=x+1$. Але $0^1(x)=0$, отже, $h(x) = S^2(S^1, 0^1) = S^1(0^1) =0+1=1$

Приклад 5.2.

Найти значення $S^3(I_2^2,I_1^1,0^1)$. У цьому випадку кінцева функція буде двомісною ($n = 3–1 =2$), отже $h(x_1,x_2) = I_2^2(I_1^1,0^1) = I_2^2(x_1,0)=0$

5.3. Примітивна рекурсія

Нехай задані певні числові часткові функції: n-місна $g(x_1,\ldots,x_n)$ і $(n+2)$ – місна $h(x_1,\ldots,x_n,k,y)$. Говорять, що $(n+1)$-місна часткова функція f утворюється з функцій g і h за допомогою *примітивної рекурсії*, якщо для всіх натуральних значень $x_1,\ldots,x_n, y$ справедливо:

$$f(x_1,\ldots, x_n ,0) = g(x_1,\ldots, x_n),$$
$$f(x_1,\ldots, x_n,y+1) = h(x_1,\ldots, x_n , y, f(x_1,\ldots, x_n ,0)) \quad (5.1)$$

Оскільки областю визначення функцій є множина усіх натуральних чисел, часткова функція f, яка задовольняє умовам (5.1), існує для будь-яких часткових функцій g і h і функція ця буде єдиною. Умови (5.1) задають також послідовність визначення значень f на різних кроках рекурсії:

$$\begin{gathered} f(x_1,\ldots, x_n,0) = g(x_1,\ldots, x_n), \\ f(x_1,\ldots, x_n,1) = h(x_1,\ldots, x_n,1, f(x_1,\ldots, x_n,0)), \\ \ldots \\ f(x_1,\ldots, x_n,m+1) = h(x_1,\ldots, x_n, m+1, f(x_1,\ldots, x_n, m)) \end{gathered} \quad (5.2)$$

Символічно примітивна рекурсія позначається $f=R(g,h)$; у цьому запису R розглядається як символ двомісної часткової операції, визначеної на множині усіх часткових функцій. З співвідношень *(5.2)* витікає, зокрема, що якщо g і h є усюди визначеними, то і f також є усюди визначеною. З *(5.2)* видна також та важлива обставина, що якщо ми вміємо знаходити значення функцій g і h, то значення функції $f(a_1,\ldots,a_n,m+1)$ можна обчислювати „механічно”, знаходячи послідовно значення на попередніх кроках.

Часткова функція $f(x_1,\ldots, x_n)$ називається примітивно рекурсивною, якщо її можна отримати кінцевим числом операцій суперпозиції і примітивної рекурсії, виходячи лише з найпростіших функцій S^1, 0^n і I_m^n .

Якщо операції суперпозиції і примітивної рекурсії застосувати до всюди визначених функцій, в результаті буде отримана також усюди визначена функція. Зокрема, всі *примітивно рекурсивні функції усюди визначені.*

Приклад 5.3. Довести, що двомісна функція $f(x,y)=x+y$ є примітивно-рекурсивною. Дана функція може бути представлена у вигляді:

$$x + 0 = x = I_1^1(x)$$
$$x + (y+1) = (x+y) +1 = S^1(x+y)$$

Отже, функція $f(x,y)$ утвориться з примітивно рекурсивних функцій операцією примітивної рекурсії і, отже, вона сама є примітивно-рекурсивна.

Приклад 5.4.

Найти значення функції $f(3,2)$, якщо вона задана наступними співвідношеннями:

$$f(0,x) = 0$$
$$f(y+1,x) = f(y,x) + x$$

В даному випадку $g(x) = 0$, $h(x,y,z) = y + z$. Тому що $f(0,x) = g(x) = 0$ при будь-якому x, то і $f(0,2) = 0$, а інші значення можна обчислити послідовно:

$$f(1,2) = h(1,0,2)= 0+ 2 = 2$$
$$f(2,2) = h(2,2,2)= 2+ 2 = 4$$
$$f(3,2) = h(3,4,2)= 4+ 2 = 6$$

Нескладно довести, що в даному прикладі $f(x,y)=x\cdot y$

5.4. Операція мінімізації

Нехай задана певна функція $f(x,y)$. Зафіксуємо значення x і визначимо при якому y, значення $f(x,y)=0$. Більш складною є задача пошуку *найменшого* з тих значень y, при яких $f(x,y)=0$. Оскільки результат розв’язання такої задачі, очевидно, залежить від x, то і найменше y є функцією x. Приймемо позначення:

$$\varphi(x) = \mu_y\left[f(x,y) = 0\right]$$

(читається: *„найменше y таке, що f(x,y)=0”*, а μ_y називає μ*-оператором* або оператором *мінімізації*).

Аналогічним чином визначається функція багатьох змінних:

$$\phi(X_1,...X_n) = \mu_y\left[f(x_1,...x_n y) = 0\right]$$

Для обчислення функції ϕ можна запропонувати наступну процедуру:

1. Обчислюємо $f(x_1,...x_n,0)$; якщо значення дорівнює нулю, то вважаємо $\phi(x_1,...x_n)=0$. Якщо $f(x_1, ...x_n,0) \neq 0$, то переходимо до наступного кроку.

2. Обчислюємо $f(x_1,...x_n,1)$; якщо значення дорівнює нулю, то вважаємо $\phi(x_1,...x_n)=1$. Якщо $f(x_1,... x_n,0) \neq 0$, то переходимо до наступного кроку тощо.

Якщо виявиться, що для усіх y функція $f(x_1,...x_n,0) \neq 0$, то функція $\phi(x_1,...x_n)$ вважається невизначеною.

Приклад 5.5.

Розглянемо функцію $f(x,y)=x-y$, яка може бути отримана за допомогою оператора мінімізації

$$f(x,y) = \mu_z (y+z=x) = \mu_z [I_3^2(x,y,z) + I_3^3(x,y,z) = I_3^1(x,y,z)]$$

Обчислимо, наприклад, $f(7,2)$, тобто значення функції при $y = 2$ і $x = 7$. Покладемо $y = 2$, а x будемо надавати послідовні значення:

$z = 0$,	$2 + 0 = 2 \neq 7$,
$z = 1$,	$2 + 1 = 3 \neq 7$,
$z = 2$,	$2 + 2 = 4 \neq 7$,
$z = 3$,	$2 + 3 = 5 \neq 7$,
$z = 4$,	$2 + 4 = 6 \neq 7$,
$z = 5$,	$2 + 5 = 7 = 7$.

Таким чином, знайдено значення функції $f(7,2) = 5$

Часткова функція $f(x_1,..., x_n)$ називається частково рекурсивною якщо її можна отримати кінцевим числом операцій суперпозиції, примітивною рекурсією і мінімізації, виходячи лише з найпростіших функцій S^1, 0^n і I_m^n.

Клас частково рекурсивних функцій ширше класу примітивно рекурсивних функцій, оскільки усі примітивно рекурсивні функції є усюди визначеними, а серед частково рекурсивних функцій зустрічаються функції, які не усюди визначені, а також такі, які ніде не визначені.

Поняття частково рекурсивної функції є одним з головних понять теорії алгоритмів. Значення його полягає у наступному. З одного боку, кожна стандартно задана частково рекурсивна функція обчислюється шляхом певної процедури механічного характеру, яка відповідає інтуїтивному уявленню про алгоритми. З іншого боку, які б класи точно обкреслених алгоритмів не будувалися, у всіх випадках незмінно виявлялося, що обчислені за допомогою них числові функції були частково рекурсивними. Тому загальноприйнятою є наукова гіпотеза, сформульована *Черчем*:

Клас алгоритмічно (або машино–) обчислених часткових числових функцій збігається з класом усіх частково рекурсивних функцій.

Ця теза дає алгоритмічне тлумачення поняття частково рекурсивної функції. Його не можна довести, оскільки вона поєднує нестроге математичне поняття інтуїтивно обчисленої функції зі строгим математичним поняттям частково рекурсивної функції. Однак дослідження, які проводилися багатьма математиками протягом декількох десятиліть, виявили повну доцільність вважати *поняття частково рекурсивної функції науковим еквівалентом інтуїтивного поняття обчисленої часткової функції.*

Теза Черча виявилась достатньою для того, щоб надати необхідну точність формулюванням алгоритмічних проблем і в низці випадків зробити можливим доказ їх нерозв'язності. Причина полягає у тому, що зазвичай, в алгоритмічних проблемах математики мова йде не про алгоритми, а про обчислюваність деяких, спеціальним чином побудованих функцій. В силу тези Черча питання обчислювання функції еквівалентно питанню її рекурсивності.

Поняття рекурсивної функції строге. Тому звичайна математична техніка дозволяє іноді безпосередньо довести, що функція, яка розв'язує задачу, не може бути рекурсивною. Саме цим шляхом самому Черчу вдалося довести нерозв'язність основної алгоритмічної проблеми логіки предикатів – проблеми тотожної істинності формул вирахування першого степеня.

5.5. Застосування рекурсивних функцій в теорії алгоритмів

Застосування рекурсивних функцій в теорії алгоритмів засновано на ідеї нумерації слів у довільному алфавіті послідовними натуральними числами. Наприклад, таку нумерацію можна здійснити, розміщуючи слова у порядку зростання їх довжини, а слова, що мають однакову довжину, – у довільному лексикографічному порядку.

Після нумерації вхідних та вихідних слів будь-який алфавітний оператор перетворюється на функцію $y=f(x)$, де x та y приймають цілі невід'ємні значення. Причому функція $f(x)$ може бути визначена тільки для тих x, що належать області визначення цієї функції.

Подібні частково визначені цілочисельні функції називають арифметичними. При побудові алгоритмів використовується певний набір елементарних арифметичних функцій, таких як:

1. функції є тотожно однаковими (рівними) 0: $f(x)=0$; для всіх x;
2. функції тотожно дорівнюються аргументу :$f(x)=x$;
3. функції безпосереднього слідування: $f(x) = x+1$;

Із цих елементарних арифметичних функцій за допомогою невеликої кількості операцій будують більш складні функції, які не виходять за межі класу рекурсивних функцій. До них належать операції суперпозиції, примітивної рекурсії, найменшого кореня. Операція суперпозиції полягає у підстановці одних арифметичних функцій замість аргументів інших функцій.

Наприклад, нехай є:

$$f(x)=0;$$
$$g(x)=x+1;$$
$$h(x)=g(f(x)).$$

Тоді:

$$h(0)=g(0)= 0+1=1;$$
$$h(x)=g(0)= 0+1=1.$$

Тобто отримали функцію, яка тотожно дорівнює 1.

Якщо $h(x) =g(g(x))$, *то* $h(x) =g(x+1) = x + 1 + 1 = 2$ тощо.

Операція примітивної рекурсії дозволяє побудувати функцію, яка залежить від $n + 1$ аргументу, якщо відомі дві функції – одна від n аргументів, друга – від $n+2$ аргументів.

Наприклад, побудуємо двомісну функцію додавання $f(x, y) = x + y$ якщо є одномісна $g(x) = x$; та тримісна $h(x, y, z) = z +1$. (Кількість аргументів можна збільшувати як завгодно.)

Завдячуючи операції примітивної рекурсії здійснимо побудову функції. Нехай

$$f(x,0)=g(x);$$
$$f(x, y + 1) = h(x, y,f(x, y)); f(x, 0) = g(x) = x;$$
$$f(x, 1) = h(x, 0,f(x, 0)) = h(x, 0, x) = x + 1;$$
$$f(x,2) = h(x, 1,f(x,1)) = h(x, 1, x + 1) = x + 2;$$
$$\ldots\ldots\ldots\ldots\ldots$$
$$f(x, y - 1) = h(x, y - 2,f(x, y- 2)) = h(x, y - 2, x + y -2) =x + y - 1;$$
$$f(x, y) = h(x, y-1, f(x, y - 1)) = h(x, y – 1, x + y- 1) = x + y.$$

Таким чином, побудовано функцію, яка залежить від двох аргументів і значення якої обчислюються для будь-якого значення x, коли y змінюється від 0 до будь-якого цілого.

Функції, які можна побудувати з елементарних арифметичних функцій за допомогою операцій суперпозиції та примітивної рекурсії, називають примітивно рекурсивними.

Операція найменшого кореня, або операція мінімізації, дозволяє побудувати функцію f від n аргументів, якщо відома функція g, залежна від $n + 1$ аргументу, тобто зменшити кількість аргументів функції.

Нехай маємо функцію від $n + 1$ аргументу $g\ (x_1, x_2,\ldots x_n, y)$. Побудуємо функцію від n аргументів $f(x_1, x_2,\ldots x_n)$. Задамо значення для n аргументів функції g:

$$x_1= a_1;\quad x_2= a_2 \ldots x_n= a_n;$$

Підставимо ці значення у формулу, яка визначає функцію g, отримаємо:

$$g(a_1, a_2\ldots a_n, y).$$

Прирівняємо цей вираз до нуля і розв'яжемо рівняння відносно y:

$$g(a_1, a_2\ldots a_n, y)=0.$$

Нехай отримали $y =a$.

Вважатимемо це значення y за значення функції f від тих самих значень $x_i \ldots x_n$:

$$f(a_1 \ldots a_n) = a.$$

Діючи таким чином, ми можемо визначити функцію f від будь-яких значень її аргументів. Отже, функцію f побудовано.

Наприклад:

$g(x, y, z) = x+y+z$		$f(x,y)=?$
$g(0, 0, z)=0+0+z=0$	$z=0$	$f(0,0)=0$
$g(1, 0, z)=1+0+z=0$	$z=-1$	$f(1, 0)=-1$
$g(0, 1, z)=0+1+z=0$	$z=-1$	$f(0,1)=-1$
……		
$g(5,2, z)=5+2+z=0$	$z=-7$	$f(5,2)=-7$
$g(-6,10, z)=-6+1+z=0$	$z=5$	$f(-6,1)=5$

Питання для самоконтролю

1. *Що розуміють під рекурсією?*
2. *Що розуміють під частковою функцією?*
3. *Що розуміють під сукупністю значень функції?*
4. *Що розуміють під всюди визначеною функцією?*
5. *Що розуміють під ефективно обчислюваною функцією?*
6. *Що розуміють під примітивно рекурсивною функцією?*
7. *Що розуміють під частково рекурсивною функцією?*
8. *В чому полягає сутність тези Черча?*

6. АБСТРАКТНА МАШИНА Е. ПОСТА

6.1. Необхідність створення абстрактної обчислювальної машини

Будь-якому алгоритму відповідає задача, для розв'язання якої він був побудований. Зворотне твердження, у загальному випадку, є невірним за двома причинами:

✓ *по-перше*, одна й та сама задача може розв'язуватись різними алгоритмами;

✓ *по-друге*, можна припустити (сьогодні), що існують задачі, для яких алгоритм взагалі не може бути побудований.

Як уже відзначалося, термін *„алгоритм"* з'явився у математиці досить давно і під ним розумілася процедура, що дозволяла шляхом виконання послідовності визначених елементарних кроків отримувати однозначний результат, який не залежав від того хто ці кроки виконував. Тобто, виконане розв'язання було доказом існування алгоритму.

Однак, існував цілий ряд математичних задач, розв'язати які в загальному вигляді не вдавалося. Прикладами можуть слугувати три стародавні геометричні задачі: про трисекції кута, про квадратуру кола і про подвоєння куба – жодна з них не має загального способу розв'язання.

Зацікавленість математиків до задач подібного роду призвела до постановки питання: а чи можливо не розв'язуючи задачу, довести, що вона є алгоритмічно нерозв'язною, тобто чи не можна побудувати алгоритм, який би забезпечив її загальне розв'язання? Відповідь на це питання є дуже важливою з практичної точки зору, бо безглуздо намагатися розв'язувати задачу на комп'ютері і розробляти для неї програму, якщо можна довести, що вона алгоритмічно є нерозв'язною.

Для цього потрібно спочатку дати строге визначення алгоритму, бо без нього обговорення існування алгоритму просто не має сенсу. Намагання побудови строгого визначення алгоритму, як ми раніше вияснили, призвело до появи формальних алгоритмічних систем, зокрема побудови рекурсивної функції. Тобто, задача вважається алгоритмічно нерозв'язною, якщо не існує рекурсивної функції, яка її вирішує.

Точний опис класу частково рекурсивних функцій разом з тезою Черча[1] дає одне з можливих розв'язків задачі з уточнення поняття алгоритму. Однак це розв'язання не цілком пряме, оскільки поняття обчислюваної функції є вторинним відносно поняття алгоритму. Виникає питання, а чи не можна уточнити безпосередньо саме поняття алгоритму і вже потім за його допомогою точно визначити клас обчислюваних функцій? Такий напрямок пошуку призвів до побудови іншого, ніж рекурсивні функції, класу моделей алгоритму, а саме до абстрактної обчислюваної машини.

Для формалізації інтуїтивного поняття алгоритму А. Тюрінгом та Е. Постом були введені математичні моделі абстрактного виконавця (машини Тюрінга та Поста), А. Черч розробив так зване лямбда-числення, А. Марков запропонував так звані нормальні алгоритми Маркова. Згодом було доведено, що всі ці формальні визначення алгоритму є еквівалентними.

Важливість доказу алгоритмічної нерозв'язності полягає в тому, що якщо такий доказ отриманий, то він дозволяє не витрачати зусилля на пошук розв'язання, подібно тому, як закони збереження у фізиці роблять безглуздими спроби побудови вічного двигуна. Разом з цим необхідно усвідомлювати, що алгоритмічна нерозв'язність будь-якої задачі у загальній постановці не виключає можливості того, що задача може бути розв'язана при певних обмеженнях або для певних умов. Справедливе й зворотне твердження: часткове розв'язання задачі ще не дає приводу вважати за можливе її розв'язання в загальному випадку, тобто це не свідчить про її загальну алгоритмічну можливість розв'язання.

Роль абстрактних алгоритмічних систем полягає в тому, що саме вони дозволяють оцінити можливість знаходження повного (загального) рішення певного класу задач. Для фахівця у галузі інформатики важливо усвідомлювати, що наявність алгоритмічно нерозв'язних проблем призводить до того, що виявляється неможливим побудувати універсальний алгоритм, придатний для розв'язання будь-якої задачі (і, отже безглуздо витрачати сили на його створення). До подібних проблем приводять і спроби алгоритмізувати складну інтелектуальну діяльність людини, наприклад, навчання інших людей, написання віршів, музики тощо.

[1] **Теза Черча –** твердження, згідно з яким, клас алгоритмічно-обчислюваних функцій збігається з класом частково-рекурсивних функцій, функцій обчислюваних за Тюрінгом та інших формальних уточнень інтуїтивного поняття алгоритм. З неї випливає, що якщо функція належить до класу певної формалізації алгоритмічно-обчислюваної функції, то вона є алгоритмічно-обчислювана. Теза не доводиться. А еквівалентність класів формалізмів підлягає доведенню, що і було зроблено. Названа на честь американського математика Алонзо Черча.

6.2. Визначення класу обчислювальних функцій за допомогою абстрактних машин

Визначення класу обчислювальних функцій за допомогою абстрактної машини полягає у тому, що алгоритмічні процеси може здійснювати певним чином улаштована обчислювальна машина, яка моделює виконання окремих операцій людиною. Функціонування такої абстрактної машини слугує доказом виконання алгоритму. Виходячи з властивостей алгоритму, можна сформулювати загальні вимоги до таких обчислювальних машин:

❶ характер їх функціонування повинний бути *дискретним*, тобто складатися з окремих кроків, кожний з яких виконується тільки після завершення попереднього;

❷ дії повинні бути *детерміновані*, тобто кроки виконуються в строгому порядку, а їх результат визначається самим кроком і результатами попередніх кроків;

❸ перед початком роботи, машині надаються вихідні дані з області визначення алгоритму;

❹ за скінчену кількість кроків роботи машини, повинний бути отриманий результат (або інформація про те, що вважати результатом);

❺ машина повинна бути *універсальною*, тобто такою, щоб за її допомогою можна було б виконати будь-який алгоритм.

Чим простіше структура (тобто устрій) описаної машини і чим елементарніші її кроки, тим більше причин вважати, що її робота і є виконанням алгоритму. Щоб відповісти на питання, які кроки роботи машини потрібно віднести до елементарних, повернемося до тієї обставини, що нас цікавить перетворення інформації, представленої за допомогою певного *кінцевого алфавіту*. Вимога скінченності алфавіту є очевидним наслідком тієї обставини, що розв'язання повинно бути отримане за скінчену кількість кроків. Якщо інформація не представлена у дискретній формі, наприклад, дійсне число, то його обробка у загальному випадку може містити нескінченну кількість кроків (наприклад, знаходження цифр числа π, або добування квадратного кореня з числа 2).

Таким чином, алгоритм виявляється *кінцевою* послідовністю дій, що здійснюються над даними, представленими за допомогою скінченого алфавіту. З урахуванням наведеного стає зрозумілим визначення алгоритму, яке дає В.М. Глушков [41, с.38].

Алгоритм – це будь-яка скінчена система правил перетворення інформації (даних) над будь-яким скінченим алфавітом.

Віктор Михайлович Глушков (24 серпня 1923, Ростов-на-Дону – 30 січня 1982, Москва) – піонер комп'ютерної техніки, автор фундаментальних праць у галузі кібернетики, математики і обчислювальної техніки, ініціатор і організатор реалізації науково-дослідних програм створення проблемно-орієнтованих програмно-технічних комплексів для інформатизації, комп'ютеризації і автоматизації господарської і оборонної діяльності країни. Глава наукової школи кібернетики. Лауреат численних премій і нагород. Дійсний член АН СРСР, АН УРСР. Почесний член багатьох іноземних академій. Засновник Інституту кібернетики імені В. М. Глушкова НАН України, який і очолював до 1982 р. Помер в Москві куди був перевезений для лікування. Похований у Києві на Байковому кладовищі.

Рис.6.1. Віктор Михайлович Глушков

Нехай вихідні дані з області визначення алгоритму представлені за допомогою алфавіту *A* і утворюють при цьому скінчену послідовність знаків $\{a_1 \ldots a_n\}$, яка називається *словом*. В результаті виконання алгоритму формується нове слово $\{b_1 \ldots b_m\}$, представлене у загальному випадку в іншому алфавіті *B*.

Для проведення такого перетворення, елементарними є наступні операції (кроки):

1. *заміна одного знака вихідного слова a_i знаком b_j з алфавіту B;*
2. *видалення знака вихідного слова;*
3. *додавання до вихідного слова знака з алфавіту B.*

Однак, якщо в алфавіти включений знак, що має значення *пустого знака*, додавання якого до слова зліва або справа не змінює цього слова (аналог числового нуля), то легко бачити, що операція *(2)* є ні що інше, як заміна a_i пустим знаком, а операція *(3)* є заміна пустого знака знаком b_j.

Таким чином, всі можливі алфавітні перетворення зводиться до операції *(1)* – заміни одного знака іншим.

Саме за цією причиною, функціонування абстрактної машини зводиться до того, що вона обстежує символи алфавіту (зчитує і розпізнає їх) які записані в пам'яті (на нескінченній стрічці), і залежно від свого стану і того, який саме обстежується символ, вона замінює його іншим символом; після цього вона переходить у новий стан, зчитує наступний символ тощо і так до команди про припинення роботи.

Оскільки подібні машини є чисто модельною, теоретичною побудовою, вони отримали назву *абстрактних машин* і розглядаються як одна з можливих універсальних алгоритмічних систем.

Концепція визначення існування алгоритму як абстрактної машини була запропонована практично одночасно (1936–1937 рр.) англійським математиком Аланом Тюрінгом і його американським колегою Емілем Постом.

Висока значущість їх побудов, полягає ще й в тому, що вони в абстрактній формі передбачили основні принципові риси реальних пристроїв з обробки даних (обчислювальних машин), які з'явились лише через 8 – 9 років.

6.3. Сутність абстрактної машини Е. Поста

Е. Пост спеціалізувався в галузі математичної логіки, був одним з перших, хто усвідомив важливість такого поняття, як алгоритмічна повнота системи операцій, використовуваних для перетворення інформації. Читав лекції з математики і логіки у Колумбійському, Нью-Йоркському та інших університетах США. Ним отримана низка фундаментальних результатів у математичній логіці*;* одне з найбільш уживаних визначень понять несуперечності і повноти формальних систем (числень)*;* докази функціональної дедуктивної повноти (в широкому і вузькому значенні) числення висловлень; вивчення систем багатозначної логіки з більше чим трьома значеннями істинності; одне з перших (незалежно від А. М. Тюрінга) визначення поняття алгоритму в термінах „абстрактної обчислювальної машини" і формулювання основної тези теорії алгоритмів про можливість описати будь-який конкретний алгоритм за допомогою цього визначення; результати можливості виразити загально рекурсивні функції і предикати через примітивно рекурсивні, зокрема так звана теорема про нормальну форму; перші (одночасно з О. О. Марковим) докази алгоритмічної нерозв'язності ряду проблем математичної логіки і алгебри, тощо.

Рис. 6.2. Пост (Post) Еміль Леон (11.2.1897, Августів, Польща, – 21.4.1954 р., Нью-Йорк), американський математик і логік польського походження.

Свої ідеї Е. Пост опублікував у 1935 р., задовго до виникнення автоматичних перетворювачів інформації. Він розробив найпростішу з принципово можливих систем обробки інформації і показав, що запропонована ним система володіє досить важливою властивістю – алгоритмічною повнотою.

В чому ж сутність машини Поста?

По-перше, Е. Пост запропонував будь-яку інформацію, яка підлягає обробці за суттю, попередньо перетворювати за формою. Він вимагав кожне слово, яке підлягає обробці, попередньо переписати зі звичного алфавіту на алфавіт двійковий, тобто на двобуквений алфавіт. Подібну формальну обробку тексту можна побачити на телеграфі. Телеграфіст, не змінюючи змісту тексту, заміняє його форму. При цьому телеграфіст використовує двобуквений алфавіт: його букви – точка і тире.

По-друге, важливою частиною його системи є пропозиція обробляти кожне дане слово (вже записане у двійковому алфавіті): букву за буквою. Це приводить до дуже простої системи елементарних дій, використовуючи які можна здійснити будь-яке перетворення двійкового слова, якщо це взагалі можливо.

По-третє, після того як певна інформація переписана у двійковий алфавіт, вона заноситься буква за буквою на інформаційну стрічку машини. Інформаційна стрічка розбита на однакові секції (комірки). Кількість секцій на стрічці нескінченна. Це означає, що при необхідності можна „підклеювати" до інформаційної стрічки праворуч або ліворуч потрібну кількість секцій.

По-четверте, розподіл відмічених і невідмічених секцій обумовлює стан стрічки. Вздовж інформаційної стрічки рухається каретка. Вона може рухатися тільки стрибками (дискретно), на одну секцію праворуч, або ліворуч.

По-п'яте, конкретний стан стрічки, з указівкою, де розташована каретка, визначає стан машини. За допомогою каретки машина може розпізнати, чи є конкретна секція стрічки, під якою розташована каретка, відміченою або невідміченою. Каретка може стерти мітку, якщо вона є в секції, може помістити мітку в пусту секцію. За допомогою каретки здійснюється побуквене перетворення конкретного двійкового слова.

Задачею машини Поста є перетворення стану інформаційної стрічки. Це перетворення машина здійснює, керуючись алгоритмом, розробленим людиною. Оскільки машина Поста є програмно-керованою машиною, то алгоритм її роботи оформляється у вигляді програми.

Е. Пост довів, що його система володіє алгоритмічною повнотою.

6.4. Робота абстрактної машини Е. Поста

Машина Поста складається з пам'яті і пристрою управління, який, у відповідності з сучасною термінологією, будемо називати процесором (рис. 6.3).

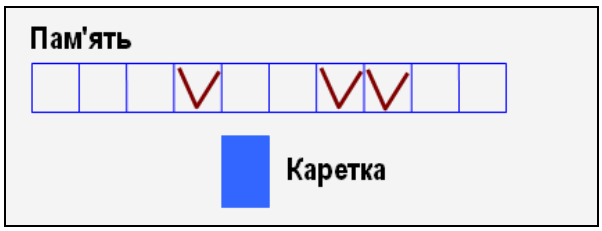


Рис. 6.3. Машина Поста

Взагалі кажучи, це зовсім ніяка не машина, а тим більше, обчислювальна. Це скоріше креслення машини, а якщо бути ще більш точнішими – розповідь про машину. Машина Поста – це *модель* комп'ютера.

Приводом створення машини слугувало бажання встановити алгоритмічну нерозв'язність деяких масових математичних проблем. І саме Емілю Посту належать перші результати у цьому напрямку. Пост висунув ідею подання алгоритму, як абстрактної машині, яка описана в точних математичних термінах і яка одночасно відповідає уяві про обчислювальний процес, який протікає за правилами-командами певного стандартного виду.

Ця абстрактна машина отримала назву на його честь і вирішувала проблему: ***якщо для розв'язку задачі можна побудувати машину Поста (МП), то вона є алгоритмічно розв'язною***.

Команда машини Поста має наступну структуру:

$$n\ Km,$$

де n – порядковий номер команди, K-дія, яка виконується головкою, m – номер наступної команди, яка підлягає виконанню.

В абстрактній машині Поста інформацію представляють у двійковому алфавіті:

$$A = \{0,1\}.$$

Команда – указівка на виконання певного кроку абстрактної машини.

Абстрактна машина Поста представляє собою нескінченну стрічку, розділену на однакові комірки, кожна з яких може бути або пустою, або заповненою міткою «V», комірки, які послідовно пронумеровані цілими числами, як додатними такі від'ємними, і головки, яка може рухатись вздовж стрічки на одну комірку праворуч або ліворуч, заносити в комірку стрічки мітку, якщо цієї мітки там попередньо не було, витирати мітку, якщо вона була або перевіряти наявність у комірці мітки. Інформація про заповнені мітками комірки стрічки характеризує стан стрічки, який може змінюватись в процесі роботи машини.

У кожний момент часу головка («–») перебуває над однією з клітинок стрічки і, як кажуть, оглядає її. Інформація про місце розташування головки разом зі станом стрічки характеризує стан машини Поста, рис. 6.4.

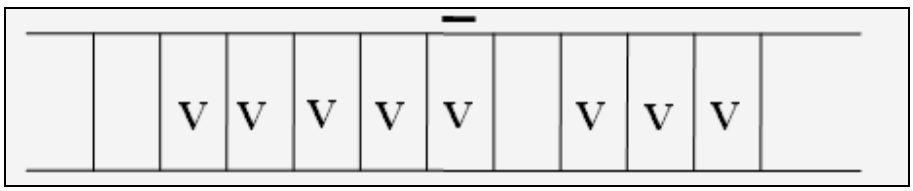


Рис. 6.4. Стан абстрактної машини Поста

У кожній комірці стрічки стоїть або ознака того, що в комірці записана мітка, або комірка порожня.

Стан стрічки – це інформація про те, які комірки зайняті (є запис), а які порожні. Кожна комірка може бути або порожньою, або позначеною міткою (наприклад, **V**). Мітці може відповідати запис значення інформації у двійковому форматі (нуля або одиниці).

Стан машини – це стан стрічки і положення головки зчитування/запису.

Процесор змінює інформацію на стрічці за допомогою спеціальної головки (каретки), виконуючи при цьому команди, *упорядкована* кінцева множина яких, складає програму його роботи.

Команди бувають шести видів (рис.6.5):

1. Команда руху вправо:

i => j, тут ***i*** – номер команди в програмі, ***j*** число відсилання, що стоїть в кінці команди, тобто номер або адреса команди до якої потрібно перейти і при цьому, потрібно перемістити каретку на одну комірку вправо.

2. Команда руху вліво:

i <= j, зрушити каретку на одну секцію ліворуч і перейти до виконання команди № **j**;

3. Команда друку мітки:

i V j, надрукувати мітку **V** в секцію, що переглядається кареткою і потім перейти до команди **j**;

4. Команда витирання мітки:

i ξ j, витерти мітку з секції що переглядається кареткою і перейти до команди **j**;

5. Команда передачі управління:

i ? j_1^0, j_2^1, якщо секція, котра переглядається кареткою порожня, то перейти до команди j_1^0, якщо секція не порожня, то перейти до команди j_2^1.

6. Команда зупинки:

i Стоп, припинити виконання програми.

Рис. 6.5. Команди машини Поста

Ситуації, у яких головка повинна наносити мітку там, де вона вже є, або, навпаки, витирати мітку там, де її немає, є аварійними (неприпустимими).

Програмою для машини Поста будемо називати непустий список команд, такий що

1) на *n*-му місті команда з номером *n*;

2) номер *m* кожної команди співпадає з номером якоїсь команди списку.

З точки зору властивостей алгоритмів, досліджуваних за допомогою машини Поста, найбільший інтерес представляють причини зупинки машини при виконанні програми:

1) зупинка за командою «стоп»; така зупинка називається результативною і вказує на коректність алгоритму (програми);

2) зупинка при виконанні неприпустимої команди; у цьому випадку зупинка називається безрезультативною;

3) машина не зупиняється ніколи; у цьому і в попередньому випадку маємо справу з некоректним алгоритмом (програмою).

В теорії алгоритмів до алгоритмічно нерозв'язного відноситься „проблема зупинки”: чи можна за описом алгоритму (*Q*) і вхідним даним (*x*) встановити, чи завершиться виконання алгоритму результативною зупинкою? Ця проблема має прозаїчну програмістську інтерпретацію. Часто помилки в розробці програми призводять до зациклення – ситуації, коли цикл не завершується і не відбувається завершення роботи програми і її зупинки.

Нерозв'язність проблеми зупинки програми означає, що не можна створити загальний (тобто придатний для будь-якої програми) алгоритм налагодження програм. Нерозв'язною виявляється і проблема розпізнавання еквівалентності алгоритмів: не можна побудувати алгоритм, який для будь-яких двох алгоритмів (програм) з'ясовував би, чи завжди вони приведуть до одного й того ж результату чи ні.

Під початковим станом головки будемо розуміти стан навпроти пустої комірки ліворуч самої лівої мітки на стрічці.

Розглянемо реалізацію деяких типових елементів програм машини Поста.

1. Нехай заданий вихідний стан головки і потрібно на пустій стрічці написати дві мітки: одну в секцію під головкою, другу праворуч від неї. Це можна зробити наступною програмою (праворуч від команди показаний результат її виконання):

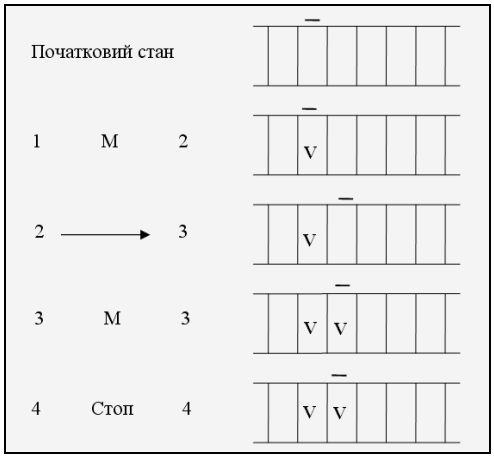


Рис. 6.6. Приклад елемента програми машини Поста

2. Покажемо, як можна скористатися командою умовного переходу для організації циклічного процесу. Нехай на стрічці є запис з декількох міток підряд і головка перебуває над самою крайньою правою міткою. Потрібно перевести головку ліворуч до першої пустої позиції.

Програма буде мати наступний вид:

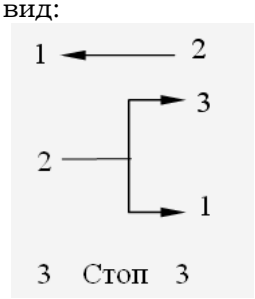


Рис. 6.7. Приклад програми для машини Поста з командою умовного переходу

Команда умовного переходу є одним з основних засобів організації циклічних процесів, наприклад, для знаходження першої мітки справа (або зліва) від головки, розташованої над пустою коміркою; знаходження зліва (або справа) від головки пустої комірки, якщо вона розташована над міткою тощо.

3. Зупинимося на поданні чисел на стрічці машини Поста і виконання операцій над ними.

Число k подається на стрічці машини Поста мітками $k + 1$ (одна мітка означає число «*0*»), що йдуть одна за одною. Між двом числами робиться інтервал як мінімум з однієї пустої секції на стрічці. Наприклад, запис чисел 3 і 5 на стрічці машини Поста буде виглядати наступним чином:

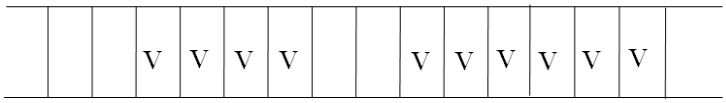


Рис. 6.8. Приклад програми для машини Поста для запису чисел 3 і 5

Звернемо увагу, що використовувана в машині Поста система запису чисел не є позиційною.

Складемо програму для додавання до довільного числа одиниці. Уявимо, що на стрічці записано тільки одне число і головка перебуває над однією з комірок, у якій знаходиться мітка, яка належить цьому числу:

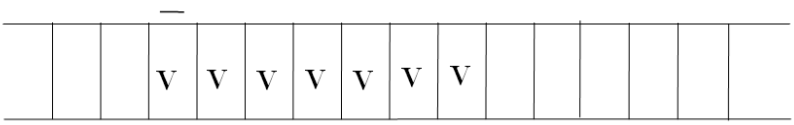


Рис. 6.9. Приклад програми для машини Поста для додавання до довільного числа одиниці

Для розв'язання задачі можна перемістити головку ліворуч (або праворуч) до першої пустої комірки, а потім нанести мітку.

Програма, яка додає до числа мітку ліворуч, має вид:

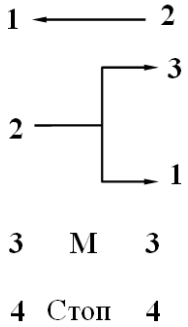


Рис. 6.10. Приклад програми для машини Поста, яка додає до числа мітку ліворуч

Програма, яка додає до числа мітку справа, має вид:

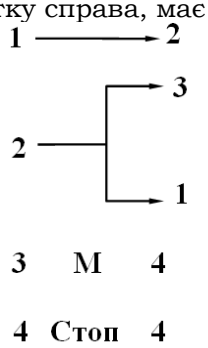


Рис. 6.11. Приклад програми для машини Поста, яка додає до числа мітку праворуч

Відмінність полягає тільки у напрямку руху головки у першій команді.

Уявимо, що зчитувальна головка розташована на відстані декількох комірок зліва від числа, до якого потрібно додати одиницю. У цьому випадку програма ускладнюється. З'явиться „блок пошуку числа" – дві команди, які приводять головку у стан, розглянутий у попередньому прикладі:

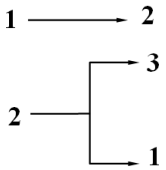


Рис. 6.12. Приклад блоку пошуку числа

Нижче – повні тексти програм, які відповідно додають одиницю ліворуч і праворуч:

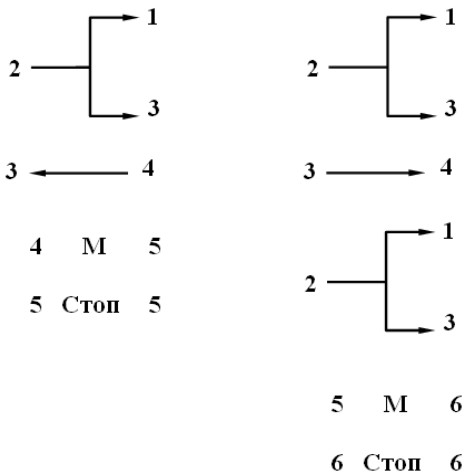


Рис. 6.13. Приклад програм для машини Поста, які відповідно додають одиницю зліва і справа

У першому випадку не потрібно переміщувати головку до крайньої лівої метки числа.

4. Наведемо програму для додавання цілих невід'ємних чисел a і b на машині Поста, коли головка знаходиться над числом a, а число b знаходиться праворуч числа a на певну кількість комірок. Ця програма реалізує наступний алгоритм: перше число поступово наближається до другого до їх злиття, а потім стирається одна мітка (інакше результат виявився б на одиницю більше від правильного).

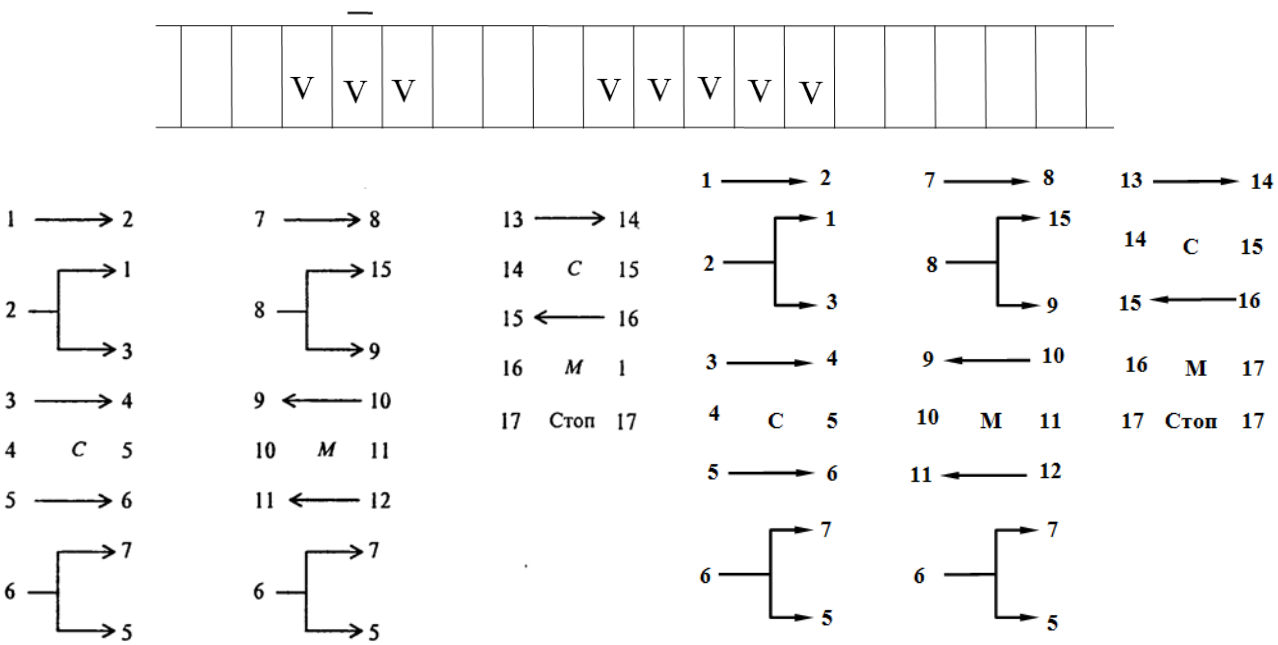


Рис. 6.14. Приклад програм для додавання цілих невід'ємних чисел *a* і *b* на машині Поста

У випадку більш складних початкових умов, коли невідомо праворуч чи ліворуч від зчитувальної головки (і на яку кількість комірок) знаходиться число, можна застосовувати такий принцип пошуку числа: рухаючи головку вправо і вліво і відзначаючи мітками ступінь віддалення головки від вихідного положення, знайти число, а потім уже застосовувати відому програму додавання. При цьому перевіряється, чи знаходиться головка над однією з міток числа і якщо так, то задача розв'язана. В протилежному випадку перевіряється, чи є пустими комірка праворуч від головки і наступна за нею; якщо обидві порожні, то робиться повернення головки на один крок і ставиться мітка, а потім така ж операція виконується зліва і за відзначеною доріжкою головка повертається вправо тощо, до тих пір, поки головка не натрапить на число, після чого можна застосувати попередньо розглянуті програми:

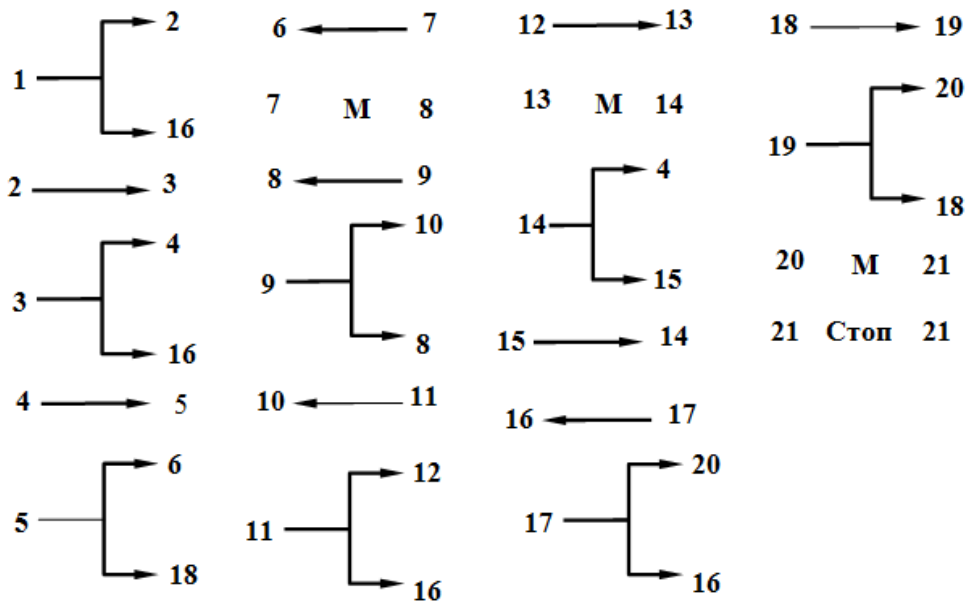


Рис. 6.15. Приклад програм для додавання цілих невід'ємних чисел *a* і *b* на машині Поста, коли невідомо, праворуч або ліворуч від зчитувальної головки (і на яку кількість комірок) зміщено число

Головна ідея абстрактної машини Е.Поста полягає в тому, що:

- алгоритмічні процеси є послідовністю досить простих „механічних” операцій;
- на результат не впливає, хто виконує алгоритм і скільки разів, – він завжди однаковий;
- алгоритмічні процеси може виконувати машина.

Машина Поста, незважаючи на зовнішню простоту, може виконувати різноманітні обчислення, для чого потрібно задати початковий стан машини і задати програму, яка ці обчислення виконає. Як приклад, на рис.6.16 наведена програма додавання одиниці до числа.

Рис.6.16. Програма додавання одиниці до числа на машині Поста

Машину Поста можна розглядати як спрощену модель ЕОМ. Дійсно, як ЕОМ, так і машина Поста мають:

✓ неподільні носії інформації (комірки – біти), які можуть бути заповненими або незаповненими;

✓ обмежений набір елементарних дій – команд, кожна з яких виконується за один такт (крок).

Обидві машини працюють на основі програми. Однак, в машині Поста інформація розташовується лінійно і читається послідовно, а в ЕОМ можна читати інформацію за адресою; набір команд ЕОМ значно ширше і виразніше, чим команди машини Поста тощо.

А чи кожний алгоритм можна представити у формі машини Поста? Відповідь на це питання дає теза Поста: ***будь-який алгоритм можна представити у формі машини Поста***. Це теза оскільки її неможливо довести, бо в ній фігурують з одного боку, інтуїтивне поняття „будь-який алгоритм”, а з іншого боку – точне поняття „машина Поста”.

Алгоритми, побудовані з будь-якого скінченого числа кроків машини Поста, називають алгоритмами Поста. Вони можуть бути зведені до алгоритмів, побудованих у системі рекурсивних функцій (і навпаки).

Питання для самоконтролю

1. В чому полягала необхідність створення машини Е. Поста?
2. Як можна визначити клас алгоритмічно нерозв'язних задач за допомогою абстрактних машин?
3. Що розуміють під терміном „команда"?
4. В чому полягає сутність машини Е. Поста?
5. Як працює абстрактна машина Е. Поста?
6. В чому полягає сутність тези Поста?

7. МАШИНА ТЮРІНГА

Якщо люди не вірять, що математика є простою, то тільки тому, що вони не усвідомлюють, яким складним є життя.
Джон фон Нейман.

7.1. Передумови створення машини А. Тюрінга

У 1900 р. в Парижі відбулась Всесвітня конференція математиків, на якій Давид Гілберт (David Hilbert) виклав у вигляді тез сформульовані ним 23 найважливіші, на його думку, проблеми, які необхідно було вирішити ученим-теоретикам XX ст. Під другим номером у цьому списку проблем значилась одна з тих задач, відповідь на яку здається очевидною, поки не копнеш трошечки глибше. Кажучи сучасною мовою, це було питання: а чи самодостатня математика?

Друга задача Гілберта зводилась до необхідності строго доведення, що система *аксіом* – базових тверджень, що приймаються в математиці за основу без доказів, – досконала і повна, тобто дозволяє математично описати усе суще. Потрібно було довести, що можна задати таку систему аксіом, яка буде, по-перше, взаємно несуперечливою, а по-друге, з них можна зробити висновок відносно істинності або хибності будь-якого твердження. Іншими словами, Д. Гілберт, запропонував звести процес доказу математичних тверджень до абстрактного маніпулювання символами. Але для цього потрібна була мова, на якій можна було б записувати поняття, визначення, аксіоми й теореми математики.

Рис.7.1. Давид Гілберт (нім. *David Hilbert*; 23 січня 1862 – 14 лютого 1943) – німецький математик. У 1910–1920-і рр. Після смерті Анрі Пуанкаре) був визнаним світовим лідером математиків.

Такі мови на той час були вже відомі і використовувалися математиками вже тривалий час (наприклад, мови математичної логіки, арифметики, алгебри, теорії груп, теорії множин, мови програмування тощо).

Будь-яка з цих мов включає в себе символи, які формують її алфавіт і правила утворення граматично вірних виразів. Вирази подаються у вигляді ланцюжків символів кінцевої довжини. Крім того, в їх арсеналі були правила виведення, які описували те, як чином можна перетворювати одні ланцюжки символів в інші. При цьому залишались й невизначувані поняття, в істинності яких не потрібно сумніватися (наприклад, поняття точки в геометрії або елемента в теорії множин), в протилежному випадку зникає відправний пункт для наступних міркувань. Маючи ці компоненти, можна спробувати виразити усі (або майже усі) математичні судження.

Д. Гілберт запропонував звести усі докази математики до маніпуляцій з символами і виразами (правильно побудованими ланцюжками символів), відволікаючись від їх математичної суті або змісту. На думку Д. Гілберта, семантичний зміст повинний був поступитися місцем чисто синтаксичному. Вся математика повинна була описуватися цією мовою (метамовою або мовою про мову). Прикладом метамови може слугувати підручник англійської мови, написаний українською, остання і є метамовою – мовою для пояснення англійської.

Так був покладений початок теорії доказів (інший поширений термін – метаматематика) – математичній дисципліні, яка вивчає саму математику. При цьому, ніхто не забороняє наповнювати такі докази будь-яким змістом, геометричною або фізичною інтерпретацією, як це часто робиться при вивченні диференціального і інтегрального числень, але на думку Гілберта, все це зайве – цілком достатньо синтаксичної правильності послідовності ланцюжків рядків, які представляють собою доказ. Такий підхід отримав назву фінітизма[1], а докази отримані таким чином, – фінітними.

Рис.7.2. Вільгельм Фрідріх Аккерман (нім. Wilhelm Friedrich Ackermann; 29 березня 1896 р., Хершайд, Германія – 24 грудня 1962 р., Люденшайд, Германія) – німецький математик і логік

У 1928 р. Гілберт зі своїм учнем Вільгельмом Аккерманом написали перший сучасний підручник з математичної логіки. Ця невеличка книга, яка виросла з лекцій, прочитаних Гілбертом протягом трьох семестрів студентам математикам і яку за першими літерами прізвищ їх авторів часто позначають як H&A, відіграла визначну роль в становленні багатьох розділів основ математики. В ній вперше був послідовно викладений фінітний підхід до побудови математичної логіки.

[1] При дослідженні формальних систем в метаматематиці фінітизм означає використання лише таких методів, які вільні від неясностей і сумнівів.

З наступної системи аксіом логіки висловлень H&A (тут «v» – це символ логічної операції «або» s «→» – символ логічного проходження; «→» скороченням для «¬X ∨ Y», де «¬» – символ логічного заперечення):

$$X \vee X \to X\,(a)$$

$$X \to X \vee Y\,(b)$$

$$X \vee Y \to Y \vee X\,(c)$$

$$(X \to Y) \to (Z \vee X \to Z \vee Y)\,(d)$$

до яких приєднуються дві аксіоми з кванторами *загальності* і *існування* (відповідно ∀ і ∃):

$$\forall\, x\ F(x) \to F(y)\ (e)$$

$$F(y) \to \exists\, x\ F(x)\ (f)$$

і два правила виведення – підстановки і правило висновку (з X і X → Y випливає Y), будується закінчений фрагмент математичної логіки – функціональне числення першого порядку (зараз прийнятий інший термін – логіка першого порядку).

∀ – універсальний квантор[1] або квантор загальності. У природній мові читається як «всі», «кожен», «всякий», «який би не був».

∃ – екзистенціальний квантор або квантор існування. У природній мові читається як «існує (бодай один)», «існують», «знайдеться (бодай один)», «знайдуться», «деякі».

Позначення кванторів представлено в табл.7.1

Таблиця 7.1.

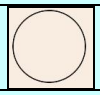
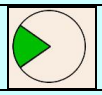
Позначення кванторів

Квантори загальності	Кількісні числівники	Квантори існування		Відсутність квантора
все	дві третини	рідко	зрідка	X
усі	одна восьма	часто	хтось	
завжди	половина	регулярно	подекуди	
вічно		чимало	буває	
ніколи		постійно	трапляється	
ніхто		більшість		
ніде		меншість		

Квантори загальності повідомляють, що «або все, або нічого», а «розмір частки», який описується кванторами існування, змінюється більш плавно: *рідко – буває – часто – постійно.* Якщо об'єднати, то ми отри маємо перехід: *ніколи – рідко – буває – постійно – завжди.* На границях, зрозуміло, квантори загальності, які спочатку повідомляють що нічого такого немає, а в кінці – що це наше *«все».* Геометрична інтерпретація кванторів представлена в табл. 7.2

Таблиця 7.2.

Геометрична інтерпретація кванторів

Квантори загальності	Кількісні числівники	Квантори існування		Відсутність квантора
				
ніхто	подекуди	часто	часто	усі
ніде	дехто	буває	регулярно	усе
нічого	зрідка	трапляється	більшість	завжди
пусто	мало	іноді	більша частина	вічно
	небагато	деякі	дві третини	постійно
	меншість	половина	багато	цілком
	менша			
	частина			
	одна вісьма			

Треба звернути увагу на те, що поява кванторів впроваджує в математичну теорію поняття нескінченності. Обійтися без нескінченності неможливо: такі фундаментальні поняття математики, як пряма, натуральний ряд, границя послідовності, множина, – усі вони ґрунтуються на цьому поданні. Поняття нескінченності слугувало (і до сьогодні слугує) джерелом як математичних, так і філософських дискусій і стало причиною виникнення декількох напрямків вивчення основ математики.

В H&A була озвучена і проблема можливості розв'язання (від нім. entscheidungsproblem). Хоча мова у книзі перед усім йшла про математичну логіку, Д. Гілберт сподівався, що питання

[1] **Ква́нтор –** загальна назва для логічних операцій, які обмежують область істинності певного предиката.

можливості розв'язання може бути застосоване до усієї математики. Пізніше, у фундаментальному двотомнику [8], проблема можливості розв'язання була сформульована як „…проблема знаходження загальних методів розпізнавання загальної значимості або ж виконуваності логічних формул". Проблема можливості розв'язання успішно була вирішена для окремих, важливих розділів математичної логіки – вирахування висловлень і деяких випадків логіки першого порядку, і це вселяло надії, що щось подібне може бути здійснене й для усієї математики.

Позитивне рішення проблеми можливості розв'язання обіцяло чималі дивіденди: у випадку успіху можна було сподіватися за допомогою методів, що не викликають сумнівів, отримувати докази шляхом виписування послідовностей рядків, з яких кожна наступна випливала з аксіом і раніше виведених рядків, які вважаються уже доведеними.

Розглянемо, наприклад, систему аксіом Пеано[1] для натурального ряду [47]:

- 0 є число (1).
- Наступний елемент кожного числа є число (2).
- Жодні два числа не мають одного й того ж наступного елемента (3).
- 0 не є наступним елементом ані для якого числа (4).
- Аксіома індукції (будь-яка властивість, яке належить 0, а також наступному елементу кожного числа, яке має цю властивість, належить усім числам) (5).

Тепер, маючи три «примітивних» (застосовуючи термінологію Б. Рассела) [47] ідеї – «0», «число» і «наступний елемент» – можна послідовно побудувати арифметику натуральних, раціональних, дійсних, … чисел.

Використовуючи сучасну термінологію, проблему можливості розв'язання можна сформулювати наступним чином: знайти методи, які володіють широкою загальністю (сьогодні ми називаємо це алгоритмами), які дозволять будувати ланцюжки доказів від посилань до висновків. Тобто було потрібно звести проблему доказу теорем, яка вимагала величезних витрат часу, сил і інтелекту, до процедури, яка з наявних аксіом породила б усі теореми.

А чи існує метод, який дозволяє визначити, чи є задане твердження теоремою чи ні? Здавалося, що теоретично це можливо. Спокусливо було б отримати пристрій, вклавши у який необхідні «інгредієнти», отримати усі можливі результати.

Перший удар по програмі був нанесений у 1931 р. – усього через три роки після появи на світ H&A і формулювання Гілбертом проблеми можливості розв'язання. Удар наніс австрійський логік Курт Гедель (який, до речі, роком раніше довів, що логіка першого порядку є повною, тобто будь-яка істинна формула логіки предикатів є теоремою у вирахуванні предикатів). Удар вийшов нищівним і жорстоким: виявилося, що у найстарішій і, як вважалося, у найбільш вивченій математичній дисципліні – арифметиці, можуть існувати істинні, але бездоказові засобами самої арифметики твердження. Більш того, Гедель вказав, що можна побудувати твердження, яке стверджує, що воно бездоказове.

Результати Геделя неможливо було спростувати: його доказ був логічно бездоганним. Тоді, можливо, справа в тому, що сама арифметика – основа основ усієї математики – суперечлива?

Проблема несуперечності – центральна проблема основ математики.

Математична система суперечлива, якщо з її аксіом можна вивести дві суперечливі одна одній теореми.

На щастя, все «обійшлося». Декількома роками пізніше німецький математик Герхард Генцен довів, що арифметика несуперечлива (для цього Генцену довелося залучити більш сильні і вже нефінітні методи доказів).

Все це означало одне – засобами арифметики неможливо довести усі її теореми. Це виглядає парадоксально, але Гедель був правий: система аксіом арифметики неповна. Спроба розширити список аксіом, приєднавши до них недовідну усередині арифметики теорему, була приречена на провал: нова система все одно залишалася неповною. Іншими словами, система аксіом арифметики істотно неповна. Результат Геделя отримав назву «теореми про неповноту арифметики» (книг, присвячених результатам, отриманим Геделем, дуже багато, наприклад, [48, 49].

Результати Геделя фактично «підрили корені» під проблемою можливості розв'язання; позитивний розв'язок останньої став доволі примарним. Для окремих розділів математики розв'язання було можливим, але не для усієї математики. Програму Гільберта почало лихоманити; надії на те, що поталанить знайти метод, який би дозволив виводити істинні теореми чисто синтаксично (тобто шляхом маніпулювання символами на папері), танули: якщо вже в арифметиці існують істинні теореми, істинність яких сама арифметика неспроможна довести, не виходячи за межі своїх методів, то як тут вести мову про інші, більш складні галузі.

---

[1] Аксіоми Пеано дозволили формалізувати арифметику. Після введення аксіом стали можливі докази багатьох властивостей натуральних і цілих чисел, а також використання цілих чисел для побудови формальних теорій раціональних і речовинних чисел.

Остаточно надію Гілберта побудувати розв'язний метод «поховали» практично одночасно (у 1936 р.) американський логік Алонзо Черч і Алан Тюрінг. При тому, що їх підходи до проблеми розв'язання були зовсім різними, обидва – і А. Черч, і А. Тюрінг – прийшли до одного й того ж висновку: проблема можливості розв'язання – нерозв'язна. Іншими словами, не варто витрачати сили і час на пошуки універсальних алгоритмів – їх просто не існує.

Можна тільки припускати, як 24-річному А. Тюрінгу, який аж ніяк не був вундеркіндом (Тюрінг зміг вступити до університету з першої спроби), прийшла в голову думка використати для дослідження проблеми можливості розв'язання машину.

В останні студентські роки Тюрінг захопився математичною логікою. Він вивчає класичну працю Бертрана Рассела і Альфреда Норта Уайтхеда „Основи математики" (Principia Mathematica), знайомиться зі статтями Джона фон Неймана про логічні основи квантової механіки і роботами видатного австрійського логіка Курта Геделя про так звану „неповноту математики". На лекції кембриджського тополога і логіка Максвелла Г. А. Нюмена (1897-1984 рр.) Тюрінг вперше почув про entscheidungsproblem („проблему можливості розв'язання" алгоритму. Сутність її полягала у „пошуку загального методу, який дозволив би визначити, чи може бути виконане дане висловлення мовою формальної логіки, тобто встановити його істинність".

Люди думають, що комп'ютери представляють собою найвищу ступінь розвитку обчислювальних машин. Але це не так. Дійсно, комп'ютери дуже часто використовуються для обчислень, оскільки вони здатні виконувати їх дуже швидко. Однак комп'ютерні програми не мають нічого спільного з числами… Комп'ютер представляє собою універсальну машину, і я довів, що вони можуть справитися з будь-якою задачею, яка може бути описана символами. Я збираюся рухатися далі. На мою думку, комп'ютер може виконати будь-яку задачу, яка підвладна людському мозку. Будь-яку задачу…

А. Тюрінг

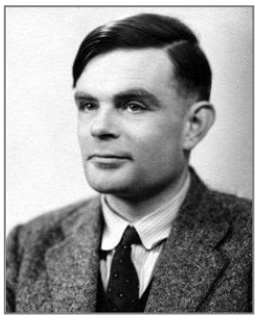


Рис. 7.3. Тюрінг Алан Метісон (Turing Alan Mathison (23 червня 1912 р., Лондон – 7 червня 1954 р., Уілмслоу, Великобританія.

Під впливом ідей Геделя, Тюрінг захопився цією проблемою і почав розробку алгоритмічного метода, здатного визначити, чи не має певна задача можливості розв'язання алгоритму з метою виключення таких задач з математики. У квітні 1936 р. він довів, що не існує універсального метода для визначення розв'язання, а отже, в математиці завжди будуть задачі, які не мають розв'язку (на відміну від невирішених на даний час). Отриманий результат він виклав у статті „Про обчислювальні числа, стосовно до проблеми можливості розв'язання" (On Computable Numbers, with an Application to the Entscheidungsproblem), опублікованій у 1936-1937 рр. і яка принесла йому світову популярність.

При цьому, Тюрінг замість того, щоб намагатись вирішити проблему можливості розв'язання аналітично (тобто тим же шляхом, яким Гедель отримав свої теореми про неповноту), уявно побудував свого роду математичну машину і дослідив за її допомогою проблему можливості розв'язання.

Праця Тюрінга спростувала думку Девіда Гілберта і його школи про те, що будь-яка математична теорія може бути виражена через набір аксіом і теорем.

Алан Метісон Тюрінг народився 23 червня 1912 р. у Лондоні. В шість років Алан навчився читати, писати і рахувати, а в початковій школі, куди його віддали через чотири роки, ставив хімічні досліди, намагаючись витягти йод з водоростей. Улюбленим читанням його дитячих років була науково-популярна книга „Природні чудеса, про які повинна знати кожна дитина".

У 1926 р., Алана віддають до приватної школи у містечку Шерборн на південному заході Великої Британії, яка готувала юних джентльменів з середнього класу до адміністративної і комерційної кар'єри.

У Шерборні Алан захопився точними науками, зокрема математикою, і з олівцем в руках вивчив теорію відносності Ейнштейна і квантову механіку за книгою А.С. Еддінгтона „Природа фізичного світу". До інших предметів він, м'яко кажучи, не виявляв зацікавленості. Це підтверджує відгук його класного керівника: *„Я можу пробачити його за твори, хоча це найгірше з того, що я будь-коли бачив; я намагаюся бути поблажливим до його неохайних робіт, повних помарок і непояснених неточностей... але я не можу вибачити його за дурне ставлення до розсудливого обговорення Нового Завіту*". І далі: *„Він належить до числа тих учнів, які створюють проблеми для будь-якої школи і для усього суспільства…*"

Не дивно, що Алан закінчив школу останнім у рейтингу успішності, та з певними проблемами отримав свідоцтво про її закінчення.

Незважаючи на „видатні результати" показані у школі, у 19-річному віці він все ж таки вступає до Королівського коледжу Кембриджського університету, де спеціалізується у галузі математики.

Після закінчення Королівського коледжу, Тюрінг поїхав продовжувати навчання до „Мекки наукового світу" – Принстонський Університет (США), де на той час працювали такі видатні вчені як Ейнштейн, фон Нейман, Гедель, Черч, Курант, Харді та інші. Під керівництвом математика Алонзо Черча у 1934 р. Тюрінг успішно захистив магістерську дисертацію з теорії імовірності, а у 1938 р. отримав ступінь доктора філософії.

Машина, яку запропонував Тюрінг, була надзвичайно простою, і в цьому полягає її перевага – йому потрібно було бути впевненим, що її використання не привнесе в проблему оцінки можливості розв'язання нових логічних складностей. Але простота конструкції зовсім не означала, що машина Тюрінга мало на що придатна. Це повноцінний комп'ютер, який ні у якому разі не слабкіший будь-яких найсучасніших і потужних машин, тільки що більш простий у побудові. А у цій простоті й крився запорука успіху: простота не закривала саму проблему, а допомагала її розв'язати.

Тюрінг скористався блискучою знахідкою Геделя – кодуванням обчислюваних функцій (тобто функцій, які обчислюються за допомогою алгоритмів). Клас таких функцій – вони називаються рекурсивними – дуже широкий: практично усі значимі математичні функції є обчислюваними. Гедель запропонував спосіб, який дозволив зіставити кожній з таких функцій унікальне натуральне число. З цього моменту стало можливим розмірковувати про обчислювані функції, оперуючи натуральними числами.

В основі кодування Гёделя лежить основна теорема арифметики: будь-яке натуральне число може бути єдиним (до порядку множників) чином розкладене на добуток простих чисел. Скажемо, число 10 єдиним чином можна розкласти (сучасний термін – «факторизувати») у вигляді 2x5; число 20 можна єдиним чином розкласти як добуток 2 x 2 x 5 тощо.

За допомогою цього метода Гедель зміг закодувати усі арифметичні твердження і вивести шляхом виточених міркувань свої теореми про неповноту.

Доказ Тюрінга ґрунтується на можливості завдання машині опису самої себе. Машина Тюрінга, спроможна виконувати програми, які описують її роботу (тобто програма, отримує саму себе на вході). Нічого незвичайного у такому «самозастосуванні» (від англ. self-referencing) немає – це приклад рекурсивної функції, подібної класичним функціям факторіала або послідовності Фібоначчі, правда, декілька незвичайного вигляду.

На жаль, як це не парадоксально, одна з найважливіших – якщо не сама важлива в історії комп'ютерних наук стаття до сьогодні не перекладена на російську або українську мову, і тому хто бажає ознайомитися з нею доводиться читати її мовою оригінала. Аналогічна ситуація, до речі, склалась й зі статтею Курта Геделя, у якій викладені його фундаментальні результати про неповноту арифметики.

У 1936 р. А. Тюрінг побудував логічну модель своєї знаменитої машини. Треба сказати, що в ті роки під словом *computer* малась на увазі людина, яка виконувала одноманітні обчислення за певними інструкціями. Наприклад, так називали бухгалтерів, рахівників тощо. Ідея Алана Тюрінга була в тому, що для проведення подібних дій присутність людини не потрібна.

Ідея машини Тюрінга була надзвичайно плідною (одним з перших, до речі, це оцінив Алонзо Чёрч, завдяки якому конструкція й отримала назву «машина Тюрінга»). Тюрінг здійснив дійсно науковий подвиг – він знайшов у собі мужність подивитись на проблему з іншої точки зору що й привело його до успіху.

Праця А. Тюрінга „Про обчислювані числа…" відіграла винятково важливу роль у розвитку обчислювальної математики і інформатики.

7.2. Концепція машини А. Тюрінга

У 1936 р. Аланом Тюрінгом для уточнення поняття алгоритму був запропонований абстрактний універсальний виконавець. Його абстрактність полягає у тому, що він представляє собою логічну обчислювальну конструкцію а не реальну обчислювальну машину. Термін „*універсальний виконавець*" говорить про те, що даного виконавця може імітувати будь-який інший виконавець. Наприклад операції, які виконують реальні обчислювальні машини, можна імітувати за допомогою універсального виконавця. Крім того, передбачається, що універсальний виконавець повинен вміти доводити існування або відсутність алгоритму для тієї або іншої задачі.

Що ж собою представляє машина Тюрінга? Машина Тюрінга – це строга математична побудова, математичний апарат (аналогічний, наприклад, апарату диференціальних рівнянь), створений для розв'язання певних задач. Цей математичний апарат був названий „машиною" за тією причиною, що за описом його складових частин і функціонуванню він схожий на обчислювальну машину. Принципова відмінність машини Тюрінга від обчислювальних машин полягає у тому, що її запам'ятовуючий пристрій представляє собою нескінченну стрічку: у реальних обчислювальних машин запам'ятовуючий пристрій може бути будь яким завгодно великим, але обов'язково кінцевим (обмеженим).

Машину Тюрінга не можна реалізувати саме через нескінченність її стрічки. У цьому значенні вона потужніше будь-якої обчислювальної машини. Згодом, обчислювальна конструкція, яка була придумана Тюрінгом була названа на його честь – машиною Тюрінга.

„Машина Тюрінга" – гіпотетичний універсальний перетворювач дискретної інформації (теоретична обчислювальна система), який складається з контрольного модуля, зчитувальної та записуючої голівок (пристрою введення-виведення), нескінченної паперової стрічки з записаними на ній символами.

У кожній машині Тюрінга є дві частини (рис. 7.4):

1) необмежена в обидві сторони стрічка, яка поділена на комірки;

2) автомат (головка для зчитування/запису,каретка, яка керується програмою).

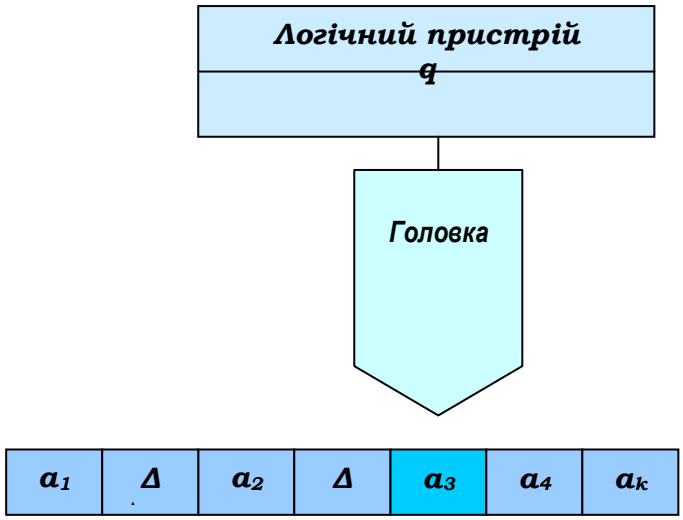


Рис. 7.4. Схема машини Тюрінга

Машина Тюрінга використовує також характеристику стану, яка може виражатись цілим числом від нуля до певної максимальної величини. Залежно від стану, машина Тюрінга може виконувати одну з трьох дій: записувати символ в комірку, пересуватися на одну комірку праворуч або ліворуч і встановлювати внутрішній стан.

Щоб задати конкретну машину Тюрінга, потрібно описати для неї наступні складові:

– зовнішній алфавіт;

– внутрішній алфавіт;

– таблицю переходів.

З кожною машиною Тюрінга пов'язані два кінцевих алфавіти: алфавіт вхідних символів (зовнішній алфавіт) $A = \{a_0, a_1, ..., a_m\}$ і алфавіт станів $Q = \{q_0, q_1, ..., q_p\}$ (внутрішній алфавіт). З різними машинами Тюрінга можуть пов'язуватись різні алфавіти A і Q.

Зовнішній алфавіт – кінцева множина (наприклад A), елементи якого називаються літерами (буквами, символами). Одна з літер цього алфавіту (наприклад, a_0) повинна представляти собою порожній символ.

Внутрішній алфавіт. Кінцева множина станів головки (автомата). Один зі станів (наприклад, q_1) повинен бути початковим (який запускає програму). Перебуваючи у цьому стані, машина починає свою роботу. Ще один зі станів (q_0) повинен бути кінцевим (який завершує програму) – стан зупинки. Стан q_0 називається пасивним. Вважається, що якщо машина попала у цей стан, то вона завершила свою роботу.

Вхідне слово розміщується на стрічці по одній літері у розташованих підряд комірках. Ліворуч і праворуч від вхідного слова знаходяться тільки пусті комірки (в алфавіт А завжди входить пуста літера a_0 – ознака того, що комірка є пустою).

Автомат може рухатися вздовж стрічки вліво або вправо, зчитувати вміст комірок і записувати в комірки літери. На рис.7.5 схематично показано огляд машиною Тюрінга першої комірки з даними.

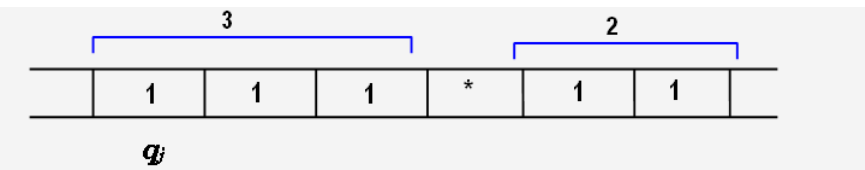


Рис.7.5. Огляд першої комірки з даними машиною Тюрінга

Таблиця переходів – опис поведінки автомата (головки) залежно від стану і зчитаного символу. Вміст таблиці переходів представляє собою команди для машини Тюрінга. Буква, яку зчитує головка в комірці (над якою вона перебуває у даний момент), і внутрішній стан головки визначають, яку команду потрібно виконати. Команда визначається перетинанням символів зовнішнього і внутрішнього алфавітів в таблиці.

Автомат машини Тюрінга в процесі своєї роботи може виконувати наступні дії:

– записувати символ зовнішнього алфавіту в комірку, замінюючи той, що знаходився у ній (зокрема й пустий).

– пересуватися на одну комірку ліворуч або праворуч.

– змінювати свій внутрішній стан.

Одна команда для машини Тюрінга як раз й представляє собою конкретну комбінацію цих трьох складових: вказівок, який символ записати в комірку (над якою перебуває автомат), куди пересунутися і в який стан перейти. Хоча команда може містити й не всі складові (наприклад, не змінювати символ, не пересуватися або не міняти внутрішнього стану).

Тобто у машини Тюрінга є три види операцій. Кожного разу для чергової пари (q_j, a_i) машина Тюрінга виконує команду, яка складається з трьох операцій з певними параметрами.

Програма для машини Тюрінга представляє собою таблицю переходів (рис.7.6), у кожній комірці якої записана команда.

	a_0	a_1	…	a_i	…	a_m
q_1						
q_2						
…				$a_i \left\{ \begin{matrix} Ë \\ Ï \\ Í \end{matrix} \right\} q_m$		
q_j						
…						
q_p						

Рис.7.6. Програма для машини Тюрінга

Комірка (q_j, a_i) визначається двома параметрами – символом алфавіту і станом машини. Команда представляє собою вказівку: куди перемістити головку читання/запису, який символ записати у поточну комірку, у який стан перейти машині. Для позначення напрямку руху автомата використаємо одну з трьох літер: "*Л*" (ліворуч), "*П*" (праворуч) або "*Н*" (нерухомий).

Після виконання автоматом чергової команди він переходить у стан q_m (який в окремому випадку може співпадати з попереднім станом q_j). Наступну команду потрібно шукати в m-му рядку таблиці на перетинанні зі стовпцем a_l (літеру a_l автомат бачить після зсуву).

Домовимося, що коли стрічка містить вхідне слово, то автомат знаходиться проти якоїсь комірки у стані q_1. В процесі роботи автомат буде перескакувати з однієї комірки програми (таблиці) в іншу, поки не дійде до комірки, у якій записано, що автомат повинен перейти до стану q_0. Ці комірки називаються комірками зупину. Дійшовши до будь-якої такої комірки, машина Тюрінга зупиняється.

Сукупність станів усіх комірок стрічки, станів логічного пристрою і головок називається <u>конфігурацією машини</u>.

Незважаючи на свій простий устрій, машина Тюрінга може виконувати усі можливі перетворення слів, реалізуючи тим самим усі можливі алгоритми і дає один із шляхів уточнення поняття алгоритму. У зв'язку з цим виникають питання:

- наскільки загальним є поняття машини Тюрінга?
- чи можна вважати, що спосіб завдання алгоритмів за допомогою машини Тюрінга є універсальним?
- чи може будь-який алгоритм задаватися таким чином?

На ці питання сучасна теорія алгоритмів пропонує відповідь у вигляді наступної гіпотези:

Будь-який алгоритм може бути заданий за допомогою тюрінгової функціональної схеми і реалізований у відповідній машині Тюрінга.

Ця гіпотеза отримала назву *тези Тюрінга*. Як і тезу Черча і Поста, її не можна довести, тому що вона зв'язує нестроге визначення поняття алгоритму із строгим визначенням машини Тюрінга.

В принципі, ця гіпотеза може бути спростована, якщо вдасться навести приклад алгоритму, який не може бути реалізований за допомогою тюрінгової функціональної схеми. Однак всі відомі до цього часу алгоритми можуть бути задані за допомогою тюрінгових функціональних схем.

Отже, <u>машини Тюрінга володіють важливою властивістю – можливістю побудови нової машини шляхом об'єднання вже наявних. Така операція називається ***композицією***.</u>

7.3. Приклад роботи машини Тюрінга

Припустимо, на стрічці є слово, яке складається з символів #, $, *1* і *0*. Потрібно замінити усі символи # і *$* на нулі. В момент запуску головка перебуває над першою літерою слова зліва. Завершується програма тоді, коли головка опиняється над пустим символом після крайньої правої літери слова.

Примітка: довжина слова і послідовність символів значення не мають. На рис. 7.7 наведений приклад послідовності виконання команд для конкретного випадку. Якщо на стрічці буде інше слово, то й послідовність виконання команд буде іншою. Незважаючи на це, дана програма для машини Тюрінга (на рис. 7.8. – таблиця ліворуч) застосовна до будь-яких слів описаного зовнішнього алфавіту (дотримується властивість застосовності алгоритму до усіх однотипних задач – масовість).

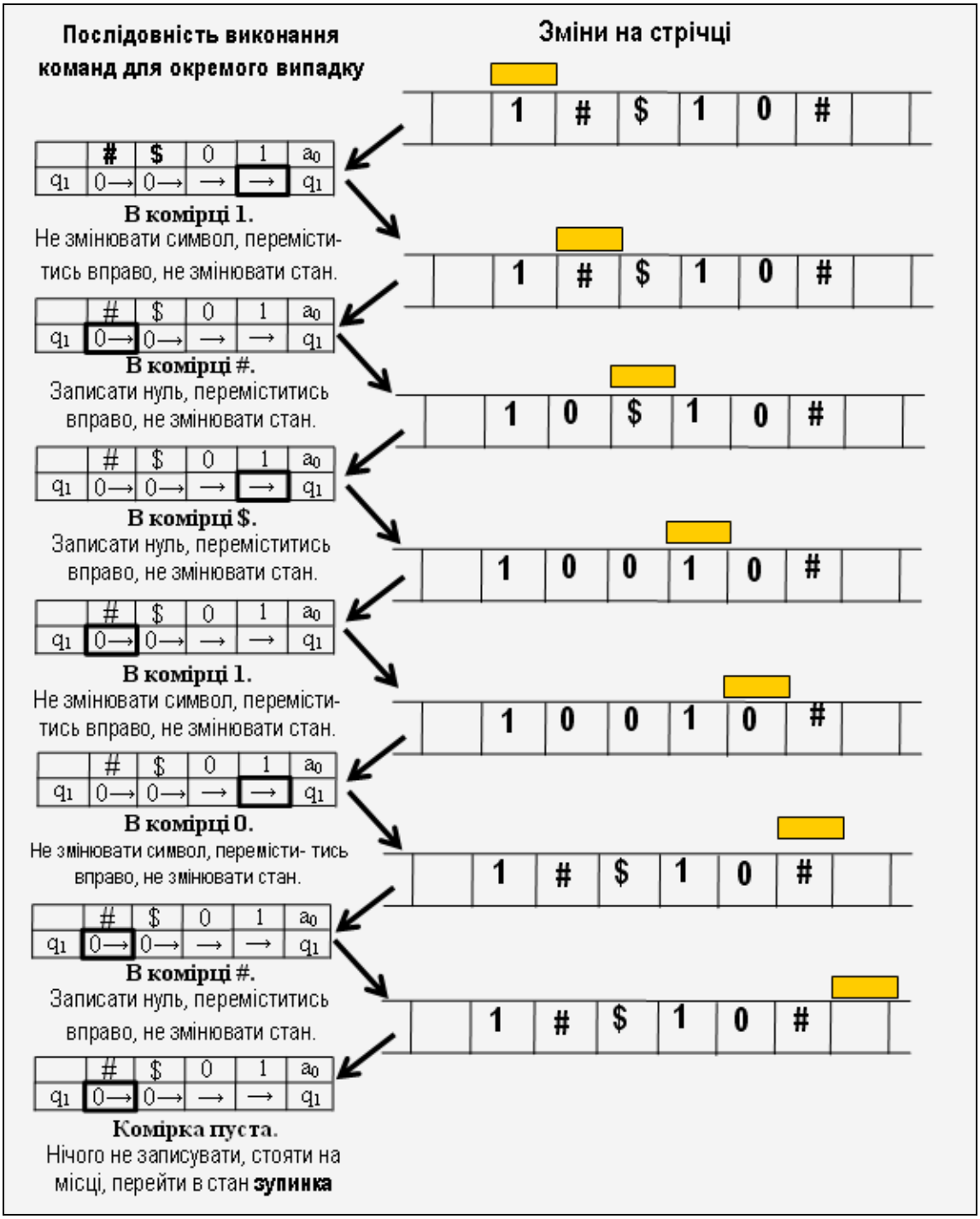


Рис.7.7. Приклад роботи машини Тюрінга

Можна ускладнити програму. Припустимо, головка розташовується не обов'язково над першим, а над будь-яким символом слова. Тоді програма для даної машини Тюрінга може бути такою (а могла би бути й іншою):

	#	$	0	1	a_0
q_1	←	←	←	←	→ q_2
q_2	0 →	0 →	→	→	q_0

Рис.7.8. Приклад програми для машини Тюрінга

Тут відбувається зсув головки вліво до тих пір, поки вона не опиниться над пустим символом. Після цього машина переходить у стан q_2 (команди якого співпадають з командами q_1 попередньої програми).

7.4. Універсальна машина Тюрінга

До цього часу ми мали справу зі спеціалізованими машинами Тюрінга, які призначались для розв'язку конкретних задач і відрізнялись набором команд, внутрішнім і зовнішнім алфавітами. Однак можна побудувати й універсальну машину Тюрінга, здатну виконувати будь-який алгоритм, а значить роботу будь-якої машини Тюрінга. У цій машині вхідне слово повинне включати зображення програми і вхідне слово інтерпретованої машини.

Щоб отримати зображення програми інтерпретованої машини, потрібно послідовно, рядок за рядком, закодувати цю програму в алфавіті універсальної машини. Універсальна машина Тюрінга повинна мати фіксований зовнішній алфавіт, який використовується при запису програми, включаючи й алфавіт інтерпретованої машини. Вона повинна бути придатною до прийому в якості вихідної інформації усіляких станів пристрою управління і конфігурацією, в яких можуть зустрічатися букви (літери, символи) з різноманітних алфавітів зі скільки завгодно великою кількістю різних літер. Це досягається шляхом кодування конфігурації і програми будь-якої заданої машини Тюрінга в символах вхідного алфавіту універсальної машини. Само кодування повинне виконуватись наступним чином:

1) різні букви повинні замінюватися різними кодовими групами але одна й та ж буква повинна замінюватись усюди, де б вона не зустрічалась, однією й тією ж кодовою групою;

2) рядки кодових записів повинні однозначним чином розбиватись на окремі кодові групи;

3) повинна існувати можливість розпізнавати, які кодові групи відповідають різним зсувам керуючої головки (R, L, E), і розрізняти кодові групи, що відповідають буквам внутрішнього і зовнішнього алфавітів.

Розглянемо приклад такого кодування для машини Тюрінга T, яка має зовнішній алфавіт $A = \{a_1, a_2, \dots, a_k\}$ і внутрішній алфавіт $Q = \{q_1, q_2, \dots, q_z\}$. Якщо зовнішній алфавіт складається з символів $A = \{0, 1\}$, то умови кодування будуть дотримані при наступному способі кодування.

1. В якості кодових груп візьмемо $3+k+m$ різних слів виду 100...01, де між одиницями проставляються нулі; k – кількість символів зовнішнього алфавіту; m – кількість станів пристрою управління. Тоді розбивка рядків на кодові групи здійснюється шляхом виділення послідовностей нулів, поміщених між двома одиницями.

2. Співставлення кодових груп вихідним символам зовнішнього і внутрішнього алфавітів здійснюється відповідно наступної таблиці кодування (рис.7.9):

	Буква	**Кодова група**		
Зовнішній алфавіт	R	101		
	E	1001		
	L	10001		
	a_1	100001	- 4 нулі	Парна кількість нулів, більше чим 2
	a_2	10000001	- 6 нулів	
	.....			
	a_k	10.....01	2(k + 1) нулів	
Внутрішній алфавіт (стану)	q_1	1000001	- 5 нулів	Непарне кількість нулів, більше чим 3
	q_2	100000001	- 7 нулів	
	.....			
	q_m	10.....01	2(m + 1) + 1 нулів	

Рис.7.9. Таблиця кодування

Наприклад, для машини Тюрінга, яка перетворює слово *bcad* в слово *bccd*, вхідне слово в універсальній машині Тюрінга з даним кодом буде представлено наступним рядком:

10000001 1000000001 100001 100000000001,

де *100001 – a, 10000001 – b, 1000000001 – c, 100000000001 – d.*

Програма ж буде представлена наступними рядками:

1) *1000001 10000001 1000001 10000001 101*
($q_0 b \rightarrow q_0\ b\ R$)
2) *1000001 1000000001 1000001 1000000001 101*
($q_0 c \rightarrow q_0\ c\ R$)
3) *1000001 100001 100000001 1000000001 101*
($q_0\ a \rightarrow q_1\ c\ R$)
3) *100000001 100000000001 10000000001 100000000001 10001*
($q_1 d \rightarrow q_2 dL$).

Розглянемо ще один приклад. Нехай на стрічку універсальної машини Тюрінга надходить слово, складене з букв англійського алфавіту. Задача машини Тюрінга – переставляти місцями букви *n* і *o* таким чином, щоб словосполучення *on* перетворювалось в *no*. Таким чином, після переробки вхідного слова у ньому не повинно залишитись жодного буквосполучення *on*.

Візьмемо слово *mnonnop*, яке повинне перетворитись універсальною машиною Тюрінга в нове слово *mnnnoop*.

Нехай зовнішній алфавіт універсальної машини Тюрінга складається із символів $A=\{0,1\}$, а внутрішній алфавіт $Q=\{q_0, q_1, q_2, q_3, q_z\}$, де q_z – заключний стан. Розбивка вхідних символів на кодові групи і співставлення кодових груп вихідним символам зовнішнього і внутрішнього алфавітів здійснюється відповідно наведеній вище таблиці кодування.

Тоді вхідне слово буде представлено наступним чином:

100001 10000001 1000000001 10000001 10000001 1000000001 100000000001,

де *100001 – m, 10000001 – n, 1000000001 – o, 100000000001 – p.*

Нижче представлені команди універсальної машини Тюрінга, які будуть виконуватись при обробці і перетворенні вихідного слова. Напроти кожної команди приводиться вхідне слово таким, яким воно є на момент початку виконання даної команди. Символ, на який вказує головка машини Тюрінга, представлений прописною буквою.

$q\ 0\ m \rightarrow q\ 0\ m\ R$	$M\ n\ o\ n\ n\ o\ p$
$q\ 0\ n \rightarrow q\ 0\ n\ R$	$m\ N\ o\ n\ n\ o\ p$
$q\ 0\ o \rightarrow q\ 1\ o\ R$	$m\ n\ O\ n\ n\ o\ p$
$q\ 1\ n \rightarrow q\ 2\ o\ L$	$m\ n\ o\ N\ n\ o\ p$
$q\ 2\ o \rightarrow q\ 0\ n\ R$	$m\ n\ O\ o\ n\ o\ p$
$q\ 0\ o \rightarrow q\ 1\ o\ R$	$m\ n\ n\ O\ n\ o\ p$
$q\ 1\ n \rightarrow q\ 2\ o\ L$	$m\ n\ n\ o\ N\ o\ p$
$q\ 2\ o \rightarrow q\ 0\ n\ R$	$m\ n\ n\ O\ o\ o\ p$
$q\ 0\ o \rightarrow q\ 1\ o\ R$	$m\ n\ n\ n\ O\ o\ p$
$q\ 1\ o \rightarrow q\ 0\ o\ R$	$m\ n\ n\ n\ o\ O\ p$
$q\ 0\ p \rightarrow q\ z\ p\ E$	$m\ n\ n\ n\ o\ o\ P.$

Програма ж виглядає так:

1000001 100001 1000001 100001 101
1000001 10000001 1000001 10000001 101
1000001 1000000001 100000001 1000000001 101
100000001 10000001 10000000001 1000000001 10001
10000000001 1000000001 1000001 10000001 101
1000001 1000000001 100000001 1000000001 101
100000001 10000001 10000000001 1000000001 10001
10000000001 1000000001 1000001 10000001 101
1000001 1000000001 100000001 1000000001 101
100000001 1000000001 1000001 1000000001 101
1000001 100000000001 1000000000001 100000000001 1001.

Таким чином, якщо будь-яка машина Тюрінга *T* розв'язує певну задачу, то й універсальна машина Тюрінга спроможна вирішити цю задачу при умові, що крім кодів вихідних даних цієї задачі на її стрічку буде поданий код програми машини *T*. Задаючи універсальній машині Тюрінга T_u зображення програми будь-якої машини Тюрінга T_n і зображення будь-якого її вхідного слова x_n, отримаємо зображення вихідного слова y_n, у яке машина T_n переводить слово x_n .

Якщо ж алгоритм, що реалізується машиною T_n, не можна застосувати до слова x_n, то алгоритм, який реалізується універсальною машиною T_u, також не можна застосувати до слова, утвореному з зображення x_n і програми машини T_n.

Таким чином, машина Тюрінга T_n може розглядатись як одна з програм для універсальної машини T_u.

У зв'язку з існуванням універсальної машини Тюрінга, таблиці відповідності, які описують різні стани пристрою управління машини, мають подвійне призначення:

1) для опису станів пристрою управління спеціальної машини Тюрінга, яка реалізує відповідний алгоритм;

2) для опису програми, яка подається на стрічку універсальної машини Тюрінга, при реалізації відповідного алгоритму.

Сучасні ЕОМ будуються як універсальні; в запам'ятовуючий пристрій разом з вхідними даними поставленої задачі вводиться також і програма її розв'язку. Однак на відміну від машини Тюрінга, у якої зовнішня пам'ять (стрічка) нескінченна, у будь-якій реальній обчислювальній машині вона кінцева.

Питання для самоконтролю

1. Чому виникла проблема розробки алгоритмічного метода, здатного визначити, чи не має певна задача рішення?
2. Що розуміють під машиною Тюрінга?
3. В чому полягає сутність універсальної машини Тюрінга?
4. Як працює універсальна машина Тюрінга?
5. Що розуміють під терміном „клас розв'язуваних задач"?
6. В чому полягає сутність тези Тюрінга?

8. АБСТРАКТНИЙ АЛФАВІТ ТА ФОРМАЛЬНІ ГРАМАТИКИ

Якщо починають з неправильного, то надія на правильне завершення є дуже малою…
Конфуцій

Наука, яка займається вивченням мов, називається семіотикою. В семіотиці розрізняють мови формальні, правила вживання яких строго і явно визначені, і неформальні, вживання яких ґрунтується на практиці, яка склалась. Розрізняють також природні мови, які з'являються „самі по собі" в процесі розвитку суспільства, і штучні мови, які створюються певними групами людей для конкретних цілей. У більшості випадків формалізація штучних мов не викликає ускладнень, задача формалізації природних мов дуже складна і мабуть у найближчі десятиліття вирішена не буде.

Під формалізацією мови розуміють складання її математичної моделі, яку можна вивчати, аналізувати і перетворювати, використовуючи доступний математичний апарат. В математичній моделі мови виділяються три стандартні складові: синтаксис, семантика і прагматика. При поданні моделей формальних мов найбільш широко використовується синтаксичний підхід, для природних мов дослідники застосовують симбіоз морфологічної, синтаксичної і семантичної складових мовних моделей.

Ефективність використання найбільш перспективних існуючих моделей, коло вирішуваних ними задач, ступень розвитку в них вищеперерахованих складових «ідеальної» математичної моделі мови – це ті задачі, які необхідно висвітити для складання повної і єдиної картини, яка склалась у сучасній теорії формальних мов.

8.1. Загальні відомості про абстрактний алфавіт

Для обміну інформацією між людьми використовуються природні мови (українська, російська, англійська, китайська тощо). Основу мови складає алфавіт, або набір символів (знаків), які людина розрізняє за їх накресленням. В основі української мови лежить кирилиця, яка включає 33 знаки (букви), в англійській мові використовується латиниця (26 знаків), в китайській мові – алфавіт з десятків тисяч знаків (ієрогліфів).

Послідовності символів алфавіту утворюють у відповідності з правилами граматики основні об'єкти мови – слова. Правила, згідно яких будуються речення зі слів даної мови, називаються синтаксисом. Необхідно відзначити, що в природних мовах граматика і синтаксис мови формулюються за допомогою великої кількості правил, з яких є виключення, оскільки такі правила складались історично.

Поряд з природними мовами були розроблені формальні мови (нотний запис, мови програмування тощо). Основна відмінність формальних мов від природних полягає в наявності не тільки жорстко зафіксованого алфавіту, але й строгих правил граматики і синтаксису. Так, правила запису математичних виразів можна розглядати як формальну мову, яка має алфавіт (цифри) і яка дозволяє не тільки іменувати і записувати об'єкти (числа), але й виконувати над ними арифметичні операції за строго визначеними правилами. В деяких мовах замість букв і цифр використовуються інші символи, наприклад, знаки хімічних елементів, музикальні ноти, зображення елементів електричних або логічних схем, дорожні знаки, точки і тире (код азбуки Морзе) тощо.

Таким чином, подання інформації за допомогою природних і формальних мов здійснюється за допомогою алфавіту – визначеного набору знаків. Інформація передається у вигляді повідомлень. Дискретна інформація записується за допомогою певного кінцевого набору знаків, які будемо називати буквами, не вкладаючи в це слово обмеженого значення (типу „українські букви" або „латинські букви"). Буква у даному розширеному розумінні – будь-який зі знаків, які певною угодою встановлені для спілкування. Наприклад, при звичайній передачі повідомлень українською мовою, такими знаками будуть українські букви – великі і малі літери, розділові знаки, пробіл; якщо в тексті є числа – то й цифри. Взагалі, буквою будемо називати елемент певної кінцевої множини (набору) відмінних один від одного знаків. Множина знаків, у якій визначений їх порядок, назвемо алфавітом (загальновідомий порядок знаків в українському алфавіті: А, Б,…, Я).

Знаки можуть мати різну фізичну природу. Наприклад, для письма слугують знаки, які є зображеннями на папері, в усній мові знаками виступають різні звуки (фонеми), а при обробці тексту на комп'ютері знаки подаються у формі послідовностей електричних імпульсів.

Розглянемо деякі приклади алфавітів.

1. Алфавіт прописних українських букв (рис.8.1):

А, Б, В, Г, Ґ, Д, Е, Є, Ж, З, И, І, Ї, Й, К, Л, М, Н, О, П, Р, С, Т, У, Ф, Х, Ц, Ч, Ш, Щ, Ь, Ю, Я

Рис.8.1. Алфавіт прописних українських букв

2. Алфавіт Морзе (рис.8.2):

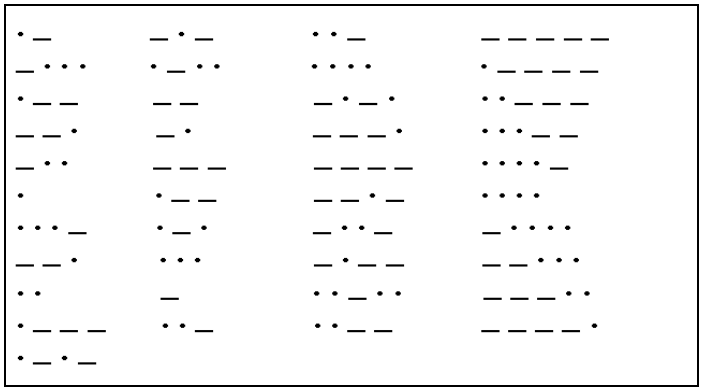


Рис.8.2. Алфавіт Морзе

3. Алфавіт клавіатурних символів ПЕОМ IBM для клавіатури (рис.8.3):

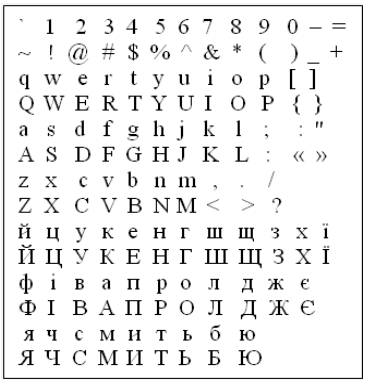


Рис.8.3. Алфавіт клавіатурних символів ПЕОМ IBM

4. Алфавіт знаків правильної шестигранної гральної кості (рис.8.4):

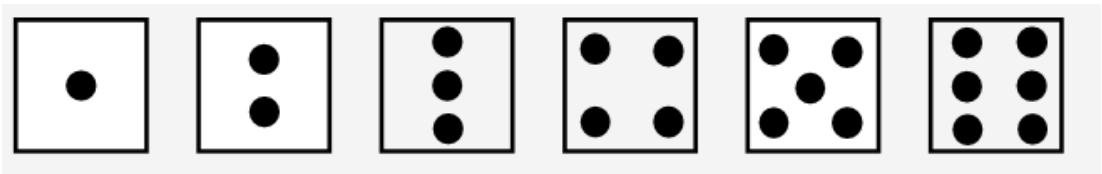


Рис.8.4. Алфавіт знаків правильної шестигранної гральної кості

5. Алфавіт арабських цифр (рис.8.5):

0123456789

Рис.8.5. Алфавіт арабських цифр

6. Алфавіт шістнадцяткових цифр (рис.8.6):

0 1 2 3 4 5 6 7 8 9 A B C D E F

Рис.8.6. Алфавіт шістнадцяткових цифр

Цей приклад, зокрема, показує, що знаки одного алфавіту можуть утворюватися із знаків інших алфавітів.

7. Алфавіт двійкових цифр (рис.8.7):

0 1

Рис.8.7. Алфавіт двійкових цифр

Алфавіт 7 є одним з прикладів, так званих, “двійкових” алфавітів, тобто алфавітів, що складаються з двох знаків. Іншими прикладами є двійкові алфавіти 8 і 9:

8. Двійковий алфавіт “точка, “тире” (рис.8.8):

. –

Рис.8.8. Двійковий алфавіт “точка, “тире”

9. Двійковий алфавіт “плюс”, “мінус” (рис.8.9):

+ -

Рис.8.9. Двійковий алфавіт “плюс”, “мінус”

10. Алфавіт прописних латинських букв (рис.8.10):

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Рис.8.10. Алфавіт прописних латинських букв

11. Алфавіт римської системи числення (рис.8.11):

I V X L C D M

Рис.8.11. Алфавіт римської системи числення

12. Алфавіт мови блок-схем зображення алгоритмів (рис.8.12):

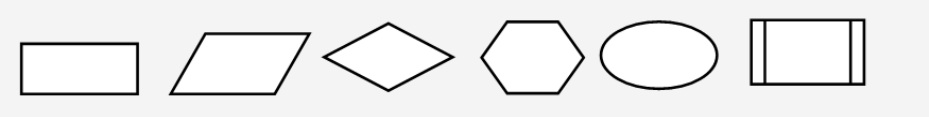


Рис.8.12. Алфавіт мови блок-схем зображення алгоритмів

У сучасній математиці алгоритмами прийнято називати конструктивно задані відповідності між словами в абстрактних алфавітах. Це визначення, в свою чергу, використовує два поняття – поняття абстрактного алфавіту і слів в такому алфавіті.

Абстрактним алфавітом називається певна кінцева сукупність об’єктів, які називаються буквами або символами даного алфавіту. Іншими словами алфавіт – це кінцева множина розрізнених символів (слово „абстрактний” для скорочення тут і далі опущене). Алфавіт, як і будь-яка інша множина, може бути заданий перерахуванням його елементів. Наприклад, алфавіт A є $A=\{a, b, c\}$, алфавіт B є $B=\{x, y\}$.

Під словом або рядком (ланцюжком) в алфавіті розуміють будь-яку кінцеву послідовність символів. В послідовності (ланцюжку) між символами стоїть операція зчеплення або конкатенації, тобто міняти місцями символи у послідовності не можна. Наприклад, в алфавіті A словами є будь-які послідовності: *a, ac, cb, acb, bb*, а в алфавіті B: *x, y, yx, xx* тощо.

Алгоритмічна система складається з абстрактного алфавіту та скінченої сукупності припустимих операцій.

При цьому природа об'єктів алфавіту нас зовсім не цікавить. Символом абстрактних алфавітів можна вважати букви алфавіту певної мови, цифри, будь-які значки і навіть слова якоїсь конкретної мови. Основною вимогою до алфавіту є його кінцівка. Таким чином, можна стверджувати, що алфавіт – це кінцева множина різних символів. Алфавіт, як всяка множина, задається перерахуванням його елементів.

Отже, об'єкти реального світу можна зображувати словами в різних алфавітах. Це дозволяє вважати, що об'єктами роботи алгоритмів можуть бути тільки слова.

У різних алгоритмічних системах застосовують різні сукупності об’єктів (символів або станів), різні сукупності припустимих операцій та різні правила їх виконання. Одні й ті самі дані можуть бути подані у різних алфавітах, а алгоритм розв’язання однієї й тієї самої задачі можна уявити послідовностями різних припустимих операцій. Отже, одна й та сама задача може бути розв’язана в межах різних алгоритмічних систем. Складність алгоритму та простота його реалізації будуть залежати від того, наскільки вдало було обрано алгоритмічну систему для розв’язку конкретної задачі.

Доведено можливість та правомірність переходу з однієї алгоритмічної системи до іншої у процесі побудови та реалізації алгоритму розв'язання задачі (тобто виконання еквівалентних перетворень алгоритму).

У будь-якій алгоритмічній системі розрізняють два типи припустимих операцій:

- ✓ операції дії;
- ✓ операції розпізнавання.

За допомогою операції дії здійснюється перехід з одного стану інформації про задачу до іншого.

Операції розпізнавання використовуються для встановлення тих чи інших особливостей стану інформації для визначення операції дій.

Для того, щоб будувати і застосовувати алгоритми, виконувати порівняння, оцінювати і вибирати найбільш прийнятні з них, необхідно мати певні, чіткі правила побудови цих об'єктів, називаних даними, з фіксованої множини базових об'єктів, яка називається алфавітом.

Послідовність слів на вході і виході алгоритму складається з символів. В залежності від обраних символів, по різному і буде представлятись перетворення інформації.

Кількість символів у слові визначає його довжину. Якщо довжина дорівнює нулю, то слово називають пустим (0). Слово *p* називається підсловом слова *q*, якщо слово *q* можна представити у вигляді *q=pr*, где *r* – будь-яке слово, у тому числі й пусте. Очевидно, що таке поняття слова буде відрізнятись від аналогічного у людських мовах. Тут під словом можна розуміти будь-яку послідовність символів, навіть беззмістовну.

При розширенні алфавіту поняття слова може істотно змінюватись. Так, наприклад, в алфавіті *A={0,1,2,3,4,5,6,7,8,9}* ланцюжок символів *69+73* представляє собою два слова, які поєднані знаком суми, а в алфавіті *B={+,0,1,2,3,4,5,6,7,8,9}* це буде одно слово.

8.2. Поняття алфавітного оператора

Поняття алфавітного оператора є найбільш загальним, оскільки до нього фактично зводяться будь-які процеси перетворення інформації.

Алфавітним оператором або алфавітним відображенням називається будь-яка відповідність між словами певного алфавіту і словами в тому ж самому або в будь-якому іншому фіксованому алфавіті. Перший називається вхідним, а другий – вихідним алфавітом даного оператора. У випадку збігу вхідного і вихідного алфавітів кажуть, що алфавітний оператор заданий у відповідному алфавіті.

Нехай задані слова в алфавітах *X* і *Y*, і задані відповідності між цими словами. Якщо x_i – слово в алфавіті *X*, а y_j – слово в алфавіті *Y*, то алфавітний оператор $Гx_i=y_j$ перетворює вхідне слово x_i у вихідне слово y_j. Символ *Г* в алфавітному операторі означає відображення.

Розрізняють однозначні і багатозначні алфавітні оператори. Під однозначним алфавітним оператором розуміють такий алфавітний оператор, який кожному вхідному слову ставить у відповідність не більше одного вихідного слова.

Сукупність усіх слів, на яких алфавітний оператор визначений, називається областю його визначення. Алфавітний оператор, який не зіставляє даному вхідному слову a_i жодного вихідного слова b_j (в тому числі й пустого), на цьому слові вважається невизначеним.

Основою теорії алфавітних операторів є способи їх завдання. Якщо область визначення алфавітного оператора кінцева, то оператор може бути заданий простою таблицею відповідності. У випадку нескінченної області визначення алфавітного оператора він задається системою правил, яка дозволяє за кінцеву кількість кроків знайти вихідне слово, яке відповідає заданому вхідному слову.

Алфавітні оператори, що задаються за допомогою кінцевої системи правил, називаються алгоритмами.

Алгоритми, у відповідності з якими розв'язання поставлених задач зводиться до арифметичних дій, називаються числовими алгоритмами.

Алгоритми, у відповідності з якими розв'язання поставлених задач зводиться до логічних дій, називаються логічними алгоритмами.

Під однозначним алфавітним оператором розуміють алфавітний оператор, який кожному вхідному слову ставить у відповідність не більше одного вихідного слова.

Під багатозначним алфавітним оператором розуміють алфавітний оператор, який кожному вхідному слову ставить у відповідність більше одного вихідного слова.

Алфавітний оператор, який не зіставляє даному вхідному слову a_i жодного вихідного слова b_j (у тому числі й пустого), вважають невизначеним на цьому слові.

Сукупність усіх слів, на яких алфавітний оператор визначений, називається областю його визначення.

Два алфавітних оператора вважаються рівними, якщо рівними є відповідні їм алфавітні оператори і збігається система правил, яка задає дію цих алгоритмів на вихідні слова.

Два алгоритми вважаються еквівалентними, якщо у них збігаються алфавітні оператори, однак не збігаються способи їх завдання (система правил).

Зазвичай в теорії алгоритмів розглядаються лише такі алгоритми, яким відповідають однозначні алфавітні оператори.

8.3. Формальні граматики

Програми для перших комп'ютерів записувались як послідовність машинних команд, представлених у вигляді двійкових або восьмеричних кодів. Зрозуміло, складати і налагоджувати великі програми у такому вигляді було дуже важко. Тому програмісти завжди прагнули навчити комп'ютер розуміти мову, наближену до людської. Так з'явилися Асемблер, FORTRAN, Алгол-60, C++ й інші мови програмування. Але всі вони в одному відношенні кардинально відрізняються від мов природних. В усній і писемній мові завжди припустима

певна неточність і недомовленість – слухач або читач відновить зміст з контексту (у загальному випадку цей контекст включає увесь культурний багаж людства). Комп'ютер же потребує точних і однозначних інструкцій. Значить, запис комп'ютерних програм на мовах програмування повинна підкорятися строгим правилам, які не припускають жодної двозначності, а компілятор повинен безпомилково перетворювати їх у послідовності машинних команд.

Речення у будь-якій мові складається з елементарних символів: букв, цифр, пробілів і розділових знаків. Тому можна було б формально визначити речення як послідовність або ланцюжок символів. Однак таке визначення занадто широке. У величезній кількості різноманітних ланцюжків символів є сенс виділити підмножину правильно побудованих ланцюжків – тих, які складаються з „правильних слів, розставлених у правильному порядку". Наприклад, ланцюжки „*рорилво ргнцої орвилора іиврпфг*" і „*По кареті дорогою пливе*" не є правильно складеними реченнями української мови, речення ж „*Карета пливе дорогою*" складене правильно, хоча і є сумнівним з точки зору змісту.

Граматика – це засіб завдання мови. В штучних мовах, до яких відносяться і мови програмування, множина правильно побудованих речень може бути заздалегідь точно описана за допомогою формальних математичних конструкцій. Тому такі мови називають формальними мовами.

Формальна мова вважається визначеною, якщо задані:

1) алфавіт, який включає не тільки букви, але й усі символи даної мови, у тому числі цифри, пробіли, розділові знаки;

2) критерій, який дозволяє однозначно визначити, чи є даний ланцюжок символів з алфавіту мови правильно побудованим чи ні. Яким чином задається цей критерій, не має значення. У найпростішому випадку можна просто перерахувати усі припустимі ланцюжки. Або скласти звід правил „створення" подібних ланцюжків. Такий звід правил називається граматикою, а будова ланцюжків мови – синтаксисом.

Контекстно-вільні граматики були запропоновані лінгвістами для опису природних мов. Складові частини ланцюжка позначають або буквами, які не входять в алфавіт мови, або назвами понять – метасимволами, або нетерміналами, укладеними в кутові скобки. Можна розглянути найбільш прості фрази української мови, в яких використовуються нетермінали: речення, підмет, група підмета, присудок, група присудка, визначення, обставина. Граматика складається з правил. У лівій частині правила записується нетермінал, у правій частині – довільний ланцюжок з терміналів і нетерміналів, де начебто розкривається значення поняття, зазначеного у лівій частині. Права і ліва частини правила розділяються стрілочкою, яка читається „це є". Перше правило цієї граматики визначає, що речення складається з двох частин – групи підмета, за якою іде група присудка.

$$\langle \text{речення} \rangle \rightarrow$$
$$\langle \text{група підмета} \rangle \ \langle \text{група присудка} \rangle .$$

В граматиці повинні бути розписані правила для усіх понять мови. В прикладі можна ввести два правила для групи підмета:

$$\langle \text{група підмета} \rangle \rightarrow$$
$$\langle \text{підмет} \rangle$$
$$\langle \text{група підмета} \rangle \rightarrow$$
$$\langle \text{визначення} \rangle \langle \text{група підмета} \rangle .$$

У другому правилі нетермінал група підмета зустрічається як у лівій, так і в правій частинах. Це важлива риса контекстно-вільної граматики, яка дозволяє будувати скільки завгодно довгі фрази. У даному випадку друге правило дозволяє додавати ліворуч будь-яку кількість визначень до підмету. Якщо в граматиці є декілька правил з однаковою лівою частиною, то їх можна звести в один запис, з'єднавши усі праві частини за допомогою вертикальної риси (вона читається як „або"):

$$\langle \text{група підмета} \rangle \rightarrow$$
$$\langle \text{підмет} \rangle \mid \langle \text{визначення} \rangle \langle \text{група підмета} \rangle .$$

Нижче наведені усі правила граматики:

$$\langle \text{речення} \rangle \rightarrow$$
$$\langle \text{група підмета} \rangle \langle \text{група присудка} \rangle ;$$
$$\langle \text{група підмета} \rangle \rightarrow$$
$$\langle \text{підмет} \rangle \mid \langle \text{визначення} \rangle \langle \text{група підмета} \rangle ;$$
$$\langle \text{підмет} \rangle \rightarrow \textit{човен} \mid \textit{карета};$$

⟨визначення⟩ → *зелена* | *біла* | *велика;*

⟨група присудка⟩ → ⟨присудок⟩ ⟨îáñòàâèíà ⟩;

⟨присудок⟩ → *котиться* | *пливе;*

⟨обстоятельство⟩ → *дорогою* | *річкою.*

В граматиці виділяється початковий нетермінал; ланцюжки мови виводяться з нього. В наведеному прикладі це нетермінал ⟨речення⟩. На першому кроці до нього застосовують одне з правил, ліва частина якого співпадає з даним нетерміналом. Відбувається заміна символу на правую частину правила. Послідовність таких замін називається виведенням. Виведення закінчиться, коли в ланцюжку не залишиться жодного нетермінала.

Розглянемо приклад виведення. Використаємо одне правило для нетермінала:

⟨речення⟩ →⟨група підмета⟩ ⟨група присудка⟩.

Потім до нетерміналу ⟨група підмета⟩ застосовується друге правило:

⟨група підмета⟩ ⟨група присудка⟩→ ⟨визначення⟩ ⟨група підмета⟩ ⟨група присудка⟩.

Продовжимо виведення:

⟨визначення⟩ ⟨група підмета⟩ ⟨група присудка⟩→ зелена ⟨група підмета⟩ ⟨група присудка⟩→

зелена ⟨ підмет⟩ ⟨група присудка⟩→ зелена карета ⟨група присудка⟩→

зелена карета ⟨присудок⟩ ⟨îáñòàâèíà ⟩→ зелена карета котиться ⟨îáñòàâèíà ⟩→

зелена карета котиться по дорозі.

На останньому кроці утворюється закінчене виведення. Аналогічно ццьому можна побудувати й інші варіанти ⟨речення⟩:

великий білий човен пливе по річці;
карета пливе по дорозі;
біла зелена карета котиться по річці тощо.

Два останніх приклади показують, що граматика ніяким чином не враховує фрази.

Алгоритм раніше був визначений як алфавітний оператор з кінцевою системою правил перетворення. Для запису вхідних, проміжних і вихідних слів використовується певний алфавіт. Певним чином повинні бути описані й правила перетворення. Очевидно, для цього потрібна певна *мова.* А чи придатна для опису алгоритму звичайна людська мова?

Будь-яка природна мова виникала як засіб спілкування людей. Саме за цією причиною, їй притаманні наступні особливості:

- мінливість, яка складається з мінливості словарного складу мови;
- неоднозначність трактування фраз різними людьми;
- надмірність.

Перераховані особливості не дозволяють застосувати природну мову для запису алгоритму, оскільки однією з властивостей алгоритму є його детермінованість, тобто однозначність виконання кроків будь-яким виконавцем.

Найбільш простий шлях подолання небажаних властивостей природних мов, є побудова мов штучних (*формальних)* зі строгим синтаксисом і повною значеннєвою визначеністю. Теорія формальних граматик була сформульована у 1959 р. американським лінгвістом М. Хомським (рис.8.14), хоча основи, на яких вона базується, були винайдені значно раніше.

Хомський ввів поняття формальної граматики і породжуваної нею мови. Запровадив чотири класи мов. Назви мов згідно визначення Хомського.

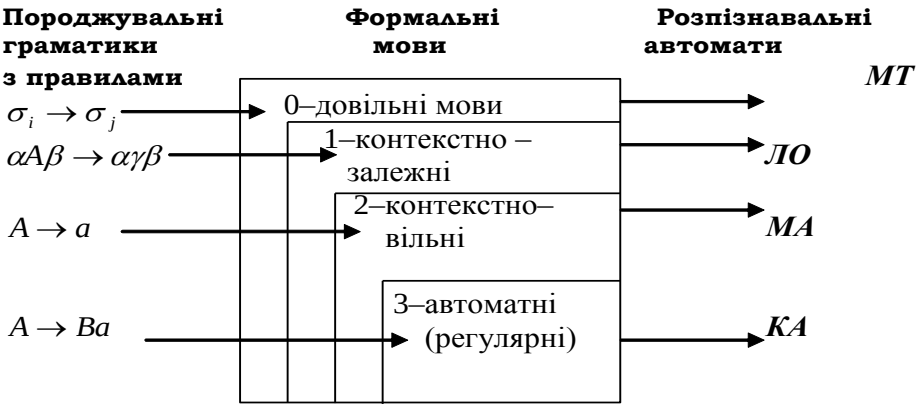


Рис.8.13. Визначення класів мов за А. Хомським

Мови клас «0» (довільні) не вживаються для практичних цілей.

Мови класу «1» (контекстно-залежні); ЛО, КЗ-граматики використовуються для побудови програм – перекладачів з природних мов.

Мови класу «2» (контекстно-вільні); МА, КС-граматики використовуються для побудови синтаксичних аналізаторів мов програмування.

Мови класу «3» (автоматні мови); КА, А – граматики використовуються для побудови текстових редакторів і налагоджувальників програм. Наприклад, МЕ, Perl тощо.

Граматики типу 0 – це граматики, на правила виводу яких відсутні обмеження. Граматики типу 1, або контекстні граматики – це граматики, усі правила яких мають такий вигляд:

$$xAy \to x\varphi y,$$

де $A \in V_N$, $x, y, \varphi \in (V_N \cup V_T)^+$.

Граматики типу 2 – це безконтекстні, або контекстно-вільні граматики (КС-граматики). Правила виводу для цих граматик мають наступний вигляд:

$$A \to \varphi,$$

де $A \in V_N$, $\varphi \in (V_N \cup V_T)^*$.

Граматики типу 3 – це автоматні граматики, які поділяються на два типи:

а) ліволінійні (ліворекурсивні), правила виводу для яких мають наступний вигляд:

$$A \to Aa \mid a,$$

де $A \in V_N$;

б) праволінійні (праворекурсивні), правила виводу для яких мають наступний вид:

$$A \to Aa \mid a.$$

Мова L називається мовою типу i, якщо існує граматика типу i, яка породжує мову L.

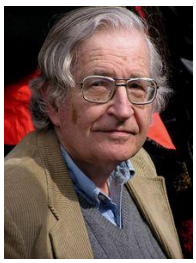


Рис.8.14. Авра́м Ноа́м Хо́мський (також транскрибується як *Чо́мскі*[11], англ. *Avram Noam Chomsky*; *7 грудня, 1928, Філадельфія, Пенсильванія) – американський лінгвіст, філософ та політичний активіст, професор мовознавства Масачусетського технологічного інституту (MTI) у відставці. Хомський добре відомий академічній та науковій спільноті як один із засновників сучасної лінгвістики та визначна постать в аналітичній філософії.

Теорія формальних мов і граматик є розділом математичної лінгвістики – специфічної математичної дисципліни, орієнтованої на вивчення структури природних і штучних мов. За характером використовуваного математичного апарата теорія формальних граматик і мов близька до теорії алгоритмів і теорії автоматів.

На інтуїтивному рівні мову можна визначити як певну множину пропозицій з заданою структурою, які мають певне значення. Правила, як і припустимі конструкції мови, складають синтаксис мови. Значення або зміст фрази, визначається семантикою мови. Якщо б усі мови складались з кінцевої кількості пропозицій, то проблеми синтаксису б не виникало. Але, оскільки мова містить необмежену (або доволі велику) кількість правильно побудованих фраз, то виникає проблема опису нескінченних мов за допомогою певних кінцевих засобів.

В термінах теорії алгоритмів, формальні граматики – це системи, що знаходяться між скінченими абстрактними автоматами та нескінченними машинами Тюрінга. Хоча спочатку вони призначалися для вирішення суто лінгвістичних проблем природних мов, з'ясувалося, що формальні граматики можна пристосувати до алгоритмічних мов, тому що вони простіші за структурою і значно легше формалізуються на відміну від природних мов.

Таким чином, формальні граматики являють собою клас алгоритмічних систем, які мають характерні відмінності.

У будь-якій мові, в тому числі і формальній, можна виділити дві складові: *синтаксис* і *семантику*.

Синтаксис (граматика мови) – це сукупність правил, згідно яких будуються конструкції, припустимимі у даній мові.

Семантика – змістовна сторона мови – співвідносить одиниці і конструкції мови з певним зовнішнім світом, для опису якого мова використовується.

Для опису формальної мови необхідна ще одна мова, за допомогою якої будуть створюватися мовні конструкції. Ця мова, за допомогою якої відбувається опис формальної мови, називається *мовою – об'єктом*, а мова, засобами якої провадиться опис – *метамовою*. Метамова повинна забезпечувати як опис структурних одиниць мови і правил об'єднання їх в припустимі речення, так і змістовну (значеннєву) сторону мовних конструкцій.

Будь-яка граматика починається з вказівки алфавіту, тобто набору символів за допомогою яких будуються конструкції мови.

Синтаксис формальної мови задається певною системою правил, які породжують систему, яка з невеличкого набору первинних конструкцій породжує всі припустимі їх комбінації, тобто мова утворюється як множина дозволених правилами сполучень початкових конструкцій.

Крім цього, синтаксис містить формулювання умов, які виконуються для закінчених конструкцій мови і не виконуються у протилежному випадку.

Крім синтаксису, встановлюється система правил, яка дозволяє конструкціям мови додавати зміст – ці правила утворюють *семантику мови.*

Формальна граматика – це система правил, яка описує множину кінцевих послідовностей символів формального алфавіту.

Кінцеві ланцюжки символів називаються реченнями формальної мови, а сама множина ланцюжків – мовою, описуваною даною граматикою.

Набір синтаксичних правил формальної мови аналогічний системі підстановок, що використовуються в нормальних алгоритмах Маркова.

У формальних граматиках, ланцюги символів інтерпретуються як мовні об'єкти різних рівнів: словоформи, словосполучення, фрази. Словоформа складається з ланцюжка морфем.

Морфема – найменша граматична частина слова (префікс, корінь, суфікс, закінчення).

Граматична мова – скінчена множина правил побудови об'єктів, їх аналізу та перетворень.

Формальна граматика задається упорядкованою четвіркою:

$$\{T, N, S, P\},$$

де T і N – непересічні кінцеві множини, які утворюють алфавіт або словник породжуваної формальної мови;

T – множина (словник) *термінальних символів*;

N – множиною (словником) *нетермінальних* (допоміжних) *символів*;

S – початковий (виділений) допоміжний символ з множини N;

P – набір правил виведення конструкцій мови (підстановок) з виділеного допоміжного символу, що має вид $g \to h$, де g і h – ланцюжки, що складаються як з термінальних, так і нетермінальних символів.

Підстановки працюють наступним чином: якщо у перетворюваному ланцюжку є слово $\boldsymbol{g}$, то воно замінюється словом $\boldsymbol{h}$. Єдине обмеження на вид підстановок полягає в тому, що слово $\boldsymbol{g}$ не може складатися тільки з термінальних символів. Це означає, що отримання на певному кроці ланцюжка, об'єкта, який складається *тільки з термінальних символів*, свідчить про припинення процесу породження – цей ланцюжок є правильною, завершеною конструкцією породжуваної мови. Підстановки $\boldsymbol{P}$ можуть застосовуватися до трансформованого ланцюжка у довільному порядку.

Приклад 8.1. Нехай формальна граматика задається наступним чином: $\boldsymbol{T = \{a, b\}}$ (тобто множина термінальних символів – алфавіт мови – складається з двох символів – $\boldsymbol{a}$ і $\boldsymbol{b}$); $\boldsymbol{N = \{S\}}$, тобто множина нетермінальних символів складається з єдиного символу $\boldsymbol{S}$ – він, природно, виявляється виділеним; система підстановок нехай має наступний вид:

$$\boldsymbol{S \to aSa,\ \ S \to bSb,\ \ S \to a\ \ S \to b}$$

Описана граматика породжує мову, що складається з усіх *„слів-перевертнів"* в алфавіті $\{a, b\}$, які мають непарну довжину, тобто слів, які зліва направо читаються так само, як справа наліво, наприклад, *aba, abababa, bbbbb, baaaaaab* тощо. Легко бачити, що застосування перших двох правил (у будь-якому числі й будь-якій послідовності) породжує ланцюжки (слова) типу $\boldsymbol{\alpha S \alpha^1}$ де $\boldsymbol{\alpha^1}$ означає слово $\boldsymbol{\alpha}$, записане справа наліво; застосування третього і четвертого правил завершує процес породження слова і формують слова типу

$$\boldsymbol{\alpha a \alpha^1 \text{ або } \alpha b \alpha^1}$$

Приклад 8.2. Розглянемо формальну граматику, яка породжує фрагмент природної мови. Нехай $\boldsymbol{T = \{а, б, \ldots я, А, Б, \ldots Я\}}$ – множина термінальних символів – букв українського алфавіту. Нетермінальний алфавіт будується з символів $N = \{Q, R, S\}$, де $Q = \{q_1 \ldots q_n\}$ – множина імен людей в українському алфавіті, $R = \{r_1 \ldots r_m\}$ – множина дієслів, що перебувають в третій особі однини дійсного часу. r_i і q_j записуються за допомогою термінальних символів.

Нехай система підстановок має вид:

$$1)\ S \to QR$$
$$2)\ Q \to q_1,\ Q \to q_2,\ \ldots . Q \to q_m$$
$$3)\ R \to r_1,\ R \to r_2,\ \ldots\ R \to r_m$$

Очевидно, ця граматика породжує мову, що складається з фраз типу: *„такий-то робить те і те”*, наприклад, *„Маша читає”, „Вася спить”* тощо. Працює граматика наступним чином: на першому кроці визначається тип фрази; другий крок породжує конкретне ім'я, а третій крок – конкретну дію (дієслово). З даного прикладу видне змістовне значення нетермінальних символів – вони можуть позначати різні класи конкретних слів, зокрема, традиційні граматичні класи – частини мови, члени речення тощо.

Хоча ми підійшли до розгляду формальних граматик у зв'язку з необхідністю побудови строгого (тобто такого, що однозначно сприймається) опису алгоритму, в дійсності області їх застосування в інформатиці набагато ширші.

На основі формальних граматик створюються мови програмування і транслятори до них. При розв’язку задач штучного інтелекту вони використовуються в системах машинного перекладу, а також для генерації синтаксично правильних речень у відповідях експертних систем на запити користувачів. Формальні граматики можуть застосовуватися в навчальних і інших програмах (наприклад, *Microsoft Word*), де потрібна перевірка правильності тексту, що вводиться в комп’ютер та пошуку у ньому помилок.

8.4. Способи опису формальних мов

Як вже було сказано, для опису мови-об'єкта повинна застосовуватися метамова. Але метамова також повинна володіти певними властивостями формальної мови, щоб однозначно визначати конструкції мови-об'єкта. Отже, метамова спочатку повинна бути описана сама, для чого також потрібна мова. Природно, може скластися враження, що такий процес ніколи не закінчиться. Однак доведено, що *для опису будь-якої метамови можна використовувати природну мову.* Таким чином, для побудови формальної мови необхідно засобами природної мови описати метамову, а потім за допомогою метамови описати мову формальну. Розглянемо два варіанта опису метамов.

8.4.1. Нотації Бекуса-Наура

Одна з широко поширених метамов відома як *нотації Бекуса-Наура*. Для формування речень у формі Бекуса-Наура використовуються універсальні метасимволи:

{ <, >, ::=, | }.

Перші два метасимволи називають *„кутовими дужками”* – вони слугують для обрамлення нетермінального символу. Символ «***::=***» читається *«за визначенням є»*; символ «|» – *«або»*. В реченнях, записаних у формі Бекуса-Наура, нетермінальний символ, який стоїть у кутових дужках, відіграє роль конструкції мови-об'єкта, яка визначається.

В формах Бекуса-Наура можуть використовуватися термінальні символи з алфавіту мови-об'єкта, відмінні від універсальних метасимволів. Термінальні символи формальної мови нічим не обмежуються.

Опис формальної мови будується з послідовності форм, кожна з яких у лівій частині містить один метасимвол, який позначає певну конструкцію мови-об'єкта. Права частина такої формули містить або перелік метасимволів і термінальних символів мови-об'єкта (ніяких роздільників при цьому не ставиться) або сукупності переліків, розділених символом « **|**». Права і ліва частини об'єднуються в єдину формулу знаком «***::=***».

Мову-об'єкт можна вважати повністю визначеною у формі Бекуса-Наура, якщо будь-який нетермінальний символ можна представити послідовністю термінальних символів.

Як приклад можна розглянути визначення поняття *«ідентифікатор»*, яке використовується у багатьох мовах програмування. На природній мові визначення звучить наступним чином: ***„Ідентифікатор – це будь-яка послідовність букв і цифр, яка починається з букви”***. В формі Бекуса-Наура воно буде виглядати наступним чином:

<ідентифікатор>::=<буква>|<ідентифікатор><буква>|<ідентифікатор> <цифра>
<буква>::= a|b|c|d|e...<цифра>::= 0|1|2|3|...|9

Видно, що у визначенні цього поняття присутня рекурсивність, оскільки поняття *„ідентифікатор”* визначається само через себе. Елементарним є ідентифікатор, який складається з однієї букви.

Перевага нотацій Бекуса-Наура полягає у тому, що вони подаються у буквеному вигляді; незручні нотації одноманітністю способів побудови речень мови-об'єкта – запис, виявляється громіздким і погано сприйманим.

Найбільш наочним потрібно вважати інший спосіб опису формальної мови, який був запропонований Нікласом Віртом – творцем мови програмування *PASCAL* – і який отримав назву *«синтаксичні діаграми»*.

8.4.2. Синтаксична діаграма

Синтаксична діаграма **– *це схема (графічне представлення) опису якогось нетермінального символу мови-об'єкта.*** Схема завжди має один вхід і один вихід. Елементами схеми можуть бути термінальні символи мови-об'єкту, поміщені в окружність (або овал) або нетермінальні символи (поняття) мови-об'єкта, поміщені у прямокутник. Елементи поєднуються між собою спрямованими лініями, які вказують порядок проходження об'єктів в обумовленому нетермінальному символі. Прийняті наступні позначення (рис.8.15).

Структура синтаксичних діаграм ідентична структурам мов програмування, а це дозволило широко використовувати діаграми для написання трансляторів різних мов. Першою мовою, описаною за допомогою синтаксичних діаграм, була мова *PASCAL*.

Читання діаграми провадиться за напрямком стрілок; у точці розгалуження може обиратися певний маршрут. В якості метамови може використовуватися природна мова; мови програмування будуються на англомовній основі. Термінальні символи переписуються в конструкції формальної мови дослівно.

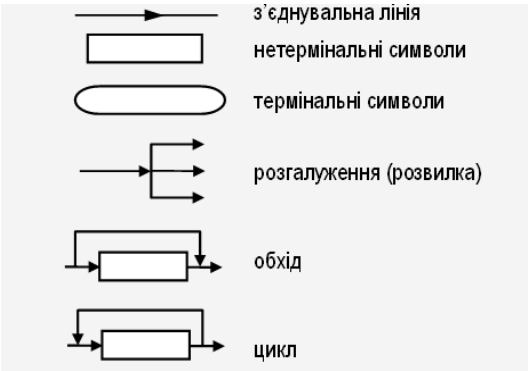


Рис. 8.15. Позначення синтаксичної діаграми

Нетермінальні символи можуть виражатися через термінальні або інші нетермінальні – у цьому випадку для них будуються уточнюючі діаграми; в кінцевому підсумку, всі нетермінальні символи повинні бути виражені через термінальні. При використанні синтаксичних діаграм приймається умова, що серед термінальних символів мови-об'єкта не повинно бути однакових, а також жодний з термінальних символів не може слугувати початком іншого. При порушенні даної умови, можливе неоднозначне читання діаграми і, як наслідок, побудова або розпізнавання невірної конструкції мови.

Розглянемо ряд прикладів побудови синтаксичних діаграм, першим з яких буде визначення поняття *«ідентифікатор»* для порівняння з приведеною вище нотацією Бекуса-Наура (рис.8.16).

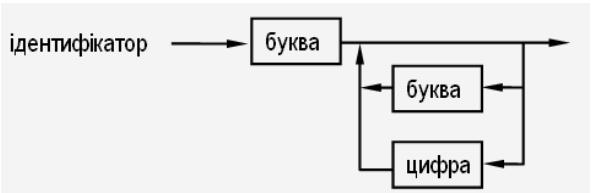


Рис. 8.16. Визначення поняття ідентифікатор за допомогою синтаксичної діаграми

Для визначення поняття „ідентифікатор” за допомогою синтаксичної діаграми необхідні уточнюючі діаграми (рис.8.17):

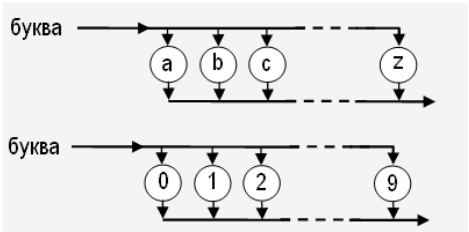


Рис. 8.17. Уточнюючі діаграми

Прикладами побудови англомовних ідентифікаторів у відповідності з цією діаграмою є: *q, a123, identificator, e2e4*.

Діаграма, яка задає загальний вигляд програми на мові *PASCAL*, виглядає наступним чином (рис.8.18):

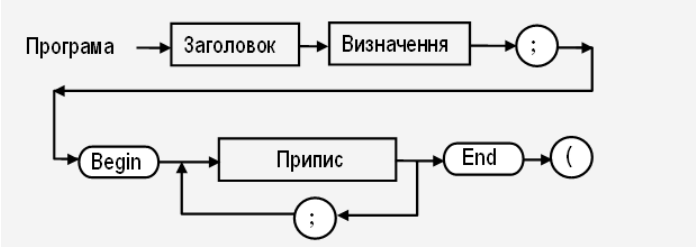


Рис. 8.18. Загальний вигляд програми на мові PASCAL в термінах синтаксичної діаграми

У відповідності з цими і подібними діаграмами будуються припустимі синтаксичні конструкції мови.

Таким чином, нотації Бекуса-Наура і синтаксичні діаграми – це два альтернативних способи опису конструкцій метамови, за допомогою яких будується формальна мова. Після того, як побудована формальна граматика й породжена нею мова, вона може бути використана для вирішення прикладних задач – комунікації, збереження і обробки інформації.

Як приклад приписів розглянемо умовний і циклічний (рис.8.19).

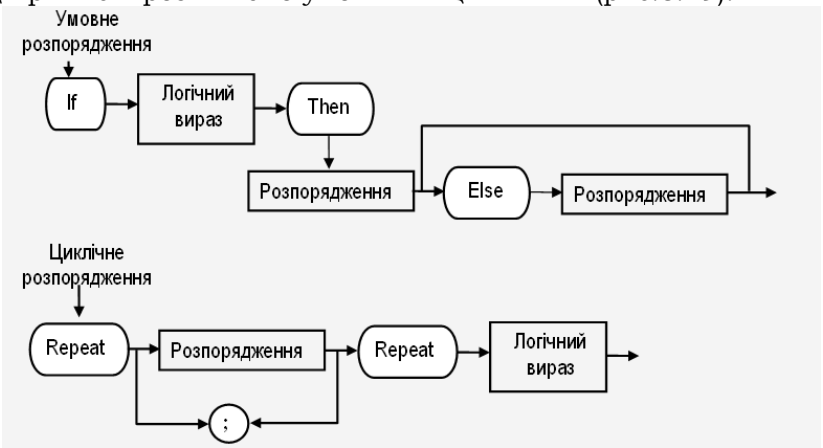


Рис. 8.19. Схема умовного і циклічного приписів

Останній клас задач призводить до необхідності формулювання за допомогою мов послідовностей обробки інформації, тобто алгоритмів і їх подання у формі, доступній для розуміння і виконання особою або технічним пристроєм, які здійснюють обробку.

8.5. Класифікація формальних граматик

Розрізняють розпізнавальні, породжувальні, перетворювальні, контекстно-вільні, та контекстно-залежні формальні граматики.

Розпізнавальна граматика – якщо для будь-якого ланцюжка символів вона може вирішити, чи є він правильним чи ні, і якщо так, то як він буде побудований.

Породжувальна граматика – якщо вона може побудувати певний правильний ланцюжок символів, даючи вказівки щодо його побудови і не будує жодного неправильного ланцюжка.

Перетворювальна граматика – якщо для будь-якого правильно побудованого ланцюжка вона вміє побудувати відображення у вигляді правильного ланцюжка, задаючи вказівки щодо порядку проведення відображень.

Контекстно-вільні граматики: правила підстановок застосовуються до ланцюжків символів (до окремих сполучень символів, до окремих символів) незалежно від тих символів, які їх оточують (від контексту) і при виконанні підстановок цей контекст залишається без змін.

Контекстно-залежні граматики: до ланцюжків символів застосовуються різні правила підстановок залежно від їх контексту. Наприклад, при перекладі текстів з однієї мови спілкування на іншу, використовуються правила контекстно-залежних граматик, бо одному й тому самому слову у різних контекстах будуть ставитись у відповідність різні слова іншої мови. Водночас мови програмування побудовано за правилами контекстно-вільних граматик, і кожна конструкція такої мови завжди має один і той самий зміст, значення і тлумачення, тому при перекладі (трансляції) на будь-яку іншу мову програмування та мову комп'ютерних команд (EXE-модулі) у першу чергу за правилами перетворювальних граматик можна отримати тільки один варіант вихідного тексту, з одним ланцюжком символів вхідного тексту, незалежно від контексту, зіставити один і той самий вихідний ланцюжок.

Дві граматики називають слабо еквівалентними, якщо вони породжують одну й ту саму мову, тобто одну й ту саму множину термінальних ланцюжків.

Дві граматики називають сильно еквівалентними, якщо вони породжують одну й ту саму мову та однаковим ланцюжком приписують однакові виводи (однакові алгоритми).

Формальні граматики є складовою частиною науки, що відокремилась від теорії алгоритмів, – математичної лінгвістики, яка вивчає природні мови, їх розвиток та ужиток за допомогою законів математики, а також є основою створення мов програмування.

8.6. Основні поняття породжуючих граматик

Алфавіт – це непуста кінцева множина. Елементи алфавіту називаються символами. Ланцюжок над алфавітом $\Sigma=\{a_1,a_2,\ldots,a_n\}$ є кінцева послідовність елементів a_i.

Ланцюжком символів (або рядком) називають довільну послідовність символів, записаних один за одним.

Ланцюжок – це послідовність, до якої можуть входити будь-які припустимі символи.

Множина усіх ланцюжків над алфавітом Σ позначається Σ*.

Довжина ланцюжка x дорівнює числу її елементів і позначається $|x|$. Ланцюжок нульової довжини називається пустим і позначається символом ε. Відповідно, непустий ланцюжок визначається як ланцюжок ненульової довжини. Нехай є алфавіт $\Sigma=\{a, b\}$. Тоді множина усіх ланцюжків визначається наступним чином: $\Sigma^*=\{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, \ldots\}$.

Ланцюжки x і y є однаковими, якщо вони однакової довжини і збігаються з точністю до порядку символів, з яких вони складаються.

Над ланцюжками x і y визначена операція зчеплення (конкатенації, добутку), яка утворюється дописуванням символів ланцюжка y за символами ланцюжка x.

Мова L над алфавітом Σ представляє собою множину ланцюжків над Σ. Необхідно розрізняти пусту мову $L=0$ і мову, що містить тільки пустий ланцюжок: $L=\{\varepsilon\}\neq 0$.

Формальна мова L над алфавітом Σ – це мова, виділена за допомогою кінцевої множини певних формальних правил.

Нехай M і L – мови над алфавітами. Тоді конкатенація $LM=\{xy \mid x\in L, y\in M\}$. Зокрема, $\{\varepsilon\}L = L\{\varepsilon\}=L$. Використовуючи поняття добутку, визначимо ітерацію L^* і усічену ітерацію L^+ множини L:

$$L^+ = \bigcup_{i=1}^{\infty} L^i,$$

$$L^* = \bigcup_{i=1}^{\infty} L^i,$$

де i – ступінь мови, L визначається рекурсивно наступним чином:

$L^0 = \{\varepsilon\};$

$L^1 = L;$

$L^{n+1} = L^n L = L L^n;$

$\{\varepsilon\}L=L\{\varepsilon\}=L.$

Наприклад, якщо задана мова $L=\{a\}$, тоді $L^*=\{\varepsilon, a, aa, aaa, \ldots\}$, $L^+=\{a, aa, aaa, \ldots\}$.

Породжуюча граматика – це упорядкована четвірка $G= (V_T, V_N, P, S)$, де V_T – кінцевий алфавіт, який визначає множину термінальних символів; V_N – кінцевий алфавіт, який визначає множину нетермінальних символів; P – кінцева множина правил виведення, тобто множину пар такого виду:

$$u \to v, \qquad де\ u,v\in(V_T \cup V_N)^*;$$

S – начальний символ (аксіома граматики), $S \in V_N$.

З термінальних символів складаються ланцюжки мови, породжені граматикою. Аксіомою називається символ у лівій частині першого правила виводу граматики.

Для того щоб розрізняти термінальні і нетермінальні символи, прийнято позначати термінальні символи рядковими, а нетермінальні символи заголовними буквами латинського алфавіту.

В граматиці G ланцюжок x безпосередньо породжує ланцюжок y, якщо $x = a\alpha\beta$, $y = a\upsilon\beta$ і $u\to\upsilon \in P$, тобто ланцюжок y безпосередньо виводиться з x, що позначається $x \Rightarrow y$. Мовою, породженою граматикою $G = (V_T,V_N,P,S)$, називається множина термінальних ланцюжків, що виводяться в граматиці G з аксіоми:

$$L(G) = \{x \mid x \in V_T^*; S \Rightarrow^* x\},$$

де $\Rightarrow^*$ – виводимість.

Приклад. Дана граматика $G = (V_T,V_N,P,S)$, у якій $V_T = \{a, b\}$, $V_N = \{S\}$, $P = \{S \to aSb, S \to ab\}$. Визначити мова, яка породжується цією граматикою.

Розв'язання. Використовуючи рекурсію, виведемо декілька ланцюжків мови, яка породжується цією граматикою:

$$S \to ab;$$

$$S \to aSb \Rightarrow aabb;$$
$$S \to aSb \Rightarrow aaSbb \Rightarrow aaabbb; \ldots$$

Визначимо мову, яка породжується даною граматикою:

$$L(G) = \{a^n b^n \mid n > 0\}.$$

Кажучи про подання граматик, потрібно відзначити, що множина правил виводу граматики може наводитися без спеціальної вказівки множини терміналів і нетерміналів. У такому випадку зазвичай передбачається, що граматика містить в точності ті термінальні і нетермінальні символи, які зустрічаються в правилах виводу.

Також передбачається, що праві частини правил, для яких збігаються ліві частини, можна записати в один рядок з використанням роздільника. Так, правила виводу з наведеного прикладу можна записати наступним чином:

$$S \to aSb \mid ab.$$

Питання для самоконтролю

1. *З яких компонентів складається алгоритмічна обчислювальна система?*
2. *Які типи припустимих операцій притаманні будь-якій алгоритмічній системі?*
3. *Що розуміють під абстрактним алфавітом?*
4. *Що розуміють в алгоритмізації під терміном „синтаксис”?*
5. *Що розуміють в алгоритмізації під терміном „семантика”?*
6. *Що розуміють в алгоритмізації під терміном „формальна граматика”?*
7. *Що розуміють в алгоритмізації під терміном „морфема”?*
8. *Що розуміють в алгоритмізації під терміном „граматична мова”?*
9. *На основі чого створюються транслятори і мови програмування?*
10. *Дайте опис формальних мов за допомогою нотації Бекуса-Наура.*
11. *Дайте опис формальних мов за допомогою синтаксичної діаграми.*

9. НОРМАЛЬНІ АЛГОРИТМИ МАРКОВА

9.1. Загальні визначення

Нормальний алгоритм Маркова (НАМ) – один зі стандартних способів формального визначення поняття алгоритму, аналогічній машині Тюрінга, але дані в ньому опрацьовуються на основі інших принципів.

Поняття нормального алгоритму введено А.А. Марковим наприкінці 1940-х років. Традиційно, ведучи мову про алгоритми Маркова, використовують слово „алгорифм".

Нормальний алгоритм Маркова (НАМ) – математична побудова, призначена для уточнення поняття „алгоритм".

В алгоритмічній схемі Маркова відсутнє поняття стрічки і здійснюється безпосередній доступ до різних частин перетворюваного слова. Марков назвав цю алгоритмічну схему нормальним алгоритмом. Поняття нормального алгоритму, який має багато переваг як принципового, так і методичного характеру, виявилось плідним і зручним. Витримавши іспит часом і довівши свою життєздатність, нормальні алгоритми міцно ввійшли в науковий вжиток сучасної теорії алгоритмів.

Нормальний алгоритм описує метод переписування рядків (ланцюжків), схожий за способом завдання на формальні граматики. НАМ є Тюрінг-повною мовою, що робить її за виразністю еквівалентною машині Тюрінга і отже сучасною мовою програмування. На основі НАМ була створена функціональна мова програмування Рефал.

9.2. Сутність підходу Маркова

Нормальний алгоритм Маркова задається алфавітом *A* і нормальною схемою підстановок.

Нормальна схема підстановок – послідовність правил „що на що змінювати. Найкраще всього це показати на прикладі заміни слів, тим більше, що й сам Марков будь-яку послідовність букв, яку в жодному словнику не відшукаєш, називав „словами". Так при наявності двох підстановок: що змінюють „*ха*" на „*ссон*" і „*мусс*" на „*сл*" з „*муха*" можна зробити „*слон*".

Механізм нормальних алгоритмів настільки простий, що нагадує більше дитячу гру, чим математику. Але в дійсності, це дуже потужний механізм, оскільки завдячуючи йому можна виразити розв'язання будь-якої алгоритмічно розв'язної задачі. Відзначимо, що це не треба сприймати, як пропозицію розв'язувати будь-яку задачу через підстановки (хоча на цих принципах працює чудова мова програмування РЕФАЛ). Це лише означає, що будь-яку алгоритмічно розв'язну задачу можна представити у вигляді такої системи підстановок. А якщо не можна (і ви це змогли довести), то така задача взагалі не має алгоритму вирішення.

Алфавіт – кінцева, непуста множина елементів, які називаються буквами. Різні комбінації букв утворюють слова.

Нормальна схема підстановок – це кінцевий набір, що складається з пар слів, де ліве слово переходить в праве (але не навпаки).

Нормальним алгоритмом в алфавіті *A* називається наступний алгоритм побудови послідовності слів: в якості начального слова береться саме слово *P*, і до нього застосовують по порядку кожну пару зі схеми підстановки. Якщо підстановка можлива, то її здійснюють й починають підстановки спочатку. Якщо процес обривається (немає жодної припустимої підстановки) на слові *Q* або приходить в кінцеву підстановку, то даний нормальний алгоритм перетворив *P* в *Q*.

Якщо існує задача: від *P* перейти до *Q* і доведено, що не можна побудувати нормальну схему, то має місце алгоритмічно нерозв'язна задача.

Клас нормальних алгоритмів Маркова і клас алгоритмів, представлених у формі машин Тюрінга і Поста, збігаються.

Для формалізації поняття алгоритму радянський математик А.А. Марков (рис.9.1) запропонував використовувати асоціативні числення.

Рис.9.1. Андрій Андрійович Марков (1903, Санкт-Петербург – 1979, Москва)

Марков Андрій Андрійович (молодший) – син видатного математика, спеціаліста з теорії ймовірності, також Маркова Андрія Андрійовича (старшого). У 1924 р. закінчив відділення фізики фізико-математичного факультету Ленінградського університету, у 1928 р. вступив до аспірантури Астрономічного інституту (Ленінград).

Ступінь доктора фізико-математичних наук була присвоєна йому без захисту дисертації у 1935 р., звання професора – у 1936 р. У 1936–1953 рр. (з перервою у 1942–1944 рр.) завідував кафедрою геометрії Ленінградського університету, в 1953–1955 рр. займав посаду професора цієї ж кафедри. Паралельно (з 1940 по 1955 рр.) працював в Ленінградському відділенні Математичного інституту ім. В.А. Стеклова АН СССР (ЛОМІ), протягом десяти років з цього періоду – на посаді заступника директора. У 1955 р. був переведений до Москви, працював в Математичному інституті ім. В.А.Стеклова (МІАН), пізніше в Обчислювальному центрі АН СРСР.

З 1959 р. і до кінця життя завідувач кафедри математичної логіки мехмата Московського державного університету. У 1953 р. був обраний членом-кореспондентом АН СРСР. Працював в багатьох галузях (теорія пластичності, прикладна геофізика, небесна механіка, топологія тощо), але найбільший вклад вніс в математичну логіку (зокрема, заснував конструктивний напрямок в математиці), теорію складності алгоритмів і кібернетику. Заснував математичну школу, його учні працюють зараз в багатьох країнах.

Під асоціативним численням розуміють сукупність усіх слів разом з системою підстановок. Введемо низку визначень:

Алфавіт – кінцева, непуста множина елементів, які називаються буквами.

Слово – це будь-яка кінцева послідовність знаків алфавіту.

Довжина слова – число символів у слові.

Пусте слово – слово, довжина якого дорівнює нулю.

Слово s називається підсловом q, якщо q можна представити у вигляді q=rst, де r і t – будь-які слова в тому ж алфавіті (в тому числі й пусті (порожні)).

Підстановкою називається заміна першого по порядку підслова P_r первинного слова P на слово P_k. Позначається підстановка $P_r \rightarrow P_k$.

Нормальна схема підстановок – це кінцевий набір, що складається із пар слів, де ліве слово переходить в праве (але не навпаки).

Елементарні підстановки – це алфавітні оператори, послідовність яких може реалізувати будь-які алгоритми перетворення інформації.

Елементарні розпізнавачі перевіряють наявність тих чи інших властивостей перетворюваної алгоритмами інформації, а результат розпізнавання визначає послідовність виконання елементарних підстановок.

Алгоритмом в алфавіті A називається ефективно обчислювана функція, областю визначення якої слугує певна підмножина множини усіх слів в алфавіті A і значеннями якої також є слова в алфавіті A.

Алгоритм у даній формі подання задається *системою підстановок*, яка представляє собою послідовність (список) підстановок. Якщо в цьому списку є підстановки з лівими частинами, які входять в *P*, то перша з них застосовується до *P*, в результаті чого воно переходить в інше слово P_1. До нього знову застосовується схема підстановок тощо. Процес припиняється у двох випадках: або у списку не знайшлося підстановки з лівою частиною, що входить в P_n, або при отриманні P_n була застосована остання підстановка.

Розглянемо деякі поняття асоціативного числення. Нехай є алфавіт (кінцевий набір різних символів). Символи, що його складають будемо називати буквами. Будь-яка кінцева послідовність букв алфавіту (лінійний їх ряд) називається словом в цьому алфавіті.

Розглянемо два слова *N* і *MA*. Якщо *N* є частиною *M*, то кажуть, що *N* входить в *M*.

Задамо у певному алфавіті кінцеву систему підстановок:

$$N - M,\ S - T, \ldots,$$

де *N*, *M*, *S*, *T*, – слова в цьому алфавіті.

Будь-яку підстановку N – M можна застосувати до певного слова *K* наступним чином: якщо в *K* є одне або декілька входжень слова *N*, то будь-яке з них може бути замінене словом *M*, і, навпаки, якщо є входження *M*, то його можна замінити словом *N*.

Наприклад, в алфавіті *A = {a,b,c}* є слова *N=ab*, *M=bcb*, *K=abcbcbab*. Замінивши в слові *K* слово *N* на *M*, отримаємо *bcbcbcbab* або *abcbcbbcb*, і, навпаки, замінивши *M* на *N*, отримаємо *aabcbab* або *abcabab*.

Підстановка *ab – bcb* неприпустима до слова *bacb*, оскільки ні *ab*, ні *bcb* не входить у це слово. До отриманих за допомогою припустимих підстановок словам можна знову застосувати припустимі підстановки тощо. Сукупність усіх слів в даному алфавіті разом з системою припустимих підстановок називають асоціативним численням. Щоб задати асоціативне числення, достатньо задати алфавіт і систему підстановок.

Слова *P1* і *P2* в певному асоціативному численні називаються суміжними, якщо одно з них може бути перетворене в інше однократним застосуванням припустимої підстановки. Послідовність, слів *P, P1, ...,M* називається дедуктивним ланцюжком, який веде від слова *P* до слова *M*, якщо кожне з двох слів цього ланцюжка, що стоять поруч – суміжне.

Слова *P* і *M* називають еквівалентними, якщо існує ланцюжок від *P* до *M* і навпаки.

Приклад 9.1.
Алфавіт:
{*a, b, c, d, e*} .
Підстановки:
ac – ca;
abac – abace;
ad – da;
eca – ae;
be – cb;
eda – be;
bd – db;
edb – be.
Слова *abcde* і *acbde* – суміжні (підстановка *be – cb*). Слова *abcde – cadbe* еквівалентні.

Може бути розглянутий спеціальний вид асоціативного числення, у якому підстановки є орієнтованими: N→M (стрілка означає, що підстановку дозволяється робити лише зліва направо). Для кожного асоціативного числення існує задача: для будь-яких двох слів визначити, чия є вони еквівалентними чи ні.

Будь-який процес виведення формул, математичні викладки і перетворення також є дедуктивними ланцюжками в певному асоціативному численні. Побудова асоціативних числень є універсальним методом детермінованої переробки інформації і дозволяє формалізувати поняття алгоритму.

Введемо поняття алгоритму на підставі асоціативного числення.

Алгоритмом в алфавіті *A* називається зрозумілий точний припис, який визначає процес над словами з алфавіту *A* і який припускає будь-яке слово як вихідне. Алгоритм в алфавіті *A* задається у вигляді системи припустимих підстановок, доповнений точним приписом про те, в якому порядку потрібно застосовувати припустимі підстановки і коли відбувається зупинка.

Приклад 9.2.
Алфавіт:
{a, b, c}.
Система підстановок *B*:
cb – ee;
cca – ab;
ab – bca.
Припис про застосування підстановок: в довільному слові *P* потрібно зробити можливі підстановки, замінивши ліву частину підстановок на праву; повторити процес зі знову отриманим словом.

Застосовуючи таким чином систему підстановок *B* з розглянутого прикладу до слів *babaac* і *bcacabc*, отримаємо:

babaac → bbcaaac → зупинка

bcacabc → bcacbcac → bcacccac → bcacabc → нескінченний процес (зупинки немає), оскільки ми отримали вихідне слово.

Нормальний алгоритм Маркова можна розглядати як універсальну форму завдання будь-якого алгоритму. Універсальність нормальних алгоритмів декларується принципом нормалізації: для будь-якого алгоритму в довільному кінцевому алфавіті *A* можна побудувати еквівалентний йому нормальний алгоритм над алфавітом А.

Роз'яснимо останнє твердження. В деяких випадках не вдається побудувати нормальний алгоритм, еквівалентний даному в алфавіті *A*, якщо використовувати в підстановках алгоритму тільки букви цього алфавіту. Однак можна побудувати необхідний нормальний алгоритм, здійснюючи розширення алфавіту *A* (додаючи до нього певну кількість нових букв). В цьому випадку кажуть, що побудований алгоритм є алгоритмом над алфавітом *A*, хоча він буде застосовуватись тільки до слів у вихідному алфавіті *A*.

Якщо алгоритм *N* заданий у якомусь розширенні алфавіту *A*, то кажуть, що *N* – нормальний алгоритм над алфавітом *A*.

Умовимося називати той або інший алгоритм таким, що нормалізується, якщо можна побудувати еквівалентний йому нормальний алгоритм, і таким, що не нормалізується у протилежному випадку. Принцип нормалізації тепер може бути висловлений у видозміненій формі: всі алгоритми можна нормалізувати.

Даний принцип не може бути строго доведений, оскільки поняття довільного алгоритму не є строго визначеним і ґрунтується на тому, що всі відомі сьогодні алгоритми можна нормалізувати, а способи композиції алгоритмів, які дозволяють будувати нові алгоритми з уже відомих, не виводять за межі класу алгоритмів, яких можна нормалізувати. Нижче перераховані способи композиції нормальних алгоритмів.

I. *Суперпозиція*[1] *алгоритмів*. При суперпозиції двох алгоритмів слово першого алгоритму розглядається як вхідне слово другого алгоритму *B*, результат суперпозиції С може бути представлений у вигляді *C(p)* і *B(A(p))*.

II. *Об'єднання алгоритмів*. Об'єднанням алгоритмів *A* і *B* в одному й тому ж алфавіті називається алгоритм *C* в тому ж алфавіті, яке перетворює будь-яке слово *p*, що міститься в перетинанні областей визначення алгоритмів *A* і *B*, в записані поруч слова *A(p)* і *B(p)*.

III. *Розгалуження алгоритмів*. Розгалуження алгоритмів представляє собою композицію *D* трьох алгоритмів *A*, *B* і *C*, причому область визначення алгоритму *D* є перетинанням областей визначення усіх трьох алгоритмів *A*, *B* і *C*, а для будь-якого слова *p* з цього перетинання:

D(p) = A(p), якщо C(p) = e,

D(p) = B(p), якщо C(p) = e,

де e – пустий (порожній) рядок.

IV. *Ітерація алгоритмів*. Ітерація (повторення) представляє собою таку суперпозицію *C* двох алгоритмів *A* і *B*, що для будь-якого вхідного слова *p* відповідне слово *C(p)* утворюється в результаті послідовного многократного застосування алгоритму *A* до тих пір, поки не утвориться слово, що перетворюється алгоритмом *B*.

Нормальні алгоритми Маркова є не тільки засобом теоретичних побудов, але й основою спеціалізованої мови програмування, яка застосовується як мова символьних перетворень при розробці систем штучного інтелекту.

Існує строгий доказ того, що за можливостями перетворення нормальні алгоритми Маркова еквівалентні машинам Тюрінга.

Розглянемо приклад використання алгоритму Маркова для перетворень над рядками.

Приклад 9.3

Правила:
«А» → «апельсин»
«кг» → «кілограм»
«М» → «магазинчику»
«Т» → «том»
«магазинчику» → «ларьку» (заключна формула)
«у тому ларьку» → «на тому ринку»
Вихідний рядок:
«Я купив кг апельсинів в Т М.»
При виконанні алгоритму рядок перетерплює наступні зміни:
«Я купив кг апельсинів в Т М.»
«Я купив кілограм апельсинів в Т М.»
«Я купив кілограм апельсинів в Т магазинчику.»
«Я купив кілограм апельсинів в тому магазинчику.»
«Я купив кілограм апельсинів в тому ларьку.»

На цьому виконання алгоритму завершиться (оскільки буде досягнута формула, яку ми зробили завершальною).

Приклад 9.4.

Даний алгоритм перетворить двійкові числа в «одиничні», тобто на виході утворюється рядок з *N* одиничок, якщо на вході у нас було *N* у двійковій системі. Наприклад, 101 перетвориться в 5 одиниць:

Правила:
«|0» → "0||"
«1» → "0|"
«0» → ""
Вихідний рядок:
«101»
Виконання:
«0|01»
«00||1»
"00||0|"
"00|0|||"
"000|||||"
"00|||||"
"0|||||"
"|||||"

[1] Суперпозиція – це застосування однієї функції до результату іншої.

Приклад 9.5. Нехай заданий алфавіт ***A ={ *,1}*** і єдина підстановка: ****1→1***; Знайти результат обробки, якщо вихідним є слово ***P=11*111*1***

Застосування нормального алгоритму з зазначеною підстановкою до даного слова дає послідовність (підкресленням виділяється перетворювана комбінація:

11*1111*1→11111*1 → 111111,

тобто алгоритм знаходить кількість одиниць в первинному слові (підсумовує числа в унарній системі числення).

Приклад 9.6. Алфавіт містить символи російської мови: ***A ={а,б...я}***. Найти систему підстановок, що забезпечують перетворення: ***путь → муть, поло → мала***. Найти результат застосування такого алгоритму до первинних слів: *папа, пузо*.

Система підстановок достатньо очевидна: ***п → м, о → а.***

Застосування алгоритму: ***папа → мапа → мама пузо → музо → муза***

Приклад 9.7 Скласти нормальний алгоритм, що забезпечує виконання операції додавання в трійковій системі числення.

Алфавіт буде включати символи: ***A ={0,1,2,+}; 0+1→1, 1+1→2, 2+1→+10, +1→1***

Алгоритм може бути застосований для різних вихідних слів:

112+1→11+10→120

22+1→2+10 →+100 →100

Різні нормальні алгоритми відрізняються один від одного алфавітами і системами припустимих підстановок. Нормальний алгоритм Маркова можна розглядати як стандартну форму для завдання будь-якого алгоритму. Дана форма представлення алгоритму важлива не тільки з точки зору проведення досліджень в теорії алгоритмів, але й вона послужила основою спеціалізованої мови символьних перетворень в системах штучного інтелекту.

А чи можна будь-який алгоритм представити у вигляді нормального алгоритму Маркова? На це питання дається відповідь у вигляді так називаної тези Маркова: ***будь-який алгоритм в алфавіті А можна представити у вигляді нормального алгоритму в цьому ж алфавіті***. Це теза тому, що його неможливо довести, тому що в ній фігурують з однієї сторони, інтуїтивно розпливчасте поняття „будь-який алгоритм". А з іншої сторони – точне поняття „нормальний алгоритм".

9.3. Порівняння алгоритмічних схем

Клас нормальних алгоритмів Маркова і клас алгоритмів представлених у формі машин Поста і машин Тюрінга співпадають.

Порівнюючи алгоритмічні схеми Тюрінга і Маркова, можна відмітити, що у машини Тюрінга порівняно розвинуте управління при слабких можливостях перетворення інформації, тоді як в алгоритмічній схемі Маркова значно ширші можливості перетворення інформації при менш розвинутому управлінні.

Основні особливості підходу Маркова:

- не використовується абстракція актуальної нескінченності;
- в якості об'єктів дослідження допускаються тільки конструктивні об'єкти визначеного стандартного типу; такими об'єктами можуть бути натуральні числа, які розглядуються як логічно первинні поняття (тобто такі, що не визначаються), але зазвичай використовуються слова в певних алфавітах. Використання слів в якості об'єктів теорій, дозволяє викладати ці теорії мовою, наближеною до мови класичної математики, на відміну від використання натуральних чисел в якості кодів досліджуваних об'єктів із залученням поняття нумерації об'єктів;
- використовується поняття алгоритму в уточненому змісті, або інакше кажучи, викладання питань, пов'язаних з обчислюваністю, засновано на теорії, об'єкти розгляду якої нормальні алгоритми Маркова.

Автори обох алгоритмічних схем намагались простими засобами забезпечити можливість опису будь-яких алгоритмів. Тюрінг досягнув цієї мети в першу чергу за рахунок спрощення дії, а Марков – за рахунок спрощення логіки управління.

Питання для самоконтролю

1. *Що розуміють під нормальним алгоритмом Маркова?*
2. *Що розуміють під терміном „елементарні підстановки"?*
3. *Що розуміють під терміном „елементарні розпізнавачі"?*
4. *В чому полягає сутність підходу Маркова?*

10. АБСТРАКТНІ АВТОМАТИ

Виникнення й розвиток теорії автоматів пов'язані зі створенням технічних засобів автоматичного управління, проектуванням складних дискретних обчислювальних систем з програмним управлінням, розробкою математичних моделей процесів переробки інформації в складних динамічних системах тощо.

Як цілісна конструктивна структурна теорія автоматів формується з поч. 50-х рр. 20 ст.

Теорія автоматів найтісніше пов'язана з теорією алгоритмів. Це пояснюється тим, що автомат перетворює дискретну інформацію по кроках в дискретні моменти часу і формує по кроках результуючу інформацію заданого алгоритму. Ці перетворення можливі за допомогою технічних та/або програмних засобів.

10.1. Загальні поняття про автомати

Автомат – пристрій, який без безпосередньої участі людини виконує обробку інформації за закладеною у ньому програмі.

У 40-50 роках XX ст. багато дослідників займалось вивченням елементарних (найпростіших) машин, які сьогодні називають „*автомати*". Такі автомати спочатку були запропоновані, як моделі функціонування мозку людини. Однак з часом вони виявилися досить корисними для багатьох галузей науки, зокрема для обчислювальної техніки.

Автомат – це алгоритм, який визначає певну множину і, перетворює її в іншу множину. Неформальний опис автоматів виглядає наступним чином: автомат має вхідну стрічку, керуючий пристрій з кінцевою пам'яттю для збереження номера стану, а також може мати допоміжну (робочу) і вихідну стрічки.

Автомат можна уявити у вигляді пристрою (чорна скринька), на вхід якого подаються вхідні сигнали і з виходу знімаються вихідні, який до того ж може мати певні внутрішні стани. На вхід цього пристрою (рис.10.1) надходять символи одного алфавіту, на виході (в загальному випадку) він видає символи іншого алфавіту.

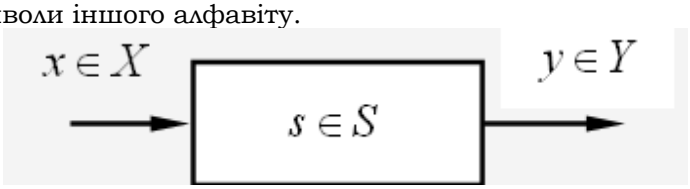


Рис.10.1. Абстрактний автомат

Символ – будь-який атомарний блок даних, який може здійснювати ефект на машину. Найчастіше усього символ – це буква звичайної природної мови, але може бути й будь-який графічний елемент діаграми.

Слово – рядок символів, який створюється через конкатенацію (з'єднання).

Алфавіт – кінцевий набір різних символів (множина символів).

Мова – множина слів, яка формується символами даного алфавіту. Може бути кінцевою або нескінченною.

Автомат – послідовність (кортеж) з п'яти елементів:

$$A=(S,X,Y,\delta,\lambda),$$

де S – кінцева множина станів автомата, X, Y – кінцеві вхідний і вихідний алфавіти відповідно, з яких формуються рядки, які зчитуються і видаються автоматом, $\delta: S\times X\to S$ – функція переходів, $\lambda: S\times X\to Y$ – функція виходів.

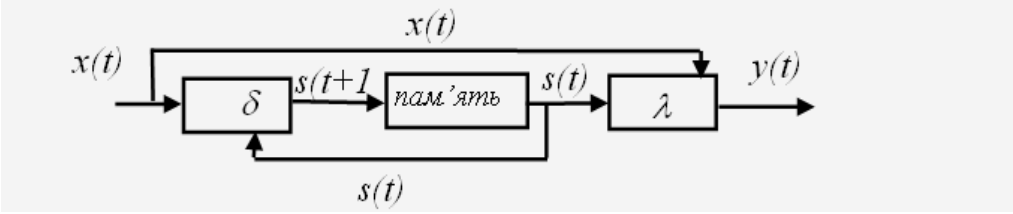


Рис.10.2. Функціональна схема абстрактного автомата

Абстрактний автомат – математична абстракція, модель дискретного пристрою, що має один вхід, один вихід і який в кожний момент часу перебуває в одному зі станів з множини можливих.

Абстрактний автомат – це алгоритмічна система, яку визначають три множини: вхідний алфавіт X, вихідний алфавіт Y, множина внутрішніх станів автомата S, а також дві функції (функції переходів та функції виходів).

Абстрактні автомати утворюють фундаментальний клас дискретних моделей як самостійна модель, і як основна компонента машин Тюрінга, автоматів з магазинною пам'яттю, кінцевих автоматів та інших перетворювачів інформації.

Модель абстрактного автомата широко використовується, як базова, для побудови дискретних моделей, які розпізнають, породжують і перетворюють послідовності символів.

Для уточнення властивостей абстрактних автоматів введена їх класифікація.

Абстрактний автомат з виділеним початковим станом називається <u>ініціальним автоматом</u>. Таким чином, абстрактний автомат визначає сімейство ініціальних автоматів $(s_i, A), s_i \in S$.

Якщо функції переходів і виходів однозначно визначені для кожної пари $(s,x) \in S \times X$***, то автомат називають детермінованим***. У протилежному випадку автомат називають *недетермінованим* або *частково визначеним.*

Якщо функція переходів і/або функція виходів є випадковими, то автомат називають <u>імовірнісним</u>.

Обмеження кількості параметрів абстрактного автомата визначило таке поняття як кінцевий автомат. ***Кінцевий автомат – математична модель пристрою з кінцевою пам'яттю, який перетворює дискретну інформацію***.

<u>Кінцевий автомат</u> – <u>математична</u> <u>абстракція</u>, яка дозволяє описувати шляхи зміни <u>станів</u> об'єкта залежно від його поточного стану і вхідних даних, при умові що загальна можлива кількість станів кінцева. Кінцевий автомат є частковим випадком <u>абстрактного автомата</u>.

Функціонування автомата полягає у породженні двох послідовностей: послідовності чергових станів автомата $s_1[1]s_2[2]s_3[3]$... і послідовності вихідних символів $y_1[1]y_2[2]y_3[3]$..., які для послідовності символів $x_1[1]x_2[2]x_3[3]$... розгортаються в моменти дискретного часу $t = 1, 2, 3, \ldots$ Моменти дискретного часу отримали назву *тактів.*

Функціонування автомата в дискретні моменти часу t може бути описано системою рекурентних співвідношень:

$$s(t+1) = \delta(s(t), x(t)); \quad y(t+1) = \lambda(s(t), x(t)).$$

Кінцевий автомат є одним з найважливіших видів *керуючих систем.* Змістовно кінцевий автомат можна охарактеризувати як пристрій, який має вхідний і вихідний канали і який перебуває у кожний з моментів дискретного часу, (які називаються тактовими моментами), в одному зі станів $S_1, S_2, \ldots, S_n$.

По вхідному каналу у кожний тактовий момент у пристрій надходять сигнали – букви вхідного алфавіту. В ті ж моменти по вихідному каналу пристрій видає сигнали – букви вихідного алфавіту. До таких пристроїв можуть бути віднесені формальні системи, реальні автомати, живі організми тощо.

Кінцеві автомати є моделями для багатьох компонентів апаратного й програмного забезпечення. З точки зору теорії, кінцевий автомат може розглядатися як частковий випадок формально заданого алгоритму, тому теорія кінцевих автоматів має відношення до теорії алгоритмів.

Кінцеві автомати зручно представляти у вигляді орієнтованого графа, у якого вершини позначають стани, а ребра визначають переходи з одного стану в інший під дією відповідних вхідних сигналів. На цих же ребрах вказують виходи.

Реалізація кінцевого автомата може бути програмною (представленою у вигляді блок-схеми або на одній з мов програмування) і апаратною, для якої попередньо використовується двійкове кодування.

Два кінцевих автомати *еквівалентні*, якщо відображення вхід-вихід, що реалізуються ними є еквівалентними.

Функції автомата визначені на всіх вхідних сигналах, а *розширені* функції переходів і виходів – на множині вхідних ланцюжків.

<u>Ланцюжок</u> – послідовність символів певного алфавіту.

Стан автомата називають *недосяжним*, якщо не існує шляху з початкового стану, в протилежному випадку стан автомата – *досяжний.*

Стан кінцевого автомату називається досяжним, якщо існує такий ланцюжок, під впливом якого автомат попадає у такий стан.

Кінцеві автомати називаються *еквівалентними*, якщо виконані дві умови:

1. їх вхідні алфавіти співпадають;
2. відображення, що реалізуються ними – співпадають.

Добуток автоматів – це два не взаємодіючих кінцевих автомата, що стоять поруч, і які синхронно працюють на одному вході.

Мінімізацією кінцевого автомата називають задачу знаходження мінімального автомат, який реалізує задане автоматне відображення.

Існує два типи автоматів:

1) *розпізнавачі* – автомати без виходу, які розпізнають, чи належить вхідний ланцюжок заданій множині L;

2) *перетворювачі* – автомати з виходом, які перетворять вхідний ланцюжок x в ланцюжок y при умові, що $x \in L$.

Вхідну стрічку можна розглядати як лінійну послідовність комірок, причому кожна комірка може зберігати один символ з певного кінцевого вхідного алфавіту.

Стрічка автомата нескінченна, але зайнято на ній у кожний момент часу тільки кінцева кількість комірок. Граничні комірки ліворуч і праворуч від зайнятої області можуть займати спеціальні кінцеві маркери. Маркер може стояти тільки на одному кінці стрічки або може бути відсутнім взагалі.

Вхідна головка у кожний момент часу зчитує (оглядає) одну комірку вхідної стрічки. За один такт роботи автомата вхідна головка може зсунутися на одну комірку вправо або залишитися на місці, при цьому вона виконує тільки зчитування, тобто в ході роботи автомата символи в комірках вхідної стрічки не змінюються.

***Робоча стрічка* – це допоміжне сховище інформації.** Дані з неї можуть зчитуватися автоматом, можуть і записуватися на неї.

***Керуючий пристрій* – це програма, яка керує поведінкою автомата**. Він являє собою кінцеву множину станів разом з відображенням, яке описує, як змінюються стан у відповідності з поточним вхідним символом, що зчитується вхідною головкою, і поточною інформацією, здобутої з робочої стрічки. Керуючий пристрій також визначає напрямок зсуву робочої головки і те, яку інформацію записати на робочу стрічку.

Автомат працює, виконуючи певну послідовність робочих тактів. На початку такту зчитується вхідний символ і досліджується інформація на робочій стрічці. Потім, залежно від зчитаної інформації і поточного стану, визначаються дії автомата:

1) вхідна головка зсовується праворуч або залишається на місці;

2) на робочу стрічку записується певна інформація;

3) змінюється стан керуючого пристрою;

4) на вихідну стрічку (якщо вона є) записується символ.

Поведінку автомата зручно описувати в термінах конфігурації автомата, яка включає в себе:

а) стан керуючого пристрою;

б) вміст вхідної стрічки з положенням вхідної головки;

в) вміст робочої стрічки разом з положенням робочої головки;

г) вміст вихідної стрічки, якщо вона є.

Керуючий пристрій може бути недетермінованим. В цьому випадку для кожної конфігурації існує кінцева множина можливих наступних тактів, будь-який з яких автомат може виконати, виходячи з цієї конфігурації. Керуючий пристрій буде детермінованим, якщо для кожної конфігурації буде можливий тільки один наступний такт.

Існують такі типи автоматів:

1) машина Тюрінга (МТ);

2) лінійно-обмежений автомат (ЛОА);

3) автомат з магазинною пам'яттю (МП-автомат);

4) кінцевий автомат (КА).

Складність робочої стрічки визначає складність автомата. Так, наприклад:

1) машина Тюрінга має необмежену в обидві сторони стрічку;

2) у лінійно-обмеженого автомата довжина робочої стрічки представляє собою лінійну функцію довжини вхідного ланцюжка;

3) у МП-автомата робоча стрічка працює за принципом магазину LIFO;

4) у кінцевого автомата робоча стрічка відсутня.

10.2. Формальне визначення автомата

Неініціальний автомат – це п'ятірка виду

$$A = (K, X, Y, \delta, \gamma),$$

де K – множина станів (алфавіт станів); X – вхідний алфавіт;

Y – вихідний алфавіт;

δ – функція переходів, яка задає відображення $K \cdot X \to K$;

γ – функція виходів, яка задає відображення $K \cdot X \to Y$.

Функціонування автомата можна задати множиною команд виду:

$$qx \to py,$$

де q і $p \in K$, $x \in X$, $y \in Y$.

Нехай на певному такті t_i керуючий пристрій перебуває в стані q, а з вхідної стрічки зчитується символ x. Якщо в множині команд є команда $qx \to py$, то на такті t_i на вихідну стрічку записується символ y, а до наступного такту t_{i+1} керуючий пристрій перейде до стану p, тобто:

$$y(t) = \gamma(q(t), x(t)), \quad q(t+1) = \delta(q(t), x(t)).$$

Якщо ж команда $qx \to py$ відсутня, то автомат виявляється блокованим і не реагує на символ, прийнятий в момент t_i, а також перестає сприймати символи у наступні моменти часу.

У відповідності з визначенням неініціального автомата у початковий момент стан автомата може бути довільним.

Якщо зафіксований певний початковий стан, то такий автомат називають ініціальним, тобто $q(0) = q_0$.

Ініціальний автомат – це шістка (кортеж) виду:

$$A = (K, X, Y, \delta, \gamma, q_0),$$

де K – множина станів (алфавіт станів);

X – вхідний алфавіт;

Y – вихідний алфавіт;

δ – функція переходів (відображення $K \cdot X \to K$);

γ – функція виходів (відображення $K \cdot X \to Y$);

q_0 – початковий стан.

10.3. Розпізнавачі

10.3.1. Мови і автомати

Задача граматичного розбору полягає у знаходженні виводу ланцюжка у заданій граматиці і визначення дерева виводу цього ланцюжка.

Мови можуть бути задані двома способами:

1) граматиками (які породжують засіб мови);

2) автоматами (які розпізнають мову).

Різним за складністю автоматам відповідають різні типи мов. Найпростішим типом автоматів є кінцеві автомати.

Кінцевий автомат має вхідну стрічку, з якої за один такт може бути зчитаний один вхідний символ. Повернення за вхідною стрічкою не допускається.

Кінцевим автоматом називається п'ятірка виду:

$$A = (K, \sum, \delta, p_0, F),$$

де K – кінцева множина станів;

$\sum$ – алфавіт;

δ – функція переходів;

p_0 – початковий стан;

F – множина заключних станів.

Автомат можна визначити як формальну систему через стан, через символи, які записуються (зчитуються) зі стрічки або з декількох стрічок, і через набір команд.

Кінцевий автомат можна представити графом, таблицею переходів, командами, а також матрицею переходів.

10.3.2 Регулярні множини

Регулярні множини утворюють клас мов, які мають важливе значення для теорії формальних мов. Розглянемо декілька методів завдання мов, кожна з яких визначає регулярні множини. Серед них – регулярні вирази, праволінійні граматики, детерміновані і недетерміновані кінцеві автомати.

Нехай Σ – певний алфавіт. Регулярна множина в алфавіті Σ визначається рекурсивно наступним чином:

1) 0 – пуста множина;

2) $\{\varepsilon\}$ – множина з пустого ланцюжка;

3) $\{a\}$ – регулярна множина для кожного елемента $a \in \Sigma$;

4) якщо P і Q – регулярні множини в алфавіті Σ, то регулярними є множини:

а) $P \cup Q$

б) PQ

в) P^*.

Інших регулярних множин в алфавіті Σ немає. Таким чином, певна множина ланцюжків у заданому алфавіті Σ називається регулярним тоді і тільки тоді, коли або вона є одним з множин: 0, $\{\varepsilon\}$ або $\{a\}$ для певного $a \in \Sigma$, або його можна отримати з цих множин застосуванням кінцевої кількості операцій об'єднання, конкатенації і ітерації.

Для кожної регулярної множини існують щонайменше один регулярний вираз, який позначає цю множину.

Мова, яка розпізнається кінцевим автоматом, – це множина ланцюжків, які зчитуються автоматом при переході з початкового стану в один із заключних станів:

$$L(A)=\{a_1 a_2 \ldots a_n \mid p_0a_1 \to p_1,\ p_1a_2 \to p_2, \ldots, p_{n-1}a_n \to p_n;\ p_n \in F\}.$$

Множина називається регулярною, якщо існує кінцевий детермінований автомат, який розпізнає його.

10.3.3. Операції над регулярними мовами

Оскільки довільному кінцевому автомату однозначно відповідає детермінований кінцевий автомат, операції над кінцевими автоматами еквівалентні операціям над регулярними множинами або регулярними мовами.

Відомо, що для довільного кінцевого автомату можна побудувати еквівалентний автомат без циклів в початкових і (або) кінцевих станах.

Теорема. Для довільного кінцевого автомата існує кінцевий автомат без циклів у початковому стані.

Доведення. Нехай $A = (K,\sum,\delta,p_0, F)$ – довільний кінцевий автомат. Побудуємо автомат:

$$A_1 = (K \cup \{q_0\}, \Sigma, \delta \cup \{q_0a \to p_i \mid p_0a \to p_i \in \delta\}, q0, F \cup \{q0 \mid p0 \in F\}).$$

Будь-який ланцюжок $x = a_1\ a_2\ \ldots\ a_k$ належить мові $L(A)$ тоді і тільки тоді, коли існує наступна послідовність команд автомата A: $p_0a_1 \to p_1;\ p_1a_2 \to p_2;\ \ldots;\ p_{h-1}a_h \to p_h,\ p_h \in F$ і відповідна їй послідовність команд автомата A_1:

$$q_0a_1 \to p_1; \quad p_1a_2 \to p_2; \ldots; \quad p_{k-1}a_k \to ph.$$

Таким чином, маємо: $A = A_1$.

Теорема. Для довільного кінцевого автомата існує еквівалентний автомат без циклів у заключному стані.

Доведення. Будемо вважати, що автомат не має циклів у початковому стані.

Порівняємо заданий довільний кінцевий автомат $A = (K, \sum, \delta, p_0, F)$ з новим автоматом A_1:

$$A_1 = (K\cup \{f\}, \Sigma, \delta\cup \{q_ja \to f \mid p_ja \to p_i \in \delta \ \&\ p_i \in F\}, p_0, \{f\}\cup\{p_0 \mid p_0 \in F\}).$$

В результаті маємо: $A = A_1$.

Теорема. Множина регулярних мов є замкненою відносно операцій ітерації, усіченої ітерації, об'єднання, добутку, перетинання, доповнення і розбіжності.

Доведення. Для доказу необхідно виконати операції над відповідними кінцевими автоматами і показати, що в результаті таких перетворень побудований автомат припускає потрібну мову. Передбачається, що в автоматі видалені цикли з початкових і заключних станів. Для виконання перерахованих в теоремі операцій необхідно виконати відповідні перетворення над заданими автоматами.

1. Операція ітерації реалізується видаленням циклів з початкових і заключних станів і об'єднання отриманих станів. Об'єднання начального і заключного станів означає, що побудований автомат припускає пустий ланцюжок.

Однократний перехід з початкового в заключний стан вихідного автомата відповідає допуску ланцюжків мови L. Оскільки ці стани об'єднані, автомат припускає ланцюжки мов LL, LLL тощо, тобто він розпізнає мову $\{\varepsilon\}\cup L\cup L^2 \ldots = L^*$

2. Операція добутку над $L(A_1)$ і $L(A_2)$ виконується за допомогою двох перетворень:

а) видаляються цикли з початкового стану A_2 і заключного стану A_1;

б) кожному заключному стану A_1 ставимо у відповідність свій екземпляр A_2 і поєднуємо заключні стани A_1 з початковим станом відповідного екземпляра A_2.

3. Об'єднання $L(A_1)$ і $L(A_2)$ будуються за допомогою видалення циклів в початкових станах A_1 і A_2 і об'єднання отриманих начальних станів.

4. Усічена ітерація може бути побудована двома способами:

а) $L(A_1)^+ = L(A_1)^* L(A_1)$,

б) $L(A_1)+ = L(A_1)\ L(A_1)^*$.

5. Розглянемо доповнення $L(A_1)$ до Σ^*. Нехай автомат A_1 детермінований, тоді будь-який ланцюжок $x=a_1a_2\ .\ .\ a_n$ розізнається за єдиним маршрутом:

$p_0a_1 \to p_1$

$p_1a_2 \to p_2$

. . .

$p_{n-1}a_n \to p_n,\ p_n \in F.$

Автомат не розпізнає тільки ті ланцюги, які:

1) або представляють собою початкову частину ланцюжка $a_1a_2\ ..a_j$, при зчитуванні якого автомат переходить в стан, який не є заключним;

2) або мають вид $y = a_1a_2\ .\ .\ a_kbc_1c_2\ .\ .\ c_m$ $(k < n)$, де початок ланцюжка $a_1a_2 \ldots a_k$ збігається з початком ланцюжка $x \in L(A_1)$, але за символом a_k стоїть такий символ b, що автомат A_1 його зчитати не може.

Тому для того щоб побудувати автомат, який розпізнає доповнення мови, необхідно:

а) усі заключні стани зробити не заключними, а не заключні стани – заключними;

б) ввести додатковий стан, зробити його заключним і з кожного стану провести в цей стан такі дуги, кожна з яких відповідає символам алфавіту, не зчитуваних у цьому стані;

в) у побудованому додатковому стані побудувати петлі для усіх символів алфавіту, щоб забезпечити зчитування довільного закінчення ланцюжка $c_1c_2 \ldots c_m$.

6. Різниця $L(A_1)$ і $L(A_2)$ будується відповідно з наступним перетворенням:

$$L(A_1)\overline{\setminus L(A_2)} = L(A_1) \cap L(A_2).$$

7. Операція перетинання будується у відповідності з наступним перетворенням:

$$L(A_1) \cap L(A_2) = \overline{\overline{L(A_1)} \cup \overline{L(A_2)}}.$$

На підставі цієї теореми можна будувати кінцеві автомати, послідовно синтезуючи їх на основі вже побудованих автоматів.

10.3.4. Автоматні граматики

Лінійні граматики (праворекурсивні і ліворекурсивні) називаються автоматними граматиками, оскільки мови, що породжуються ними, збігаються з мовами, що розізнаються кінцевими автоматами.

Розглянемо низку теорем.

Теорема. Для кожної праволінійної граматики існує еквівалентний кінцевий автомат.

Доведення. Кожному нетерміналу довільної праволінійної граматики *G* поставимо у відповідність один стан кінцевого автомату *A*. Додамо ще один стан – єдиний кінцевий стан. Стан, який відповідає аксіомі, назвемо початковим станом.

Кожному правилу $A \to aB$ поставимо у відповідність команду $Aa \to B$, а кожному термінальному правилу $A \to a$ поставимо у відповідність команду $Aa \to F$.

Таким чином, виведенню ланцюжка в граматиці:

$$S \Rightarrow a_1A_1 \Rightarrow a_1a_2A_2 \Rightarrow \ldots \Rightarrow a_1a_2\ldots a_{k-1}A_{k-1} \Rightarrow a_1a_2\ldots a_k$$

взаємно однозначно відповідає послідовність команд побудованого автомата А:

$$Aa1 \to A1;\ A1a2 \to A2;\ \ldots;\ Ak\text{-}1ak \to F.$$

Таким чином, мова $L(G) = L(A)$.

Приклад. Нехай для заданої граматики G потрібно побудувати кінцевий автомат.

$$S \to aS \mid bB;$$
$$A \to aA \mid bS;$$
$$B \to bB \mid c \mid cA.$$

Розв'язання. Граф автомата буде мати чотири вершини, три з них позначені нетермінальними символами граматики *S, A, B*, четверта вершина, яка позначена символом *F*, є єдиним заключним станом (рис.10.3). Початковим станом є вершина, яка відповідає аксіомі *S*.

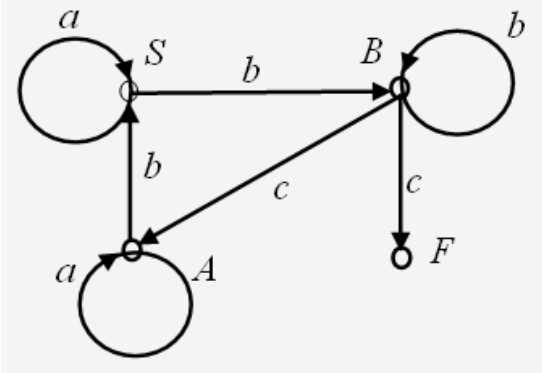


Рис.10.3. Подання автомата через граф

Кожному правилу граматики поставимо у відповідність команду кінцевого автомата: $Sa \to S$ – в початковому стані при надходженні на вхід термінального символу *a* автомат залишається у тому ж стані S;

$Sb \to B$ – з початкового стану при надходженні на вхід термінала *b* автомат переходить до стану *B*;

$Bb \to B$ – в стані В при надходженні на вхід термінала *b* автомат залишається в тому ж стані *B*;

Кожному правилу граматики поставимо у відповідність команду кінцевого автомата: $Sa \to S$ – у початковому стані при надходженні на вхід термінального символу *a* автомат залишається в тому ж стані *S*;

$Sb \to B$ – з початкового стану при надходженні на вхід термінала *b* автомат переходить в стан *B*;

$Bb \to B$ – в стані В при надходженні на вхід термінала b автомат залишається в тому ж стані B;

$Bc \to F$ – зі стану В при надходженні на вхід термінала c автомат переходить у заключний стан F;

$Bc \to A$ – зі стану В при надходженні на вхід термінала c автомат переходить у стан А;

$Aa \to A$ – в стані А при надходженні на вхід термінала a автомат залишається у цьому ж стані A;

$Ab \to S$ – зі стану А при надходженні на вхід термінала b автомат переходить у стан S.

Отриманий недетермінований кінцевий автомат розпізнає ланцюжки мови, які породжуються праворекурсивною граматикою G.

Теорема. Для довільного кінцевого автомата існує еквівалентна праволінійна граматика.

Доведення. Кожному стану довільного автомата поставлений у відповідність нетермінал граматики, причому початковому стану буде відповідати аксіома.

Тоді для кожної команди $Ac \to B$ в множині правил граматики включимо правило $A \to cB$, причому у випадку, якщо В – заключний стан, додамо правило $A \to c$. Еквівалентність вихідного кінцевого автомата і побудованої граматики очевидна.

Теорема. Для кожної ліворекурсивної граматики існує еквівалентний кінцевий автомат.

Доведення. Кожному нетермінальному символу довільної ліворекурсивної граматики поставимо у відповідність стан кінцевого автомата, причому стан, що відповідає аксіомі S, зробимо заключним. Додамо ще один стан N і зробимо його початковим станом.

Кожному правилу граматики $A \to Ba$ поставимо у відповідність команду автомата $Ba \to A$. Тоді кожному виводу в граматиці: $S \Rightarrow A_1a_k \Rightarrow A_2a_{k-1}a_k \Rightarrow \ldots \Rightarrow A_{k-1}a_2a_3\ldots a_k \Rightarrow a_1a_2\ldots a_k$ однозначно відповідає наступна послідовність команд побудованого автомата А:

$$Na_1 \to A_{k-1};\ \ldots; A_2a_{k-1} \to A_1; A_1a_k \to S.$$

Таким чином, мова $L(G) = L(A)$.

Теорема. Для довільного кінцевого автомата існує еквівалентна ліворекурсивна граматика.

Доведення. Кожному стану довільного автомата поставимо у відповідність нетермінальний символ граматики, додамо нетермінал S і зробимо його аксіомою.

Кожній команді $Aa \to B$ в множині правил включимо відповідне правило $B \to Aa$, причому у тому випадку, якщо В – заключний стан, додатково введемо правило $S \to Aa$, а якщо А – начальний стан, то додатково введемо правило $B \to a$.

Тоді послідовності команд:

$$A_0a_1 \to A_1; A_1a_2 \to A_2;\ \ldots;\ A_{k-1}a_k \to F$$

відповідає наступний висновок:

$$S \Rightarrow A_{k-1}a_k \Rightarrow \ldots \Rightarrow A_1a_2a_3\ldots a_k \Rightarrow a_1a_2a_3\ldots a_k.$$

Важливою особливістю автоматних граматик є можливість їх подання за допомогою кінцевих графів. За графом граматики легко відшукується вивід потрібного ланцюжка.

Будь-який вивід ланцюжка в автоматній граматиці відповідає шляху в графі цієї граматики, який починається з вершини S (вершини, яка помічена аксіомою) і закінчується в кінцевій вершині.

Приклад. Побудувати кінцевий автомат, який розпізнає мову $L(A) = \{(ab)^*\}$.

Спочатку побудуємо деяку граматику G, яка б породжувала мову $L(A)$:

$$S \to aA;$$
$$A \to bS \mid b.$$

Перевіримо, чи дійсно ця граматика породжує мову $L(A)$. Для цього побудуємо декілька виводів можливих варіантів ланцюжків:

1) $S \Rightarrow aA \Rightarrow ab$;
2) $S \Rightarrow aA \Rightarrow abS \Rightarrow abaA \Rightarrow abab$;
3) $S \Rightarrow aA \Rightarrow abS \Rightarrow abaA \Rightarrow ababS \Rightarrow ababaA \Rightarrow abababa$ тощо.

Таким чином, граматика G дійсно породжує мову $L(A)$, отже, можна побудувати кінцевий автомат, що відповідає цій граматиці. Для цього введемо заключний стан F, початковий стан відповідає аксіомі S (рис.10.4).

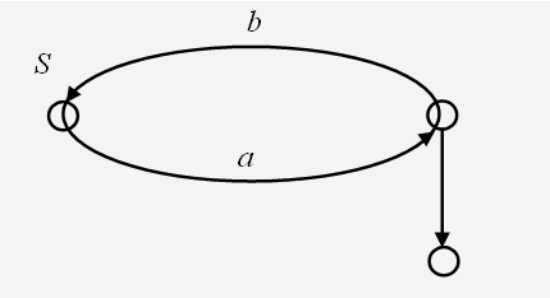


Рис.10.4. Кінцевий автомат, породжений граматикою G

Запишемо перетворення правил виводу в команди:

$Sa \to A$ – зі стану S при надходженні на вхід термінала a автомат переходить до стану A;

$Ab \to S$ – зі стану A при надходженні на вхід термінала a автомат переходить до стану S;

$Ab \to F$ – зі стану A при надходженні на вхід термінала b автомат переходить у заключний стан F.

Таким чином, будується недетермінований кінцевий автомат, який розпізнає задану мову $L(G)$.

10.4. Автомати з магазинною пам'яттю

Автомат з магазинною пам'яттю (МП-автомат) має робочу стрічку, яка організована у вигляді магазину.

МП-автомат – це сімка виду:

$$M = (K, \Sigma, \Gamma, \delta, p_0, F, B_0),$$

де K - кінцева множина станів; Σ – алфавіт; Γ – алфавіт магазина; δ – функція переходів; p_0 – початковий стан; F – множина заключних станів; B0 – символ з множини Γ для позначення маркера дна магазина.

В загальному випадку дане визначення відповідає недетермінованому автомату. На відміну від конечного автомата для довільного МП-автомата не можна побудувати еквівалентний детермінований автомат.

Основне використання розпізнавальних засобів завдання мов полягає в побудові алгоритмів граматичного розбору. Тому необхідно для довільної КС-граматики вміти будувати еквівалентний МП-автомат.

МП-автомат представляє інтерес як засіб розбору в КС-граматиках довільного виду. Цей факт сформульований в наступної теоремі.

Теорема. Мови, породжувані КС-граматиками, збігаються з мовами, які розпізнаються МП-автоматами.

Доведення. Існують дві стратегії розбору: висхідний і спадний розбір. Розглянемо обидві стратегії розбору.

10.4.1. Висхідний розбір в МП-автоматі

При висхідній стратегії необхідно знайти основу і редукувати її до якогось нетерміналу у відповідності з правилами даної граматики. Це можна зробити в том випадку, якщо реалізувати наступний алгоритм функціонування МП-автомата:

1) будь-який вхідний символ записується в магазин;

2) якщо у верхівці магазину сформована основа, співпадаюча з правою частиною правила, то вона замінюється на нетермінал в лівій частині цього правила;

3) розбір закінчується, якщо в магазині залишається аксіома, а вхідний ланцюжок розглянутий повністю.

У відповідності з цим алгоритмом для КС-граматики $G = (V_T, V_N, P, S)$ побудуємо МП-автомат:

$$M = (K, V_T, \Gamma, \delta, p_0, F, B_0), \text{ де } \Gamma = V_T \cup V_N \cup \{B0\},$$
$$K = \{p_0, F\}, F = \{f\}.$$

Функція переходів δ буде містити наступні команди:

а) $p_0, a, \varepsilon \to p_0, a$ – для будь-яких $a \in V_T$;

б) $p_0, \varepsilon, \varphi' \to p_0, A$ – для усіх правил $A \to \varphi \in$ P, де φ' – дзеркальне відображення φ;

в) $p_0, \varepsilon, SB_0 \to f, B_0$.

В загальному випадку команда виглядає так:

$$p_i, \sigma, \gamma \to p_j, \lambda ,$$

де $p_i \in K$ – стан автомата до виконання команди, $\sigma \in V_t$ – символ на вхідній стрічці, $\gamma \in \Gamma$ – символ верхівки магазина, $p_j \in K$ – стан автомата після виконання команди, $\lambda \in \Gamma$ – символ, який записується в магазин.

Таким чином, будь-якому виводу в граматиці *G* взаємно однозначно відповідає послідовність команд побудованого МП-автомата. Зворотна побудова КС-граматики за довільним МП-автоматом також молива, але не представляє практичного інтересу.

Розглянемо приклад висхідної стратегії розбору.

Нехай дана граматика G:

$$S \to S+A \mid S/A \mid A$$
$$A \to a \mid (S);$$
$$V_N=\{S, A\},\ V_T=\{a, (,), +, /\}.$$

Для заданої КС-граматики *G* необхідно побудувати МП-автомат. Еквівалентний МП-автомат повинен містити наступні команди:

1. Команди переносу термінальних символів в магазин:

$$p_0, a, \varepsilon \to p_0, a;$$
$$p_0, +, \varepsilon \to p_0, +;$$
$$p_0, /, \varepsilon \to p_0, /;$$
$$p_0, (, \varepsilon \to p_0, (;$$
$$p_0,), \varepsilon \to p_0,).$$

Ці команди забезпечують занесення термінального символу з вхідної стрічки в магазин.

2. Команди редукції за правилами граматики:

$$p_0, \varepsilon, A+S \to p_0, S;$$
$$p_0, \varepsilon, A/S \to p_0, S;$$
$$p_0, \varepsilon, A \to p_0, S;$$
$$p_0, \varepsilon,)S(\to p_0, A;$$
$$p_0, \varepsilon, a \to p_0, A.$$

Ці команди заміняють дзеркальне відображення правила, отриманого у верхівці магазина, на нетермінал у лівій частині даного правила граматики.

3. Команди перевірки на завершення розбору:

$$p_0, \varepsilon, SB_0 \to f, B_0.$$

Розбір завершується, якщо в магазині залишилися аксіома і маркер дна магазина, а вхідний ланцюжок повністю розглянутий.

Подамо на вхід автомата ланцюжок *a/(a+a)* і виконаємо розбір. Процес розбору представлений в табл. 10.1.

Таблиця 10.1.

вхід	**a**			**/**	**(**	**a**			**+**	**A**			**)**				
М											**a**	**A**					
А										**+**	**+**	**+**		**)**			
Г							**a**	**A**	**S**	**S**	**S**	**S**	**S**	**S**			
А						**(**	**(**	**(**	**(**	**(**	**(**	**(**	**(**	**(**	**A**		
З					**/**	**/**	**/**	**/**	**/**	**/**	**/**	**/**	**/**	**/**	**/**		
И		**a**	**A**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	
Н	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0
р	p_0	p_0	p_0	p_0	p_0	p_0	p_0	p_0	p_0	p_0	p_0	p_0	p_0	p_0	p_0	p_0	**f**

10.4.2. Спадний розбір в МП-автоматі

На будь-якому кроці спадного розбору повинне застосовуватись певне правило. На початковому етапі таким нетерміналом є аксіома. МП-автомат, який виконує спадний розбір, працює за наступним алгоритмом:

1) на початку в магазин заноситься аксіома: $p_0, \varepsilon, \varepsilon \to p_1, S$;

2) для кожного правила $A \to \varphi \in P$ нетермінал у верхівці магазина замінюється на праву частину правила за допомогою команди: $p_1, \varepsilon, A \to p_1, \varphi$;

3) для кожного терміналу $a \in V_T$ виконується порівняння символу на вхідній стрічці з символом у верхівці магазина і його поглинання: $p_1, a, a \to p_1, \varepsilon$;

4) розбір закінчується за командою: $p_1, \varepsilon, B_0 \to f, B_0$.

Для граматики, розглянутої у попередньому прикладі, розбір того ж вхідного ланцюжка за спадною стратегією буде виконуватись за допомогою наступної множини команд:

1) команда занесення аксіоми в магазин:

$$p_0, \varepsilon, \varepsilon \to p_0, S;$$

2) команди заміни нетермінала правою частиною правила:

$$p_1, \varepsilon, S \to p_1, S+A$$
$$p_1, \varepsilon, S \to p_1, S/A$$
$$p_1, \varepsilon, S \to p_1, A$$
$$p_1, \varepsilon, A \to p_1, a$$
$$p_1, \varepsilon, A \to p_1, (S);$$

3) команди порівняння і поглинання символу з вхідної стрічки і символу у верхівці магазина:

$$p1,\ a,\ a \rightarrow p1,\ \varepsilon$$
$$p1,\ +,\ + \rightarrow p1,\ \varepsilon$$
$$p1,\ /,\ / \rightarrow p1,\ \varepsilon$$
$$p1,\ (,\ (\rightarrow p1,\ \varepsilon$$
$$p1,\ ),\ ) \rightarrow p1,\ \varepsilon;$$

4) команда завершення розбору:

$$p_1,\ \varepsilon,\ B_0 \rightarrow f,\ B_0.$$

Процес розбору ланцюжка представлений в таблиці 10.2.

Таблиця 10.2.

вхід					**a**	**/**		**(**					**a**	**+**		**a**	**)**	
M																		
A																		
Г										**S**	**A**	**a**						
A			**S**	**a**	**a**			**(**		**+**	**+**	**+**	**+**					
З			**/**	**/**	**/**	**/**		**S**	**S**	**A**	**A**	**A**	**A**	**A**	**a**			
И		**S**	**A**	**A**	**A**	**A**	**A**	**)**	**)**	**)**	**)**	**)**	**)**	**)**	**)**	**)**		
H	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0	B_0
p	p_0	p_1	p_1	p_1	p_1	p_1	p_1	p_1	p_1	p_1	p_1	p_1	p_1	p_1	p_1	p_1	p_1	f

Вищерозглянуті МП-автомати працюють недетерміновано, тобто якщо ланцюжок належить мові, яка породжена заданою граматикою, то якийсь з варіантів функціонування автомата здійснить правильний розбір. Якщо ж ланцюжок не належить мові, то жодний з варіантів розбору не приведе до мети.

Відсутність детермінованого еквівалентного автомата для довільної КС-граматики означає неможливість побудови універсальної простої однопрохідної програми синтаксичного аналізу. Тому для ефективного розбору необхідно виділяти спеціальні класи КС-граматик, які задовольняють вимогам конкретних типів аналізаторів.

Якщо потрібно виконати розбір для довільної КС-граматики, то доведеться використовувати детерміновану програмну модель недетермінованого МП-автомата.

10.5. Перетворювачі

Автомати з виходом називаються перетворювачами. Залежно від виду функції, яка відображає множину станів і вхідних символів в множині вихідних символів і нових станів, а також від типу робочої стрічки розрізняють різні види перетворювачів. Розглянемо кінцеві автомати-перетворювачі.

Кінцевим перетворювачем називається шістка виду

$$P = (K, X, Y, f, g, q_0),$$

де K – кінцева множина станів;
X –- вхідний алфавіт;
Y – вихідний алфавіт;
f – функція переходів;
g – функція виходів;
q_0 - початковий стан.

Типи відображень f і g визначають різні види автоматів. Якщо g – відображення $K \cdot X$ *в* Y, то кінцевий перетворювач називається синхронним. В загальному випадку це відображення має вигляд $K \cdot X \rightarrow Y^*$.

Нехай $P = (K, X, Y, f, g, q_0)$ - кінцевий перетворювач. Тоді відображення $S(x) = g(q_0, x)$, що визначається для будь-якого ланцюжка $x \in X^*$, називається кінцевим перетворювачем.

Зазначимо, що для того щоб вихідний ланцюжок y можна було вважати перекладом вхідного ланцюжка x, ланцюжок x повинен перевести перетворювач з початкового стану в кінцевий.

10.6. Автомати Мілі і Мура

Автомати Мілі і Мура є неініціальними автоматами. У відображення $S(x)= g(q_0,x)$ зафіксуємо початковий стан q_0, у якому автомат перебуває у початковий момент часу. Воно істотно впливає на процес кінцевого перетворення, оскільки визначає не тільки результуючий ланцюжок, але й множину вхідних ланцюжків.

Розглянемо поведінку ініціальних автоматів, які можуть починати працювати з будь-якого зазначеного стану. Такий автомат отримує на вхід один ланцюжок нескінченної довжини

і перероблює її. Реакція такого перетворювача на певний вплив непередбачувана, якщо невідомий його початковий стан. Тому необхідно вирішити дві задачі, що мають важливе практичне значення:

1) визначення того стану автомата, в якому він перебуває в момент, починаючи з якого досліджується його поведінка;

2) розпізнавання кінцевого стану, у який перейшов автомат після завершення іспитової операції. Цей стан буде початковим для наступної серії іспитів.

Ці задачі аналізу отримали назву експериментів з розпізнаванню станів.

10.6.1. Автомат Мили

Автомат Мілі – це п'ятірка виду:

$$M = (K, X, Y, f, g),$$

де: K – множина станів автомата;

X – вхідний алфавіт;

Y – вихідний алфавіт;

f – функція переходів (відображення $K \cdot X \to K$);

g – функція виходів (відображення $K \cdot X \to Y$).

Як і будь-який інший автомат, автомат Мілі можна представити у вигляді таблиці або графа. В графі переходів автомата Мілі на дугах вказуються через символ '/' вхідні і вихідні символи. Таблиця переходів складається з двох частин: у лівій частині записуються значення функції виходів, у правій частині - значення функції переходів.

Приклад. Побудуємо перетворювач, який розпізнає арифметичні вирази, які породжені граматикою: $S \to a+S|a-S\ |+S\ |-S\ |a$ і видаляють з цих виразів надлишкові унарні операції. Наприклад, вираз $-a+-a-+-a$ від переводить в $-a-a+a$. У вхідній мові символ a представляє ідентифікатор і перед ідентифікатором допускається довільна послідовність знаків унарних операцій + и −. Зазначимо, що вхідна мова є регулярною множиною.

Нехай $M = (K, X, Y, f, g)$, де

1. $K = \{q_0, q_1, q_2, q_3, q_4\}$,
2. $X = \{a,+, -\}$,
3. $Y = X$.

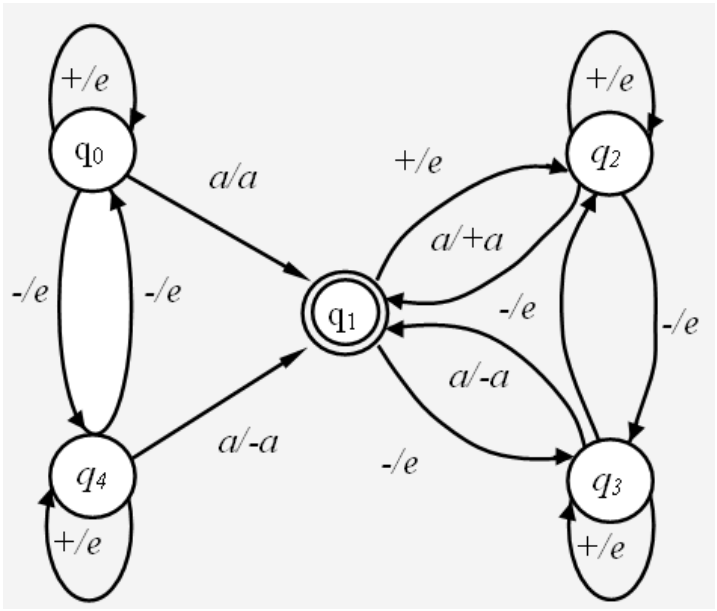


Рис.10.5. Схема перетворювача

Перетворювач М починає роботу в стані q_0 і, чергуючи стани q_0 і q_4 на вхідному символі «−», визначає, парне або непарне число знаків – передує першому символу а. Коли з'являється a, перетворювач M переходить у стан q_1, допускаючи вхід, і видає a або $-a$ в залежності від того, парна або непарна кількість мінусів, що з'явилися. Для наступних символів a він підраховує, парна або непарна кількість попередніх мінусів, за допомогою станів q_2 і q_3. Єдина відмінність між парами q_2, q_3 і q_0, q_4 полягає у тому, що якщо символу a передує парна кількість мінусів, то перша з них видає $+a$, а не тільки a (табл.10. 3).

Таблиця 10.3.

Таблиця переходів

	Y			K		
K\X	a	+	−	a	+	−
q_0	a	ε	ε	q_1	q_0	q_4
q_1	−	ε	ε	−	q_2	q_3
q_2	+a	ε	ε	q_1	q_2	q_3
v	−a	ε	ε	q_1	q_3	q_2
q_4	−a	ε	ε	q_1	q_4	q_0

10.6.2. Автомат Мура

Автомат Мура – це «п'ятірка» виду:

$$U = (K_1, X, Y, f_1, h),$$

де: K_1 – множина станів автомата;
X – вхідній алфавіт;
Y – вихідний алфавіт;

f1 – функція переходів (відображення $K \cdot X \to K$);

h – функція виходів (відображення $K \cdot X \to Y$).

При поданні автомата Мура графом дуги помічаються символами вхідного алфавіту, а кожна вершина графа – станом і символом вихідного алфавіту.

При формальному порівнянні визначень автоматів Мілі і Мура може здатися, що автомат Мура може бути заданий як вхідонезалежний автомат Мілі, тобто такий автомат Мілі, вихідна функція якого задовольняє наступним умовам: $\forall a \in X, \forall b \in X, \forall z \in Z\ (g(z, a) = g(z, b))$. Однак це не відповідає способу функціонування автоматів Мура у відповідності з встановленим визначенням.

В автоматі Мура реалізована інший часовий зв'язок між переходами з одного стану в інший і виходом, порівняно з автоматом Мілі, у якого вихід, що відповідає певному входу і певному стану, породжується під час переходу автомата в наступний стан. В автомата Мура спочатку породжується вихід, а потім – перехід у наступний стан, причому вихід визначається тільки станом автомата.

10.6.3. Рівносильність автоматів Мілі і Мура

Рівносильність полягає у тому, що множина реакцій цих автоматів співпадає:

$$L(M) = \{qz \mid qZ \in K\};$$
$$L(U) = \{ht \mid ht \in K1\};$$
$$L(M) = L(U).$$

Теорема. Для кожного автомата Мура можна побудувати рівносильний автомат Мілі.

Доказ. Граф рівносильного автомата Мілі *M* можна отримати у тому випадку, якщо кожному ребру автомата Мура *U* зіставити ребро автомата М.

Нехай $\omega = x_1\ x_2\ \dots\ x_n$ – вхідний ланцюжок, тоді множина реакцій для автоматів *M* і *U* будуть відповідно представлені наступним чином:

$q_0 / y_1, x_1 \to q_1 / y_2, x_2 \to \dots \to q_{n-1} / y_n, x_n$;
$q_0, x_1 / y_1 \to q_1, x_2 / y_2 \to \dots \to q_{n-1}, x_n / y_n$.

Теорема. Для будь-якого автомата Мілі можна побудувати еквівалентний автомат Мура.

Доказ. В якості множини K_1 автомата Мура візьмемо $K_1 = K \cdot Y$. Для забезпечення рівносильності автоматів $M = U$ функції переходів і виходів визначимо наступним чином:

$$f_1(p \cdot y, a) = \{qb \mid f(p, a) = q, b \in X\};$$
$$h(p \cdot y) = y.$$

Якщо реакція автомата *M* на вхідний ланцюжок виду $\omega = x_1\ x_2\ \dots\ x_n$ зі стану q_0 має вид

$$q_0, x_1 / y_1 \to q_1, x_2 / y_2 \to \dots \to q_{k-1}, x_k / y_k \quad (1),$$

то існує такий стан $q_0 \cdot x_1$ недетермінованого автомата *U*, що, починаючи роботу з цього стану, автомат *U* виконує наступні дії:

$$q_0 \cdot x_1 / y_1 \to q_1 \cdot x_2 / y_2 \to \dots \to q_{k-1} \cdot x_k / y_k, x_k \quad (2).$$

Аналогічно можна довести і зворотну теорему про те, що з існування реакції (2) випливає існування реакції (1), що підтверджує рівносильність автоматів Мілі і Мура.

10.7. Спосіб завдання абстрактних автоматів

Для завдання абстрактного автомату існують такі способи:

- функціональний;
- за допомогою таблиць переходів та виходів;
- графічний (залежно від способу завдання функцій).

10.7.1. Функціональний спосіб завдання абстрактних автоматів

Автомат працює у дискретному часі. Послідовні моменти часу ототожнюють з послідовними натуральними числами $t = 1, 2, 3, \ldots$ (це завжди можна зробити, обравши за одиницю часу будь-який зручний відрізок виміру).

У кожний момент автоматного часу $t = 1, 2, 3, \ldots$ автомат A знаходиться у деякому стані:

$$S(t) \in S.$$

Початковий стан автомата A - $S(0)$.

У кожний момент часу t, починаючи з $t=1$, до входу автомата надходить одна літера $x(t) \in X$ вхідного алфавіту (вхідний сигнал). Вхідними словами автомата будуть скінченні впорядковані послідовності вхідних сигналів $x(1)x(2)...x(k)$, які подаються до входу автомата, а це викликає появу вихідного слова $q = y(1)y(2)\ldots y(k)$ – упорядкованої послідовності вихідних сигналів автомата A тієї самої довжини.

Відповідальність між припустимими вхідними словами p і вихідними словам і q називається алфавітним відображенням, індукованим автоматом A. Це відображення однозначно задається двома функціями – функцією переходів з одного стану в інший та функцією виходів, котра визначає вихідний символ або сигнал.

Стан автомату $S(t)$ у будь-який момент часу t однозначно визначається попереднім станом $S(t-1)$ та вхідним сигналом $x(t)$. Але щодо моменту часу t переходу зі стану $S(t-1)$ у стан $S(t)$, вихідний сигнал $y(t)$ може з'явитися або до переходу, або після нього. Тому $y(t)$ може бути визначений двома способами:

$$\begin{cases} y(t) = \psi(S(t-1), x(t))^* \\ y(t) = \psi(S(t), x(t))^{**} \end{cases}$$

Функція ψ є функцією виходів. Причому ψ^* - це звичайна функція, яка визначає вихідну літеру $y(t)$ залежно від вхідного сигналу $x(t)$ в момент t та від стану у попередній момент часу $S(t-1)$, ψ^{**} - функція зсунута, яка визначає вихідну літеру $y(t)$ залежно від стану $S(t)$ в момент часу t і вихідного сигналу $x(t)$ в той самий момент часу t.

Залежно від визначення $y(t)$ розрізняють два типи абстрактних автоматів:

$S(t) = \varphi(S(t-1), x(t))$ – це автомат 1 роду, або автомат Мілі,

$Y(t) = \varphi(S(t-1), x(t))$ – це автомат 2 роду, або автомат Мура.

Якщо всі три множини X, Y, S, які визначають абстрактний автомат A, є скінченними, то й автомат називають скінченим.

10.7.2. Табличний спосіб завдання абстрактного автомата

Абстрактний автомат можна задати за допомогою двох таблиць:

- таблиці переходів;
- таблиці виходів автомата.

Рядки обох таблиць позначаються символами вхідного алфавіту $\{X\}$, а стовпчики - символами станів. На перетині X-го рядка та $\in$-го стовпчика таблиці переходів стоїть елемент $U(s,x)$, тобто деякий символ вихідного алфавіту Y.

Розглянемо приклад.

Нехай маємо скінчений автомат Мілі, заданий трьома множинами S {1,2,3}; X={a,b}; Y= {u,v} (табл.10.4).

Таблиця 10.4

Таблиця функціонування скінченного автомата Мілі

	1	2	3
a	2	3	1
b	2	2	3
	1	2	3
a	u	v	u
b	v	v	u

Рис. 10.4. Таблиці функціонування автомата Мілі

Якщо до входу автомата Мілі буде подано слово *aabb,* то на виході матимемо слово *uvuu.*

Якщо задано автомат Мура (II роду) тими самими множинами S, *X, Y,* то його об'єднана таблиця виходів та переходів матиме вигляд:

Таблиця 10.5

Таблиця функціонування автомата Мура

	u	***u***	***v***
	1	***2***	***3***
a	***u***	***v***	***u***
b	***v***	***v***	***u***

Рис. 10.5. Таблиця функціонування автомата Мура

Станам 1 та 2 відповідає вихідний сигнал *u*, а стану *3 – v.*

Різниця між двома класами автоматів полягає у тому, що у випадку автоматів Мілі вихідний сигнал виникає одночасно із вхідним сигналом, що його ініціює, а у випадку автоматів Мура – із затримкою на 1 (одиницю) автоматного часу.

Абстрактний автомат може мати заборонені області застосування, або сукупність слів у вхідному алфавіті, які не сприймаються автоматом, котрі знаходиться у деяких станах з множини станів, тобто у якомусь стані, автомат не сприймає певний вхідний символ (ситуація схожа з діленням на 0). Тоді у таблицях переходів та виходів у відповідних клітинах стоятиме «-». Тобто, аварійна зупинка.

10.7.3. Графічний спосіб завдання абстрактного автомата

Другим способом завдання абстрактного автомата є графічний – він більш наочний і використовує графи. Вершини графа (у вигляді кола) ототожнюються зі станами автомата. Стрілка, що з'єднує вершину *i з* вершиною *j*, означає, що існує вхідний сигнал *x*, який переводить автомат зі стану *i* у стан *j*, тобто $j = \varphi(i,x)$. Той самий автомат за допомогою граф-схеми зображено на рис. 10.6.

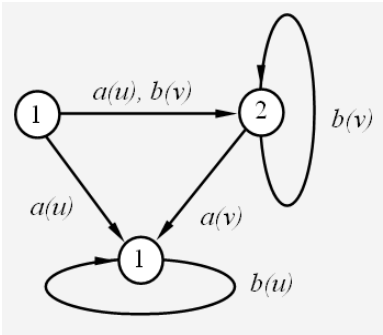


Рис. 10.6. Подання автомата за допомогою граф-схеми

Питання для самоконтролю

1. *Чому виникла необхідність створення абстрактних автоматів?*
2. *Що розуміють під абстрактним автоматом?*
3. *Які способи завдання абстрактних автоматів Вам відомі?*
4. *Що розуміють під автоматичним відображенням?*
5. *В чому полягає відмінність між автоматами Мура і Мілі?*

РОЗДІЛ 2

ПРОГРАМУВАННЯ В СЕРЕДОВИЩІ ТУРБО-ПАСКАЛЬ

11. ТЕХНОЛОГІЇ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

„Роби просто, наскільки це можливо, але не простіше цього”.
А. Ейнштейн

Розробка програмного забезпечення (ПЗ) (англ. software engineering, software development) – це рід діяльності (професія) і процес, спрямований на створення і підтримку працездатності, якості і надійності програмного забезпечення, що використовує технології, методологію і практику з інформатики, управління проектами, математики, інженерії й інших галузей знання.

Індустрія ПЗ розвивається стрімкими темпами, однак ні для кого не є секретом, що процес розробки ще дуже далекий від досконалості і для нього характерна множина внутрішніх проблем. За даними дослідження Standish Group (www.standishgroup.com), менше третини програмних проектів виявляються успішними, інші – або не вписуються в фінансові і часові рамки, або взагалі закінчуються повним провалом.

Найбільш успішними є невеличкі проекти, а ризик провалу тим вище, чим вони значніше. Це свідчить про те, що зі зростанням масштабів проектів зі створення програмного забезпечення менеджери не справляються з управлінням виділеними ресурсами.

Власне, проблема зростання складності управління залежно від збільшення розміру організації далеко не нова, однак в програмуванні вона набуває нові риси: якщо для реалізації проекту у відведений термін недостатньо ресурсів, то просте їх збільшення тільки ускладнює проблему і термін виконання буде відсунутий ще далі (мова йде перед усім про людські ресурси, оскільки інші в програмуванні є набагато менш істотними).

11.1. Предмет технології розробки програмного забезпечення

ПЗ сьогодні присутнє всюди: практично всі електронні пристрої містять ПЗ того чи іншого виду. Без відповідного ПЗ в сучасному світі неможливо уявити індустріальне виробництво, школи, університети, систему охорони здоров’я, фінансові і урядові заклади. Багато людей використовують його для самоосвіти, розваг тощо. В технічних системах вартість ПЗ часто складає більшу частину вартості виробу.

Сьогодні використання інформаційних технологій визначає успіхи провідних компаній світу. Більше того, економічний стан розвинутих країн визначається як комп'ютерними технологіями так і станом комп'ютерної індустрії в цих країнах. Наприклад, в США за такими показниками, як зростання цін на акції а також ріст заробітної плати співробітників, компанії, які займаються розробкою та впровадженням комп'ютерних технологій в декілька десятків разів перевершують традиційні високоприбуткові галузі промисловості як-то будівництво, автомобілебудування тощо. Люди і компанії, які свого часу вкладали кошти в ризиковані проекти, пов'язані з інформаційними технологіями сьогодні отримують надприбутки.

Прогрес, досягнутий за останні роки в розвитку комп'ютерної техніки настільки великий, що якщо б такого ж прогресу за ці роки наприклад досягло літакобудування то ми мали б літак, який міг би облетіти довкола земної кулі за 1 годину, витративши при цьому лише 2 літри пального і коштував би 400 доларів.

В той же час швидкість створення ПЗ залишається майже такою ж низькою, як і 20-30 років тому. При цьому розробка ПЗ коштує надзвичайно дорого і значно перевищує вартість комп'ютерів, тоді як якість ПЗ залишається низькою.

Такі диспропорції пояснюються дуже просто. Комп'ютери збираються з готових компонент і деталей (як і будь-який продукт в сучасному світі), хоча людство могло витратити на винахід і розробку такої компоненти багато десятків а то й сотень років. Іншими словами, компаніям, які займаються виготовленням комп'ютерної техніки не доводиться кожного разу винаходити транзистори, мікросхеми а також технологічні процеси, які дозволяють переходити від етапу до етапу.

В той же час в галузі ПЗ програмісти створюють кожну нову програму з нуля (це все одно, якби створення нового комп'ютера починалося з необхідності винаходу технології для обробки кремнієвого піску та перетворення його на транзистори). Завдяки цьому програми створюються надзвичайно повільно, крім того, в такому продукті завжди багато помилок, на виправлення яких потрібний новий час і нові гроші.

Яким чином можна виправити такий стан речей?

Відповідь дуже проста – зробити можливим складати нові програми з уже створених і давно перевірених модулів (компонент). При цьому, щоб зробити такі компоненти якомога ширше доступними і взаємозамінними, необхідно створити певні стандарти, яким вони повинні відповідати.

На сьогодні існують два такі найбільш поширені специфікації, які описують побудову компонент а також їх взаємодію: CORBA та COM/DCOM.

Як приклад компонентного ПЗ може бути наведена сучасна продукція компанії Micrtosoft: MS Office, Internet Explorer, ОС Windows та інші. При цьому такі програми як Excel, Word, Access здатні надавати „послуги" іншим програмам (через виклики своїх функцій).

Особливо актуальною така технологія стає сьогодні у зв'язку з бурхливим розвитком комп'ютерних комунікацій й мережі Internet. При цьому спостерігається відхід компаній від традиційної концепції *клієнт-сервер* і впровадження *intranet-технології*. Основна відмінність між цими двома підходами визначається самою природою Web–середовища, як множини незалежних документів і програм, динамічно пов'язаних між собою.

Традиційні рішення на основі технології *клієнт-сервер* виглядають дещо інакше. Зазвичай, прикладні програми архітектури *клієнт-сервер* зв'язані жорстко, модулі збираються компонувальником в один або кілька бінарних файлів. Крім того, такі програми виготовляються в спеціалізованому середовищі розробки і це визначає закритість і негнучкість таких систем.

В противагу цьому intranet–середовище не накладає вимог на засоби розробки документів або об'єктів.

Компонентний підхід дозволяє докорінно змінити інструменти розробки програмного забезпечення, процес розробки а також саме поняття про працю програміста, оскільки фахівців, які займатимуться програмуванням можна буде розділити на 2 категорії:

1 – ті, хто створює компоненти;

2 – ті, хто збирають з готових компонент закінчені програми.

Це дозволить скоротити витрати на створення нового ПЗ, час необхідний для розробки а також витрати на підготовку фахівців, які будуть займатися зборкою ПЗ.

Найкращою якістю ПЗ є його гнучкість: запрограмувати можна практично все що завгодно. Гіршою ж якістю ПЗ також є його гнучкість. Це саме „практично все що завгодно " дуже ускладнює планування, моніторинг і керування розробкою ПЗ. Непередбачуваність – це першопричина того, що в останні 40 років називали «кризою ПЗ».

У середині 90-х років XX ст. було розпочато принаймні три важливі спроби аналізу стану індустрії розробки ПЗ. Результати були представлені в Patterns of Software Systems Failure and Success (Зразки успіхів і невдач систем ПЗ) [Jones, 1996], в «Chaos» («Хаос») [Standish Group, 1995] і в Report of the Defense Science Board Task Force on Acquiring Defense Software Commercially (доповідь наукової ради з оборони, яка стосується завдання збільшення кількості ПЗ для оборонних цілей, одержуваного на комерційній основі) [Defense Science Board, 1994].

Ці три аналітичні дослідження привели до аналогічного висновку: відсоток успішних проектів зі створення ПЗ надзвичайно низький. В деяких моментах були розбіжності, але основні твердження добре узгоджувались та доповнювали один одного. Ці роботи чітко продемонстрували складність проблеми й сучасні нормативні показники ефективності традиційного керування розробкою ПЗ. Тут є що вдосконалювати.

11.2. Моделі життєвого циклу програмного забезпечення

11.2.1. Модель „кодування–усунення помилок "

Така модель описується наступним чином:

1) поставити задачу;

2) виконувати її до успішного завершення або скасування;

3) перевірити результат;

4) повторити при необхідності з 1-го кроку.

Природно, що така модель жодним чином не структурувала процес розробки і вести мову про можливості її ефективного застосування, особливо у великих проектах, немає сенсу.

11.2.2. Водоспадна (каскадна) модель

Першою моделлю, яка отримала велику популярність і дійсно структурувала процес розробки, була *каскадна* або *водоспадна модель*. Вона була створена після конференції NATO з питань науки і техніки, яка відбулась у 1968 р., де розглядалися подібні питання. Ця модель поділяла процес створення програмного продукту на послідовні етапи (треба відзначити, що на той час вона вже застосовувалася різними розробниками ПЗ, однак ані кількість, ані зміст етапів не уніфікувався).

У 1970 р. Уінстон Ройс опублікував в IEEE WESCON статтю „Managing the Development of Large Scale Software Systems" („Керування розробкою широкомасштабних систем програмного забезпечення") [Royce, Winston, 1970]. Ця стаття, заснована на знаннях, здобутих їм у процесі керівництва великими проектами з розробки ПЗ і донині залишається найбільш цитованим джерелом з водоспадної моделі. У статті представлений глибокий і короткий виклад філософії традиційного керування розробкою ПЗ того часу, і багато з рекомендацій п'ятдесятилітньої давнини витримали перевірку часом, незважаючи на принципові зміни в технології.

Більшість праць з програмної інженерії подає водоспадну модель як основу „традиційного” процесу створення ПЗ. Ця модель передбачає послідовне виконання різних видів діяльності, починаючи з розробки вимог і закінчуючи супроводом, з чітким визначенням границь між етапами, на яких набір документів, вироблений на попередній стадії, передається як вхідні дані для наступної (рис. 11.1).

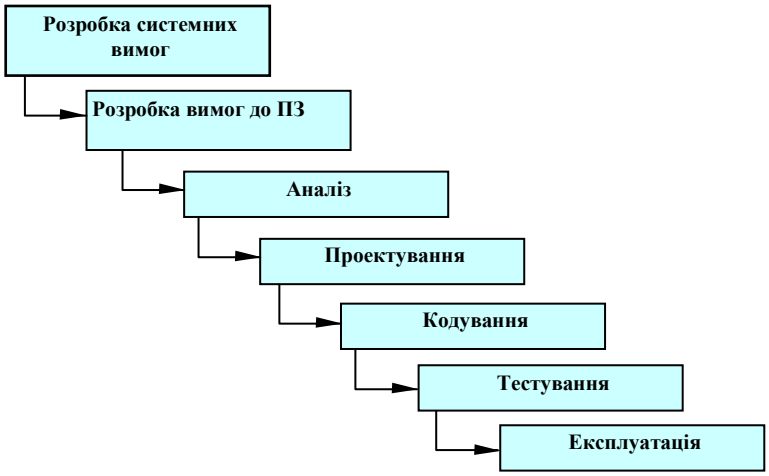


Рис. 11.1. Послідовність розробки програмного забезпечення за водоспадною моделлю

„Класична” водоспадна (каскадна) модель передбачає тільки рух вперед за цією схемою: все необхідне для проведення наступної роботи, повинне бути підготовлене в процесі попередніх робіт.

Кодування полягає в перекладі результатів проектування в текст на мові програмування.

Тестування – виконання програми для виявлення дефектів у функціях, логіці і формі реалізації програмного продукту.

Експлуатація (Супровід) – це внесення змін в експлуатоване ПЗ. Цілі змін:

– виправлення помилок;

– адаптація до змін зовнішнього для ПЗ середовища;

– удосконалення ПЗ за вимогами замовника.

Супровід ПЗ полягає у повторному застосуванні кожного з попередніх кроків (етапів) життєвого циклу до існуючої програми але не в розробці нової програми.

Водоспадна (каскадна) модель розробки ПЗ має характерні особливості:

1) Розробка комп’ютерних програм складається з двох основних етапів: аналіз і кодування.

2) Для того щоб мати можливість керувати й контролювати усією тією інтелектуальною волею, яка властива процесу розробки ПЗ, необхідно ввести „наднормативні” етапи, які включають у себе *визначення системних вимог*, *вимог до ПЗ*, *розробку програми й тестування*. Ці етапи доповнюють етапи аналізу й кодування.

3) Основний підхід, описаний водоспадною моделлю, є досить ризикованим і припускає невдале завершення. Стадія тестування, що перебуває наприкінці циклу розробки, – це перший момент, де можна визначити реальний час виконання, обсяг займаної пам’яті, швидкість введення/виведення, щоб порівняти їх зі значеннями, встановленими при аналізі. Зміни, внесені в програму, можуть виявитися настільки руйнівними, що вимоги до ПЗ, на яких ґрунтувалася розробка програми, можуть стати такими, що їх взагалі неможливо бути виконати. У такому випадку необхідно переглянути вимоги або внести істотні зміни в структуру програми.

Можна виділити п’ять умов, які необхідні для вдосконалення основного водоспадного процесу, і які дозволять виключити більшість ризиків при розробці ПЗ, що зазначені у третьому пункті. Це такі умови:

1) *Проектні рішення є первинними.* Насамперед необхідно включити стадію попереднього проектування програми між стадією формування вимог до ПЗ й стадією аналізу. Таким чином розробник програми може переконатися в тому, що вона не виявиться непридатною через зміну обсягу займаної пам’яті, часу виконання й даних. При переході до стадії аналізу програміст повинен передати аналітикові дані з обмеження за часом, пам’яті й операціям таким чином, щоб той зміг оцінити наслідки. Якщо необхідні сумарні ресурси виявляться неприпустимими або якщо в проекті з’явилися помилки, це буде встановлено на ранній стадії і ітерація з визначенням вимог і попереднім проектуванням може бути повторена до початку основного проектування, кодування й тестування.

2) *Документуйте проект.* Обсяг документації, необхідної для комп'ютерних програм, досить великий; він набагато більший того обсягу, який готові створити програмісти, аналітики або розроблювачі програм, якщо залишити їх наодинці зі своїми комп'ютерами. Навіщо ж така кількість документації? По-перше, кожний розробник повинен обмінюватися інформацією з розробниками інтерфейсів, менеджерами й, можливо, замовниками. По-друге, на ранніх стадіях документація – це і є проект. По-третє, дійсна цінність документації полягає в тому, що вона дозволяє підтримувати на більш пізніх етапах зміни, які виконуються окремою групою тестування, окремою групою супроводу й групою експлуатації, які не знають ПЗ.

3) *Виконуйте роботу двічі.* Якщо комп'ютерна програма розробляється вперше, потрібно, щоб версія, яка зрештою потрапить до замовника для реального використання, була б насправді другою, хоча б для найбільш критичних проектних/експлуатаційних рішень. Працююча над першою версією команда повинна бути особливо компетентна, що дозволить швидко виявити проблемні сторони в проекті, змоделювати їх і альтернативні їм рішення, не звертаючи уваги на очевидні аспекти розробки, які є неістотними на ранніх стадіях, і в врешті-решт отримати програму, вільну від помилок.

4) *Плануйте, контролюйте й стежте за тестуванням.* Поза всяким сумнівом, найголовнішим споживачем ресурсів, виділених на проект (людино-днів, машинного часу та/або управлінського аналізу), є стадія тестування. Ця стадія найбільш ризикована з погляду вартості виконання й дотримання графіка робіт. Вона зазвичай є завершальним етапом, коли повернутися назад і піти іншим шляхом є сумнівним рішенням (або взагалі неможливим). Три попередні рекомендації мали метою пошук і виправлення помилок до початку стадії тестування. Однак навіть при дотриманні цих рекомендацій однаково доводиться переходити до стадії тестування й однаково залишаються деякі важливі речі, які необхідно виконати, а саме:

– задіяти групу фахівців для проведення тестування, які не несуть ніякої відповідальності за вихідну розробку;

– виконати візуальні перевірки для виявлення очевидних помилок, таких як відсутність знаків „мінус", пропуск множників двійки, перехід за невірною адресою (не слід використовувати комп'ютер для виявлення подібного роду помилок, це дуже невигідно);

– протестувати всі логічні функції програми;

– зробити остаточну перевірку на комп'ютері, де буде застосовуватися програма.

Серед цих порад є як актуальні так і застарілі. Пункти 1 і 4 як і раніше залишаються важливими. Пункт 2 сьогодні є поширеним способом перевірки якості (застосування інспекцій ПЗ), але його мета у тому виді, у якому вона викладена тут, застаріла. Подібний підхід міг бути ефективним по вартості при використанні технології 1970 р., але не сучасних підходів. Комп'ютери, компілятори, аналізатори та інші інструменти є більш ефективними механізмами пошуку очевидних помилок. Що стосується пункту 3, то тестування всіх логічних функцій програми і в 1970 р. було вже досить непростим завданням навіть без урахування тих складностей, які привнесли розподілені, повторно використовувані компоненти й інші ускладнюючі фактори. І звичайно ж, це абсолютно нездійсненне для більшості сучасних систем. Особливо це стосується розподілених обчислень, для яких час є додатковим виміром, а кількість логічних функцій практично нескінченна. У сучасному процесі, тестування відбувається протягом усього життєвого циклу, що, при правильному його виконанні, вимагає набагато менше сумарних ресурсів і дозволяє знаходити помилки на більш ранніх стадіях життєвого циклу, коли ще існує можливість використання альтернативних рішень.

5) *Залучайте до роботи замовника.* З ряду причин проектні рішення є предметом, що допускають широку інтерпретацію навіть після досягнення попередньої згоди. Важливо залучити клієнта до роботи формальним шляхом для того, щоб він був залучений у розробку на ранніх її стадіях і до остаточної здачі проекту. Існують три стадії, що випливають за визначенням вимог, на яких введення в курс справи, оцінки й участь замовника можуть сприяти успіху розробки. Це „попередній огляд ПЗ ", що йде за етапом попередньої розробки програми, серія „критичних оглядів структури ПЗ" у процесі розробки програми й „остаточний огляд прийнятого ПЗ", який розпочинається після тестування.

При реальній роботі у відповідності з наведеною моделлю, яка припускає рух тільки в одну сторону, зазвичай виникають проблеми при виявленні недоробок і помилок, зроблених на ранніх етапах. Втім ще більш важко мати справу зі змінами оточення, у якому розробляється ПЗ (це можуть бути зміни вимог, зміна підрядників, зміни політик організації, які розробляють або експлуатують ПЗ, зміни галузевих стандартів, поява конкуруючих продуктів тощо).

Тому через нетривалий час після своєї появи, каскадна модель була модифікована Уінстом Ройсом з урахуванням взаємозалежності етапів і необхідності повернення на попередні ступені, що могло бути викликане, наприклад, неповнотою вимог або помилками у формуванні задачі. У такому „оборотному" вигляді каскадна модель проіснувала тривалий час і стала основою для багатьох проектів (рис.11.2).

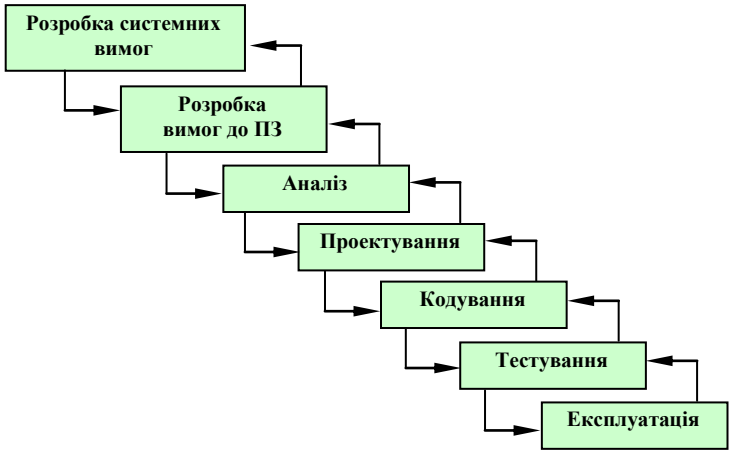


Рис. 11.2. Модифікована каскадна модель

Підсумовуючи наведене, можна зробити висновок, що працювати у відповідності з цією моделлю можна тільки у випадку, якщо вдається передбачити заздалегідь можливі перипетії хода проекту і ретельно збирати і інтегрувати інформацію на перших етапах, з тим, щоб згодом можна було користуватися їх результатами без оглядки на можливі зміни.

Практичне використання даної моделі виявило велику кількість її недоліків, головний з яких полягав у тому, що вона більше підходить для традиційних видів інженерної діяльності, чим для розробки ПЗ. Зокрема, однією з найбільших проблем виявилася її „схильність" до можливих невідповідностей отриманого продукту і вимог, які до нього виставлялися.

Головна причина цього полягає в тому, що повністю сформований продукт з'являється лише на останніх етапах розробки, але оскільки роботу на різних етапах зазвичай виконували різні фахівці і проект передавався від однієї групи до іншої, то за принципом зіпсованого телефону виявлялося так, що на виході утворювалося не зовсім те, що передбачалося спочатку (рис.11.3).

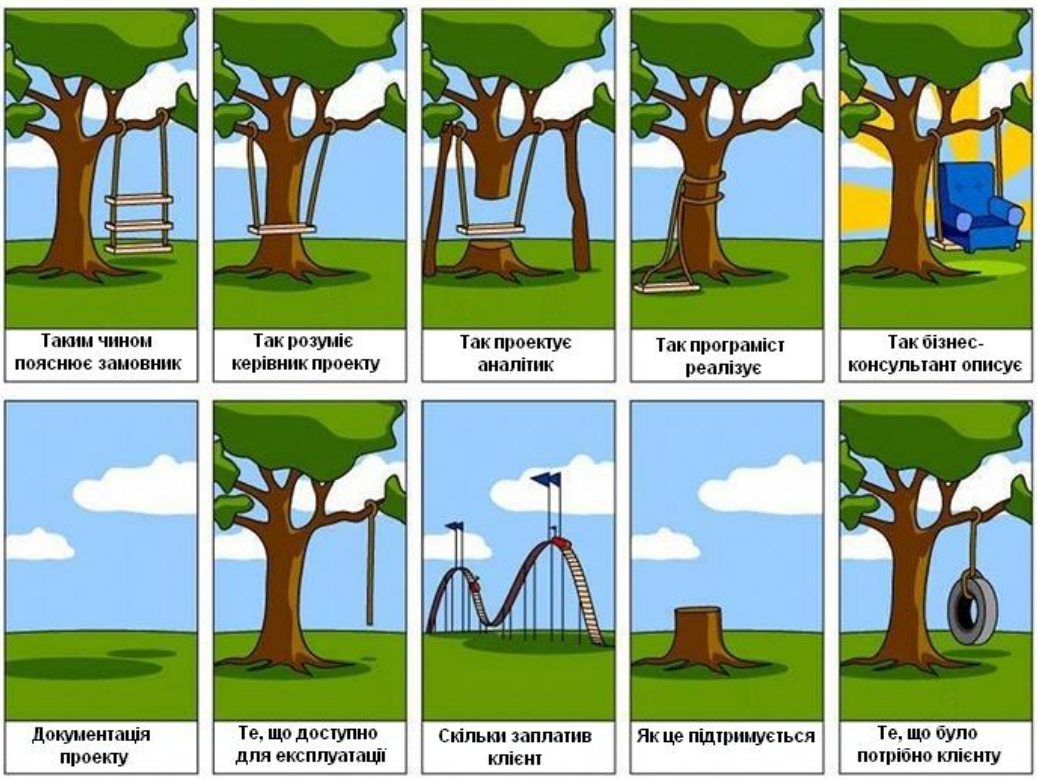


Рис.11.3. Принцип зіпсованого телефону при розробці ПЗ

11.2.3. V-подібна модель

V-подібна модель запропонована саме для того, щоб усунути недоліки каскадної моделі, а назва – V-подібна або шарнірна – з'явилася через її специфічне графічне подання (рис. 11.4). Ця модель дозволяє набагато краще контролювати результат на предмет його відповідності очікуванням, оскільки сфокусована на тестуванні.

V-подібна модель дала можливість значно підвищити якість ПЗ за рахунок своєї орієнтації на тестування, а також багато в чому вирішила проблему відповідності створеного програмного продукту висунутим вимогам завдяки процедурам верифікації і атестації на ранніх стадіях розробки (пунктирні лінії на рис. 11.4 вказують на залежність етапів планування/постановки задачі і тестування/приймання).

Однак в цілому V-подібна модель залишається усього лише модифікацією каскадної і володіє багатьма її вадами. Зокрема, і та, і інша слабо пристосовані до можливих змін вимог замовника. Якщо процес розробки ПЗ займає тривалий час (іноді до декількох років), то отриманий в результаті програмний продукт може виявитися фактично непотрібним замовнику, оскільки його потреби істотно змінилися.

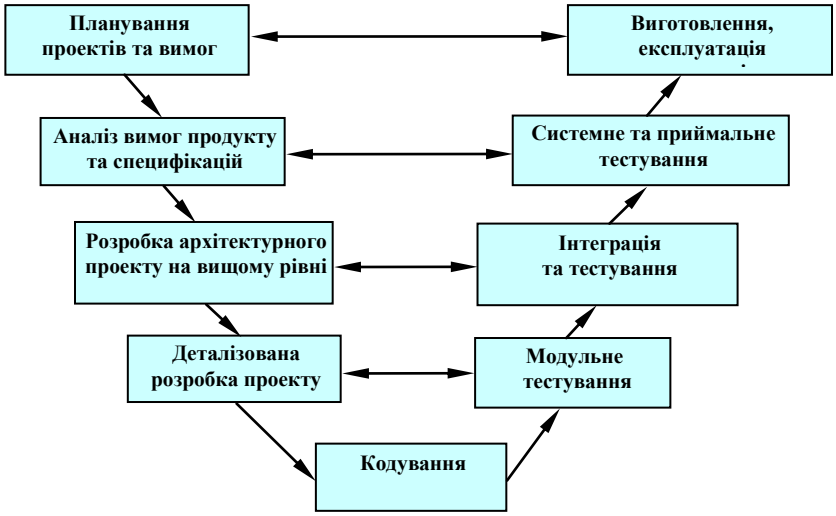


Рис. 11.4. V-подібна модель

В тій же мірі актуальним є й питання впливу науково-технічного прогресу: вимоги до ПЗ висуваються з урахуванням поточного стану наукових і практичних досягнень в галузі апаратно-програмного забезпечення, однак IT-сфера розвивається надзвичайно швидко, і тривалий процес розробки здатний привести до створення програмного продукту, який ґрунтується на застарілих технологіях і виявляється неконкурентоспроможним ще до своєї появи.

Актуальним також є питання планування показників очікуваної функціональності, оскільки в цих моделях він є не більше чим припущенням: зокрема, визначити, яку швидкість обробки даних забезпечить створюваний програмний продукт або скільки він буде займати пам'яті, на етапі постановки завдання практично неможливо. Якщо подібні вимоги чітко зафіксовані в умовах договору між замовником і виконавцем, то цілком імовірно, що отримане рішення не буде їм задовольняти, хоча відомо це стане тільки на завершальних етапах розробки, коли основні ресурси вже витрачені.

У підсумку замовник змушений буде або миритися з обмеженнями створеного на основі розглянутих моделей рішення, або додатково інвестувати кошти, щоб отримати дійсно те, що потрібно.

11.2.4. Модель на основі створення прототипів (макетів)

Оскільки перші моделі були запозичені з традиційної інженерної галузі, вони не враховували повною мірою специфіку створення ПЗ. Проте наступні моделі були вже набагато більше орієнтовані на особливості цього виду діяльності, що мав багато принципових відмінностей від конструювання предметів матеріального світу.

У зв'язку з тим що замовник найчастіше не є фахівцем в галузі ПЗ, він зазвичай погано сприймає „голі” специфікації продукту. Замовник не може сформулювати докладні вимоги з введення, обробка або виведення даних для майбутнього програмного продукту. З іншого боку, розробник може сумніватися у можливості пристосування програмного продукту під операційну систему, форму діалогу з користувачем або в ефективності реалізованого алгоритму.

Для подолання інформаційного бар'єру між замовником і розробником і зниження ризику отримання програмного продукту, що не відповідає висунутим вимогам, став застосовуватися підхід, спрямований на створення прототипів (макетів), які представляли собою повністю або частково робочі моделі готової системи. Він дозволяє значно покращити взаєморозуміння між усіма учасниками процесу за рахунок послідовного, еволюційного розвитку системи на основі ітеративного уточнення прототипів.

Основна мета макетування – зняти невизначеності у вимогах замовника.

<u>Макетування (прототипування)</u> – це процес створення моделі необхідного програмного продукту.

Модель може приймати одну з трьох форм:

1) паперовий макет або макет на основі ПК (зображує або малює людино-машинний діалог);

2) працюючий макет (виконує певну частину необхідних функцій);

3) існуюча програма (характеристики якої потім повинні бути покращені).

Як показано на рис. 11.5, макетування ґрунтується на багаторазовому повторенні ітерацій, участь в яких беруть замовник і розробник. Застосування ітерацій подібне використанню зменшених макетів будинків в архітектурі – вони дають можливість наочно представити кінцевий результат, їх побудова і зміна є набагато менш трудомісткою у порівнянні зі зведенням і зміною самого будинку.

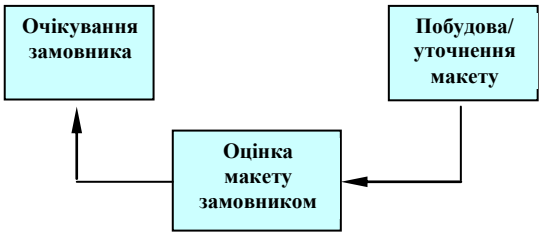


Рис. 11.5. Модель на основі створення прототипів (макетів)

Послідовність дій при макетуванні представлена на рис. 11.6.

Рис. 11.6. Послідовність дій при макетуванні

Макетування починається зі збору й уточнення вимог до створюваного ПЗ. Розробник і замовник зустрічаються і визначають усі цілі ПЗ, встановлюють, які вимоги відомі, а які потрібно довизначити. Потім виконується швидке проектування. Увага в ньому зосереджується на тих характеристиках ПЗ, які повинні бути видимі користувачу.

Швидке проектування призводить до побудови макета. Макет оцінюється замовником і використовується для уточнення вимог до ПЗ. Ітерації повторюються до тих пір, поки макет не виявить усі вимоги замовника і, тим самим, не дасть можливість розробнику зрозуміти, що повинне бути зроблене.

Переваги макетування: забезпечує визначення повних вимог до ПЗ.

Недоліки макетування:

– замовник може прийняти макет за продукт;

– розробник може прийняти макет за продукт.

Пояснимо сутність цих недоліків. Коли замовник бачить працюючу версію ПЗ, він перестає усвідомлювати, що деталі макета скріплені „жувальної гумкою і дротом”; він забуває, що в погоні за працюючим варіантом залишені невирішеними питання якості і зручності супроводу ПЗ. Коли замовнику кажуть, що продукт повинен бути перебудований, він починає обурюватися і вимагати, щоб макет „в три прийоми” був перетворений в робочий продукт. Дуже часто це негативно позначається на управлінні розробкою ПЗ.

З іншого боку, для швидкого одержання працюючого макета розробник часто йде на певні компроміси. Можуть використовуватися не самі підходящі мова програмування або операційна система. Для простої демонстрації можливостей може застосовуватися неефективний алгоритм. Через деякий час розробник забуває про причини, за якими ці засоби не підходять. В результаті далеко не ідеальний обраний варіант інтегрується в систему.

Таким чином, незважаючи не усі переваги, дана модель також не стала панацеєю. Основні її проблеми лежали в площині взаємовідношень „замовник–виконавець”: перший зацікавлений у створенні як можна більш всебічних прототипів для того, щоб знизити ризик отримання неадекватного ПЗ, одночасно для другого, кожний новий прототип означає додаткові витрати часу і ресурсів, а в підсумку – зниження рентабельності проекту.

11.2.5. Об’єктно-орієнтовані моделі

В середині 1990-х років індустрія ПЗ стала достатньо розвиненою, складні проекти успішно реалізовувалися за допомогою об’єктно-орієнтованої методології, яка набуває популярності, а команди розробників стали застосовувати підходи, засновані на використанні найбільш значимих переваг попередніх моделей.

Об’єктно-орієнтована методологія передбачає конструювання програмного рішення з готових об'єктів, для яких визначаються правила їх взаємодії, яка переводить об'єкти з одного стану в інший. Модель, що передбачає повну відповідність процесу розробки положенням об’єктно-орієнтованої методології (об’єктно-орієнтований аналіз, проектування, програмування), ефективна у великих проектах, а також там, де застосовуються так звані засоби швидкої розробки (Rapid Application Development – RAD), які засновані на цих технологіях і які містять готові бібліотеки класів.

Однак RAD-системи не сприяють створенню об’єктно-орієнтованих рішень. Програмісти, розпещені інструментарієм, який дозволяє за декілька годин створювати з готових компонентів програмні продукти, на які раніше витрачались дні і місяці роботи, вважають зайвим для себе детально вивчати методологію і UML, а вже тим більше не прагнуть оформляти свої рішення у вигляді класів, придатних для повторного використання.

Таким чином, об’єктно-орієнтована модель застосовується переважно в дуже великих проектах, де приділяється належна увага етапам аналізу і проектування, а також жорстко контролюється дотримання розробниками встановлених правил.

11.2.6. Інкрементальні моделі

Серед розробників і дослідників, які мали справу з розробкою складного ПЗ, практично с самого зародження індустрії виробництва програм велику популярність посідали інкраментальні моделі, оскільки вони володіють великою гнучкістю і спроможністю працювати в оточенні, яке постійно змінюється.

Інкрементальні моделі передбачають дроблення створюваного ПЗ на відносно незалежні складові, які розробляються і вводяться в експлуатацію по окремості. Ітеративний процес передбачає, що різні види діяльності не прив'язані намертво до певних етапів розробки, а виконуються в міру необхідності, іноді повторюються до тих пір, поки не буде отриманий потрібний результат. На першій ітерації розробляється частина системи, яка є незалежною від інших (рис.11.7).

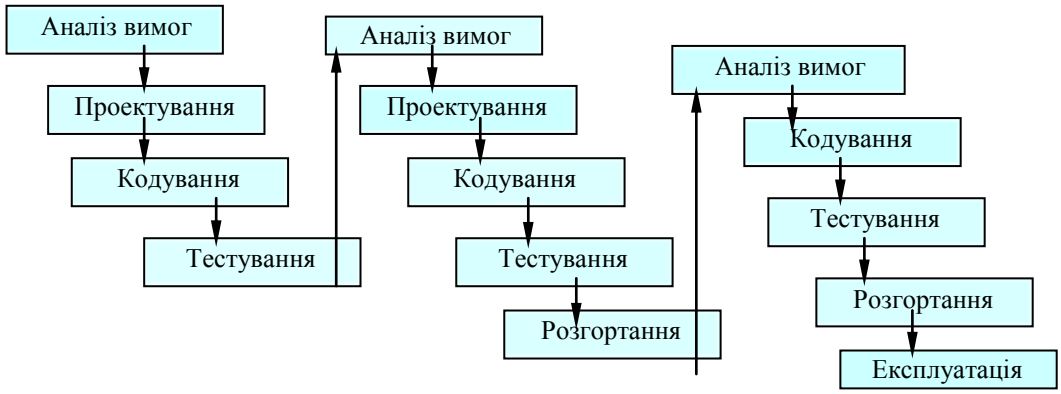


Рис. 11.7. Можливий хід робіт за інкраментальною моделлю

При цьому більша частина або навіть повний цикл робіт відбувається на першій ітерації, потім оцінюються результати і на наступній ітерації або ця частина переробляється, або розробляється наступна, яка може залежати від першої, або певним чином об'єднується доробка першої частини з додаванням нових функцій. В результаті на кожній ітерації можна аналізувати проміжні результати робіт і реакцію на них усіх зацікавлених осіб, включаючи користувачів, і вносити коригувальні зміни на наступних ітераціях. Кожна ітерація може містити повний набір видів діяльності від аналізу вимог, до введення в експлуатацію чергової частини ПЗ.

Така модель є зручною як для замовника, так і для виконавця, оскільки дозволяє просуватися вперед, забезпечуючи дотримання інтересів обох сторін. Однак у неї є свої вади. Розподіл на функціональні блоки в загальному випадку гальмує процес створення програмного продукту, оскільки виникає необхідність забезпечення взаємодії складових цього продукту. Для багатьох рішень цей метод неможливо застосувати, оскільки з них не можна вичленувати окремі складові, які можуть незалежно функціонувати. Істотно зростає навантаження і на керівний персонал у зв'язку з ускладненням задач з координування робіт над окремими складовими системи, збільшується вартість внесення змін в готові компоненти, які вже встановлені і працюють у замовника.

Разом з гнучкістю і можливістю швидко реагувати на зміни ітеративні моделі привносять додаткові складності в управління проектом і відстеження його ходу. При використанні ітеративного підходу важко адекватно оцінити поточний стан проекту і спланувати довгостроковий розвиток подій, а також передбачити терміни і ресурси, необхідні для забезпечення певної якості результату, значно складніше, чим для каскадних проектів (зрозуміло, при відсутності змін, які впливають на хід останніх).

11.2.7. Спіральна модель

Наступним еволюційним розвитком ідеї ітерацій є спіральна модель життєвого циклу ПЗ, яка була запропонована Баррі Боемом у 1988 р. [68]. Вона передбачає кожну ітерацію починати з виділення цілей і планування чергової ітерації, визначення основних альтернатив і обмежень при її виконанні, їх оцінки, а також оцінки виникаючих ризиків і визначення способів рятування від них, а закінчувати ітерацію оцінкою результатів робіт проведених в її рамках.

Спіральна модель стала істотним проривом в розумінні природи розробки ПЗ, хоча, за великим рахунком, є симбіозом двох моделей: каскадної і на основі створення прототипів (макетів).

Спіральна модель Боема сфокусована на проектуванні. Власне розробка ПЗ відбувається лише на останньому етапі за звичайною каскадною моделлю, однак цьому передує декілька ітерацій проектування на основі створення прототипів – при цьому кожна ітерація включає стадію виявлення й аналізу ризиків найбільш складних задач.

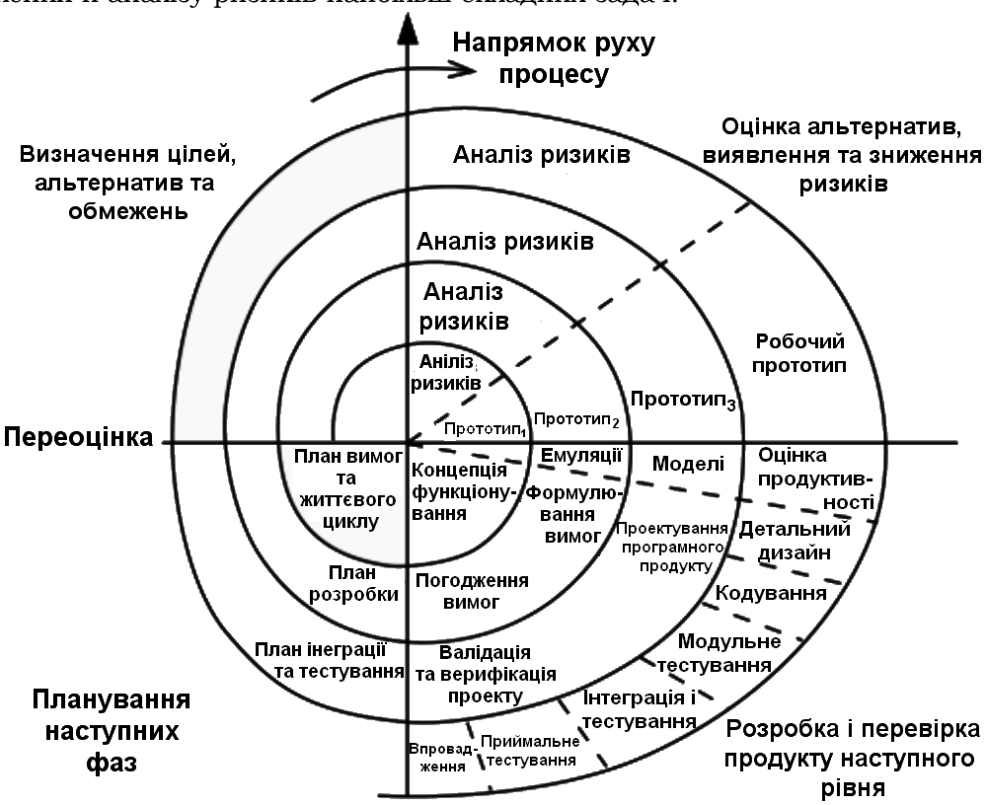


Рис. 11.8. Зображення ходу робіт за спіральною моделлю

Оскільки спіральна модель головним чином охоплює саме проектування, то у первісному вигляді вона не отримала широкого поширення, як метод управління усім життєвим циклом створення ПЗ. Однак її головна ідея, яка полягає в тому, що процес роботи над проектом може складатися з циклів, які проходять одні й ті ж етапи, послужила вихідним пунктом для подальших досліджень і стала основою більшості сучасних моделей процесу розробки ПЗ.

Основним її новим елементом є загальна структура дій на кожній ітерації – планування, визначення задач, обмежень і варіантів рішень, оцінка запропонованих рішень і ризиків, виконання основних робіт ітерації й оцінка їх результатів.

Назву спіральної ця модель одержала через зображення ходу робіт в „полярних координатах", в яких кут відповідає виконуваному етапу в рамках загальної структури ітерацій, а віддалення від початку координат – витраченим ресурсам.

Рис. 11.8. показує можливий розвиток проекту за спіральною моделлю – кількість витків, а також розташування і набір видів діяльності в правому нижньому квадранті можуть змінюватися в залежності від результатів планування й аналізу ризиків, які виконувались на попередніх етапах.

11.2.8. Ітеративна модель

Ітеративна модель вперше була запропонована Ф. Крачтеном у 1995 р. Дана модель об'єднує головні переваги каскадної, інкрементної й спіральної моделей, а також методів розробки на основі створення прототипів й об'єктно-орієнтованого підходу (рис. 11.9).

Рис.11.9. Ітеративна модель Ф. Крачтена

Вона здобула велику популярність і в тому або іншому вигляді використовується в багатьох сучасних проектах зі створення програмних продуктів.

У відповідності з ітеративною моделлю є чотири основні фази життєвого циклу розробки ПЗ (початок, дослідження, побудова й впровадження). На кожній фазі проект проходить певну кількість ітерацій, які приводять до створення працездатних версій: на початкових створюються прототипи, уточнюються вимоги, пробляються найбільш складні проблеми; кінцеві приводять до створення програмного продукту, його удосконалюванню і розширенню функціональності.

Ітеративна модель, крім основних фаз, виділяє ще дві групи процесів: робочі (управління вимогами, аналіз і проектування, реалізація, тестування, розгортання) і допоміжні (управління конфігурацією і змінами, проектом і процесом). Кількість і сутність процесів варіюються залежно від потреб розробника, вони також можуть мати свої цикли, які не обов'язково навіть відповідають основним фазам. Однак результатом робочих процесів завжди є створення версій продукту.

Ітеративна модель подібно спіральній дає можливість успішно справлятися з ризиками. Якщо під час роботи над черговою версією буде встановлено, що трудовитрати на реалізацію необхідної функціональності занадто великі, то перевищення бюджету й недодержання термінів можна буде уникнути шляхом співвіднесення пріоритетів розробки і трудовитрат на початку кожної ітерації.

Отже, дана модель добре підходить для більшості типів програмних проектів, але особливо її переваги помітні при роботі над програмними продуктами, які призначені для виходу на вільний ринок, в силу орієнтації на випуск послідовних версій.

Філіпп Крачтен тривалий час працює у фірмі Rational Software, яка сьогодні належить IBM. Саме за цією причиною ітеративна модель стала основою RUP (Rational Unified Process) – одного з найпоширеніших методів комплексного управління процесом розробки ПЗ. На її ж основі розроблений головний конкурент RUP з боку Microsoft – MSF (Microsoft Solutions Framework), а також аналогічний підхід компанії Borland – ALM (Application Lifecycle Management).

11.2.9. Уніфікований процес розробки програмного забезпечення

Уніфікований процес розробки ПЗ Rational (Rational Unified Process – RUP) є прикладом ітеративного процесу, але побудований він на підставі різних припущень про природу розробки програмного забезпечення, і, відповідно, достатньо сильно відрізняється.

Уніфікований процес розробки ПЗ є прикладом так званого „важкого" процесу, який детально описаний і який передбачає підтримку власне розробки вихідного коду ПЗ великою кількістю допоміжних дій. Прикладами таких дій є розробка планів, технічних завдань, численних проектних моделей, проектної документації тощо. Основна мета такого процесу – відокремити успішні практики розробки і супроводу ПЗ від конкретних людей, які вміють їх застосовувати. Багаточисельні допоміжні дії дають надію зробити можливим успішне вирішення задач з конструювання і підтримки складних систем за допомогою наявних працівників, які не обов'язково повинні бути суперпрофесіоналами.

Для досягнення цього виконується ієрархічний покроковий детальний опис дій, що приймаються у тій або іншій ситуації, щоб можна було навчити звичайного працівника діяти аналогічним чином. В ході проекту створюється багато проміжних документів, які дозволяють розробникам послідовно розбивати задачі, що стоять перед ними на більш прості. Ці ж документи слугують для перевірки правильності рішень, які приймаються на кожному кроці, а також відстеження загального ходу робіт й уточнення оцінок ресурсів, необхідних для одержання бажаних результатів.

Уніфікований процес розробки ПЗ є досить складною, детально проробленою ітеративною моделлю життєвого циклу ПЗ.

Історично уніфікований процес розробки ПЗ є розвитком моделі процесу розробки, прийнятій в компанії Ericsson в 70-х-80-х рр. XX ст. Ця модель була створена Джекобсоном (Ivar Jacobson), який згодом, у 1987 р., заснував власну компанію Objectory AB саме для розвитку технологічного процесу розробки ПЗ як окремого продукту, який можна було б переносити в інші організації. Після злиття Objectory з Rational у 1995 р. розробки Джекобсона були інтегровані з роботами Ройса (Walker Royce, син автора «класичної» каскадної моделі), Крухтена (Philippe Kruchten) і Буча (Grady Booch), а також з універсальною мовою моделювання (Unified Modeling Language, UML), яка розвивалася паралельно.

Уніфікований процес розробки ПЗ заснований на трьох ключових ідеях.

• Весь хід робіт спрямовується підсумковими цілями проекту, вираженими у вигляді варіантів використання (use cases) – сценаріїв взаємодії результуючої програмної системи з користувачами або іншими системами, при виконанні яких користувачі отримують значимі для них результати і послуги. Розробка починається з виділення варіантів використання і на кожному кроці контролюється ступенем наближення до їх реалізації.

• Основним рішенням, який приймається в ході проекту, є архітектура результуючої програмної системи. Архітектура встановлює набір компонентів, з яких буде побудоване ПЗ, відповідальність кожного з компонентів (тобто розв'язувані ними підзадачі в рамках загальних задач системи), чітко визначає інтерфейси, через які вони можуть взаємодіяти, а також способи взаємодії компонентів один з одним. Архітектура є одночасно основою для одержання якісного ПЗ і базою для планування робіт і оцінок проекту в термінах часу і ресурсів, необхідних для досягнення певних результатів. Вона оформляється у вигляді набору графічних моделей на мові UML.

• Основою процесу розробки є плановані і керовані ітерації, об'єм яких (функціональність і набір компонентів, які реалізуються в рамках ітерації) визначається на основі архітектури.

Уніфікований процес розробки ПЗ виділяє в життєвому циклі 4 основні фази, в рамках кожної з яких можливе проведення декількох ітерацій. Крім того, розробка системи може пройти через декілька циклів, яка включає усі 4 фази.

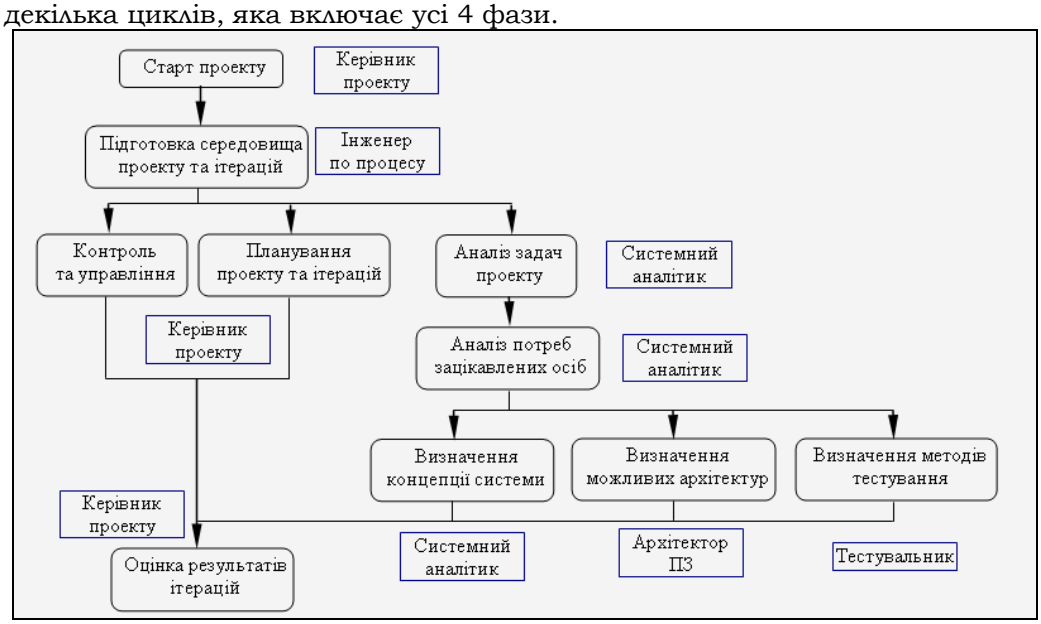


Рис. 11.10. Приклад ходу робіт на фазі початку проекту

1. *Фаза початку проекту (Inception).* Головна мета цієї фази – досягти компромісу між усіма зацікавленими особами відносно задач проекту і виділюваних на нього ресурсів.

На цій фазі визначаються основні цілі проекту, керівник проекту і бюджет проекту, основні засоби його виконання – технології, інструменти, ключові виконавці, а також відбувається, можливо, апробація обраних технологій з метою підтвердження можливості досягнення цілей за їх допомогою, складаються попередні плани проекту.

На цю фазу може витрачатись близько 10% часу і 5% трудомісткості одного циклу. Приклад ходу робіт на цій фазі показаний на рис. 11.10.

2. *Фаза проектування (Elaboration).* Головна мета цієї фази – на базі основних, найбільш істотних вимог – розробити стабільну базову архітектуру продукту, яка дозволяє вирішувати поставлені перед системою задачі і в подальшому використовується як основа розробки системи. На цю фазу може витрачатись близько 30% часу і 20% трудомісткості одного циклу. Приклад ходу робіт на цій фазі представлений на рис. 11.11.

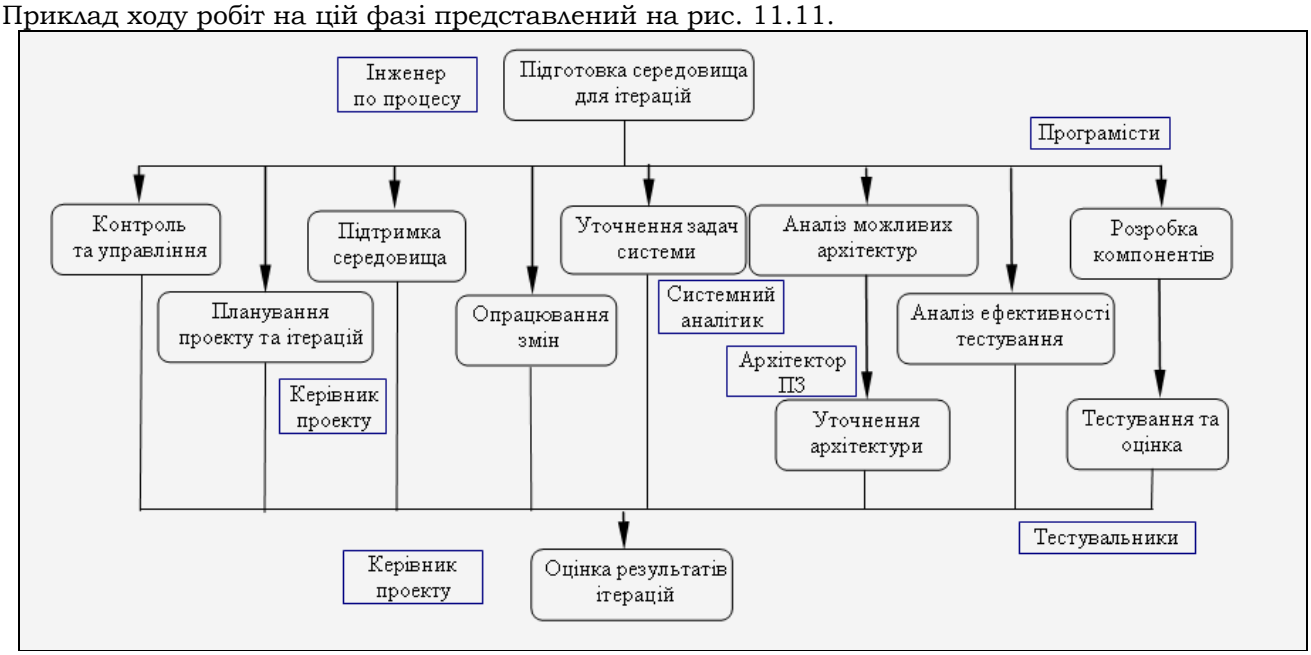


Рис. 11.11. Приклад ходу робіт на фазі проектування

3. *Фаза побудови (Construction).* Головна мета цієї фази – детальне прояснення вимог і розробка системи, яка б їм задовольняла, на основі спроектованої раніше архітектури. В результаті повинна утворитися система, яка реалізує усі виділені варіанти використання.

На цю фазу витрачається близько 50% часу і 65% трудомісткості одного циклу. Приклад ходу робіт на цій фазі представлений на рис. 11.12.

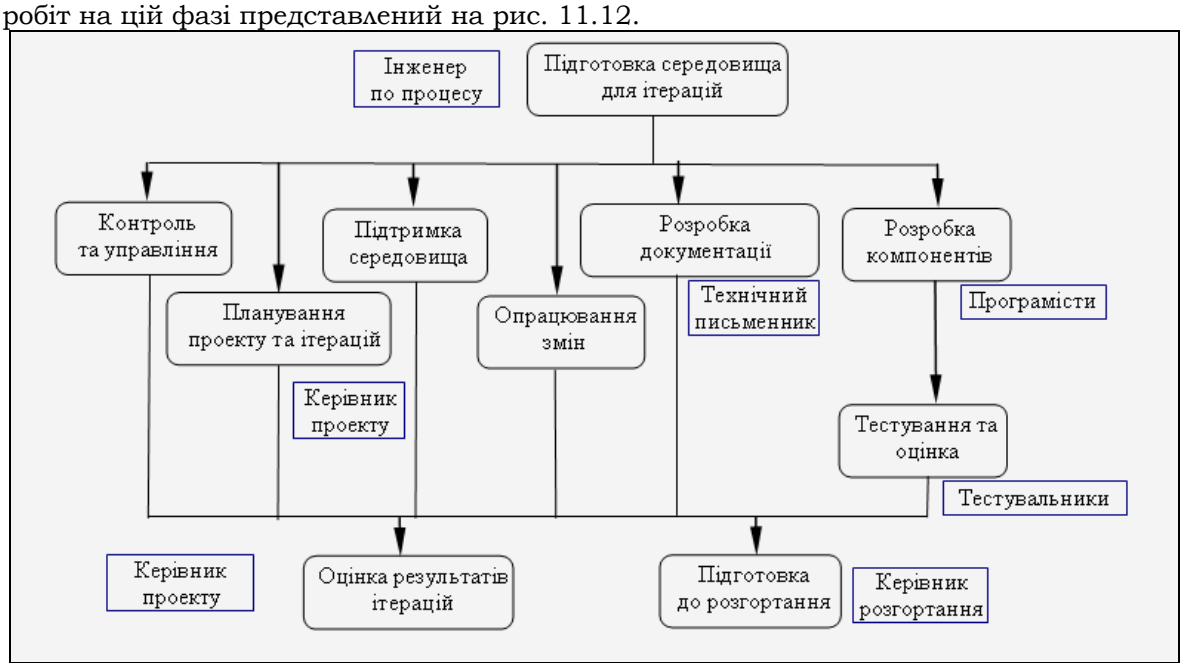


Рис. 11.12. Приклад ходу робіт на фазі побудови

4. *Фаза впровадження (Transition)*. Мета цієї фази – зробити систему цілком доступною кінцевим користувачам. На цій стадії відбувається розгортання системи в її робочому середовищі, бета-тестування, припасування дрібних деталей під потреби користувачів.

На цю фазу може витрачатись близько 10% часу і 10% трудомісткості одного циклу. Приклад ходу робіт на цій фазі представлений на рис. 11.13.

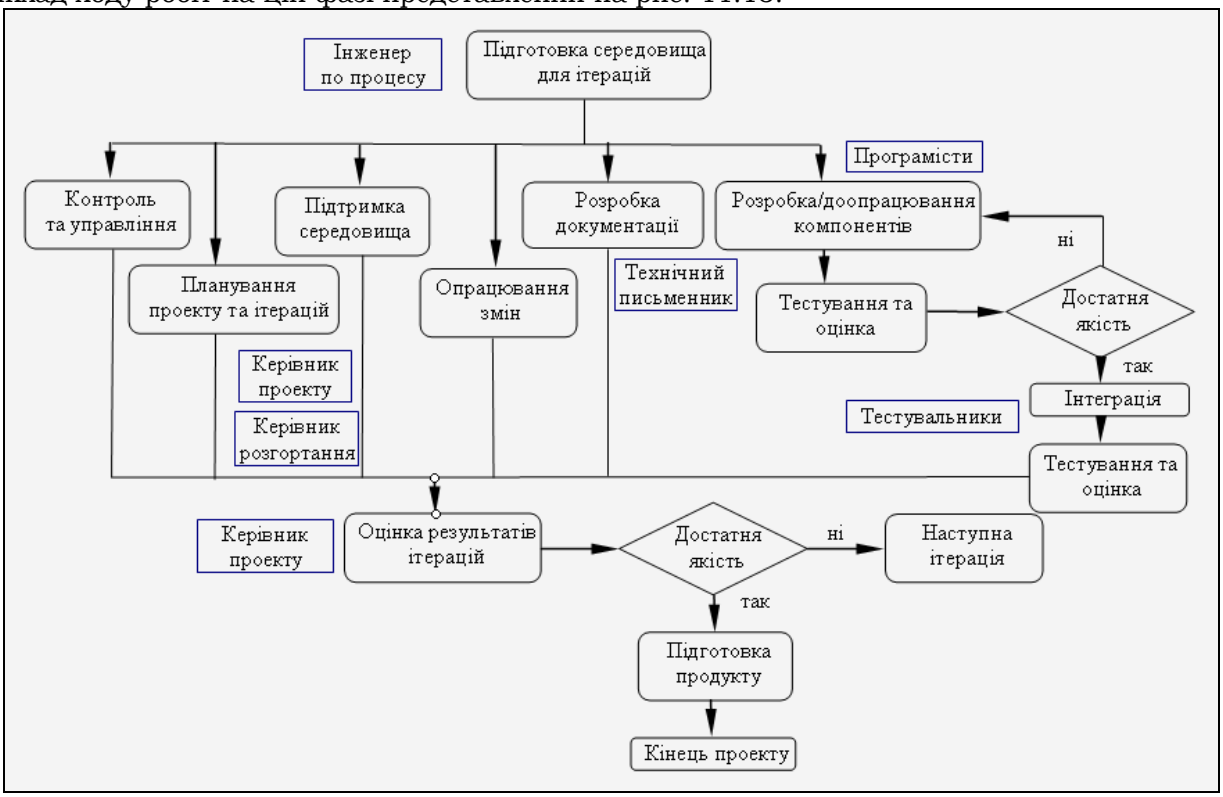


Рис. 11.13. Приклад ходу робіт на фазі впровадження

Артефакти, які виробляються в ході проекту, можуть бути представлені у вигляді баз даних таблиць з інформацією різного типу, різних видів документів, вихідного коду і об'єктних модулів, а також моделей, що складаються з окремих елементів. Основні артефакти і потоки даних між ними згідно з уніфікованим процесом розробки ПЗ зображені на рис. 11.14.

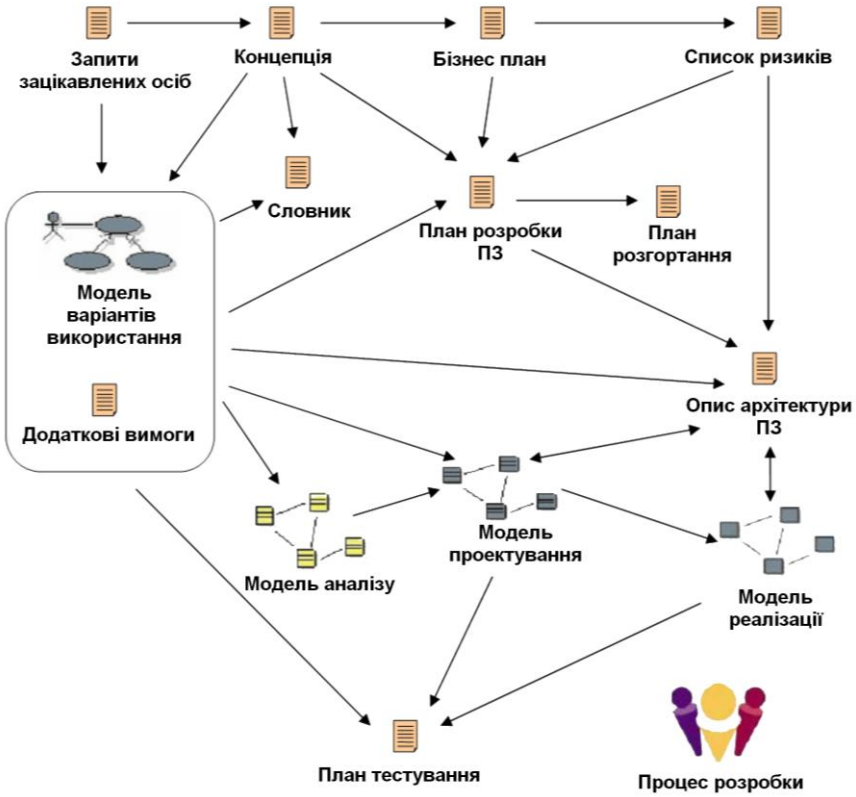


Рис. 11.14. Основні артефакти проекту за уніфікованим процесом розробки ПЗ і потоки даних між ними

Найважливішими з точки зору уніфікованого процесу розробки ПЗ артефакти проекту є моделі, які описують різні аспекти майбутньої системи. Більшість моделей представляють собою набори діаграм UML. Основні використовувані види моделей наступні.

• **Модель варіантів використання** (Use-Case Model). Ця модель визначає вимоги до ПЗ – те, що система повинна робити – у вигляді набору варіантів використання. Кожний варіант використання визначає сценарій взаємодії системи з діючими особами (actors) або ролями, які дають у підсумку значимий для них результат. Діючими особами можуть бути не тільки люди, а й інші системи, що взаємодіють з розглядуваною. Варіант використання визначає основний хід подій, які розвиваються у нормальній ситуації, а також можуть включати декілька альтернативних сценаріїв, які починають працювати тільки при специфічних умовах.

Модель варіантів використання слугує основою для проектування й оцінки готовності системи до впровадження. Прикладом варіанта використання може слугувати сценарій дій клієнта Інтернет-магазину відносно до сайту цього магазину, в результаті яких клієнт замовляє товар, наприклад, книги. Такий варіант використання можна назвати „Замовлення товару". Якщо нас цікавить сайт магазина тільки як програмна система, результатом можна вважати те, що запис про зроблене замовлення занесений до бази даних, а оператору замовлень відправлений електронний лист, який містить всю інформацію, необхідну для того, щоб можна було сформувати замовлення (рис.11.15).

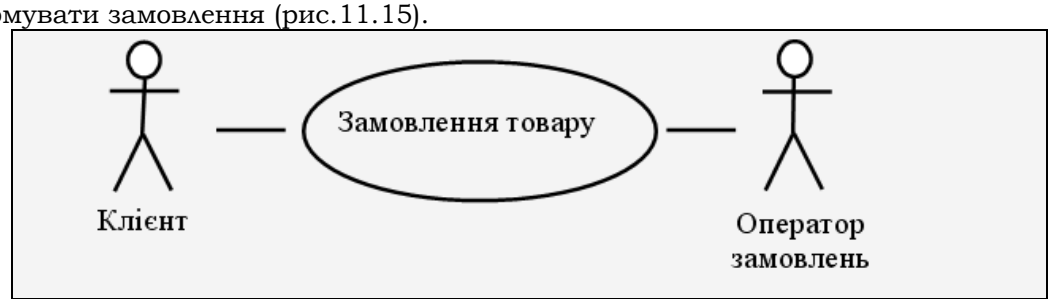


Рис. 11.15. Приклад варіанта використання і діючих осіб

До неї входить контактна інформація покупця, ідентифікатор запису про замовлення і, наприклад, список замовлених книг з їх ISBN, їх кількість для кожного найменування і номера партій для зручності їх пошуку на складі. При цьому виконання іншої частини варіанта використання – це справа інших складових системи під назвою „Інтернет-магазин". Ця робота може включати телефонний дзвінок або лист клієнту і підтвердження, що саме він зробив замовлення, питання про зручні для нього форми, час і адресу доставки і форму оплати, формування замовлення, передача його для доставки кур'єру, доставка і підтвердження одержання замовлення й оплати.

В наведеному прикладі діючими особами є клієнт, який робить замовлення, і оператор замовлень. Альтернативні сценарії в рамках даного варіанта можуть включиться, якщо, наприклад, замовленого користувачем товару немає на складі, або сам користувач перебуває на поганому рахунку в магазині, не оплативши певні попередні замовлення, або навпаки, він є привілейованим клієнтом або представником великої організації.

• **Модель аналізу** (Analysis Model). Ця модель включає основні класи, необхідні для реалізації виділених варіантів використання, а також можливі зв'язки між класами. Виділювані класи розбиваються на три різновиди – *інтерфейсні*, *керуючі* і *класи даних*. Ці класи представляють собою набір сутностей, в термінах яких робота системи повинна подаватись користувачам. Вони є поняттями, за допомогою яких достатньо зручно пояснювати собі й іншим що відбувається всередині системи, не занадто вдаючись в подробиці.

Інтерфейсні класи (boundary classes) відповідають пристроям або способам обміну даними між системою і її оточенням, у тому числі користувачами.

Класи даних (entity classes) відповідають наборам даних, які описують деякі однотипні сутності усередині системи. Ці сутності є абстракціями представлень користувачів про дані, з якими працює система.

Керуючі класи (control classes) відповідають алгоритмам, які реалізують певні значимі перетворення даних в системі і керуючим обміном даними з її оточенням в рамках варіантів використання.

В наведеному прикладі з Інтернет-магазином можна було б виділити наступні класи в моделі аналізу: інтерфейсний клас, який надає інформацію про товар і можливість зробити замовлення, інтерфейсний клас, який надає повідомлення оператору, керуючий клас, який обробляє введену користувачем інформацію і перетворюючий її в дані про замовлення і повідомлення оператору, клас даних про замовлення. Відповідна модель приведена на рис. 11.16.

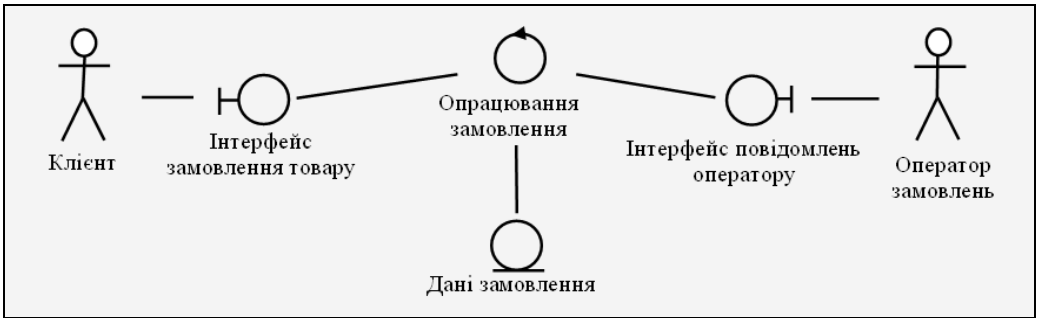


Рис.11.16. Приклад моделі аналізу для одного варіанта використання

Кожен клас може відігравати декілька ролей в реалізації одного або декількох варіантів використання. Кожна роль визначає його обов'язки і властивості, і також є частиною моделі аналізу.

В рамках інших підходів модель аналізу часто називається концептуальною моделлю системи. Вона складається з набору класів, які спільно реалізують усі варіанти використання і які слугують основою для розуміння поведінки системи і пояснення його усім зацікавленим особам.

• **Модель проектування** (Design Model). Модель проектування є деталізацією і спеціалізацією моделі аналізу. Вона також складається з класів, однак більш чітко визначених, з більш точним і детальним розподілом обов'язків, на відміну від класів моделі аналізу. Класи моделі проектування повинні бути спеціалізовані для конкретної використаної платформи (яка включає операційні системи усіх залучених машин, використовувані мови програмування, інтерфейси і класи конкретних компонентних середовищ, таких як J2EE, .NET, COM або CORBA, інтерфейси обраних для використання систем управління базами даних, СУБД, наприклад, Oracle або MS SQL Server, використовувані бібліотеки розробки користувацького інтерфейсу, такі як *swing* або *swt* в Java, *MFC* або *gtk*, інтерфейси взаємодіючих систем тощо).

В наведеному прикладі, перед усім, необхідно деталізувати класи, уточнити їх функціональність. Наприклад, для того, щоб клієнту було зручніше замовляти товар, потрібно надати йому список наявних товарів, певні способи навігації і пошуку у цьому списку, а також детальну інформацію про товар. Це означає, що інтерфейс замовлення товару реалізується у вигляді набору класів, які надають, наприклад, різні сторінки сайту магазину. Точно так же дані замовлення повинні бути деталізовані у вигляді декількох таблиць в СУБД, які зазвичай включають, дані самого замовлення (дату, посилання на дані клієнта, рядки з кількістю окремих товарів і посиланнями на товари), дані товарів, клієнта тощо. Крім того, для реляційної СУБД знадобляться класи-посередники між її таблицями і об'єктною структурою іншої програми. Оброблювач замовлення може бути реалізований у вигляді набору об'єктів декількох класів, наприклад, з виділеним окремо набором часто змінюваних політик (знижки на певні категорії товарів і певним категоріям клієнтів, сезонні знижки, рекламні комплекти тощо) і більш стабільним загальним алгоритмом обробки. Далі, прийнявши, наприклад, рішення реалізовувати систему за допомогою технологій J2EE або .NET, ми тим самим визначаємо додаткові обмеження на структуру класів, так і на саму їх кількість.

• **Модель реалізації** (Implementation Model). Під моделлю реалізації в рамках RUP і UML мають на увазі набір компонентів результуючої системи і зв'язків між ними. Під компонентом тут мається на увазі компонент складання – мінімальна за розмірами частина коду системи, яка може брати участь або не брати участь у даній конфігурації, одиниця конфігураційного управління. Зв'язки між компонентами розгортання представляють собою залежності між ними. Якщо компонент залежить від іншого компонента, він не може бути поставлений окремо від нього. Часто компоненти представляють собою окремі файли з вихідним кодом.

• **Модель розгортання** (Deployment Model). Модель розгортання представляє собою набір вузлів системи, які є фізично окремими пристроями, здатними обробляти інформацію – серверами, робочими станціями, принтерами, контролерами датчиків тощо, зі зв'язками між ними, утвореними різного роду мережними з'єднаннями. Кожний вузол може бути навантажений певною множиною компонентів, визначених в моделі реалізації.

Мета побудови моделі розгортання – визначити фізичне положення компонентів розподіленої системи, які забезпечують виконання нею потрібних функцій в тих місцях, де ці функції будуть доступні і зручні для користувачів.

В наведеному прикладі Web-сайта магазину вузлами системи є один або декілька комп'ютерів, на яких розгорнуті Web-сервер, який пересилає за запитом користувача текст потрібної сторінки, набір програмних компонентів, які відповідають за генерацію сторінок, обробку дій користувача і взаємодію з базою даних, і СУБД, в рамках якої працює база даних системи. Крім того, в систему входять усі комп'ютери клієнтів, на яких працює Web-браузер, і який робить можливим перегляд сторінок сайту і пересилання кодованих дій користувача для їх обробки.

• **Модель тестування** (Test Model або Test Suite). В рамках цієї моделі визначаються тестові варіанти або тестові приклади (test cases) і тестові процедури (test scripts). Перші є визначеними сценаріями роботи одного або декількох діючих осіб з системою, яка розгортається в рамках одного з варіантів використання. Тестовий варіант включає, крім вхідних даних на кожному кроці, де вони можуть бути введені, умови виконання окремих кроків і коректні відповіді системи для будь-якого кроку, на якому можна спостерігати відповідь системи. На відміну від варіантів використання, в тестових варіантах чітко визначені вхідні дані, і, відповідно, тестовий варіант або взагалі не має альтернативних сценаріїв, або може передбачати альтернативний порядок дій, якщо система може вести себе недетерміновано і надавати різні результати у відповідь на одні й ті ж дії. Усі інші альтернативи зазвичай закінчуються винесенням вердикту про некоректну роботу системи.

Тестова процедура представляє собою спосіб виконання одного або декількох тестових варіантів і їх складових елементів (окремих кроків і перевірок). Це може бути інструкція з ручного виконання вхідних дій, які входять в тестовий варіант або програмний компонент, який автоматизує запуск тестів.

Для виділеного варіанта використання „Замовлення товару” можна визначити наступні тестові варіанти:

– замовити один з наявних на складі товарів і перевірити, що повідомлення про це замовлення надійшло оператору;

– замовити велику кількість товарів і перевірити, що все працює на належному рівні;

– замовити відсутній на складі товар і перевірити, що у відповідь надходить повідомлення про його відсутність;

– зробити замовлення від імені користувача, занесеного до „чорного списку” і перевірити, що у відповідь надходить повідомлення про несплачені колишніх замовленнях.

Уніфікований процес розробки ПЗ також визначає дисципліни, які включають різні набори діяльностей, які в різних комбінаціях і з різною інтенсивністю, виконуються на різних фазах. В документації по процесу кожна дисципліна супроводжується досить великою діаграмою, яка пояснює дії, які потрібно виконати в ході робіт в рамках даної дисципліни, артефакти, з якими потрібно мати справу, і ролі залучених у ці дії осіб.

• **Моделювання предметної області** (бізнес-моделювання, Business Modeling). Задачі цієї діяльності – зрозуміти предметну область або бізнес-контекст, у яких повинна буде працювати система і переконатися, що всі зацікавлені особи розуміють його однаково, усвідомити наявні проблеми, оцінити їх можливі рішення та їх наслідки для бізнесу організації, у якій буде працювати система.

В результаті моделювання предметної області повинна з'явитися її модель у вигляді набору діаграм класів (об'єктів предметної області) і діяльностей (які представляють бізнес-операції і бізнес-процеси). Ця модель слугує основою моделі аналізу.

• **Визначення вимог** (Requirements). Задачами цього етапу є розуміння того, що повинна робити система, і переконатися у взаєморозумінні з цього приводу між зацікавленими особами, визначити границі системи і основу для планування проекту і оцінок витрат ресурсів у ньому. Вимоги прийнято фіксувати у вигляді моделі варіантів використання.

• **Аналіз і проектування** (Analysis and Design). Задачами цього етапу є вироблення архітектури системи на підставі вимог, переконатися в тому, що дана архітектура може бути основою працюючої системи в контексті її майбутнього використання.

В результаті проектування повинна з'явитися модель проектування, яка включає діаграми класів системи, діаграми її компонентів, діаграми взаємодій між об'єктами в ході реалізації варіантів використання, діаграми станів для окремих об'єктів, і діаграми розгортання.

• **Реалізація** (Implementation). Задачі – визначити структуру вихідного коду системи, розробити код її компонентів і протестувати їх, інтегрувати систему в працююче ціле.

• **Тестування** (Test). Задачі – знайти і описати дефекти системи (прояв недоліків її якості), оцінити її якість в цілому, оцінити чи виконані гіпотези, які лежать в основі проектування, оцінити ступінь відповідності системи вимогам.

• **Розгортання** (Deployment). Задачі – встановити систему в її робочому оточенні і оцінити її працездатність на тому місці, де вона повинна буде працювати.

• **Управління конфігураціями і змінами** (Configuration and Change Management). Задачі – визначення елементів, які підлягають збереженню в репозиторії проекту і правил побудови з них узгоджених конфігурацій, підтримка цілісності поточного стану системи, перевірка узгодженості змін, які вносяться.

• **Управління проектом** (Project Management). Задачі – планування, управління персоналом, забезпечення взаємодії на благо проекту між усіма зацікавленими особами, управління ризиками, відстеження поточного стану проекту.

• **Управління середовищем проекту** (Environment). Задачі – підстроювання процесу під конкретний проект, вибір і заміна технологій і інструментів, що використовуються в проекті.

Перші п’ять дисциплін вважаються робочими, інші – підтримуючими. Розподіл об’ємів робіт за дисциплінами в ході проекту виглядає згідно керівництву з уніфікованого процесу розробки ПЗ приблизно так, як це представлено на рис. 11.17.

Насамкінець перерахуємо методики, які використовуються в уніфікованому процесі розробки ПЗ згідно [51].

• Розробка концепції проекту (project vision) на його початку для чіткої постановки задач.
• Управління за планом.
• Зниження ризиків і відстеження їх наслідків, щонайшвидшого раннього початку робіт з подолання ризиків.
• Ретельне економічне обґрунтування усіх дій – виконується тільки те, що потрібно замовнику.
• Щонайшвидше формування базової архітектури.
• Використання компонентної архітектури.
• Прототипування, інкрементна розробка і тестування.
• Регулярні оцінки поточного стану.
• Управління змінами, постійне відпрацьовування змін поза проектом.
• Націленість на створення продукту, працездатного в реальному оточенні.
• Націленість на якість.
• Адаптація процесу під потреби проекту.

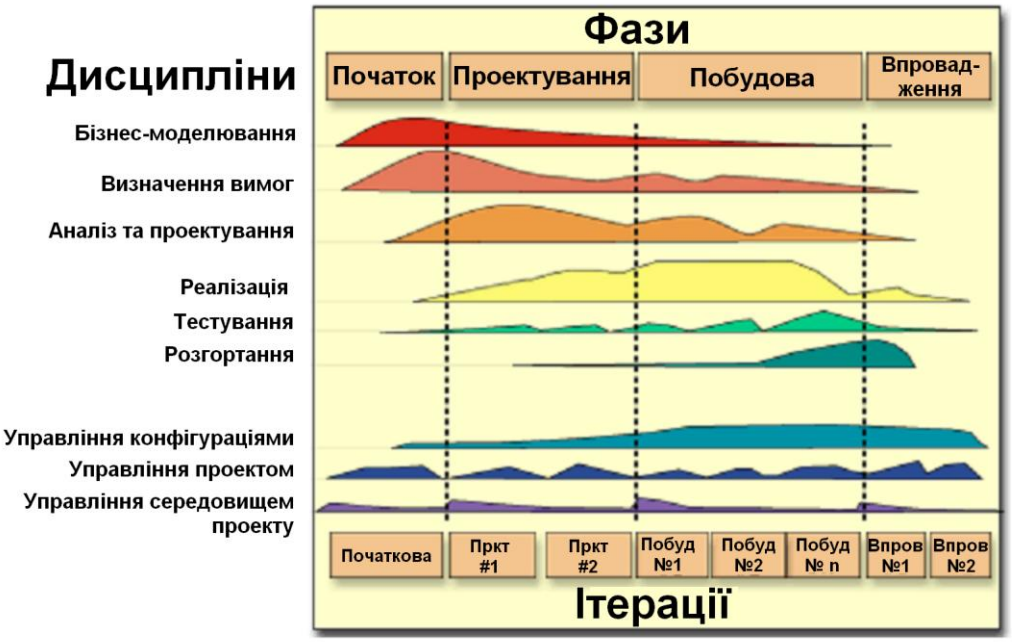


Рис. 11.17. Розподіл робіт між різними дисциплінами в проекті за уніфікованим процесом розробки ПЗ

11.2.10. Екстремальне програмування

Величезна кількість обмежень в сучасних методологіях створення ПЗ призвели до того, що компанії-розробники багато в чому стали схожі на велетенські бюрократичні системи. Наявність великої кількості формальних процедур і правил істотно звужує свободу дій кожного конкретного програміста, перетворила його в гвинтик у величезній і неповороткій машині. Незважаючи на те що подібні „машини” спроможні цілком успішно справлятися з задачами, що стоять перед ними, зазвичай їх ККД досить низький і питома продуктивність окремого розробника настільки мала, що нормальним може вважатися написання програмістом декількох рядків коду в день.

Кинути виклик подібним перевантаженим формальностями підходам й покликані моделі швидкої розробки, такі, як, наприклад, екстремальне програмування.

Сутність екстремального програмування полягає у відмові від усього зайвого, що не відноситься безпосередньо до створення якісного програмного продукту, а за основу беруться лише найефективніші методи створення ПЗ. Особлива увага приділяється питанням взаємодії з замовником, організації продуктивної роботи і тестуванню.

Більшість ідей швидкої розробки не були чимось новим, наприклад юніт-тести до цього вже застосовувалися в багатьох проектах, однак зібрані водночас і які стали обов'язковими для застосування, вони набули позитивного ефекту.

В сучасних умовах швидка розробка – це дуже модний підхід, і її використовують все активніше. Основна перевага полягає в тому, що порівняно невеликі групи розробників

спроможні справлятися з проектами за той же час, який необхідний при застосуванні більш традиційних методів командами на порядок більшої чисельності.

Однак тут є й свої недоліки, зокрема швидка розробка погано підходить для великих проектів і орієнтована головним чином на невеликі і середні, крім того, її ефективне використання можливе тільки за умови, що розробники ПЗ мають досить високу кваліфікацію і значний досвід.

Екстремальне програмування представляє так звані „живі" (agile) методи розробки, які роблять акцент на використанні пристойних розробників, а не добре відлагоджених процесів розробки. Живі методи уникають фіксації чітких схем дій, для забезпечення більшої гнучкості в кожному конкретному проекті, а також виступають проти розробки додаткових документів, які не вносять безпосереднього вкладу в створення готової працюючої програми.

Екстремальне програмування (Extreme Programming, XP) виникло як еволюційний метод розробки ПЗ „знизу-нагору". Цей підхід є прикладом так званого метода „живої" розробки (Agile Development Method). До групи „живих" методів входять, крім екстремального програмування, методи SCRUM, DSDM (Dynamic Systems Development Method, метод розробки динамічних систем), Feature-Driven Development (розробка, яка керується характеристиками результату) тощо.

Основні принципи „живої" розробки ПЗ зафіксовані в маніфесті [43], який з'явився у 2000 р.

• Люди, які беруть участь в проекті, і їх спілкування є більш важливими чинниками, чим процеси і інструменти.

• Працююча програма є більш важливою, чим вичерпна документація.

• Співробітництво з замовником є більш важливим, чим обговорення деталей контракту.

• Відпрацьовування змін є більш важливим, чим наслідування планів.

„Живі" методи дозволяють більшу частину зусиль розробників зосередити власне на задачах розробки і задоволення реальних потреб користувачів. Відсутність великої кількості документів і необхідності підтримувати їх у систематичному стані дозволяє більш швидко і якісно реагувати на зміни в вимогах і в оточенні, в якому доведеться працювати майбутній програмі.

Незважаючи на це, екстремальне програмування має свою схему процесу розробки (хоча широко використовуване розуміння „процесу розробки", як достатньо жорсткої схеми дій, суперечить ідеї „жвавості" розробки), наведену на рис. 11.18.

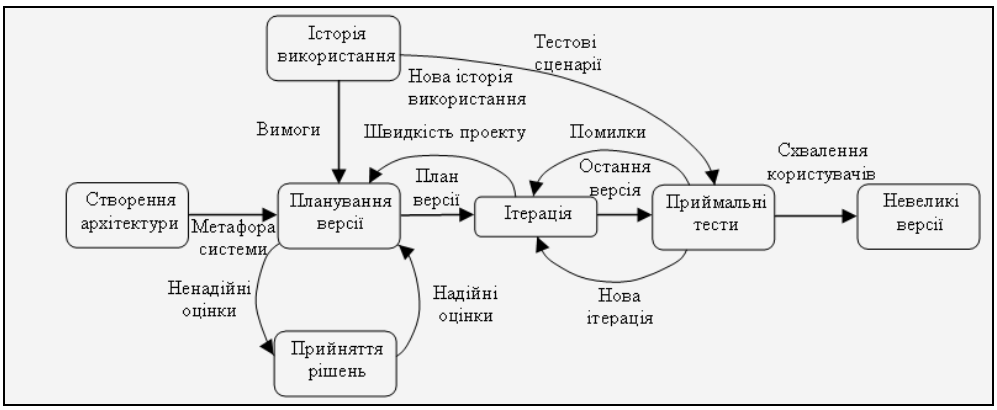


Рис. 11.18. Схема потоку робіт при екстремальному програмуванні

За твердженнями авторів екстремального програмування, ця методика представляє собою не стільки слідкування якимось загальним схемам дій, скільки застосуванню комбінації нижченаведених методик. При цьому кожна методика є важливою, і без її використання розробка вважається такою, яке не відповідає ідеї екстремального програмування, згідно твердженню Кента Бека, Kent Beck [42], одного з авторів цього підходу, разом з Уордом Каннінгемом (Ward Cunningham), і Роном Джефрісом (Ron Jeffries).

• **Живе планування** (planning game). Його задача щонайшвидше визначити обсяг робіт, який потрібно зробити до наступної версії ПЗ. Рішення ухвалюється, в першу чергу, на основі пріоритетів замовника (тобто його потреб, того, що потрібно йому від системи для більш успішного ведення свого бізнесу) і, в другу, на основі технічних оцінок (тобто оцінок трудомісткості розробки, сумісності з іншими елементами системи тощо). Плани змінюються, як тільки вони починають розходитися з дійсністю або побажаннями замовника.

• **Часта зміна версій** (small releases). Сама перша працююча версія повинна з'явитися щонайшвидше, і тут же повинна відразу використовуватися. Наступні версії готуються через достатньо короткі проміжки часу (від декількох годин при невеликих змінах і невеликій програмі, до місяця-двох при серйозній переробці великої системи).

• **Метафора** (metaphor) **системы**. Метафора в достатньо простому і зрозумілому команді вигляді повинна описувати основний механізм роботи системи. Це поняття нагадує архітектуру, але повинне бути значно простіше, найчастіше усього у вигляді однієї-двох фраз описувати основну сутність прийнятих технічних рішень.

• **Прості проектні рішення** (simple design). У кожний момент часу система повинна бути сконструйована настільки просто, наскільки це можливо. Не потрібно додавати функції заздалегідь – тільки після явного прохання про це. Вся зайва складність видаляється, як тільки виявляється.

• **Розробка на основі тестування** (test-driven development). Розробники спочатку пишуть тести, потім намагаються реалізувати свої модулі так, щоб спрацьовували тести. Замовники заздалегідь пишуть тести, які демонструють основні можливості системи, щоб можна було побачити, що система дійсно запрацювала.

• **Постійна переробка** (refactoring). Програмісти постійно переробляють систему для усунення надмірної складності, збільшення зрозумілості коду, підвищення його гнучкості, але без змін в його поведінці, що перевіряється прогоном після кожної переробки тестів. При цьому перевага віддається більш елегантним і гнучким рішенням, порівняно з такими, які просто дають потрібний результат. Невдало перероблені компоненти повинні виявлятися при виконанні тестів і відкочуватися до останнього цілісного стану (разом з залежними від них компонентами).

• **Програмування парами** (pair programming). Кодування виконується двома програмістами на одному комп'ютері. Об'єднання в пари довільне і змінюється від задачі до задачі. Той, в чиїх руках клавіатура, намагається найкращим способом розв'язати поточну задачу. Другий програміст аналізує роботу першого і дає поради, обмірковує наслідки тих або інших рішень, нові тести, менш прямі, однак більш гнучкі рішення.

• **Колективне володіння кодом** (collective ownership). У будь-який момент будь-який член команди може змінити будь-яку частину коду. Ніхто не повинен виділяти свою власну область відповідальності, вся команда цілком відповідає за весь код.

• **Постійна інтеграція** (continuous integration). Система збирається і проходить інтеграційне тестування щонайчастіше, по декілька разів на день, кожного разу, коли пара програмістів закінчує реалізацію чергової функції.

• **40-годинна робочий тиждень**. Понадурочна робота розглядається як ознака великих проблем в проекті. Не допускається понаднормова робота 2 тижні підряд – це виснажує програмістів і робить їх роботу значно менш продуктивною.

• **Включення замовника в команду** (on-site customer). В складі команди розробників постійно перебуває представник замовника, який є доступним протягом усього робочого дня і здатний дати відповіді на всі питання про систему. Його обов'язком є достатньо оперативні відповіді на питання будь-якого типу, що стосуються функцій системи, її інтерфейсу, потрібної продуктивності, правильної роботи системи в складних ситуаціях, необхідності підтримувати зв'язок з іншими додатками тощо.

• **Використання коду як засобу комунікації**. Код розглядається як найважливіший засіб спілкування всередині команди. Ясність коду – один з основних пріоритетів. Дотримання стандартів кодування, які забезпечують таку ясність, є обов’язковою. Такі стандарти, крім ясності коду, повинні забезпечувати мінімальність виразів (заборона на дублювання коду і інформації) і повинні бути прийняті усіма членами команди.

• **Відкритий робочий простір** (open workspace). Команда розміщується в одному, доволі просторому приміщенні, для спрощення комунікації і можливості проведення колективних обговорень при плануванні й прийнятті важливих технічних рішень.

• **Зміна правил за необхідності** (just rules). Кожний член команди повинен прийняти перераховані правила, але при виникненні необхідності команда може їх поміняти, якщо усі її члени пришли до згоди з приводу цієї зміни.

Як свідчать застосовувані методики, екстремальне програмування розраховане на використання в рамках невеличких команд (не більше 10 програмістів), що підкреслюється й авторами цієї методики, більша чисельність команди руйнує необхідну для успіху простоту комунікації і робить неможливим застосування багатьох перерахованих прийомів.

Перевагами екстремального програмування, якщо його вдається застосувати, є велика гнучкість, можливість швидко й акуратно вносити зміни в ПЗ у відповідь на зміни вимог і окремі побажання замовників, висока якість отримуваного в результаті коду і відсутність необхідності переконувати замовників в тому, що результат відповідає їх очікуванням.

Недоліками цього підходу є неможливість виконання у такому стилі достатньо великих і складних проектів, унеможливлення планування термінів і трудомісткість проекту на достатньо довгу перспективу і чітко спрогнозувати результати довготривалого проекту в термінах співвідношення якості результату і витрат часу і ресурсів. Також можна відзначити непристосованість екстремального програмування для тих випадків, в яких можливі рішення знаходяться не відразу на основі апріорного досвіду, а вимагають проведення попередніх досліджень екстремального програмування як сукупності описаних методик вперше було

використано в ході роботи на проектом C3 (Chrysler Comprehensive Compensation System, розробка системи обліку виплат працівникам компанії Daimler Chrysler). З 20-ти учасників цього проекту 5 (у тому числі згадані вище 3 основних автора екстремального програмування) опублікували ще під час самого проекту і в подальшому 3 книги і велику кількість статей, присвячених екстремальному програмуванню. Цей проект неодноразово згадується в різних джерелах як приклад використання цієї методики.

11.2.11. Адаптовані і комбіновані моделі

В процесі еволюції моделей життєвого циклу розробки ПЗ нові ідеї не підмінюють старі цілком і повністю. Більш правильно вважати, що кожна з них має власну сферу застосування. Крім того, у кожному конкретному випадку може виявитися, що в загальному випадку не існує методики, яка б ідеально підійшла для розв'язку даної задачі. У цьому випадку менеджерам програмних проектів треба розглянути варіанти адаптації моделей під конкретні потреби або застосовувати комбіновані методи, які включають елементи різних підходів. Наприклад, успіх швидкої розробки призвів до того, що більш консервативні моделі запозичили найефективніші її прийоми і стали використовувати їх вже в рамках своїх процесів.

Питання для самоконтролю

1. *В чому полягає сутність моделі „кодування–усунення помилок”?*
2. *В чому полягає сутність „водоспадної” моделі?*
3. *В чому полягає сутність модифікованої „водоспадної” моделі?*
4. *В чому полягає сутність V–моделі?*
5. *В чому полягає сутність моделі на основі створення прототипів?*
6. *В чому полягає сутність спіральної моделі?*
7. *В чому полягає сутність ітеративної моделі?*
8. *В чому полягає сутність уніфікованого процесу розробки ПЗ?*
9. *В чому полягає сутність екстремального програмування?*

12. ОСНОВИ ПРОГРАМУВАННЯ В СЕРЕДОВИЩІ TURBO-PASCAL

Обчислення на комп'ютері здійснюються за допомогою виконання послідовності команд. Комп'ютер „розуміє" тільки команди на спеціальній машинній мові (машинний код). Для зручності спілкування користувача з комп'ютером розроблені мови програмування (мови високого рівня). Всі вони зрозумілі людині і будуються на базі англійської мови. Одним з найпопулярніших мов є Pascal. Вона первісно була задумана для студентів і є достатньо простою в освоєнні через інтуїтивну зрозумілість її конструкцій. Сьогодні ця мова використовується інженерами і науковими співробітниками.

Вивчення мови програмування представляє собою знайомство з формальними правилами запису алгоритмів для їх наступного виконання комп'ютером. Ця формальність утворюється з самих принципів, закладених в архітектуру обчислювальних пристроїв і жорсткості математичної логіки. Тому, докладіть зусиль сприйняти усі доволі суворі правила як неминучість, настроїти себе на серйозну, скрупульозну, часами складну роботу. Однак не потрібно лякатися, розстроюватись і нарікати на долю: трохи акуратності, уваги, знання попереднього матеріалу – і ви вже програміст.

12.1. Загальні відомості по мову програмування Pascal (Turbo Pascal)

Для того, щоб комп'ютер „зрозумів" вказівки програміста, він повинен „спілкуватись" із програмістом на мові, яку називають *мовою програмування.*

Комп'ютерна програма – це набір інструкцій, які вказують ЕОМ послідовність дій з розв'язання певної поставленої перед ним задачі.

Сьогодні у світі існують сотні різних мов програмування, які в загальному випадку можна поділити на мови програмування низького (Asembler) та високого рівня (Pascal, Delphi, Сі тощо). Крім цього мови програмування можна поділити на функціональні, структурні, логічні, процедурні, об'єктно-орієнтовані.

Мова *Pascal* має свою багату історію розвитку, початок якій поклало оголошення у 1965 р. конкурсу зі створення нової мови програмування – спадкоємця АЛГОЛА-60. Участь у конкурсі взяв швейцарський учений Ніклаус Вірт, який на той час працював доцентом на факультеті інформатики Стенфордського університету.

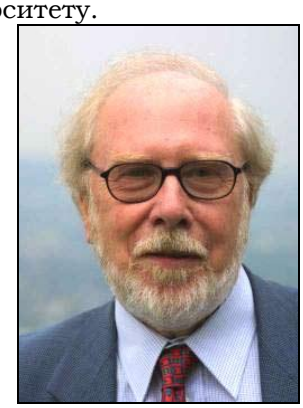


Рис.12.1. Професор Ніклаус Вірт

Ніклаус Вірт, професор, директор Інституту інформатики Швейцарської вищої політехнічної школи, лауреат премії Тюрінга, автор багатьох широко відомих праць з програмування. Н. Вірт також є автором таких мов програмування, як Ейлер, Модула, Модула-2. Окрім цього, ним запропонована методика покрокової розробки програм – від глобального до локального, від загального до часткового, – тобто занурення в алгоритм зверху донизу. Ця методика була визнана найсильнішою ідеологією програмування 70-х рр. XX ст.

Мова *Pascal* названа на честь великого французького математика й механіка Блеза Паскаля. Н. Вірт вирішив взятися за створення нової мови програмування з методичною метою: дати своїм студентам інструмент для вивчення програмування як систематичної, логічної дисципліни, яка ґрунтується на фундаментальних поняттях.

У 1971 р. Н. Вірт випустив опис своєї мови, а у 1975 р. було розроблене керівництво для користувачів версії *Pascal*, яке практично лягло в основу стандарту ISO на мову *Pascal*, що з'явився у 1982 р.

Пройшов певний час з моменту появи мови *Pascal* на ринку програмних продуктів, перш ніж вона отримала загальне визнання. Визнання програмістів і простих користувачів прийшло внаслідок появи діалекту мови – мови програмування *Turbo Pascal* (ТП), створеної американською фірмою Борланд. Ця фірма інтегрувала у 1985 р. дуже швидкий компілятор з редактором тексту й додала до стандартного *Pascal* потужне розширення, що спричинило успіх першої версії цієї мови. З тих пір *Pascal* став застосовуватися в загальноосвітніх, професійно-

технічних школах й у сфері вищої освіти, як „перша” мова програмування. Завдяки простоті використання, мова *Turbo Pascal* отримала широке поширення й в аматорських колах.

Підвищенню популярності *Turbo Pascal* сприяв набір невеликих супутніх програм, які дозволяють отримувати компактні, швидкі й легкі у читанні програми.

У пакеті *Turbo Pascal* 4.0, що з'явився згодом, була усунута більшість обмежень компілятора, яка піддавалися критиці, й була підвищена продуктивність системи. Крім того, новий компілятор версії 4.0 мав істотні відмінності від попередньої версії. Найбільш важливим нововведенням була UNIT-концепція, запозичена з мови Модула-2. Це дало можливість реалізувати в рамках мови розробку великих програмних продуктів.

З виходом у світ версії 5.0 *Turbo Pascal* одержав ще більші шанси на позитивну реакцію з боку професійних користувачів завдяки вбудованому в середовище програмування інтегрованому відладчику, який дозволяв підвищити продуктивність роботи.

Істотно поліпшила технічні характеристики *Turbo Pascal* реалізація апарата перекриттів (overlays), яка дозволяла будувати потужні програмні комплекси, розраховані на експлуатацію в малих за обсягом областях пам'яті. Суть механізму перекриттів зводиться до поділу програми на частини, які по мірі необхідності завантажуютьсь по черзі з диску в одну й ту ж саму область пам'яті, замінюючи при цьому частину програми, що перебувала там.

Крім того, у *Turbo Pascal* 5.0 були розширені можливості налагодження (debugging) програм і забезпечена можливість підтримки розширеної пам'яті в стандарті Lotus-Intel-Microsoft (LIMS/EMS 4.0). Скорочення EMS позначає Expanded Memory Specification (специфікація розширеної пам'яті).

У цій версії були також виправлені й поліпшені бібліотеки графічних процедур, що поставлялись разом з пакетом *Turbo Pascal*. При цьому забезпечувалася повна сумісність із графічними адаптерами класу VGA (Video Graphics Array).

У рамках версії *Turbo Pascal* 5.5 були здійснені подальші удосконалення технічних характеристик пакета. Поряд із внутрішнім поліпшенням й новими можливостями вбудованої довідкової системи *Help* і більшим набором навчальних прикладів, важливим нововведенням виявилася реалізація в мові концепції об’єктно-орієнтованого програмування (ООП).

Через деякий час на ринку з'явилася версія 6.0, у якій чисто теоретична концепція об’єктно-орієнтованого програмування була реалізована практично з повним набором об'єктів, що могли використовуватися для розв’язання прикладних задач користувача. Крім того, була приведена реалізація системи меню у відповідність зі стандартом SAA (Turbo Vision). Як практичний приклад використання нових можливостей був реалізований текстовий редактор, вбудований в IDE – Integrated Development Environment – інтегровану інструментальну оболонку. При цьому користувачі *Turbo Pascal* 6.0 отримали можливість не тільки працювати з вбудованим багатовіконним текстовим редактором, але й використовувати мишу, яка значно полегшувала роботу користувача.

У 1992 р. фірма Borland International представила користувачам чергову версію мови програмування *Pascal – Turbo Pascal* 7.0. Поряд з усіма перевагами, які *Turbo Pascal* 7.0 успадкував від попередньої версії (багатовіконний режим роботи, можливість використання миші, можливість використання при написанні програм мови програмування низького рівня Асемблер, можливість створювати об’єктно-орієнтовані програми), у ньому були зроблені зміни й поліпшення.

По-перше: з'явилася можливість виділяти певним кольорами різні елементи вихідного тексту (зарезервовані слова, ідентифікатори, числа тощо), що дозволяє навіть недосвідченим користувачам усувати помилки на етапі введення вихідного тексту.

По-друге: мова програмування *Turbo Pascal* 7.0 була розширена (з'явилася можливість використовувати типізований адресний оператор, відкриті масиви й рядки тощо), що надало користувачеві додаткових можливостей при розв’язанні повсякденних завдань.

По-третє: був удосконалений компілятор, внаслідок чого „коди програм” стали більш ефективними.

По-четверте: був удосконалений інтерфейс користувача.

Крім того, в мові *Turbo Pascal* 7.0 були розширені можливості об’єктно-орієнтованого програмування (зокрема, розширені й поліпшені можливості *Turbo Vision*).

Мова програмування *Pascal*, на відміну від усіх попередніх популярних мов, пропонує програмісту прості конструкції команд, які виглядають дуже природно. Набір команд мови Паскаль зовсім невеликий і складає її базову частину. Усі додаткові можливості *Pascal* дають змогу програмісту користуватися типами змінних, властивих лише цій мові програмування, а також створювати свої власні типи.

Таким чином, мова *Pascal* дуже швидко набула популярності серед програмістів і була реалізована для різних типів комп'ютерів.

Популярність цієї мови пояснюється ще й тим, що вона належить до мов *візуального програмування.* Мови візуального програмування значно полегшують процес програмування інтерфейсу створюваних програм. Адже зовнішній вигляд вікон та інших об’єктів програміст створює лише з допомогою миші та панелей об’єктів-заготовок. Серед поширених мов

візуального програмування можна також відмітити *Visual Basic* –мову, на якій створюють макроси для документів дуже популярного офісного пакета *Microsoft Office*.

В даній праці розглянуто основні можливості програмування на стандартній мові програмування *Turbo Pascal версії 7.0.* Цей посібник розрахований перш за все на початківців, які тільки знайомляться із світом програмування. Знання *Pascal* допоможуть їм набути базових навичок програмування та стануть основою для подальшого вивчення більш потужних мов, таких як Delphi.

12.2. Основні поняття мови Pascal

***Транслятор* – програма, що перетворює програму, написану мовою асемблеру або мовою високого рівня в машинні коди**. Транслятори поділяються на дві категорії: *інтерпретатори і компілятори.*

***Інтерпретатор* перетворює невеликий фрагмент вихідної програми в машинні коди і, лише дочекавшись, коли процесор їх виконає, переходить до обробки наступного фрагмента**.

Turbo Pascal 7.0, залишаючись сумісним із попередніми версіями *Turbo Pascal* та *Turbo Pascal for Windows*, надає низку нових можливостей. Найважливішою з них є наявність значно швидшого компілятора програм, який став своєрідним світовим стандартом для компіляторів.

Компілятор* – це спеціальна програма (програмний модуль), яка здійснює перетворення тексту програми, написаного на відповідній мові програмування у програмний код, зрозумілий комп'ютеру.** Сам процес перетворення тексту програми, у *програмний код* називається *компіляцією.** Потрібно відмітити, що при компіляції відбувається автоматичний пошук компілятором помилок у тексті програми.

Послідовність операцій, яка виконується при розв'язанні задачі, називають методом її розв'язання або алгоритмом.

***Алгоритм* – це чітка послідовність дій, спрямована на досягнення мети або на розв'язок поставленого завдання.**

***Програма* – це алгоритм, записаний мовою програмування**.

***Мова програмування* – це система позначень, за допомогою яких можна точно описати алгоритм для виконання його комп'ютером**.

Мови програмування поділяються на *машинні коди, мови асемблеру (Asembler), мови високого рівня (Turbo Pascal), візуальне програмування (Delphi).*

Мови високого рівня поділяються на *алгоритмічні* (процедурні) мови програмування**,** *дескриптивні* (непроцедурні) мови програмування.

Основними характеристиками мови є відносно невелика кількість базових понять, простий синтаксис, швидкий компілятор для перекладу вихідних текстів в машинний код.

Популярність мови *Pascal* викликана наступними чинниками:

– завдяки своїй компактності і вдалому первісному опису, *Pascal* є достатньо легким для вивчення;

– *Pascal* відбиває фундаментальні і найбільш важливі концепції (ідеї) алгоритмів в очевидній і легко сприйманій формі, які надають програмісту засоби, що допомагають проектувати програми;

– мова дозволяє чітко реалізувати ідеї структурного програмування і структурної організації даних;

– *Pascal* відіграв велику роль у розвитку методів аналітичного доказу правильності програм і дозволив реально перейти від методів налагодження програм до систем автоматичної перевірки правильності програм;

– застосування мови *Pascal* дозволило істотно підняти планку надійності розроблювальних програм за рахунок вимог мови до опису використовуваних в програмі змінних, перевірки узгодженості програми при компіляції без її виконання;

– використання в *Pascal* простих і гнучких структур управління: розгалужень, циклів.

Наступні версії набули розвитку у мові *Delphi* які працюють під ОС Windows.

Як відомо, мови програмування поділяться на два типи: інтерпретатори і компілятори. *Паскаль* відноситься до компіляторних мов.

12.3. Алфавіт мови Turbo Pascal. Типи даних

Комп'ютерні програми створюються з допомогою спеціальних мов, які називаються мовами програмування. Мова програмування, які і будь-яка інша мова, містить свій алфавіт.

***Алфавіт мови програмування* – це набір констант, типів даних, змінних, стандартних процедур і функцій, операндів та операторів, з яких складається програма.** До алфавіту мови *Pascal* входять:

1) 26 великих і малих латинських літер:

A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z
(a,b,c,d,e,f,g,h,i,j,k,j,m,n,o,p,q,r,s,t,u,v,w,x,y,z).

У програмі можуть використовуватися як великі, так і малі літери, компілятор їх сприймає однаково;

2) арабські цифри:

1,2,3,4,5,6,7,8,9,0;

3) шістнадцяткові цифри:

l,2,3,4,5,6,7,8,9,a,b,c,d,e,f;

4) знаки арифметичних операцій:

+ (додавання чисел і об'єднання текстів),

– (віднімання чисел);

* (множення чисел);

/ (ділення чисел);

5) знаки відношення:

= (рівне);

< (менше);

>= (не менше);

> (більше);

<= (не більше);

= (дорівнює);

<> (не дорівнює).

6) синтаксичні знаки:

. (крапка);

, (кома);

; (крапка з комою);

" (подвійні лапки);

' (апостроф);

: (двокрапка);

(пробіл);

? (знак питання),

! (знак оклику);

7) дужки:

() круглі;

[ ] квадратні;

{ } фігурні;

6) спеціальні символи: *$,%,#,@,&* тощо.

Константи – це дані, значення яких відоме до завантаження програми і не змінюється в процесі її виконання. Наприклад: *375, Pi*. В *Pascal* існують як стандартні константи, так і вказані користувачем. До стандартних констант належить, наприклад, число *Pi=3,141592....* Константи, задані користувачем, потрібно вказувати в спеціальному розділі програми.

У будь-якій мові програмування використовуються так звані ідентифікатори. Ідентифікатори використовуються для позначення різних об'єктів програми, наприклад функцій, змінних, масивів, міток, процедур, файлів тощо.

Ідентифікатор – це послідовність букв і цифр, яка починається з букви і яка не є ключовим словом. Буквою в ідентифікаторі може бути будь-яка велика або мала буква англійської мови і знак підкреслювання. Кількість символів у ідентифікаторі знаходиться у межах від 2 до 127. Найчастіше в ідентифікаторі використовується до 8 символів. Приклади ідентифікаторів:

a, z2, alfa, m21, dam_8.

Приклади неправильних ідентифікаторів:

array, 121, alk, 5ab.

Змінні – це дані, значення яких вводиться та змінюється під час виконання програми. Усі змінні, що використовуються в Pascal-програмі, потрібно описувати в спеціальному розділі програми.

Тип даних – це набір даних (констант, змінних, значень функцій тощо), які мають спільні характеристики (формат представлення в пам'яті ПК, множина припустимих значень, множина припустимих операцій, які можна використовувати для даного типу).

Тип даних визначає обсяг пам'яті, необхідний для зберігання даного об'єкта, і операції, що можна над цим об'єктом виконувати.

Типи даних в *Pascal* поділяються на *прості (скалярні)* та *складні (структуровані)*.

Скалярні типи у свою чергу поділяються на *стандартні* та *типи користувача*. *Стандартні типи* пропонуються користувачам розроблювачами системи Turbo Pascal.

Типи користувача – розробляються самим програмістом.

До стандартних скалярних типів відносяться наступні типи:

- цілі,
- дійсні,
- літерні,
- булівські.

Величини цілих типів можуть бути подані, як у десятковій, так і в шістнадцятковій системах. Якщо число представлене в шістнадцятковій системі, перед ним без проміжку записується знак $. Діапазон зміни шістнадцяткових чисел від $0000 до $FFFF.

Цілі типи даних являють собою значення, що можуть використовуватися в арифметичних виразах. Усі стандартні цілі типи зведені в таблицю 12.1.

Таблиця 12.1.

Тип	**Діапазон**	**Необхідна пам'ять (байт)**
Byte	0…255	1
Shortint	-128…127	1
Integer	-32768…32767	2
Word	0…65535	2
Longint	-2147483648…2147483647	4

Дійсні типи даних являють собою дійсні значення, які використовуються в арифметичних виразах і займають у пам'яті від 4 до 10 байт. Паскаль припускає подання дійсних значень у вигляді, як з плаваючою, так і з фіксованою крапкою.

Дійсні десяткові числа з фіксованою крапкою записуються за звичайними правилами арифметики. Єдине, що відрізняє цей формат від математичного це те, що ціла частина від дробової відокремлюється десятковою крапкою, а не комою. Якщо десяткова крапка відсутня, число вважається цілим. Перед числом може знаходитися знак "+" або "–". Якщо знак відсутній, за замовчуванням число вважається додатнім.

Дійсні десяткові числа у форматі з плаваючою крапкою подаються в наступному (експоненціальному) виді:

$$mE+p,$$

де m – мантиса (ціле або дробове число з фіксованою десятковою крапкою),

E – означає "десять у степені",

p – порядок (ціле число).

Взагалі мантиса повинна бути нормалізованою, тобто представленою у вигляді числа, що належить діапазону від 0 до 1 (це означає, що крапка завжди знаходиться перед першою цифрою числа). Однак програмісту дозволяється писати мантису у вигляді будь-якого дробового числа з фіксованою крапкою. Нормалізація при цьому виконується системою автоматично (табл. 12.2).

Таблиця 12.2.

Число у форматі з плаваючою крапкою	**Значення числа**
0.4500E+02	0.45*102=45
-2.600E05	-2.6*105=-260000
+0.45670E-02	0.4567*10-2=0.004567

Стандартний найбільш часто використовуваний дійсний тип даних представлений в табл.12.3.

Таблиця 12.3.

Тип	**Діапазон значень**	**Мантиса**	**Необхідна пам'ять (байт)**
Real	2.9*10E-39…1.7*10E38	11-12	6

Літерний (символьний) тип може набувати значень кодової таблиці ПЕОМ (таблицю ASCII кодів) яка наведена в табл.12.4.

Символьній змінній в пам'яті виділяється один байт, тому вона можна зберегти тільки один символ ASCII таблиці.

Таблиця 12.4.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	NUL	32	20	(sp)	64	40	@	96	60	`
1	1	SOH	33	21	!	65	41	A	97	61	a
2	2	STX	34	22	"	66	42	B	98	62	b
3	3	ETX	35	23	#	67	43	C	99	63	c
4	4	EOT	36	24	$	68	44	D	100	64	d
5	5	ENQ	37	25	%	69	45	E	101	65	e
6	6	ACK	38	26	&	70	46	F	102	66	f
7	7	BEL	39	27	'	71	47	G	103	67	g
8	8	BS	40	28	(	72	48	H	104	68	h
9	9	TAB	41	29	)	73	49	I	105	69	i
10	A	LF	42	2A	*	74	4A	J	106	6A	j
11	B	VT	43	2B	+	75	4B	K	107	6B	k
12	C	FF	44	2C	,	76	4C	L	108	6C	l
13	D	CR	45	2D	-	77	4D	M	109	6D	m
14	E	SO	46	2E	.	78	4E	N	110	6E	n
15	F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Булівський тип подається двома значеннями: *True* (істина) або *False* (хибність). Він широко застосовується в логічних виразах і виразах відношення.

Структуровані типи у своїй основі мають один або декілька скалярних типів даних. До структурованих типів даних відносяться рядки, масиви, файли, записи тощо. Їх ми будемо вивчати пізніше.

Змінні і константи всіх типів використовуються у виразах.

Вираз задає порядок виконання дій над елементами даних і складається з операндів (констант, змінних, звертань до функцій), круглих дужок і знаків операцій.

Круглі дужки ставляться, як і в математиці, для управління порядком виконання операцій. Якщо дужки відсутні, операції виконуються в залежності від їх пріоритетів, про що буде сказано далі.

Крім цього, типи даних у *Turbo Pascal* можна поділити на впорядковані та невпорядковані.

***Впорядковані** – це типи, в яких дані розміщені в певному, наперед визначеному, порядку і кожен з елементів характеризується своїм порядковим номером*.

***Невпорядковані** – це типи, в яких дані не мають своїх порядкових номерів.*

***Процедури та функції** – це підпрограми, що використовуються в середині програми.*

До алфавіту мови програмування відносять стандартні процедури та функції, тобто такі, що сприймаються мовою програмування без їх додаткового опису.

Операнди – це спеціальні символи або послідовності символів, які виконують над даними певні операції (математичні, логічні тощо). Прикладом операндів можуть бути операнди математичних операцій: "+" – додавання даних), "–" – віднімання даних, "*" – множення даних, "/" – ділення даних, "=" – рівність даних тощо.

Оператори – це певні неподільні елементи програми, які дозволяють виконувати певні алгоритмічні дії у програмі (певні команди). Фактично, оператор – це окрема команда в алгоритмі програми, тобто окремий крок виконання програми.

В мові *Pascal* оператори поділяються на прості та структурні. До простих операторів відносяться:

- ✓ оператор присвоєння *(:=)*;
- ✓ оператор безумовного переходу *(goto)*;
- ✓ оператор звертання до процедури (функції).

Структурними операторами називаються такі, які складаються з інших операторів. До них відносяться:

- складний оператор – набір операторів, які поміщені в операторні дужки (*begin – end*);
- умовний оператор *(if)*;
- оператор вибору *(case)*;
- оператори циклу *(repeat, while, for)*;
- оператор приєднання *(with)*.

Усі вищеописані елементи алфавіту мови програмування складаються з окремих символів. В якості символів, що складають елементи алфавіту, в мові *Pascal* можна використовувати більшість символів, що входять до стандартної *ASCII*-таблиці.

Заборонено використовувати символи розширеної ASCII-таблиці, тобто символи з кодами від 128 до 255, а також символи: (&), (!), (%), (~), (”). Ці символи можна використовувати лише в якості коментарів та в текстових стрічках, які беруться в одинарні лапки (’).

Коментар – запис, призначений для внесення до тексту програми пояснень. Обмежується символами { } або (* *). Наприклад: *{Це коментар}, (*Це коментар*).*

Інтегроване середовище програмування – це система програмування, яка поєднує редактор для зручного введення і редагування програми, транслятор й налагоджувач помилок.

Інтегроване середовище програмування *Turbo Pascal 7.0* складається з редактора, компілятора, редактора зв'язків і налагоджувача.

У кожній мові програмування із символів конструюються службові слова (їх називають ще зарезервованими або ключовими словами). Таких слів у мові може бути від декількох десятків до декількох сотень. Службові слова не дозволяється змінювати. Службові слова можуть записуватись як великими, так і малими літерами. Службові слова мови *Pascal* представлені в табл.12.5.

Таблиця 12.5.

Службові слова мови Turbo Pascal

and – і	**end** – кінець	**nil** – нуль	**repeat** – повторювати
array – масив	**file** – файл	**not** – ні	**set** – множина
begin – початок	**for** – для	**of** – з	**then** – то
case – вибір	**function** – функція	**or** – або	**to** – до
const – сталі	**goto** – перейти до	**packed** – стиснутий	**type** – тип
div – ділення без остачі	**if** – якщо	**procedure** – процедура	**until** – доки
do – виконати	**in** – в	**program** – програма	**var** – змінні
downto – вниз до	**label** – позначка	**record** – запис	**while** – доки та інші.
else – інакше	**mod** – остача		

Типи даних мови *Pascal* представлені в табл.12.6.

Операції можуть бути *унарними* та *бінарними*. У першому випадку операція відноситься до одного операнду і завжди записується перед ним, у другому операція виражає відношення між двома операндами і записується між ними.

Таблиця 12.6.

Стандартні типи даних

Тип даних	Позначення	Опис	Об'єм пам'яті	Приклади
Ціло-чисельний	***Byte***	Цілі числа в інтервалі [0; 255]	1 байт	0, 5, 245
	Shortint	–128..127	1 байт	-5, 26, 96
	word	0..65535	2 байта	0, 36, 6545
	Integer	Цілі числа в інтервалі [-32767; 32767]	2 байта	457, -568, -7, 0
	longint	–2147483648..2147483647	4 байта	-2356, 988456
Дійсний	***single***	1.5·10–45–3.4·1038	4 байта	236,36; 6954,369
	Real	Значення з плаваючою крапкою, що містить мантису (до 11 значущих цифр) і експоненту - степінь числа 10 (від 10^{-39} до 10^{38})	6 байтів	0.4,-1.8, 0.172E+3, 37E-4
	extended	$3.4\cdot10^{-4951}$ –$1.1\cdot10^{4932}$	10 байт	0.4,-1.8, 0.172E+3,
Символь-ний	***Char***	Символьні константи містять один символ, який обмежено апострофами	1 байт	A', 'є', Y, '2'
Рядковий	***String***	Рядок символів довжиною не більше 255	Залежить від довжини	
Логічний	***Boolean***	Константи логічного типу мають два можливі значення: *True* (істинне) і *False* (хибне)	1 байт	5<3 –True 5+8>14 – *False*

Арифметичні операції задають арифметичні дії у виразах над значеннями операндів цілих та дійсних типів. Найбільш часто використовуються арифметичні операції, що подані в табл. 12.7.

Таблиця 12.7.

Операція	Дія	Типи операндів	Тип результату
Бінарні			
+	Додавання	Цілий Дійсний	Цілий Дійсний
–	Віднімання	Цілий Дійсний	Цілий Дійсний
*	Множення	Цілий Дійсний	Цілий Дійсний
/	Ділення	Цілий Дійсний	Цілий Дійсний
Div	Ділення націло	Цілий	Цілий
mod	Залишок від ділення	Цілий	Цілий
Унарні			
+	Збереження знака	Цілий Дійсний	Цілий Дійсний
–	Збереження знака	Цілий Дійсний	Цілий Дійсний

Арифметичні вирази у якості операндів можуть містити імена функцій. З поняттям функції ми будемо знайомитись пізніше, але стандартні функції (*cos*, *sin*, x^2 та інші) знайомі з курсу математики і їх використання у мові *Pascal* відрізняється тільки правилами запису (синтаксису). Так, на відміну від математики, в програмуванні аргумент функції обов'язково береться в круглі дужки. Нижче поданий список цих функцій, де літерами *X* та *I* позначені відповідно: *X* – цілі та дійсні типи, *I* – тільки цілі.

Abs(X) – обчислення абсолютного значення (модулю) *X*. Тип результату збігається з типом параметра.

ArcTan(X) – обчислення кута, тангенс якого дорівнює *X*, значення кута подано в радіанах і може знаходитися в діапазоні від –п/2 до п/2. Для перетворення значення кута з радіанної міри в градусну необхідно значення кута помножити на число 180/п. Результат має дійсний тип.

Cos(X) – обчислення косинуса *X*, параметр задає значення кута в радіанах.

Для перетворення значення кута з радіанної міри в градусну необхідно значення кута помножити на число 180/п. Результат має дійсний тип.

Exp(x) – обчислення значення експоненти аргументу (*Ex*). Результат завжди має дійсний тип.

Frac(X) – знаходження дробової частини *X*. Результат має дійсний тип.

Int(X) – знаходження цілої частини *X* (дробова частина відкидається). Результат має дійсний тип.

Ln(x) – обчислюється натуральний логарифм аргументу. Результат має дійсний тип. За допомогою функцій *Exp* та *Ln* можна обчислити довільну степінь числа наступним чином:

$$a^b=Exp(b*Ln(a)).$$

Pi – повертає значення числа п (3.1415926).

Sin(X) – обчислення синуса *X*. Параметр задає значення кута в радіанах. Для перетворення значення кута з радіанної міри в градусну необхідно значення кута помножити на число 180/п. Результат має дійсний тип.

Sqr(X) – піднесення до квадрату значення *X*. Тип результату збігається з типом параметра.

Sqrt(X) – обчислення квадратного кореня з X. Тип результату дійсний.

Random – генерує значення випадкового числа з діапазону від 0 до 0.99. Тип результату дійсний.

Random(P) – генерує значення випадкового числа з діапазону від 0 до P. Тип результату цілий. Щоб випадкові числа були „більш випадковими", необхідно періодично змінювати базу генерації. Для цього використовується процедура *Randomize*, що дозволяє при кожному новому запуску програми отримувати різні випадкові числа.

Для величин перелічувальних типів (всі цілі, літерний та булівський) існують також декілька зручних стандартних функцій та процедур, наведених нижче:

Dec(x,[dx]) – процедура зменшує значення змінної *X* на величину *DX*, а якщо параметр *DX* не заданий – на 1.

Inc(x,[dx]) – процедура збільшує значення змінної *X* на величину *DX*, а якщо параметр *DX* не заданий – на 1.

Pred(X) – функція, що визначає попереднє значення для даного типа. Наприклад, *Pred(5)=4, Pred(True)=False, Pred('B')='A'*.

Succ(X) – функція, що визначає наступне значення для даного типа. Наприклад, *Succ(5)=6, Succ(True)=False, Succ('A')='B'*.

Odd(X) – число перевіряється на парність. Аргументом цієї функції може бути тільки число типу *longint*, а результат дорівнює *true*, якщо аргумент парний, і *false* – в протилежному випадку.

Операції відношень представлені в табл.12.8. Результатом операції відношень є значення логічного типу.

Таблиця 12.8.

Операції відношень

>	більше ніж	< =	менше ніж або дорівнює
> =	більше ніж або дорівнює	=	дорівнює
<	менше ніж	< >	не дорівнює

Операції відношення виконують порівняння двох операндів і визначають, чи є значення виразу істинним або хибним. Результат завжди має булівський тип і приймає одне з двох значень: *True* (істина) або *False* (хибність). Всі операції відношення зведені в табл. 12.9.

Таблиця 12.9.

Операції	Назва	Вираз	Результат
=	Дорівнює	A=B	True, якщо A дорівнює B
<>	Не дорівнює	A<>B	True, якщо A не дорівнює
>	Більше	A>B	True, якщо A більше B
<	Менше	A<B	True, якщо A менше B
>=	Більше або дорівнює	A>=B	True, якщо A більше або дорівнює B
<=	Менше або дорівнює	A<=B	True, якщо A менше або дорівнює B

Результатом виконання логічного (булівського) виразу є логічне значення *True* або *False*.

Операції над стандартними типами даних представлені в табл.12.10.і табл. 12.11.

Таблиця 12.10.

Операції над цілими числами

Операція	Позначення	Приклад
Додавання	**+**	**Y+Z**
Віднімання	**–**	**Y–Z**
Множення	*****	**Y*Z**
Цілочислове ділення з визначенням цілої частини	**div**	**Y div Z**
Цілочислове ділення з визначенням дробової частини	**mod**	**Y mod Z**
Рівність	**=**	**Y=Z**
Не рівність	**<>**	**Y<>Z**
Менше (порівняння)	**<**	**Y<Z**
Менше або дорівнює (порівняння)	**<=**	**Y*Z**
Більше (порівняння)	**>**	**Y<=Z**
Більше або дорівнює (порівняння)	**>=**	**Y>=Z**

Таблиця 12.11.

Операції над дійсними числами

Операція	Позначення	Приклад
Додавання	+	Y+Z
Віднімання	–	Y–Z
Множення	*	Y*Z
Ділення	/	Y / Z
Рівність	=	Y=Z
Менше (порівняння)	<	Y<Z
Менше або дорівнює (порівняння)	<=	Y*Z
Більше (порівняння)	>	Y<=Z
Більше або дорівнює (порівняння)	>=	Y>=Z

Арифметичні вирази (табл. 12.12) будуються з констант, змінних, функцій і операцій над ними.

Таблиця 12.12.

Вирази

Функція	Тип аргументу	Тип результату	Математичний запис, коментар
abs(x)	integer, real	integer, real	|x|
arctan(x)	integer, real	real	arctgx
cos(x)	integer, real	real	cosx
sin(x)	integer, real	real	sinx
exp(x)	integer, real	real	e^x
ln(x)	integer, real	real	lnx
sqrt(x)	integer, real	real	$\sqrt{x}$
sqr(x)	integer, real	integer, real	x^2
ord(x)	упорядкований	integer	ASCII-код символу
succ(x)	упорядкований	упорядкований	повертає наступне значення x
pred(x)	упорядкований	упорядкований	повертає попереднє значення x
round(x)	real	integer	заокруглює число x до цілого
trunc(x)	real	integer	відкидає дробову частину числа x
int(x)	real	real	відкидає цілу частину числа x
frac(x)	real	real	дробова частина числа x
odd(x)	integer	boolean	true (x – непарне), false (x – парне)
random(x)	integer	integer	Генерує випадкове число з діапазону від 0 до x
upcase(x)	char	char	замінює малу літеру латинської абетки на велику
Процедури			
inc(x,y)	integer	integer	збільшує x на y
inc(x)	integer, char	integer, char	збільшує x на 1
dec(x,y)	integer	integer	зменшує x на y
dec(x)	integer, char	integer, char	зменшує x на 1

Операндами слугують дані тільки булівського типу. Список логічних операцій наведений у табл.12.13.

Таблиця 12.13.

Операція	Дія	Вираз	A	B	Результат
not	Логічне заперечення	not A	True False		False True
and	Логічне "ТА"	A and B	True True False False	True False True False	True False False False
or	Логічне "АБО"	A or B	True True False False	True False True False	True True True False

Виконання кожної операції відбувається з урахуванням її пріоритету. Значення пріоритетів зазначені в табл. 12.14.

Таблиця 12.14.

Операція	Пріоритет	Вид операції
Not, унарні "-" і "+"	Перший (вищий)	Унарна операція
*, /, div, mod, and	Другий	Операції типу множення
+, -, or	Третій	Операції типу додавання
=, <>, <, >, <=, >=	Четвертий (нижчий)	Операції відношення

12.4. Структура програми на мові Паскаль

Програма реалізує алгоритм розв'язку задачі. В ній програміст записує послідовність дій, які виконуються над даними за допомогою операцій для реалізації певної мети.

Основними характеристиками програми є: *точність отриманого результату*, *час виконання* і *об'єм необхідної пам'яті*. Про відповідність цих показників розв'язуваній задачі і можливостям комп'ютера повинен подбати сам програміст. У більшості випадків визначальною вимогою є точність. Обмеження об'єму пам'яті і часу виконання носять менш жорсткий характер. Будь-яка програма на мові *Pascal* складається з двох розділів:

1) опису використовуваних даних;

2) послідовності операторів, які опрацьовують дані.

Такий поділ дозволяє уникнути багатьох помилок в процесі програмуванні. Для відділення розділів використовуються спеціальні службові слова (англійські), які для інших цілей не повинні застосовуватися.

Існують різні схеми написання програм мовою *Pascal*, усі вони відрізняються кількістю відступів зліва у кожному рядку і різним використанням прописних букв. Набираючи текст програми, програміст може довільно розташовувати рядки на екрані. Рядок може починатися з будь-якої колонки, тобто величина відступу від лівої границі екрана для кожного рядка встановлюється самим програмістом з метою отримати найбільш зручний для читання текст програми. Кількість операторів в рядку є довільною, але якщо в рядку записується один оператор, то така програма легше читається.

При написанні програми необхідно дотримуватися правила розміщення в тексті різних змістовних блоків. Будь-яку програму можна умовно розділити на дві основні частини: розділ опису (розділ оголошень та угод; розділ текстів процедур та функцій) та розділ виконання (розділ основного блоку).

Отже, структура *Pascal -програми* має вигляд (рис. 12.2).

```
Program .<ім'я програми>;      { Заголовок програми }
Uses ... ;                     { Підключення модулів }
Label ... ;                    { Розділ оголошення міток }
Const ... ;                    { Розділ оголошення констант }
Type ... ;                     { Розділ оголошення нових типів }
Var ... ;                      { Розділ оголошення змінних }
Procedure ... ;                { Опис своїх процедур }
Function ... ;                 { Опис своїх функцій }
Begin                          { початок основної програми }
...;
{ Оператори }
...;
End.
```

Рис. 12.2. Структура *Pascal* -програми

Program, Begin і End – службові слова. Заголовок програми складається із зарезервованого слова *Program* та імені програми, яке є ідентифікатором (цей рядок може бути відсутнім).

Program – це заголовок програми, що вказує її ім'я. *Ім'я* – це ім'я програми (послідовність латинських букв і цифр, які починаються з букви). Для *Turbo Pascal 6.0* і більш пізніх версій цей підрозділ вказувати необов'язково, хоча рекомендовано вказувати заголовок програми, щоб уже при першому знайомстві з її текстом можна було отримати інформацію про її призначення. Заголовок програми, що слідує, за словом *Program* є ідентифікатором і володіє всіма його властивостями. Крім цього, заголовок програми обов'язково повинен починатись з латинської літери, а далі можуть знаходитись символи, припустимі для алфавіту *Turbo Pascal*.

End. – кінець програми (крапка обов'язкова, вона вказує, що це останнє слово в програмі).

Точка з комою – покажчик закінчення програмної одиниці (оператора). Такий поділ дозволяє писати декілька операторів у рядку. Але зазвичай, для зручності читання програми і роботи з нею, оператори записують по одному у рядку і навіть драбинкою. Відступи дозволяють бачити структуру програми.

За допомогою оператора *Uses* до тексту програми підключаються модулі бібліотек, він може бути використаний тільки один раз, і його місце чітко визначено (він знаходиться поперед усіх операторів та підрозділів (крім заголовка програми та директив компілятора). Поняття „*модуль*”, „*бібліотека*”, „*блок*” складають основу термінології програмування на *Pascal*.

<u>Модуль</u> – замкнений блок зі своїм своє ім'ям, який компілюється окремо і підключається до програми, як “чорна скринька” із набором певних (описаних у ньому) процедур, функцій, типів даних, констант тощо.

Бібліотека представляє собою набір таких модулів. Якщо модулі в програмі не потрібно використовувати (не використовується процедури, функції тощо, які описані в модулі), то цей підрозділ не вказується.

Розділ опису міток *Label* містить перелічені через кому імена позначок переходу, які можуть являти собою ціле число (від 0 до 9999), рядок символів, символьно-цифрову конструкцію. Якщо міток у програмі немає, то підрозділ *Label* не вказується.

<u>Примітка.</u> Існує неписане правило – не використовувати у *Pascal*-програмі міток, оскільки це вважається дурним тоном. Якщо в програмі використано мітки, то така програма вважається написаною безграмотно.

У розділі *Const* містяться перелічені через кому константи, що використовуються в програмі. В інших випадках цей підрозділ не вказується. Слід відмітити, що стандартні константи, прийняті в *Pascal*, не потрібно описувати в даному розділі.

У розділі *Type* можна визначити нові типи, тут можуть використовуватися раніше визначені в розділі *Const* константи. В цьому підрозділі переважно вказують складні та нестандартні типи даних. Якщо такі типи в програмі не використовують, то даний підрозділ не вказується.

У розділі опису змінних *Var* міститься список змінних, що використовуються в програмі, та визначається їх тип. Жорстке дотримання порядку оголошень позначок, констант, типів та змінних не потрібне. Потрібно відмітити, що всі змінні, що використовуються в програмі, обов'язково повинні бути описані в підрозділі *var.* Це єдиний підрозділ розділу описів та узгоджень, який обов'язково повинен бути представлений у кожній *Pascal*-програмі.

Розділ текстів процедур та функцій вказується лише у випадку, якщо в програмі використовуються нестандартні процедури та функції, що створені самим користувачем (якщо вони не описані в одному з модулів вказаних у розділі *Uses*). Потрібно відмітити, що процедури та функції – це спеціальним чином оформлені послідовності команд (у вигляді підпрограми). Доступ до цієї підпрограми може здійснюватись з будь-якого місця основної програми, а також з будь-якої процедури та функції, що описані нижче за текстом програми. Далі йде оголошення локальних міток, констант, типів та змінних. Локально оголошені конструкції доступні тільки всередині даної підпрограми.

Тіло процедури (функції) обмежується службовими словами *Begin…End* .

Основний блок складається з послідовності операторів, які визначають послідовність дій для виконання основного алгоритму.

Тіло програми обмежується службовими словами *Begin…End*, як і тіло підпрограми, але після оператора *End* ставиться крапка, що є позначенням кінця програми. Усі наступні описи будуть сприйняті як коментар та ігноруватись транслятором.

12.5. Правила написання Pascal-програм

1. Основний текст будь-якої програми починається службовим словом *begin* і закінчується словом *end,* після чого йде крапка. Без крапки програма вважається незакінченою. І навпаки, якщо в програмі знаходиться крапка, то всі команди, що слідують за нею ігноруються (оскільки програма завершена).

2. В кінці кожної команди ставиться крапка з комою (“;”) – символ, що розділяє команди між собою.

3. Після команди, яка знаходиться перед *end*, крапку з комою (“;”) бажано не встановлювати, оскільки буде вважатись, що перед *end* є ще один порожній оператор.

4. Команди в *Pascal* можна записувати в один рядок, хоча для полегшення читабельності програми бажано кожну команду вказувати з нового рядка.

5. Якщо користувач бажає помістити в текст програми коментарі, то їх необхідно вказувати у фігурних дужках (замість фігурних дужок можна вказувати альтернативний набір символів – “(* ,,, *)”).

6. При вказівці в програмі виразів, що містять будь-які дужки, потрібно пам'ятати, що кількість закритих та відкритих дужок повинна бути однаковою.

7. Якщо потрібно використати декілька операторів у якості одного складеного оператора, то їх слід взяти в операторні дужки, що починаються словом *begin* і закінчуються *end.* При цьому кількість слів *begin* у програмі повинна співпадати з кількістю слів *end.*

8. Усі змінні, константи та типи даних, що використовуються в програмі, повинні бути описані в розділах *const, type* та *var.*

9. Кожну програму треба складати так, щоб вона дозволяла здійснювати швидку модернізацію на випадок можливої зміни її параметрів: коефіцієнтів, кроків дискретизації її змінних, початкових, кінцевих значень аргументів тощо. Тому ніколи не розміщуйте конкретні цифрові значення в тіло програми. Цим Ви істотно зменшите час розробки програми, зробите її легко зчитуваною, придатною для широкого використання.

Правила хорошого тону написання програм.

1. Після команди *begin* всі наступні команди, аж до відповідного йому *end,* бажано записувати з відступом (наприклад, на величину слова *begin).*

2. Назви всіх команд бажано записувати малими літерами, а змінні – великими.

12.6. Оператори

Оператори – це команди програми. В табл.12.15 наведені основні оператори, за допомогою яких можна створювати прості програми.

Таблиця 12.15.

Оператори

Оператор	Загальний вигляд	Опис	Приклади
:= Присвоювання	Ідентифікатор: = вираз	Присвоює змінній конкретне значення, заповнюючи комірку пам'яті, відведену для змінної, новим значенням, відночас знищуючи старе	A:=4 X:= Z + 7 – Y Name:= Аристарх
Read Введення (*зчитування*) даних з клавіатури	Read(a_1, a_2,... ,a_n), де a_1, a_2, ...,a_n –змінні зазначених типів, яким присвоюються значення, що вводяться	Програма зупиняється і чекає введення необхідної кількості даних. Числа при введенні поділяються пробілами або натиском клавіші **<Enter>.** Введення закінчується натиском клавіші **<Enter>**	Read (Name) Read (A, B, У, Z)
ReadIn	Після введення значень a_1, a_2, ...,a_n курсор переводиться на наступний рядок		
Write Виведення *(запис)* даних на монітор	Write(a_1, a_2,... ,a_n), де a_1, a_2, ...,a_n – константи або змінні зазначених типів	Виведення зазначених даних (чисел, змінних, тексту, який обмежено апострофами) виконується з позиції курсору	Write ('Відповідь',2) *Відповідь 2* Write('Привіт,', Name,'!') *Привіт, Аристарх!*
WriteIn	Після виведення значень a_1, a_2, ...,a_n курсор переводиться на наступний рядок		

Примітка. Після кожного оператора в програмі ставиться символ «**;**» (*крапка з комою*). У кінці програми ставиться «**.**» (*крапка*).

Питання для самоконтролю

1. *Що розуміють під терміном „комп'ютерна програма"?*
2. *Що розуміють під терміном „транслятор"?*
3. *Що розуміють під терміном „інтерпретатор"?*
4. *Що розуміють під терміном „компілятор"?*
5. *Що розуміють під терміном „алгоритм"?*
6. *Що розуміють під терміном „програма"?*
7. *Що розуміють під терміном „мова програмування"?*
8. *Що розуміють під терміном „транслятор"?*
9. *Що розуміють під терміном „ідентифікатор"?*
10. *Що розуміють під терміном „змінні"?*
11. *Що розуміють під терміном „константа"?*
12. *Що розуміють під терміном „тип даних"?*
13. *Які типи даних Вам відомі?*
14. *Що розуміють під терміном „оператор"?*
15. *Що розуміють під терміном „операнд"?*
16. *Що розуміють під терміном „інтегроване середовище програмування"?*
17. *Які стандартні типи даних Вам відомі?*
18. *Які оператори відношень Вам відомі?*
19. *Який вигляд має структура Pascal-програми?*
20. *Які правила Pascal-програми Вам відомі?*

13. РОБОТА В СЕРЕДОВИЩІ ТУРБО-ПАСКАЛЬ

13.1. Початок роботи з середовищем Turbo Pascal 7.0

Для виклику середовища *Turbo Pascal* необхідно відшукати деревовидній структурі каталогів ПК файл з ім'ям TURBO.EXE, де власне міститься система, а також файл TURBO.TPL (бібліотека транслятора) і TURBO.HLP (бібліотека довідкової служби). При написанні програм, що використовують графічні можливості ПК, знадобиться бібліотечний файл GRAPH.TPU й файл, який містить графічний драйвер. Тип графічного драйвера залежить від апаратних засобів, реалізованих в конкретному комп'ютері.

Зазвичай усі файли, що відносяться до *Turbo Pascal* зберігаються в каталозі (папці) з ім'ям TP, TURBO, TURBOPAS, PASCAL або їм подібним. Припустимо, що цей каталог називається TP і розташовується на диску C. Тоді для виклику *Turbo Pascal* потрібно дати команду:

C:\TP\TURBO

Не рекомендується працювати з системою, назначивши в якості каталогу вищеперераховані файли: по-перше, у такому випадку існує загроза помилково стерти будь-який з цих файлів і тим самим порушити працездатність системи, а по-друге, цей каталог дуже скоро заповниться іншими файлам, які безпосередньо не відносяться до *Turbo Pascal.* Крім того, *Turbo Pascal* має властивість „запам'ятовувати" свої настройки у двох файлах з іменами TURBO.TP і TURBO.PCK. При виклику система починає пошук цих двох файлів і в поточному каталозі. Якщо цей каталог – Ваш індивідуальний, система кожного разу буде настроюватись так, як Ви цього бажаєте. Якщо ці файли не будуть виявлені у Вашому каталозі (а при першому зверненні саме так і буде) система продовжить пошук в каталогах TURBO і EXE, і не знайшовши їх там, – настроїться стандартним чином. Пізніше Ви зможете зберегти настроюванні файли у своєму каталозі і тим самим позбавити себе від перенастроювання системи кожного разу при зверненні до неї.

Основний екран інтегрованого середовища програмування Turbo Pascal 7.0 виглядає таким чином (рис.13.1):

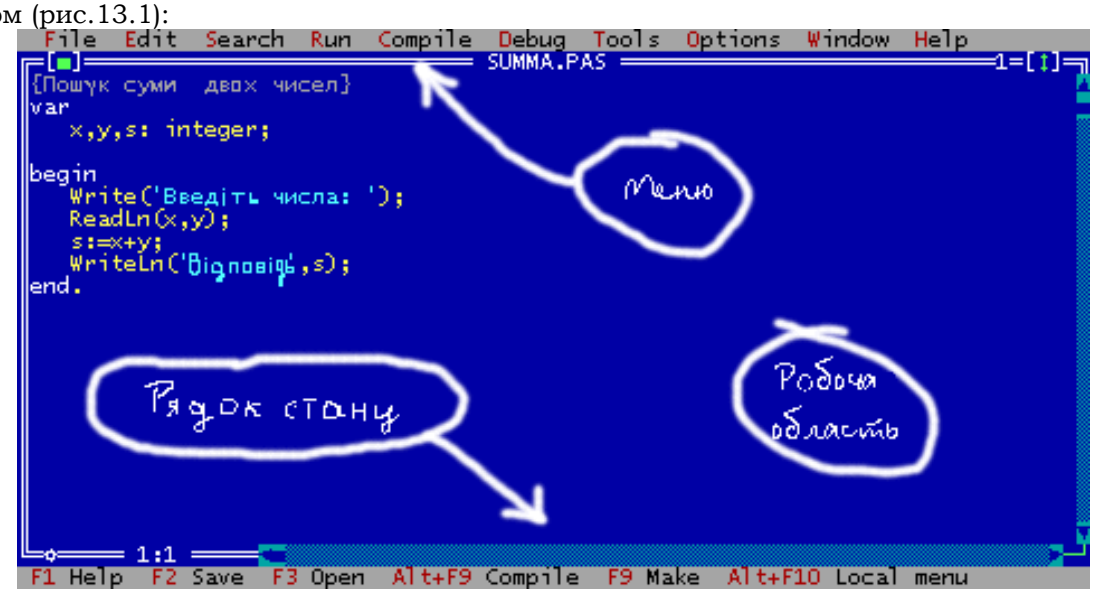


Рис.13.1. Вигляд екрана після виклику Turbo Pascal

За функціональним призначенням виділяється три області екрану:

– рядок меню;

– робоча область;

– рядок стану.

Рядок меню активізується натисненням клавіші *F10.* У меню містяться наступні розділи:

- *File.* Дозволяє виконувати основні дії з файлами (створення відкриття, збереження тощо).
- *Edit.* Дозволяє виконувати всі основні операції редагування тексту (копіювання, вставка, видалення фрагментів, відміна останніх змін тощо).
- *Search.* Дозволяє здійснювати пошук і заміну фрагментів тексту.
- *Run.* Дозволяє запускати програму, зокрема у покроковому режимі.
- *Compile.* Дозволяє здійснювати компіляцію програми.
- *Debug.* Містить команди, які полегшують процес пошуку помилок в програмі.
- *Tools.* Містить деякі додаткові засоби *Turbo Pascal.*
- *Options.* Дозволяє встановити необхідні для роботи параметри компілятора і середовища програмування.

• *Window.* Дозволяє виконувати всі основні операції з вікнами (відкривати, закривати, переміщати, змінювати розмір).

• *Help.* Дозволяє отримати наявну в системі довідкову інформацію.

Всі пункти меню доступні через гарячі клавіші. Для цього треба натиснути клавішу *Alt* і ту букву, яка виділена червоною в назві пункту меню. Меню також дозволяє працювати з мишею.

В робочої області є можливість відкривати різні вікна програми – вікна редагованого тексту, вікна допомоги, відладки і настройки. У вищенаведеному прикладі відкрито тільки одне вікно – вікно тексту програми. У *заголовку вікна* написано *ім'я файлу* – початкового тексту програми.

Рядок стану демонструє деякі доступні і важливі в даний момент операції і відповідні ним комбінації клавіш.

13.2. Основні команди і гарячі клавіші

Функціональні клавіші використовуються для управління середовищем *Turbo Pascal*. Вони мають позначення *F1, F2, ..., F12* і розташовуються в самому верхньому ряду клавіатури. З кожною з цих клавіш пов'язується певна команда меню. Дію майже усіх функціональних клавіш можна модифікувати трьома особливими клавішами: *Alt* (від ALTernative – додатковий), *Ctrl* (ConTRoL керуючий) і *Shift* (SHIFT зсувний). Ці клавіші використовуються подібно клавіші тимчасової зміни регістру на друкарській машинці: потрібно натиснути на одну з них і потім, не відпускаючи її, натиснути функціональну клавішу. В подальшому таке спільне натискання двох клавіш будемо позначати знаком додавання. Наприклад, *Alt+F3* означає, що разом з клавішею *Alt* необхідно натиснути клавішу *F3*, *Ctrl+F9* – разом з *Ctrl* натискається *F9* тощо.

Нижче наводяться команди, які передаються середовищу *Turbo Pascal* функціональними клавішами і деякими їх комбінаціями з клавішами <Ctrl> і <Alt>:

▪ *F1* – звернутися за довідкою до вбудованої довідкової служби (Help – допомога);

▪ *F2* – записати текст, що редагується у файл на диск;

▪ *F3* – прочитати текст з дискового файлу у вікно редактора;

▪ *F4* – використовується в режимі відладки: почати або продовжити виконання програми і зупинитися перед виконанням того її рядка, на якому стоїть курсор;

▪ *F5* – „розгорнути" активне вікно на весь екран;

▪ *F6* – зробити активним наступне вікно;

▪ *F7* – застосовується в режимі відладки: виконати наступний рядок програми; якщо в рядку є звернення до процедури (функції), увійти в цю процедуру і зупинитися перед виконанням першого її оператора;

▪ *F8* – використовується в режимі відладки: виконати наступний рядок програми; якщо в рядку є звернення до процедури (функції), виконати її і не простежувати її роботу;

▪ *F9* компілювати програму, але не виконувати її;

▪ *F10* – перейти до діалогового вибору режиму роботи за допомогою головного меню;

▪ *Ctrl+F1* – довідка про оператор, на якому встановлений курсор;

▪ *Ctrl+F9* – виконати прогін програми: компілювати програму, яка знаходиться в редакторі, завантажити її в оперативну пам'ять і виконати, після чого повернутися в середовище *Turbo Pascal*;

▪ *Alt+F3* – закриття активного вікна;

▪ *Alt + F5* – змінити вікно редактора на вікно виведення результатів роботи (прогону).

▪ *Alt+X* – вихід з середовища *Turbo Pascal*;

▪ *Alt+Backspace* – відміна останньої зміни;

▪ *Ctrl+Y* – видалення рядка;

▪ *Shift+стрілки* – виділення блоку тексту;

▪ *Ctrl+Insert* – копіювання виділеного блоку в буфер;

▪ *Shift+Insert* – вставка з буфера.

Повний опис функціональних клавіш міститься в спеціалізованій літературі з програмування в середовищу *Turbo Pascal,* наприклад в [78].

13.3. Вихід з середовища Turbo Pascal

Для виходу з середовища *Turbo Pascal* треба натиснути клавіші *Alt+F10.* Другий спосіб більш довгий: необхідно активізувати верхнє горизонтальне меню, натиснувши клавішу *F10*, обрати меню *File* за допомогою клавіш горизонтального переміщення курсору, натиснути клавішу *Enter*; далі використовуючи клавіші вертикального переміщення курсору обрати останній пункт *Exit* спадного підменю і ще раз натиснути клавішу *Enter.*

13.4. Відкриття нового вікна

Якщо ви вже знаходитесь в середовищі *Turbo Pascal*, то для виведення нового вікна редагування, необхідного, наприклад, для набору тексту нової програми, треба виконати команди: *F10 ⇨ File ⇨ Enter ⇨ New ⇨ Enter.*

13.5. Відкриття вже існуючого файлу (програми)

Відкрити вже існуючий файл дозволяють такі команди:

F10 ⇨ File ⇨ Enter ⇨ Open ⇨ Enter ⇨ Tab (вибрати файл) ⇨ Enter.

Перші п'ять команд можна замінити натисканням клавіші *F3.* Команда *Alt+ F3* вилучає з екрану поточне вікно.

13.6. Набір і редагування програми

Введення і редагування тексту програми здійснюється за допомогою клавіатури.

Текстовий редактор середовища *Turbo Pascal* надає користувачу зручні засоби створення і редагування текстів програм. Ознакою того, що середовище перебуває в стані редагування, є наявність у вікні редактора курсору невеликого миготливого прямокутника. Режим редагування автоматично встановлюється відразу після завантаження *Turbo Pascal.* З режиму редагування можна перейти до будь-якого іншого другому режиму роботи *Turbo Pascal* за допомогою функціональних клавіш або шляхом вибору потрібного режиму з головного меню. Якщо середовище перебуває в стані вибору з меню, курсор зникає, а в рядку меню з'являється кольоровий покажчик-прямокутник, який виділяє одне з кодових слів (опцій меню). Для переходу від стану вибору режиму з головного меню в стан редагування потрібно натиснути клавішу *Esc* (ESCape вислизати, тікати), а для переходу до вибору з головного меню – *F10.*

Розглянемо основні прийоми роботи з текстовим редактором. Для створення тексту програми потрібно ввести цей текст за допомогою клавіатури ПК подібно тому, як це робиться при друкуванні тексту у будь-якому текстовому редакторі, наприклад у Microsoft Word. Після заповнення чергового рядку треба натиснути на клавішу *Enter*, щоб перевести курсор на наступний рядок (курсор завжди показує те місце на екрані, куди буде поміщений черговий символ програми, що вводиться).

Вікно редактора імітує довгий і достатньо широкий аркуш паперу, фрагмент якого видний у вікні. Якщо курсор досяг нижнього краю, здійснюється прокручування вікна редактора: його вміст зміщається вверх на один рядок і знизу з'являється новий рядок аркуша. Якщо курсор досяг правої границі екрана, вікно починає по мірі введення символів зміщатися праворуч, показуючи правий край аркуша. Розміри аркуша за горизонталлю і вертикаллю обмежуються тільки загальною кількістю символів у файлі, яких не повинно бути більше 64 535, однак компілятор *Turbo Pascal* сприймає рядки програми довжиною не більше 126 символів.

Вікно можна зміщувати відносно аркуша за допомогою наступних клавіш:

- *Page Up* на сторінку нагору;
- *Page Down* – на сторінку вниз;
- *Home* – на початок поточного рядка;
- *End* – в кінець поточного рядка;
- *Ctrl+Page Up* – на початок тексту;
- *Ctrl+Page Down* – в кінець тексту.

Клавішами управління курсором (ці клавіші позначені стрілками і розташовуються у правій частині клавіатури) його можна переміщувати по екрану. При досягненні границь вікна воно зміщується на рядок або на символ.

Якщо Ви помилилися при введенні чергового символу, його можна витерти за допомогою клавіші зі стрілкою (або клавішею *Backspace*), яка розташована над клавішею *Enter.* Клавіша *Delete* витирає символ, на який в даний момент вказує курсор, а Ctrl+Y – весь рядок, на якому розташований курсор.

Треба пам'ятати, що редактор *Turbo Pascal* вставляє наприкінці кожного рядка невидимий символ-роздільник. Цей символ вставляється клавішею Enter, а стирається клавішами *Backspace* або *Delete.* За допомогою вставлення/стирання роздільника можна „розрізати"/„склеїти" рядки. Щоб розрізати рядок, треба підвести курсор до потрібного місця і натиснути *Enter*, щоб склеїти сусідні рядки, потрібно встановити курсор в кінець першого рядка (для цього зручно використовувати клавішу *End*) і натиснути *Delete* або встановити курсор на початок другого рядка (клавішею *Home*) і натиснути *Backspaces.*

Нормальним режимом роботи редактора є режим вставки, у якому кожний новий символ, що вводиться начебто „розсовує" текст на екрані, зміщуючи праворуч залишок рядка. Треба мати на увазі, що розрізування тексту і наступна вставка пропущених рядків можливі тільки в цьому режимі.

Редактор може також працювати в режимі накладення нових символів на існуючий старий текст: у цьому режимі новий символ замінює собою той символ, на який вказує курсор,

а залишок рядка не зміщується праворуч. Для переходу в режим накладення потрібно натиснути клавішу *Insert*, а якщо натиснути цю клавішу ще раз, знову встановиться режим вставки. Ознакою того, в якому режимі працює редактор, є форма курсору: в режимі вставки він схожий на миготливий символ підкреслення, а в режимі накладення він представляє собою великий миготливий прямокутник, який повністю заслоняє символ.

І ще про одну можливість редактора. Зазвичай редактор працює в режимі автовідступу. В цьому режимі кожний новий рядок починається у тій же позиції на екрані, що й попередній. Режим автовідступу підтримує гарний стиль оформлення тексту програми відступи від лівого краю виділяють тіло умовного або складового оператора і роблять програму більш наочною. Відмовитися від автовідступу можна клавішами *Ctrl+O, I* (при натиснутій *Ctrl* натискається спочатку клавіша з латинською літерою *O*, а потім O відпускається і натискується клавіша *I*), повторна команда *Ctrl+O, I* відновить режим автовідступу.

Нижче перераховані найбільш часто використовувані команди редактора *Turbo Pascal*.

Зсув курсору:

- ✓ *Page Up* – на сторінку нагору;
- ✓ *Page Down* – на сторінку вниз;
- ✓ *Home* – на початок поточного рядка;
- ✓ *End* – в кінець поточного рядка;
- ✓ *Ctrl+Page Up* – на початок тексту;
- ✓ *Ctrl+Page Down* – в кінець тексту.

Команди редагування:

- ◆ *Backspace* – витирає символ ліворуч від курсору;
- ◆ *Delete* – витирає символ, на який вказує курсор;
- ◆ *Home* – витирає рядок з курсором;
- ◆ Enter – вставляє новий рядок, розрізає старий;
- ◆ Ctrl+Q, L – відновлює змінений рядок (діє, якщо курсор не залишав рядок після його зміни).

Робота з блоком:

- Ctri+K>, B – починає виділення блока;
- Ctrl+K>, K – закінчує виділення блока;
- Ctrl+K>, Y – знищує виділений блок;
- Ctrl+K>, C – копіює блок;
- Ctrl+K>, V – переміщує блок на нове місце;
- Ctrl+K>, W – записує блок у файл;
- Ctrl+K>, R – зчитує блок з файлу;
- Ctrl+K>, P – друкує блок.

13.7. Робота з блоками тексту

Для зручності редагування тексту, розробки структурованих програм дуже корисно з перших же кроків освоїти роботу з блоками тексту: виділення блоку, його копіювання, зсув, переміщення і видалення.

Виділення блоку тексту (при цьому він підсвічується) можна здійснити різними способами.

1. Якщо блок невеликий (слово, декілька слів, рядок, декілька рядків або екран), то зручніше усього виділяти його за допомогою одночасного натискання клавіш *Shift* і однієї з клавіш керування курсором: ↑, ↓, →, ←, *Home, End, PgUp, PgDn*. Зняти виділення усього блока або його частини можна аналогічним чином, використовуючи клавіші управління курсором *Ctrl+K+H* (утримуючи клавішу *Ctrl* і послідовно клацаючи по клавішах *K* і *H*), за допомогою цих же клавіш його знову можна включити, якщо не був ще виділений якийсь інший блок.

2. Встановити курсор на початок блоку і натиснути клавіші *Ctrl+K+B*. Перевести курсор на кінець блоку і натиснути клавіші *Ctrl+K+K*. Можна спочатку визначити кінець блока, а потім його початок. Зняття і відновлення виділення (підсвічування) також здійснюється переключенням клавіш *Ctrl+K+K*.

Виділений блок можна видалити (*Ctrl+Del* або *Ctrl+K+Y)*, скопіювати (*Ctrl+K+C)* або перемістити *(Ctrl+K+V)* в місце розташування курсору.

Для створення відступів при структуруванні програми використовують зсув блока праворуч (*Ctrl+K+I)* або ліворуч (*Ctrl+K+U)* на одну позицію при одноразовому натисканні.

Часто при редагуванні буває зручним записувати виділений блок у буфер обміну *(Ctrl+Ins)*, а потім копіювати *(Ctrl+ Shift)* його будь-яку кількість разів у різні місця поточної або будь-якої іншої програми (в місце розташування курсору).

Блок в буфері обміну зберігається до тих пір, поки Ви не помістите до нього новий блок.

Перераховані операції з блоками істотно економлять час, який витрачається на написання програм. При цьому економія часу відбувається в результаті дії двох причин. По-перше, Ви практично не витрачаєте час на набір вже існуючих або подібних блоків. По-друге,

копіювання блоків влагоджених програм страхує Вас від прикрих помилок, які на жаль, легко допускаються в процесі набору тексту. Тому рекомендується всі ідентичні або схожі ідентифікатори і блоки з довжиною більше двох-трьох літер не набирати, а копіювати.

13.8. Збереження програми

Програма, яку Ви набрали у вікні редагування, існує тільки в оперативній пам'яті комп'ютера. Тому навіть при короткочасному відключенні електроживлення, перезапуску комп'ютера або при виході з середовища *Turbo Pascal* вона зникне назавжди. Для її збереження зручніше усього натиснути клавішу *F2.* При першому натисненні клавіші *F2* в діалоговому вікні необхідно буде ввести ім'я файлу. Доцільно це зробити так, щоб ім'я файлу співпадало з ім'ям програми. У цьому випадку легко здійснюється пошук програму у списку каталогів.

Ім'я файлу не повинне містити більше 8 дозволених символів. Ім'я файлу на диску має розширення *.pas*.

Можна також зберегти програми за допомогою команд *F10 ⇨ File ⇨Enter ⇨Save (набрати ім'я файлу) ⇨Enter*.

Рекомендуємо клавішу *F2* натискувати періодично після набору двох трьох рядків тексту, коли Ви забажали трішки відпочити або при необхідності над чимось розміркувати. Це виключить втрату вже набраної частини програми, які на жаль, часто трапляються при наборі тексту.

13.9. Перейменування імен файлів з програмами

Часто виникає необхідність зробити копію програми, наприклад:

- для збереження оригіналу програми перед його модернізацією;
- для збереження модернізованої програми у новому файлі;
- для запису програми в новому каталозі або на диску, дискеті, флешці тощо.

У всіх цих випадках мова йде про перейменування файлів з програмами, оскільки в одному і тому ж каталозі не може бути однакових імен файлів.

Для збереження файлу з програмою під новим ім'ям (при незмінності оригіналу) необхідно спочатку ввести команди:

F10 ⇨ File ⇨Enter ⇨Save as...

В результаті виконаних дій з'явиться вікно, аналогічне першому введенні команди *Save* (або *F2*). В цьому вікні для збереження файлу в поточному каталозі необхідно набрати нове ім'я файлу, для його збереження у будь-якому іншому каталозі необхідно ввести ім'я файлу і вказати шлях до нього, а потім натиснути клавішу *Enter.*

13.10. Компіляція програми

Набрану за допомогою текстового редактора програму необхідно відкомпілювати – отримати машинний код, який складається з двійкових даних і інструкцій процесора. Комп'ютер виконує програму тільки у вигляді машинного коду. Задачу перекладу тексту програми в машинний код вирішує спеціальна програма, яка називається компілятором.

При компіляції до програми додаються підпрограми з модулів, які оголошені в директиві компілятора *uses*, коміркам пам'яті привласнюються імена констант і змінних, оголошених в розділах програми *const* і *var.* Розмір цих комірок встановлюється таким, щоб в них помістились значення зазначених констант і змінних.

Тип константи (ціле або дійсне число, текст, символ тощо) визначаються автоматично на підставі її структури (наприклад, 2012 – ціле число, а 3.1415926 і 12.0 – речовинні числа) і символів, з яких вони складаються. Зазначені в програмі константи відразу поміщуються у відповідну комірку пам'яті.

Розмір комірки під змінну визначається типом змінної, зазначеної після її ідентифікатора. Наприклад, для змінної типу *real* потрібна комірка розміром 6 байт.

Компіляція програми здійснюється натисненням клавіш *Alt+F9* (або введенням команд: *F10 ⇨ Compile ⇨Enter ⇨ Compile*), в результаті буде отриманий файл з тим же ім'ям, що і файл, який містить програм, що компілюється, але з розширенням *.exe*. Цей файл може розташовуватись або тільки в оперативній пам'яті, або в оперативній пам'яті й на диску.

Якщо є бажання, щоб виконуваний файл розміщувався не в поточній, а в заданій директорії, тоді вказати шлях до цієї директорії у відповідному вікні, яке з'явиться після введення команд:

F10 ⇨ Option ⇨ Enter ⇨ Directories…

Вибір варіанту розташування цього файлу (диск і оперативна пам'ять або тільки оперативна пам'ять) здійснюється встановленням параметрів *disk* або *memory* за допомогою команд:

F10 ⇨ Compile ⇨ Enter ⇨ Destination ⇨Enter.

Для запуску виконуваного файлу, який розташовується на диску середовище *Turbo Pascal* не потрібне, він запускається стандартним чином в командному рядку *MS-DOS* або *NC.*

Модернізувати такий файл неможливо. Тому первісно, на стадії відлагоджування, програму рекомендується компілювати тільки в оперативну пам'ять. Це прискорює процес компіляції, і диск не захаращується найчастіше некорисними виконуваними файлами. В багатьох випадках можна обмежитись файлами з текстами програми. Справа в тому, що при запуску програми (файлу з розширенням *.pas*) для проведення розрахунків в середовищі *Turbo Pascal* завжди автоматично спочатку виконується компіляція програми зі створенням виконуваного файлу в оперативній пам'яті.

Якщо ж є необхідність мати на диску виконуваний файл повністю відлагодженої, не потребуючої подальшої модернізації програми, то в командному рядку *MS-DOS* введіть команди:

TPC Exe_File.pas ⇨ Enter.

Тут Exe_File.pas – ім'я файлу повністю відлагодженої програми, для якої потрібно отримати файл з розширенням *.exe.*

Головна користь від команди *Alt+F9* полягає у створенні не стільки виконуваного файлу, скільки в допомозі розробнику у вияві помилок програми. Компілятор виявляє помилки двох видів:

– синтаксичні, які виникають в результаті порушення правил написання пропозицій *Turbo Pascal;*

– семантичні, які пов'язані з неприпустимими значеннями параметрів, неприпустимими діями над параметрами тощо.

Якщо помилка виявлена, то компіляція завершується (при цьому виконуваний файл не створюється), виводиться повідомлення про сутність помилки, а курсор вказує місце в тексті програми, де вона знаходиться. Помилку потрібно виправити, а потім знову відкомпілювати програму. Однак при цьому не треба цілком довірятись рекомендаціям компілятора: іноді вони бувають невірними, помилка може знаходитись в іншому місці програми і її сутність може бути зовсім іншою.

При успішному завершенні компіляції не екран видається повідомлення: «*Compile successful: press any key*».

13.11. Запуск і відладка програми

Після підготовки тексту програми можна спробувати її виконати, тобто відкомпілювати програму, зв'язати її (якщо це необхідно) з бібліотекою стандартних процедур і функцій, завантажити в оперативну пам'ять і передати їй управління. Вся ця послідовність дій називається *прогоном програми* і реалізується комбінацією клавіш *Ctrl+F9* або послідовністю команд:

F10⇨ Enter ⇨ Run⇨ Enter.

Якщо в програмі відсутні синтаксичні помилки, то всі дії виконуються послідовно одна за одною, при цьому в невеликому вікні повідомляється про кількість відкомпільованих рядків і об'ємі доступної оперативної пам'яті. Перед передачею управління завантаженій програмі середовище *Turbo Pascal* очищує екран (точніше, виводить на екран вікно прогону програми), а після завершення роботи програми знову берет управління комп'ютером на себе і відновлює на екрані вікно редактора.

Якщо на певному етапі середовище *Turbo Pascal* виявить помилку, воно припиняє подальші дії, відновлює вікно редактора і поміщує курсор на той рядок програми, при компіляції або виконанні якого виявлена помилка. При цьому у верхньому рядку редактора з'являється діагностичне повідомлення про причину помилки. Все це дозволяє дуже швидко відлагодити програму, тобто усунути в ній синтаксичні помилки і переконатися в правильності її роботи. Якщо помилка виникла на етапі прогону програми, проста вказівка того місця, де вона виявлена, може не дати потрібної інформації, оскільки помилка може бути наслідком неправильної підготовки даних в попередніх операторах програми.

На стадії виконання програм найчастіше „випливають" такі помилки:

- ділення на нуль;
- обчислення квадратного кореня з від'ємного числа;
- переповнення (вихід за праву границю діапазону типа) через занадто великий показник експоненти.

Наприклад, якщо помилка виникла в результаті добування квадратного кореня з від'ємного числа, буде вказаний оператор, у якому здійснюється добування кореня, хоча зрозуміло, що першопричину помилки потрібно шукати десь раніше, там, де відповідній змінній присвоюється від'ємне значення. В таких ситуаціях зазвичай вдаються до покрокового виконання програми за допомогою команд, пов'язаних з клавішами *F4, F7* і *F8*. Поки ще не накопичений достатній досвід відладки, можна скористатися однією клавішею F7, після натискання на яку середовище *Turbo Pascal* здійснить компіляцію, компоновку (зв'язок з бібліотекою стандартних процедур і функцій) і завантаження програми, а потім зупинить прогін перед виконанням першого оператора.

Рядок програми, яка містить цей оператор, буде виділена на екрані покажчиком (кольором). Тепер кожне нове натискання F7 буде викликати виконання усіх операцій, запрограмованих в поточному рядку, і зсув покажчика до наступного рядка програми.

В „підозрілому" місці програми можна переглянути поточне значення змінної або виразу. Для цього потрібно встановити курсор в те місце рядка, де перебуває ім'я змінної, яка вас цікавить, і натиснути *Ctrl+F4*. На екрані з'явиться діалогове вікно, що складається з трьох полів (у верхньому полі буде стояти ім'я змінної, два інших поля будуть порожніми). Натисніть *Enter*, щоб в середньому полі отримати поточне значення змінної. Якщо перед натисненням *Ctrl+F4* курсор стояв на пустій ділянці рядка або вказував на ім'я іншої змінної, верхнє поле діалогового вікна також буде порожнім або містити ім'я цієї іншої змінної. У цьому випадку треба ввести за допомогою клавіатури ім'я потрібної змінної і натиснути *Enter*.

До речі, таким чином можна вводити не тільки імена змінних, що простежуються, але й вирази – середовище *Turbo Pascal* обчислить і покаже значення введеного виразу.

Логічні помилки, що пов'язані з неправильним використанням тих або інших алгоритмічних конструкцій, не призводять до зупинки програми. Виявити їх можна тільки за допомогою аналізу проміжних і остаточних результатів. Тому при відлагодженні програми в її текст доцільно впроваджувати друк проміжних результатів.

В процесі відладки будь-який фрагмент програми можна виключити з її роботи шляхом перетворення його в коментар з використанням фігурних дужок. Якщо ж фігурні дужки вже використані у блоці, який виключається, то „закоментувати" блок можна за допомогою комбінованих скобок (*...*). Можна навпаки – спочатку використати комбіновані дужки, а потім, для виключення фрагментів програми, – фігурні.

Результати роботи програми виводяться на екран за допомогою команди *Alt+F5*. Зупинити виконання задачі можна командою *Ctrl+Break*.

13.12. Довідкова служба Turbo Pascal

Невід'ємною складовою частиною середовища *Turbo Pascal* є вбудована довідкова служба. Якщо Ви достатньо добре володієте англійською мовою, у вас не буде проблем при роботі з *Turbo Pascal*, в скрутній ситуації достатньо натиснути *F1* і на екрані з'явиться необхідна довідка. Ця довідка залежить від поточного стану середовища *Turbo Pascal*, тому довідкову службу називають контекстно-чутливою. Наприклад, якщо натиснути *F1* в момент, коли середовища *Turbo Pascal* виявило помилку в програмі, в довідці будуть повідомлені додаткові відомості про причини помилки і надані рекомендації з їх усунення.

Існують чотири способи звертання до довідкової служби безпосередньо з вікна редактора:

- *F1* – одержання контекстно-залежної довідки;
- *Shift+F1*– вибір довідки зі списку доступних довідкових повідомлень;
- *Ctrl+F1* – одержання довідки про потрібну стандартну процедуру, функцію, стандартну константу або змінну;
- *Alt+F1* – одержання попередньої довідки.

При натисканні комбінації клавіш *Shift+F1* на екрані з'явиться вікно, яке містить упорядкований за алфавітом список стандартних процедур, функцій, типів, констант і змінних, для яких можна отримати довідкову інформацію.

Цю ж довідку можна отримати й іншим чином. Надрукуйте на екрані ім'я процедури (функції, типу тощо) або підведіть курсор до наявних в тексті стандартних імен і натисніть *Ctrl+F1*. Середовище *Turbo Pascal* проаналізує найближче оточення курсору, виділить ім'я і надасть потрібну довідку.

В більшості випадків довідка містить невеличкий приклад, який ілюструє відповідні можливості середовища *Turbo Pascal*. Не поспішаєте запам'ятовувати його або записувати на аркуші паперу: його можна „вирізати" з довідки і перенести у вікно редактора. Для цього після виклику довідки натисніть *Alt+E*, виберіть в додатковому меню, що з'явиться, пункт *Copy examples* (Копіювати приклади) і натисніть *Enter* – текст прикладу буде скопійований у внутрішній буфер редактора. Для витягу прикладу з буфера натисніть клавішу *Esc*, щоб вийти з довідкової служби, потім підведіть курсор до вільного рядка у вікні редактора, натисніть *Shift+Insert* (копіювання вмісту буфера в текст програми) і *Ctrl+K*, *H*, щоб усунути виділення скопійованого тексту кольором.

Питання для самоконтролю

1. *Як запустити інтегроване середовище програмування Turbo Pascal?*
2. *Які основні команди і гарячі клавіші середовища Turbo Pascal Вам відомі?*
3. *Як здійснюється вихід з середовища Turbo Pascal?*
4. *Як можна відкрити вже існуючу програму (файл)?*
5. *Як можна здійснити набір і редагування програми в середовищі Turbo Pascal?*
6. *Яку роботу можна виконати з блоками тексту програми?*
7. *Як можна зберегти програму в середовищі Turbo Pascal?*
8. *Як здійснюється перейменування файлів у програмі в середовищі Turbo Pascal?*
9. *Яким чином можна здійснити компіляцію програми в середовищі Turbo Pascal?*
10. *Яким чином можна здійснити запуск і від лагодження програми в середовищі Turbo Pascal?*

14. ПРОГРАМУВАННЯ ЦИКЛІВ В СЕРЕДОВИЩІ PASCAL

Навіть ще не маючи досвіду у створенні алгоритмів, людина інтуїтивно розуміє, що вони різняться своєю структурою. Так, є алгоритми, що виконуються за будь-яких обставин. Але таке трапляється нечасто, тому що людина завжди коригує свої плани залежно від оточуючих умов, і тому виникає ситуація „якщо трапиться...", „якщо зустрінуся...", „якщо встигну..." тощо. А іноді ми змушені повторювати певний процес декілька разів, доки не отримаємо бажаного результату. Найчастіше ж ми й умови враховуємо, й повторюємо щось. Ось так і виникають різні типи алгоритмів.

14.1. Загальні визначення

Найчастіше задача, яку треба розв'язати, сформульована не математичною мовою. Для її вирішення на комп'ютері її спочатку потрібно перевести у форму математичної задачі, а потім вже програмувати.

Робота з розв'язання таких задач з використанням комп'ютера проходить через такі етапи:

1) Постановка задачі.
2) Математична формалізація.
3) Побудова алгоритму.
4) Складання програми на мові програмування.
5) Налагодження і тестування програми.
6) Проведення розрахунків й аналізу отриманих результатів

Цю послідовність називають технологією розв'язання задач на комп'ютері.

На етапі постановки задачі повинно бути чітко визначено, що дано і що потрібно знайти.

Другий етап – математична формалізація. Тут задача перекладається на мову математичних формул, рівнянь, відношень. Далеко не завжди ці формули є очевидними. Нерідко їх доводиться виводити самому або відшукувати в спеціальній літературі. Якщо розв'язок задачі вимагає математичного опису певного реального об'єкта, явища або процесу, то формалізація рівносильна одержанню відповідної математичної моделі.

Третій етап – побудова алгоритму. Способи опису алгоритмів були розглянуті і 1 розділі.

Перші три етапи – це робота без комп'ютера. Далі власне йде програмування на певній мові у певній системі програмування.

Останній (шостий) етап – це використання вже розробленої програми в практичних цілях.

Будь-яку програму, яка описує алгоритм розв'язання тієї або іншої задачі, можна уявити себе як послідовність команд, які необхідно виконати над даними за допомогою певних керуючих команд, які показують послідовність обчислень залежно від отриманих результатів.

Всього існують чотири базових структури алгоритмів:

- лінійні;
- розгалужені;
- циклічні;
- змішані.

Лінійним називається алгоритм, в якому всі команди виконуються однократно одна за одною у строгій послідовності (рис.14.1).

Першим типом операторів лінійної ділянки програми можна вважати оператори опису змінних. Кожна змінна має ім'я, значення і тип. Для простоти можна уявити собі змінну, як область пам'яті – скриньку, на кришці якої написане її ім'я, в скриньці лежить значення, а збоку бовтається бирка з вказівкою типу значення, яке може лежати у цій скриньці.

В програмі ім'я змінної (наприклад, A, B$, C%) слугує для позначення певної величини. Під час виконання програми у кожний конкретний момент часу величина має одне значення (в скриньці не можуть лежати два різних значення, вони там просто не помістяться). Скринька може бути й порожньою, тобто на даний момент значення змінної ще не визначене, але це не означає, що скринька буде пустою увесь час, згодом їй довірять певне значення.

Якщо значенням змінної є тільки ціле число, то змінна називається цілочисельною. В програмі можуть бути й інші типи змінних, наприклад, дійсні (дробові) і літерні (символьні). Залежно від типу даних вони неоднаково будуть подаватись в комп'ютерній системі.

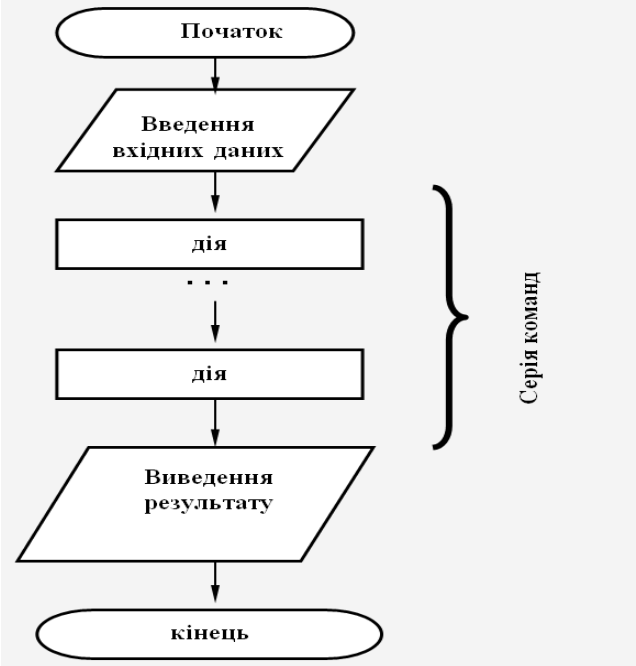


Рис.14.1. Лінійний алгоритм

Лінійний алгоритм називають іноді елементарним – у ньому усі команди для процесора розписані послідовно, а значить, і виконуються також – одна за одною, вздовж лінійки, без відхилень, без умов (рис.14.2). Такі команди називають ще серією команд.

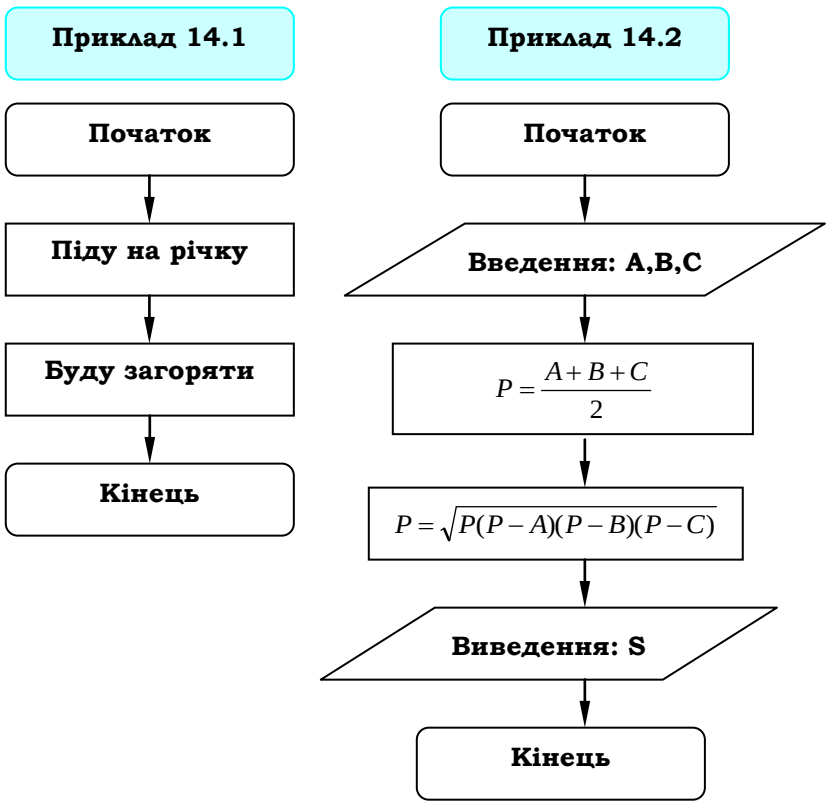


Рис.14.2. Приклади лінійних алгоритмів

На практиці рідко вдається представити схему алгоритму розв'язання задачі у вигляді лінійної структури. Найчастіше в залежності від певних значень проміжних результатів необхідно організувати обчислення або одними, або за іншими формулами. Розгалуження – така схема, яка передбачає розгалуження зазначеної послідовності дій на два напрямки залежно від підсумку перевірки заданої умови.

Алгоритм розгалуженої структури – алгоритм, в якому послідовність дій залежить від результату перевірки певної умови. Тобто така структура забезпечує залежно від результату перевірки умови (так або ні) вибір одного з альтернативних шляхів роботи алгоритму (рис.14.3).

Кожен зі шляхів веде до загального виходу, так що робота алгоритму буде тривати незалежно від того, який шлях буде обраний.

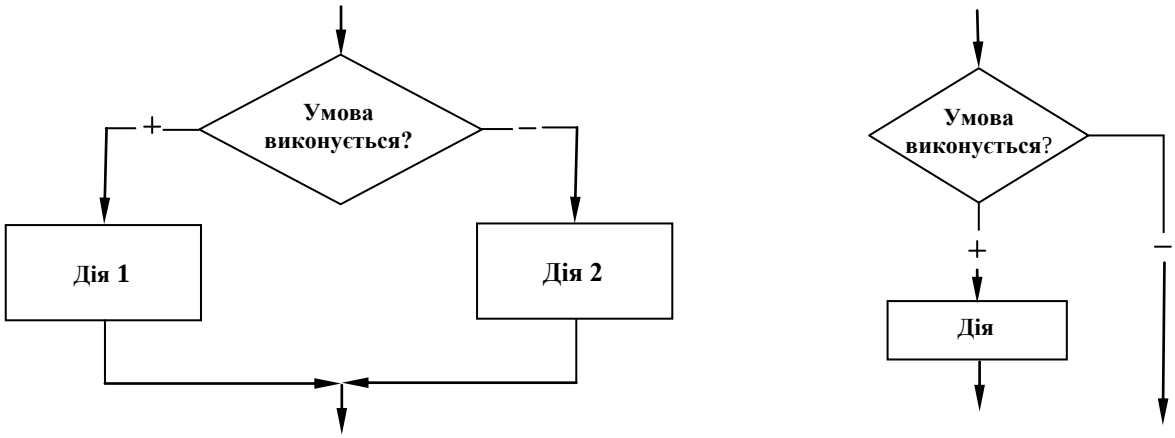


Рис.14.3. Приклади розгалужених алгоритмів

В схемах такої структури використовується логічний блок. Розгалужена структура існує в чотирьох основних варіантах:

– якщо – то:
якщо *умова* то *дії*
кінець (рис.14.4);

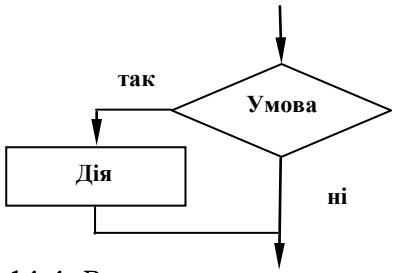


Рис.14.4. Розгалужена структура: „якщо – то”

– якщо - то – інакше:
якщо *умова*
то *дія 1*
інакше *дія* 2
кінець (рис.14.5);

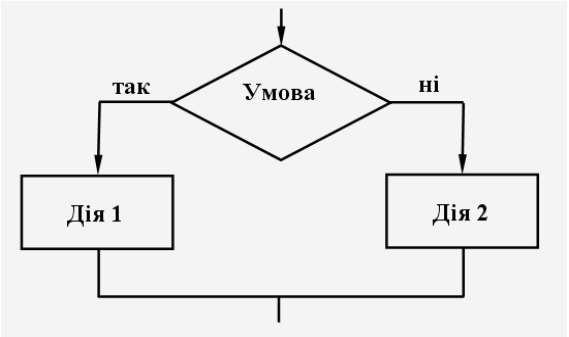


Рис.14.5. Розгалужена структура: „ якщо - то – інакше ”

– вибір:
вибір
при *умові* 1: *дія* 1
при *умові* 2: *дія* 2
.....
при *умові* N: *дія* N
кінець (рис.14.6);

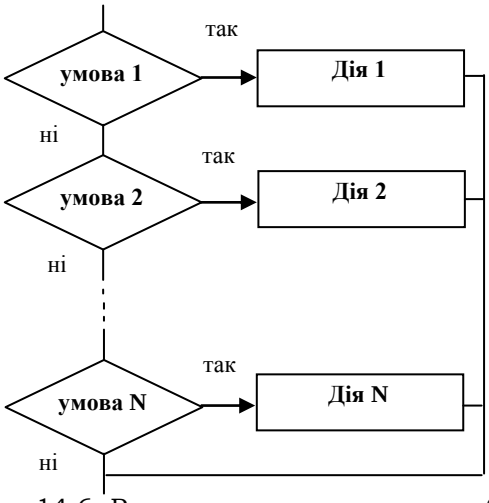


Рис.14.6. Розгалужена структура: „вибір”

– вибір – інакше:
при *умові* 1: *дія* 1
при *умові* 2: *дія* 2
.....
при *умові* N: *дія* N
інакше дія N+1
кінець (рис.14.7).

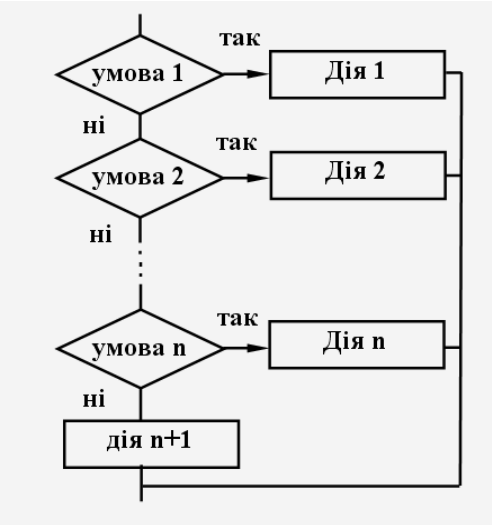


Рис.14.7. Розгалужена структура: „ вибір – інакше”

<u>Цикл</u> – різновид керуючої конструкції у високорівневих мовах програмування, який призначений для організації багаторазового виконання набору інструкцій (команд). Також циклом може називатися будь-яка багатократно виконувана послідовність команд, організована будь-яким чином (наприклад, із допомогою умовного переходу).

Обчислювальний процес, який містить цикл, називається циклічним.

Одноразове виконання тіла циклу називається ітерацією. Вираз, що визначає чи буде вчергове виконуватися ітерація, чи цикл завершиться, називається умовою виходу або умовою завершення циклу (або умовою продовження в залежності від того, як інтерпретується його істинність – як ознака необхідності завершення або продовження циклу.

Змінна, в якій зберігається номер поточної ітерації, називається лічильником ітерацій циклу або просто лічильником циклу. Цикл не обов'язково містить лічильник, також лічильник не зобов’язаний бути одним – умова виходу із циклу може залежати від декількох змінюваних в циклі змінних, а може визначатися зовнішніми умовами (наприклад, настанням певного часу), в останньому випадку лічильник взагалі не знадобиться.

Управління повторенням циклу здійснюється за допомогою змінної, яка називається параметром циклу. Спочатку цьому параметру надають певне початкове значення. Потім цикл виконується зі зміною параметру при кожному повторенні від початкового до кінцевого значення на величину, яка називається кроком циклу.

Крок циклу може бути додатним або від'ємним. Залежно від цього параметр циклу зростає або зменшується. Цикл припиняється, якщо значення параметра лежить поза межами діапазону між початковим і кінцевим значеннями.

Частинами виконання будь-якого циклу є початкова ініціалізація змінних циклу, перевірка умови виходу, виконання тіла циклу і оновлення змінної циклу на кожній ітерації. Крім того, більшість мов програмування надають засоби для дострокового керування циклом, наприклад, оператори завершення циклу, тобто виходу з циклу незалежно від істинності умови виходу і оператори пропущення ітерації.

Залежно від того, чи знаємо ми скільки разів необхідно повторювати якусь послідовність команд, розрізняють цикли з лічильником (кількість повторень відома заздалегідь) та цикли з умовою (цикл повторюється доти, доки не виконається певна умова).

Алгоритми, окремі дії в яких багаторазово повторюються, називаються алгоритмами циклічної структури (повторення). Сукупність дій алгоритму, пов'язану з повторенням, називають циклом.

Алгоритми з циклічною структурою – це такі алгоритми, й яких одні й ті ж вказівки (команди) виконуються багаторазово над різними значеннями змінних.

Послідовність дій, які повторюються в циклі, називають тілом циклу.

В алгоритмах з циклічною структурою виділяють такі основні частини:

– підготовчу;

– робочу;

– управління.

Підготовча частина містить вказівки (надалі ми будемо їх ототожнювати з операторами), які задають початкові значення змінним.

Робочу частину (або тіло циклу) утворюють оператори, в яких виконуються основні дії обчислення. Важливими характеристиками циклічних програм є параметр циклу і лічильник циклів.

Для реалізації циклічних структур алгоритмів у мовах програмування високого рівня використовують декілька типів операторів циклу. У середовищі Pascal використовуються такі оператори циклу.

1. Оператор циклу зі заздалегідь відомою кількістю циклів або оператор циклу з параметром.

Цей оператор має таку структуру:

for <змінна>:=a to b do s

де <змінна> – змінна типу *integer*, яку називають параметром циклу (або провідною змінною);

a і *b* – вирази із значенням типу integer;

s – оператор (тіло циклу).

Ключові слова перекладаються так:

for – для,

to – до,

do – виконувати.

В операторі циклу першого типу провідна змінна послідовно приймає такі значення:

a, a+1,a+2,...,b.

Цикл *for* послідовно запускає оператор, що стоїть після слова *do.* При цьому після кожного кроку циклу параметр циклу збільшується з кроком 1 (від 1 до *k*). Для кожного з цих значень виконується оператор *s.* Якщо провідна змінна приймає значення більше, ніж *b,* то виконання оператора *s* припиняється. Таким чином, у цьому випадку реалізується схема виконання оператора, яка зображена на рис. 14.1.

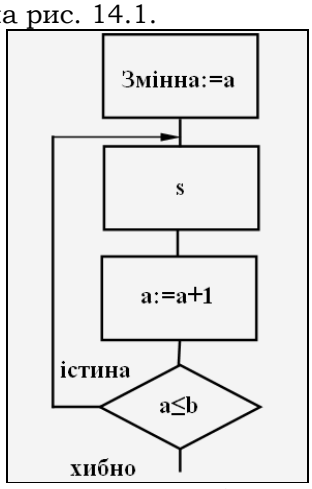


Рис. 14.1. Виконання циклу при умові зростання провідної змінної

Якщо початкове значення змінної більше кінцевого, то використовується така структура оператора:

for <змінна>:=a down to b do s;

У цьому випадку на кожному наступному кроці змінна *a* зменшує своє значення на 1, тобто реалізується схема виконання оператора, зображена на рис. 14.2.

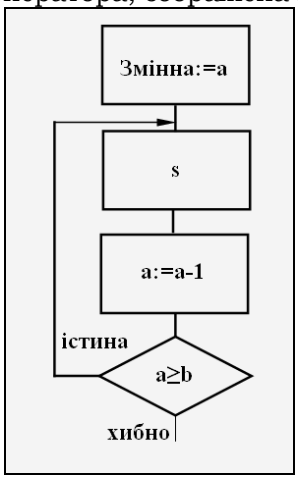


Рис. 14.2. Виконання циклу при умові убування провідної змінної

Якщо тіло циклу містить більше одного оператора, то вони поміщаються в операторні дужки *begin...end*;. Наприклад:

*for i:=l to n do begin s:=s+a; y:=2*x end;*.

2. Оператор while...do, який перевіряє умову перед початком кожного виконання операторів тіла циклу (цикл з передумовою), тобто він реалізує схему, яка зображена на рис. 14.3.

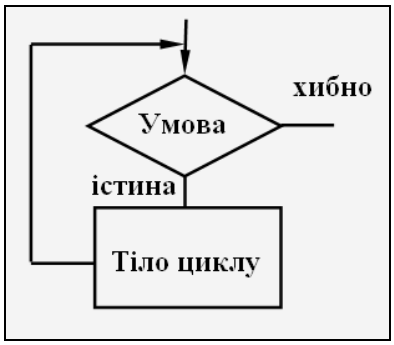


Рис. 14.3. Виконання циклу з передумовою

Структура оператора:

while <логічний вираз> do s;

де: *s* – оператор тіла циклу.

Якщо в тілі циклу міститься декілька операторів, то вони поміщаються в операторні дужки *begin...end*;.

3. Оператор repeat...until – спочатку виконує оператори тіла циклу, потім перевіряє умову (цикл з післяумовою), тобто реалізує схему, зображену на рис. 14.4.

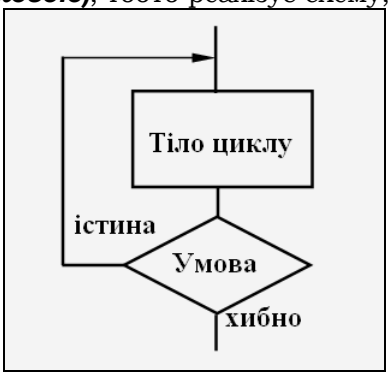


Рис. 14.4. Виконання циклу з післяумовою

Структура оператора:

repeat s until <логічний вираз>;

де: *s* – оператори тіла циклу.

Оператори тіла циклу не поміщаються в операторні дужки навіть у тому випадку, коли їх декілька.

Оператор циклу *repeat* використовується у першу чергу у тих випадках, коли заздалегідь невідомо яку кількість разів треба повторити цикл. Такі ситуації досить часто виникають при розробці програм. Але цей оператор можна застосовувати й для заздалегідь відомою кількістю кроків.

14.2. Рекурентні алгоритми з циклічною структурою

Характерною рисою обчислювальних алгоритмів з циклічною структурою є те, що значення змінних в середині циклу обчислюються рекурентно.

Рекурентність виражається в тому, що значення змінної у кожному циклі обчислюється через значення цієї самої змінної, знайдене на попередньому кроці виконання циклу.

Використання рекурентних обчислень дає змогу зменшити число операцій, що виконуються в циклі, тобто підвищити швидкодію виконання алгоритму.

Для організації рекурентних обчислень необхідно знайти коефіцієнт рекурентної залежності. Для простих рекурентних залежностей цей коефіцієнт визначається через відношення змінної на *i*-тому кроці до цієї самої змінної на *(i–1) – ому* кроці (u_i / u_{i-1}) або через їх різницю $(u_i - u_{i-1})$.

Розглянемо приклади використання рекурентних співвідношень.

Приклад 1. Розробити блок-схему алгоритму і програму обчислення *n* членів арифметичної прогресії і виведення кожного значення на друк.

Якби в даній задачі потрібно було обчислити будь-який один член прогресії, наприклад *i*-тий, то для цього доцільно скористатися формулою: $u = a + (i-l)d$.

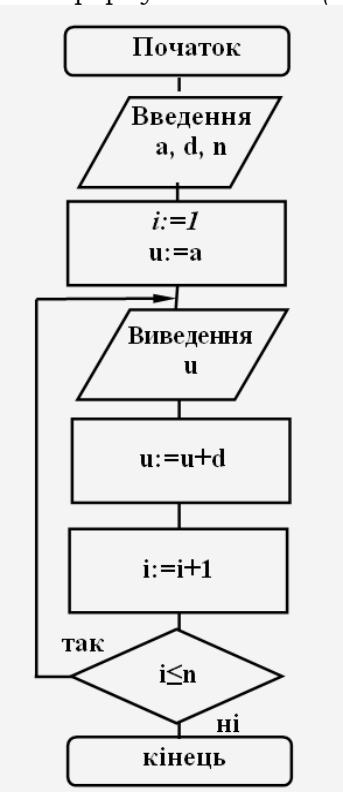


Рис. 14.5. Рекурентний алгоритм з циклічною структурою

Звичайно, цю саму формулу можна було б використати і для обчислення кожного із *n* – членів прогресії. Проте доцільніше використати рекурентну залежність між членами арифметичної прогресії. Для цієї прогресії коефіцієнтам рекурентної залежності є різниця прогресії. Значення членів прогресії рекурентно обчислюється за формулою:

$$u_i = u_{i-1} + d\ ,\ i = 1, 2,..., n;\ u_0 = a.$$

Зазначимо, що в процесі обчислення кожного члена прогресії за формулою $u_i = a+(i-l)d$ потрібно виконати одну операцію множення, одну операцію віднімання і одну операцію додавання. У процесі рекурентного обчислення потрібно виконувати тільки одну операцію додавання.

Блок-схема алгоритму обчислення членів арифметичної прогресії і виведення кожного елемента прогресії зображена на рис. 14.5.

Нижче наведено програму реалізації даного алгоритму.

```
program n2_1_01;
var i,n:integer; a,d,u:real;
begin
      writeln ('уведіть n,a,d'); readln (n,a,d);
       u:=a;                          {перший член прогресії}
       for i:=l to n do begin         {початок циклу}
      writeln ('u=',u); u:=u+d        {оператори тіла циклу}
      end;                            {закриття операторних дужок}
end.
```

14.3. Циклічні обчислювальні алгоритми зі заздалегідь невідомою кількістю циклів

Розглянуті вище циклічні алгоритми є прикладами алгоритмів зі заздалегідь відомою кількістю повторень циклу. На практиці часто зустрічаються алгоритми, для яких кількість повторень циклу можна визначити тільки після виконання алгоритму. Кількість циклів, що виконуються в цих алгоритмах, залежить від потрібної точності обчислення математичного виразу. Розглянемо приклади.

Приклад 2. Задана спадна геометрична прогресія з першим членом a і знаменником q. Визначити всі члени цієї прогресії, значення яких більші від e.

Члени спадної геометричної прогресії пов'язані таким рекурентним співвідношенням:

$$u_i = u_{i-1}q,$$

де $i = 2, 3,...; u_1 = a$.

Нехай, наприклад, $a = 4$ і $q = 1/2$. Тоді члени цієї прогресії набувають таких значень:

$$4, 2, 1, 1/2, 1/4, 1/8, 1/16, 1/32, 1/64, ...$$

Якщо задано $e = 1/15$, то обчислено буде 6 членів прогресії. Останнім у цьому випадку буде значення 1/8. Якщо ж e = 0,02, то обчислено буде 8 членів. При цьому останній член прогресії набуває значення 1/32.

Блок-схема алгоритму обчислення членів спадної геометричної прогресії зображена на рис. 14.6.

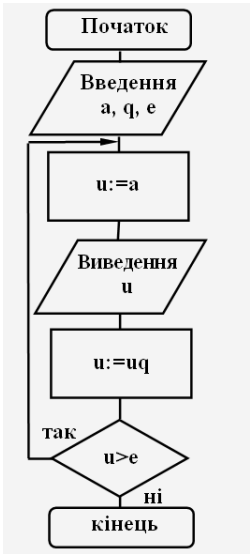


Рис. 14.6. Алгоритм обчислення членів спадної геометричної прогресії

Програму розв'язання задачі наведено нижче.

```
program n2_1_02;                    ]
var a,u,g,e:real;
begin
        writeln ('уведіть a,g,e'); read (a,g,e);
        u:=a;                              {перший член прогресії}
        while e<u do begin
        writeln ('u=',u); u:=u*g {виведення і обчислення члена прогресії}
      end;                                 {закриття операторних дужок}
end.
```

14.5. Обчислювальні алгоритми із вкладеними циклами

Розглянуті вище приклади алгоритмів із циклічною структурою відносять до класу простих алгоритмів. Простими алгоритмами з циклічною структурою називатимемо такі алгоритми, цикл яких не містить у собі інших циклів.

До складних алгоритмів з циклічною структурою відноситимемо алгоритми, в яких у циклі міститься ще хоча б один цикл. Цикл, що міститься в іншому циклі, називають внутрішнім, а цикл, який містить в собі ще один цикл, називають зовнішнім.

Приклад 3. У прямокутному трикутнику катети набувають таких значень:

a=10, 12, 14, 16, 18, 20, 22;
b = 7, 8, 9, 10, 11, 12, 13, 14.

Потрібно розробити блок-схему алгоритму і програму обчислення площі усіх трикутників.

Приймемо таку схему обчислення: при значенні катету *a=10* обчислимо площі трикутників для всіх значень катету *b;* потім при значенні катету *a=12* обчислимо площі трикутників для всіх значень катету *n* тощо. З урахуванням цієї схеми на рис 14.7 зображено блок-схему алгоритму.

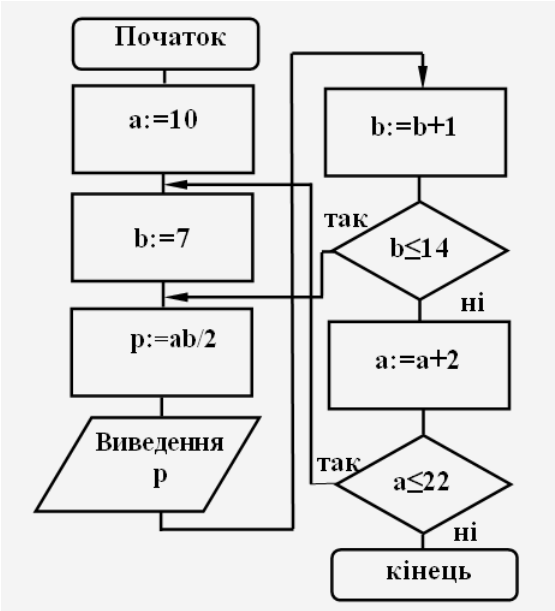


Рис. 14.7. Алгоритм із вкладеними циклами

Нижче наведено програму розв'язання задачі.

```
program n2_1_03;
var a,b:integer; s:real;
    begin
          a:=10;                    {початкове значення змінної a}
          while a<=22 do begin      {зовнішній цикл}
          for b:=7 to 14 do begin   {внутрішній цикл}
         s:=l/2*a*b;
     writeln ('s=',s)
     end;  {оператори тіла циклу}
    writeln;                        {порожній рядок}
a:=a+2 end;                         {перехід до чергового значення змінної a}
end.
```

14.6. Змішані алгоритми

Найчастіше ж у житті ми використовуємо змішані алгоритми. Так, дійсно, зранку ми контролюємо свій стан здоров'я, погоду, розклад (зустрічей, занять (коли збираємо речі), свій одяг, етапи роботи тощо). Коли ми снідаємо, ми повторюємо одну й ту саму послідовність дій (зачерпнути ложкою їжу, піднести до рота, забрати в рот, прожувати, проковтнути, зачерпнути ложкою їжу, піднести до рота тощо), коли ми миємо посуд, ми теж повторюємо одну й ту саму послідовність дій. Навіть коли ми йдемо, ми повторюємо визначені дії (подумайте, які).

Отже, найпоширенішим є змішаний тип алгоритму, у якому поєднуються елементи лінійних, розгалужених та циклічних алгоритмів. Під час складання алгоритмів іноді виникає ситуація, коли необхідно виконати повторювану послідовність дій, але не зовсім ідентичну. Наприклад, необхідно почистити картоплю, моркву, буряк та цибулю. Всі алгоритми чищення загалом схожі, але все ж таки мають певні відмінності. Щоб не переписувати алгоритми, що суттєво не розрізняються, використовують так звані допоміжні алгоритми, що викликаються і виконуються тільки тоді, коли в них є потреба. Перевага використання допоміжних алгоритмів полягає ще в тому, що склавши їх один раз, ми можемо їх потім використовувати при написанні навіть інших алгоритмів. В цьому випадку вони об'єднуються в так звані бібліотеки і ними можна користуватися так, як в реальному житті ми користуємося справжніми бібліотеками.

Питання для самоконтролю

1. *Що розуміють під алгоритмом з циклічною структурою?*
2. *Які типи операторів циклу Вам відомі?*
3. *Що розуміють під терміном „рекурентність”?*

15. ОБРОБКА ЧИСЛОВИХ МАСИВІВ

Як відомо, комп'ютер призначений головним чином для полегшення роботи людини з великими об'ємами інформації. Але як, використовуючи тільки змінні відомих вам типів, зберегти в пам'яті комп'ютера і опрацювати дані, що містять десяток, сотню, тисячу чисел або, наприклад, рядків? Адже такі задачі зустрічаються у будь-якій галузі знань. Зрозуміло, що можна завести стільки змінних, скільки даних, можна навіть занести в них значення, але тільки уявіть, якої величини буде текст такої програми, скільки часу знадобиться для його складання, як багато місця для можливих помилок? Природно, що про це замислювалися й автори мов програмування. Тому у всіх існуючих мовах є типи змінних, які відповідають за збереження більші масивів даних. В мові Pascal вони так і називаються: „масиви".

15.1. Загальні відомості про масиви

До цього часу ми працювали з простими типами даних – логічний (*boolean*), цілий (*integer, word, byte, longint*), дійсний (*real*), символьний (*char*). Будь-який алгоритм можна запрограмувати за допомогою цих чотирьох базових типів. Але для обробки інформації про розмаїтий реальний світ потрібні дані, які мають більш складну будову. Тобто, збереження й обробка складних видів інформації. Такі складні конструкції даних, які засновані на найпростіших скалярных типах, називаються структурами (записами).

***Структура* – певний складовий тип даних, який складається з базових скалярних типів даних, які називаються компонентами**. Компоненти можуть бути при цьому однорідними або різнорідними. У першому випадку тип даних визначається як тип-масив, а у другому – тип-запис.

Будь-який з структурованих типів, а їх у середовищу *Turbo Pascal* чотири типи: масиви, записи, множини і файли – характеризуються множинністю утворюючих цей тип елементів, тобто змінна або константа структурованого типу завжди має декілька компонентів. Кожний компонент у свою чергу , може належати певному структурованому типу, що дозволяє говорити про можливу вкладеність типів. В *Turbo Pascal* припускається довільна глибина вкладеності типів, однак сумарна довжина будь-якого з них у внутрішньому поданні не повинна перевищувати 65520 байт.

Подібно типу-діапазону або перераховуваного типу, структурований тип необхідно об'являти в розділі описів після кодового слова TYPE. З метою сумісництва зі стандартним Паскалем, в *Turbo Pascal* дозволяється перед описом структурованого типу ставити кодове слово PACKED, яке приписує компілятору за можливістю економити пам'ять, що відводиться під об'єкти структурованого типу; однак фактично компілятор ігнорує цю вказівку: „упакування" даних в Турбо-Паскалі здійснюється автоматично усюди, де це можливе.

Якщо структура не змінює своєї будови протягом усього виконання програми, у якій вона описана, то таку структуру називають статичною.

Необхідність в масивах виникає кожного разу, коли в пам'яті потрібно зберігати кінцеву кількість однотипних упорядкованих даних. Масив представляє собою структуру даних, яка складається із елементів одного типу, наприклад *integer*, *char* тощо.

Масив *(array)* **– це скінчений набір елементів одного (базового) типу, які зберігаються в послідовно розташованих комірках оперативної пам'яті і мають спільну назву**.

У математиці поняттю *масив* відповідають поняття вектора та матриці. Розрізняють одно (рядки, стрічки, вектори) – та багатовимірні масиви. Двовимірний масив даних – це таблиця, що складається з декількох *рядків*.

Елементи масиву називаються індексними змінними. За кількістю індексів, які треба вказати для доступу до окремого елемента масиву, розрізнюють одномірні, двомірні, ..., *n*-мірні масиви. Вимоги до індексів різні в різних алгоритмічних мовах. У *Pascal* індекс – це змінна порядкового типу. Характерною особливістю масивів є та обставина, що всі їх компоненти можна упорядкувати і забезпечити доступ до них простою вказівкою його порядкового номера (індексу).

А навіщо потрібні масиви? Масиви дозволяють зберігати і опрацьовувати табличні дані. За своєю структурою вони можуть бути як лінійними – записаними в один рядок або стовпчик, – так і двохмірними –у вигляді рядків і стовпців. Такі складні за своєю будовою типи даних або змінних називаються структурованими. До них відносяться не тільки масиви, а й рядкові змінні, записи, перерахування та деякі інші типи.

Для збереження кожного елементу масиву виділяється окрема комірка пам'яті, всі комірки мають однаковий розмір. Якщо до цього ми розглядали збереження змінних простих типів (наприклад, *real*, *integer* тощо) для яких були потрібні окремі комірки пам'яті, то тепер необхідною буде їх послідовність.

Як вже відзначалось раніше, місце для розміщення у пам'яті комп'ютера всіх значень статичних змінних, а масивів зокрема, виділяється на стадії компіляції програми. Це стає можливим завдяки роботі програми-компілятора, яка при аналізі тексту програми в розділі опису змінних *var* знаходить ідентифікатори змінних і їх типи, спеціально позначеними зарезервованими в *Turbo Pascal* словами. Потім компілятор виділяє комірки пам'яті, об'єм яких визначається типом змінних, і привласнює їм імена ідентифікаторів змінних. Для визначення масивів використовується зарезервоване слово *array*.

Опис масиву містить ім'я масиву (ідентифікатор), принцип індексації елементів (діапазон зміни індексів), тип елементів масиву:

VAR <ім'я_масиву>: ARRAY [діапазони зміни індексів] OF <тип_даних>,

де <ім'я_масиву> – правильний ідентифікатор; ARRAY, OF – кодові слова (англ. *масив*, *з*); [діапазони зміни індексів] – список з одного або декількох індексних типів, розділений комами, квадратні дужки, які обрамляють список, – вимоги синтаксису; <тип_даних> – будь-який тип *Turbo Pascal*.

Як видно з цього прикладу, після слова *array* в квадратних дужках вказується початковий і кінцевий номери (або індекси) елементів масиву, а після службового слова *of* – тип його елементів. Оскільки кожна комірка масиву має свій індекс, масиви називають змінними з індексами. Початковий індекс масиву не обов'язково повинен починатись з одиниці, важливо, щоб його значення було менше значення кінцевого індексу.

При використанні масивів необхідно мати на увазі наступні обмеження:

1. Індекси масиву можуть мати будь-який порядковий тип даних, за виключенням типу longint.

2. Максимально можлива кількість елементів масиву *n* визначається за формулою:

$$n = \frac{65 \cdot 520}{p},$$

де *p* – розмір елемента масиву в байтах.

Ця залежність зумовлена наступними обставинами. Справа в тому, що всі змінні і константи програми знаходяться в спеціальній пам'яті комп'ютера, яка називається сегментом даних. Його об'єм дорівнює 64 Кб (65536 байт). Максимальний же розмір змінної будь-якого типу в *Turbo Pascal* не може перевищувати 65520 байт. Тому тільки у випадку, коли сумарний об'єм пам'яті на усі константи й інші змінні буде менше 16 байт (65536-65520= 16 байт), Ви можете максимальну кількість *n*. Якщо ж цей об'єм перевищує 16 байт (що зазвичай і спостерігається у будь-якій програмі), то величину цього перевищення треба відняти у наведеній залежності з 6520, що приведе до зменшення *n*.

Згідно наведеної формули, максимально можливе значення *n* спостерігається при *p=1* байт (це такі прості типи, як *byte, shortint, char*) і становить 65520 байт. Це число значно менше розміру longint. Цією обставиною й зумовлене перше обмеження.

Масив називається одновимірним (лінійна таблиця), якщо для доступу до його елементів достатньо одного індексу.

1. Одномірні масиви (рядки, вектори). Загальний вигляд конструкції опису типу масиву такий:

TYPE

<назва типу>=array [<розмір>] of <назва базового типу>;

можна і так:

<список змінних>:array [<розмір>] of <назва базового типу>;

Розмір (кількість елементів) масиву найчастіше задають у вигляді діапазону або назви деякого перерахованого типу даних $x_1, x_2, \ldots, x_n$.

Доступ до елемента масиву здійснюється через назву масиву і номер елемента. Цей номер записується в квадратних дужках, наприклад, *x[2]* – другий елемент масиву *x*.

Описати масив можна у розділі опису типів *type*, у розділі констант *const*, або у розділі оголошення змінних *var*. Назви типів масивів і змінних-масивів обирає довільно користувач.

Наприклад, одномірний масив з 8 дійсних чисел у Паскалі можна оголосити таким чином:

- *VAR Name: ARRAY [1..8] OF real;*
- *Const N=8;*

VAR Name: ARRAY [1..N] OF real;

- *TYPE MASSIV = ARRAY [1..8] OF real;*

VAR Name: MASSIV;

2. Двохмірні масиви (матриці). Елементи двохмірного масиву (дані можуть бути подані у вигляді таблиці-матриці) визначаються ім'ям масиву та двома індексами: перший індекс означає номер рядка, а другий – номер стовпця, на перетині яких стоїть елемент.

Приклад двовимірного масиву:

$$x_{11},x_{12},\ldots,x_{1n}$$
$$x_{21},x_{22},\ldots,x_{2n}$$
$$\ldots$$
$$x_{m1},x_{m2},\ldots,x_{mn}$$

Місце елемента в двохмірному масиві визначається двома індексами, наприклад $x_{3,5}$. Перший індекс визначає номер рядка в масиві, другий – номер стовпця.

Загальний вигляд конструкції опису типу матриці такий:

TYPE

<назва типу>=array [<розмір>, <розмір>] of <назва базового типу>;

можна і так:

<список змінних>:array [<розмір>,<розмір>] of <назва базового типу>;

Двохмірний масив – це таблична організація даних.

При розподілі пам'яті в описовій частині програми під масив резервується стільки місця, скільки передбачає вказана кількість елементів масиву, враховуючи тип елементів. Межі зміни індексів повинні бути сталими величинами, а не змінними, інакше буде невідомо, скільки місця необхідно відвести в пам'яті для такого масиву.

У пам'яті комп'ютера елементи одномірних масивів розташовано послідовно. Двохмірні масиви розташовуються таким чином: спочатку елементи першого рядка, потім другого тощо.

Приклади оголошення масивів.

Приклад 15_1:

var x:array[1..10] of real;

У цьому прикладі оголошено одномірний масив дійсних чисел. Ім'я масиву *x*. Кількість чисел в масиві *10*. Отже: array вказує, що змінна *x* є масивом; після слова *of* зазначається тип елементів масиву; в квадратних дужках вказується діапазон зміни індексів.

Приклад 15_2: var phel:array[1..4,1..5] of char;

Оголошено двохмірний символьний масив *phel*, в якому чотири рядки і п'ять стовпців.

Індексами можуть бути константи, змінні, вирази, причому тип виразу повинен збігатися з типом індексу, заданим при оголошенні масиву.

Для звернення до будь-якого елемента масиву достатньо в програмі вказати ім'я масиву та значення індексів у квадратних дужках, наприклад *phel[3,4]*. Будь-якому елементу масиву можна присвоїти певне значення за допомогою оператора присвоювання, наприклад

phel[2,4]:='ш'.

Для масиву може бути введено ідентифікатор типу масиву. В цьому випадку загальна структура оголошення масиву має вигляд:

type <ім'я масиву> array [<діапазон індексів>] of <тип елемента>;

Приклад 15_3: type ziml=array [1..10] of real; var x:ziml;

Приклад 15_4: *const n=17;*

Type malk=array [10..n] of string [10]; var y:malk;

Елементи одного масиву можна передавати другому масиву одним оператором присвоювання. Наприклад: *var x, y:array [5.. 10] of integer; ... x:=y;*

Запис – це структура даних, яка складається з фіксованої кількості компонентів, які називаються полями запису. На відміну від масиву, компоненти (поля) можуть бути різного типу. Для забезпечення можливості посилання на той або інший компонент запису, поля іменуються. Структура об'яви типу запису така:

<ім'я_типу>: RECORD <список полів> END,

де <ім'я_типу> – правильний ідентифікатор;

RECORD, END – ключові слова (англ. *запис, кінець*);

<список полів> – список полів, який представляє собою послідовність розділів запису, між якими ставиться крапка з комою.

Кожний розділ запису складається з одного або декількох ідентифікаторів, які відділяються один від одного комами. За ідентифікатором ставиться двокрапка і опис типу.

Множини – це набори однотипних об'єктів, певним чином пов'язаних один з одним. Характер зв'язків між об'єктами має на увазі лише програмістом і жодним чином не контролюється середовищем програмування Турбо-Паскаль. Кількість елементів, що входять у множину, може змінюватися в межах від 0 до 256 (множина, яка не містить елементів, називається пустою). Саме змінністю кількості своїх елементів множини відрізняються від масивів і записів.

Дві множини вважаються еквівалентними тоді і тільки тоді, коли їх елементи однакові, причому порядок слідкування елементів в множині може бути будь-яким. Якщо всі елементи однієї множини входять також і в другу, то кажуть про включення першої множини у другу. Пуста множина включена у будь-яку іншу.

15.2. Робота з масивами

Роботу з масивами можна умовно поділити на три частини:

1) формування масиву;
2) опрацювання масиву;
3) виведення масиву.

Формування масиву. Формування значень елементів масиву можна виконувати у такий спосіб:

• введення значень елементів масиву з клавіатури або з файлу;

• формування значень випадковим чином, з використанням функції-генератора випадкових чисел *Random*;

• обчислення значень елементів масиву за формулою.

Опрацювання масиву. Класичними задачами для роботи з масивами можна назвати:

– пошук заданого елемента в масиві;
– знаходження суми (добутку) елементів масиву;
– пошук максимального (мінімального) елемента в масиві;
– упорядкування масиву за ознакою (наприклад за зростанням або спаданням та ін.).

Одним із найскладніших завдань є впорядкування елементів масиву. Для розв'язання цієї задачі існує декілька алгоритмів.

Сортування вибором:

1. Встановити номер найбільшого елемента масиву.
2. Поміняти місцями найбільший і останній елементи.
3. Повторити 1 і 2 над остачею масиву (без останнього елемента).

Застосовувати цей метод до елементів масиву, що залишилися, доки залишок не скоротиться до одного елемента.

Аналогічно сортування вибором можна застосувати до найменшого елемента, міняючи його з першим. У результаті все одно отримаємо зростаючу (незменшувану) послідовність елементів масиву.

Обмінне сортування («бульбашка»):

1. Порівняти два поруч розташованих елементи.
2. Якщо пара порушує потрібний порядок слідування, елементи міняють місцями.

Порівняння відбувається до кінця масиву. Обмін здійснюється доти, поки прохід по масиву не викличе жодного обміну.

Існують й інші методи сортування масивів: метод уставки, швидке сортування тощо.

Виведення масиву. Виведення елементів одновимірного масиву на екран можна виконувати в рядок:

For i:=1 to n do Write (mas[i]);

або у стовпчик:

For i:=1 to n do Writeln (mas[i]).

Для виведення елементів двовимірного масиву у вигляді двовимірної таблиці (матриці) можна використовувати таку конструкцію:

```
For i:=1 to n do
Begin
For j:=1 to m do
Write (mas[i,j]);
Writeln;
End.
```

15.3. Обчислювальні алгоритми циклічної структури з масивами

В обчислювальних задачах елементами масиву є числа (цілі, дійсні тощо). Тому такі масиви називають числовими. Організація чисел у вигляді масиву дає змогу проводити обробку цих чисел циклічно. Розглянемо приклади побудови алгоритмів обробки масивів.

Приклад 15_5. Задано одновимірний масив дійсних чисел $a_1\ a_2,\ldots,\ a_n$. Необхідно скласти блок-схему алгоритму і програму обчислення частинних сум із виведенням кожної суми на екран монітора.

Частинні суми набувають таких значень:

$$s_1=a_1;\ s_2=a_1+a_2;\ s_3=a_1+a_2+\ a;\ \ldots\ s_n=a_1+a_2+\ a_3+\ldots+a_n.$$

Обчислення частинних сум можна здійснювати рекурентно за такою формулою:

$$s_i = s_{i-1} + a_{(i)} \quad при\ s_0 = 0;\ i = 1, 2, 3, \ldots, n.$$

Підкреслимо, що індекс при змінній *s* визначає номер кроку обчислення частинних сум. При зображенні блок-схем алгоритмів цей індекс опускається. При елементі масиву *a* індекс визначає номер цього елемента в масиві. Цей індекс в блок-схемі зберігається, але дужки опускаються. Блок-схема алгоритму зображена на рис. 15.1.

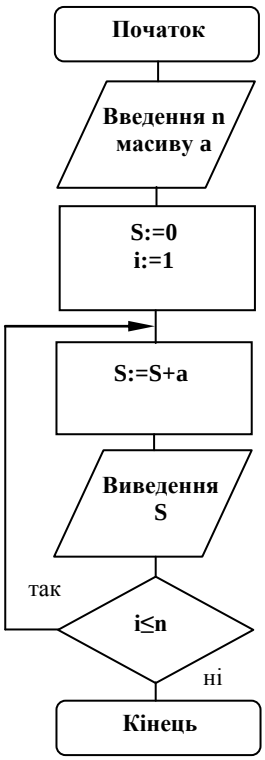


Рис. 15.1. Блок-схема алгоритму циклічної структури з масивами

Нижче також наведено програму розв'язування задачі.

```
program 15_06;
        const n=5;                              {іменована константа}
         var s:real; i:integer; a:array [l..n] of real;
             begin
             for i:=l to n do begin             {початок зовнішнього циклу}
             writeln ('уведіть a(',i,')=');     {оператор тіла циклу}
             readln (a[i]) end;                 {оператор тіла циклу}
         s:=0;                                  {початкове значення суми}
         for i:=l to n do begin
             s:=s+a[i]; writeln ('s=',s) end;   {обчислення і виведення частинних сум}
end.
```

Приклад 15_07. Задано двовимірний масив дійсних чисел:

$$a_{11},a_{12},\ldots,a_{1n}$$
$$a_{21},a_{22},\ldots,a_{2n}$$
$$\ldots$$
$$a_{m1},a_{m2},\ldots,a_{mn}$$

Треба скласти блок-схему алгоритму обчислення суми елементів масиву.

Обчислення суми двовимірного масиву можна здійснити різними методами. Наприклад, можна підрахувати суму чисел кожного окремого рядка, а потім скласти значення отриманих сум рядків. Можна підрахувати суму чисел кожного окремого стовпчика, а потім скласти значення отриманих сум стовпчиків.

Однак краще використати метод „накопичення". Сутність його полягає у тому, що вихідне значення суми приймається за нуль. Потім до цієї суми додається перший елемент першого рядка, до отриманої суми додається другий елемент цього ж рядка і т. д. До суми чисел першого рядка додається перший елемент другого рядка і так далі до завершення всього обчислювального процесу. Наприклад, для двовимірного масиву

$$a_{11},a_{12},a_{13}$$
$$a_{21},a_{22},a_{23}$$

обчислення суми здійснюватиметься наступним чином:

1. $s_0 = 0;$ 2. $s_1 = s_0 + a_{11};$
3. $s_2 = s_1 + a_{12};$ 4. $s_3 = s_2 + a_{13};$
5. $s_4 = s_3 + a_{21};$ 6. $s_5 = s_4 + a_{22};$
7. $s_6 = s_5 + a_{23}.$ 8. *Кінець.*

Блок-схема алгоритму обчислення суми елементів масиву на основі розглянутого підходу зображено на рис. 15.2.

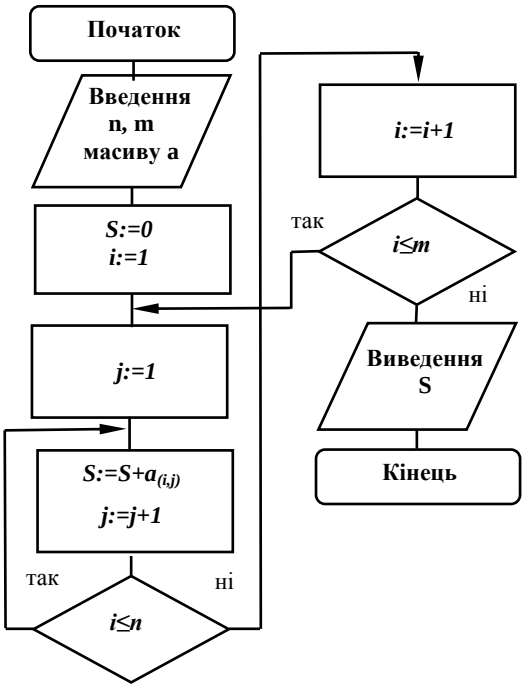


Рис. 15.2. Блок-схема алгоритму циклічної структури з масивами

Крім вже наведених, в алгоритмі також використовуються наступні змінні:
i, j – лічильники циклів,
n – кількість рядків у масиві,
m – кількість стовпців у масиві.

Нижче наведено програму реалізації алгоритму.

```
program 15_08;
const n=3; m=4;                          {іменовані константи}
var a:array[l..n,l..m] of real;          {оголошення масиву}
i,j:integer; s:real;
begin
      for i:=l to n do begin
         for j:=l to m do begin
            writeln ('уведіть a(',i, Y,j,')=');   {повідомлення про введення}
            readln (a[i,j]) end;                  {введення елемента масиву}
       end;
      s:=0;                              {початкове значення суми}
      for i:=l to n do begin
      for j:=l to m do s:=s+a[i,j] end;  {підсумування елементів масиву}
  writeln ('s=',s)                       {виведення суми елементів масиву}
end.
```

15.4. Робота з текстовою інформацією

У Паскалі при роботі з текстовою інформацією існує можливість обробки одиночних символів типу *Char* та послідовності символів – рядків типу *String*.

Символьний тип. Тип *Char* – це один з базових типів мови, призначений для збереження та опрацювання одного символу. Множиною його значень є окремі символи (букви, цифри, знаки), впорядковані у відповідності із розширеним набором символів ASCII-коду. Змінна цього типу займає 1 байт пам'яті. Завдяки тому що в пам'яті машини символи зберігаються у вигляді кодів (більшим вважається той символ, чий код більший), їх можна порівнювати. Для символів припустимі всі операції порівняння: <, <=, =, >, >=, <>.

Опис даних символьного типу:

Const Name1 = 'v'; – опис символьної константи,

Var Name2: CHAR; – опис символьної змінної.

Як правило, значення для символьних змінних та констант задаються в лапках, наприклад, 'f', '1', '+'. Також можна задати значення, вказавши безпосередньо числове значення ASCII-коду, поставивши перед цим числовим кодом знак #, наприклад, #35, #102.

У Паскалі для роботи з символьною інформацією реалізовані функції перетворення: *CHR(N)* – символ з кодом *N*, *ORD(S)* – код символі *S*. Також застосовуються функції, що визначають *SUCC(S)* – наступний символ, *PRED(S)* – попередній символ. Для цих функцій виконуються такі залежності:

SUCC(S) = CHR(ORD(S) +1);

PRED (S) = CHR(ORD(S) –1).

Для латинських літер 'a'..'z' виконується функція UPCASE(S), яка переводить ці літери у верхній регістр 'A'..'Z'.

Рядковий тип. Тип STRING (рядок) – тип даних, призначений для збереження та опрацювання послідовності символів і багато в чому схожий на одномірний масив символів ARRAY [0..N] OF CHAR. Однак на відміну від масиву кількість символів в рядку-змінній може змінюватися від 0 до N, де N – максимальна кількість символів у рядку. Значення N визначається об'явою STRING [N] і може бути будь-якою константою порядкового типу, але не перевищувати 255. Турбо-Паскаль дозволяє не вказувати N, і в цьому випадку довжина рядка приймається максимально можливою, тобто N=255.

Рядок в Турбо-Паскалі трактується як ланцюжок символів. До будь-якого символу в рядку можна звернутись аналогічно зверненню до одномірного масиву ARRAY [0..N] OF CHAR.

Опис даних рядкового типу:

Const Name1='computer'; – опис рядкової константи,

Var Name2: STRING; – опис рядкової змінної,

Name3: STRING [20]; – опис рядкової змінної заданої довжини,

За умовчанням довжина рядкової змінної дорівнює 255 символам, але можна обмежити довжину рядка за допомогою явної вказівки довжини рядка.

У Паскалі реалізовано опрацювання рядків двома шляхами: опрацювання рядка як єдиного цілого та як об'єкта, який створюється з окремих символів.

Перший шлях надає можливість:

• присвоєння рядковій змінній за одну операцію цілого рядка символів, наприклад, *Name2:='computer'; Name3:='science'*;

• об'єднання рядків у довільному порядку за допомогою операції «+» (операції скріплення, об'єднання), наприклад,

Name3:= 'computer'+'science';

Name3:= Name2 + Name3;

• порівняння рядків за допомогою операцій порівняння: <, <=, =, >, >=, <>, наприклад,

If Name3 <> Name2 then write ('no');

Другий шлях надає можливість до кожного окремого символу рядка звертатися за його номером позиції як до елемента масиву за індексом, наприклад,

Name3:= Name2 [6] + Name2 [2] + Name2 [4];

Елемент з нульовим індексом містить символ, код якого вказує на дійсну довжину даного рядка.

У Паскалі реалізовані процедури і функції для опрацювання рядків. Поточну довжину рядка S можна дізнатися за допомогою функції *LENGTH (S)*.

Група функцій та процедур, спрямована на опрацювання фрагментів рядка:

• функція *COPY(S, N, M)* – копіювання фрагменту рядка S довжиною M, що починається з позиції *N*;

• функція *POS (S1, S)* – пошук фрагменту *S1* в рядку *S* (отримуємо позицію, з якої починається фрагмент S1 в рядку S);

• функція *CONCAT (S1, S2,…)* – об'єднання рядків *S1*, *S2*,…;

• процедура *INSERT (S1, S2, M)* – вставка фрагменту *S1* у рядок *S2* із позиції M;

• процедура *DELETE (S1, N, M)* – вилучення частини рядка S1 довжиною M, починаючи з позиції N;

• процедура *VAL (S, N, Code)* – перетворення рядка цифрових символів *S* у число *N* (параметр *Code=0*, якщо рядок *S* утворений не із цифрових символів);

• процедура *STR (N, S)* – перетворення числа N у рядок цифрових символів *S*.

Для сортування символьних рядків (наприклад, за алфавітом) доцільно створити масив символьних рядків (масив типу *String*), що, з урахуванням можливості використання операцій порівняння для рядків, дозволить у простий спосіб застосовувати основні алгоритми сортування.

Питання для самоконтролю

1. *Що розуміють під терміном „масив"?*
2. *Які типи масивів Вам відомі?*
3. *Як оголошуються масиви в мові програмування Паскаль?*
4. *Як здійснюється звернення до елементів масиву?*

16. ПОШУК, ПЕРЕМІЩЕННЯ ТА СОРТУВАННЯ ДАНИХ

Пошук і сортування є одним з найпоширеніших процесів сучасної обробки даних. Сортування можна визначити як розміщення об'єктів у певному порядку їх значень: за зростанням або убуванням (коли сортованими елементами є числа), або у алфавітному порядку (при сортуванні текстової інформації).

16.1. Лінійний пошук даних

Будь-який метод пошуку даних заснований на методі перебору. Перебір може бути прямий (повний) і скорочений. Надалі будемо розглядати два найпростіших методи пошуку: лінійний і двійковий пошук.

З усього розмаїття методів впорядкування даних виділимо найпростіші, які широко застосовуються в процесі розв'язування практичних задач, а саме: метод впорядкування вибором і метод попарної перестановки.

У процесі розв'язування практичних задач особливо часто пошук і впорядкування здійснюються в трьох класах даних: лінійних списках, масивах і таблицях. Лінійні списки мають форму одновимірного масиву і включають послідовність із довільного числа елементів. Принципова відмінність лінійного списку від одновимірного масиву полягає у різному характері виконуваних операцій над їх елементами. Елементи лінійного списку можуть вилучатися, замінюватися іншими елементами. Лінійні списки і одновимірні масиви мають однакові структурні властивості, тому іноді їх об'єднують терміном „послідовність". В табличній організації даних часто застосовують такі терміни: ключ *(k)* – це одне з полів (стовпців) таблиці, яке має найбільш характерну ознаку порівняно з іншими полями; інформаційні поля *(d)* – це поля таблиці, які не входять до складу ключового; запис *(s)* – один із рядків таблиці.

Лінійний пошук базується на прямому переборі елементів у послідовності. При цьому заданий ключ запиту *c* послідовно порівнюється зі значенням ключів послідовності $k_L, k_z, \ldots, k_n$, наприклад, на початку зі значенням k_t, потім зі значенням k_2, k_3 до k_n. У процесі порівняння значення ключа *c* зі значенням ключів послідовності $k_t, k_2, \ldots, k_n$ можуть вирішуватися різні завдання, наприклад, чи є в заданій послідовності хоча б одне значення, яке дорівнює *c*. Або підрахувати, скільки значень ключів заданої послідовності збігаються зі значенням *c*.

Приклад 16_1. Послідовність ключів має такі значення: *ch-2501, cl-3081, ch-5000, pn-4201.* Треба визначити, чи є в заданій послідовності ключ запиту *ch-5000.*

Розв'язання завдання може бути здійснено наступним чином. Спочатку ключ запиту *ch-5000* порівнюється з ключем *ch-2501,* потім з ключем *ci-3081*, згодом – з *ch-5000* і наприкінці – з *pn-4201,* на цьому процес пошуку закінчується.

Блок-схему алгоритму лінійного пошуку зображено на рис. 16.1. У цьому алгоритмі змінна *i* виконує роль лічильника кількості переглянутих ключів послідовності; *n* – кількість елементів в послідовності.

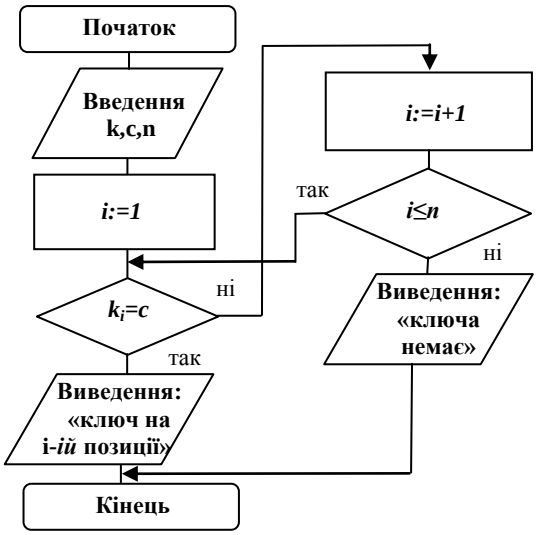


Рис. 16.1. Блок-схема алгоритму лінійного пошуку

```
program 16_01;
const n=5;                                   {іменована константа}
var k:array[l..n] of string[7];              {масив рядкового типу}
c:string[7]; p:string; irinteger; label ml;
begin
writeln ('уведіть c='); readln (c);          {введення ключа пошуку}
for i:=l to n do begin                       {початок циклу}
writeln ('уведіть k(',i,')=');               {повідомлення про введення}
readln (k[i]) end;                           {введення послідовності}
for i:=l to n do if k[i]=c then begin        {пошук ключа}
wrteln ('ключ на позиції =',i); goto ml end; {виведення повідомлення}
writeln ('ключа немає');                     {виведення повідомлення}
ml: writeln                                  {порожній рядок} end.
```

Відмітимо, що метод лінійного пошуку найпростіший, але за часом пошуку найдовший, оскільки він не враховує закономірностей у розміщенні ключів *k*. Наприклад, значення ключів *k* можуть бути упорядковані за якоюсь ознакою. У цьому випадку організація пошуку може здійснюватися більш раціонально.

16.2. Двійковий пошук даних

Цей метод застосовується лише до впорядкованої послідовності. Суть методу полягає в наступному. У послідовності $k_1, k_2, \ldots, k_n$ вибирається середній елемент $k_{сер}$. Порівнюється ключ запиту *c* зі значенням $k_{сер}$. Якщо ключ запиту дорівнює $k_{сер}$, то робиться висновок, що його знайдено, і з таблиці вибирається відповідна інформація. Якщо $c < k_{сер}$, то вибирається ліва частина послідовності; в протилежному випадку – права частина послідовності. У вибраній частині послідовності знову вибирається середнє значення ключа. Далі процес повторюється аналогічно до описаного вище.

Приклад 16_2. Задана послідовність ключів 1,3,5,7,9,11,13,15 і ключ запиту *c* =11. Розглянемо алгоритм пошуку.

У цій послідовності міститься вісім ключів. Номер середнього ключа можна визначити із співвідношення:

$$\left[\frac{1+8}{2}\right]_{ö.ç.} = 4,$$

де *ц.з.* – ціле значення.

Оскільки k_4 =7 менше від ключа запиту 11, то надалі пошук будемо проводити в послідовності 9,11,13,15. У цій послідовності визначається номер середнього ключа

$$\left[\frac{5+8}{2}\right]_{ö.ç.} = 6.$$

Ключ з номером 6 (k_6) дорівнює 11 і збігається з ключем запиту. Зазначимо, що значення правої і лівої меж аналізованих послідовностей змінюються. Ці значення будемо зберігати в змінних *p* (права межа) і *l* – (ліва межа). У розглянутому прикладі спочатку вони набувають значення *l =1, p=8,* а потім: *l =5, p=8.* У загальному випадку номер середнього ключа в будь-якій послідовності можна визначити за формулою:

$$i = \left[\frac{l+p}{2}\right]_{ö.ç.}.$$

Блок-схема алгоритму двійкового пошуку зображено на рис. 16.2.

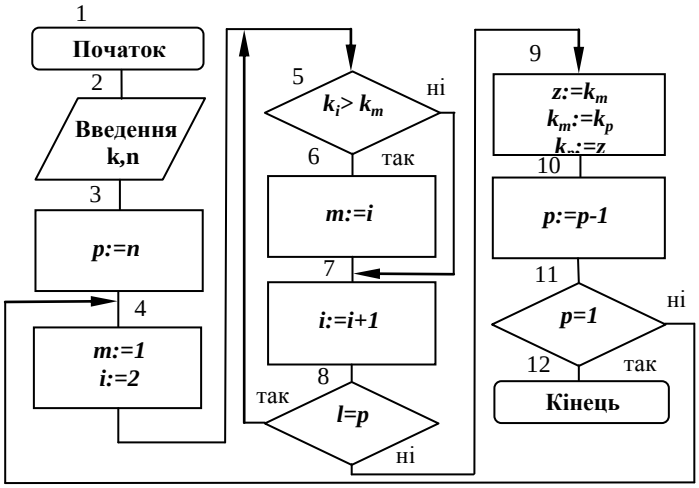


Рис 16.2. Блок-схема алгоритму двійкового пошуку

У блоці 3 змінній *l* надається початковий номер лівої межі послідовності (початковий номер завжди дорівнює одиниці), а змінній *p* надається початковий номер правої межі (цей номер завжди дорівнює *n).*

У блоці 4 обчислюється середній номер розгляданої послідовності, значення якого присвоюється змінній i.

У блоці 5 здійснюється порівняння ключа запиту *c* зі значенням ключа, розміщеного в середній позиції послідовності, що розглядається. Якщо вони збігаються, то це означає, що ключ запиту знайдено, і алгоритм припиняє роботу (блок 12).

У блоці 6 перевіряється, чи більший ключ запиту *(c)* за значення ключа, розміщеного в середній позиції розглядуваної послідовності *(k_i).* Якщо він більший, то це означає, що ключ запиту розміщений справа від k_i.

У цьому випадку значення правої межі не змінюється, а відбувається зміна значення лівої межі (блок 7). Якщо ж $c < k_i$, то пошук ключа запиту треба вести в послідовності зліва від k_i. У цьому випадку значення лівої межі не змінюється, а відбувається зміна значення правої межі (блок 8).

Алгоритм припиняє роботу у двох випадках: 1) коли виконується умова $c= k_i$; 2) коли збігаються значення лівої і правої меж (блок 9). Останній випадок відповідає тому, що всі можливі перевірки виконані, а ключа запиту в заданій послідовності немає.

Зазначимо, що метод двійкового пошуку значно "швидший" лінійного методу. Однак метод двійкового пошуку вимагає, щоб послідовність ключів була попередньо розміщена в порядку зростання або спадання її членів.

Програми реалізації алгоритму двійкового пошуку зображено нижче.

```
program 16_02;
const n=5;                              {іменована константа}
var k:array[l..n] of real; c: real;     {оголошення масиву й змінних}
i,p,l:integer; w:string[17]; label ml,m2;
begin
      writeln ('дведіть c='); readln (c);   {введення ключа пошуку}
      for i:=1 to n do begin
        writeln ('ведіть к(',i,')=');       {повідомлення про введення послідовності}
        readln (k[i]) end;                  {введення послідовності}
      l:=1; p:=n;                           {початкові значення меж послідовності}
m1: i:=(l+p) div 2;                         {обчислення середнього номера}
    if c=k[i] then begin                    {пошук ключа}
      w:='ключ на позиції ='; goto m2       {перехід на кінець програми}
      end;
    if p<=1 then begin                      {перевірка значення правої межі}
      w:='ключа немає'; goto m.2            {ключ не знайдено, на кінець програми}
    end;
    if c>k[i] then l:=i+1 else p:=i-1;      {зміна значень меж послідовності}
   goto m1;
m2: writeln(w,' ',i);                       {виведення повідомлення і номера позиції}
end.
```

16.3. Впорядкування даних методом вибору

Сутність методу розглянемо на прикладі послідовності k_1 ,k_2 ,...,k_n , яку необхідно розмістити в зростаючому порядку.

У послідовності k_1 k_2,...,k_n знаходиться максимальний елемент. Максимальний елемент і крайній правий елемент міняються місцями. Після цього правий крайній елемент з подальшого розгляду виключається.

На наступному кроці аналізується послідовність k_t ,k_2 ,..., k_{n-1}, у якій також знаходиться максимальний елемент. Цей елемент і елемент k_{n-1} міняються місцями. Далі аналогічно до попереднього виконуються дії в послідовності k_1 k_2.....k_{n-2}, потім в послідовності k_1 ,k_2 ,...,k_{n-3} тощо.

Із суті методу впорядкування вибором випливає, що значення правої межі послідовності при кожному наступному перегляді зменшується на одиницю. Значення правої межі послідовності будемо зберігати в змінній *p.*

Особливо треба підкреслити, що перестановка елементів виконується тоді, коли поточна послідовність розглянута до кінця. Для того щоб поміняти місцями значення двох змінних, необхідна третя змінна для тимчасового зберігання значення однієї змінної. В алгоритмі для цієї мети використовується змінна *z.*

Блок-схему алгоритму впорядкування методом вибору зображено на рис. 16.3.

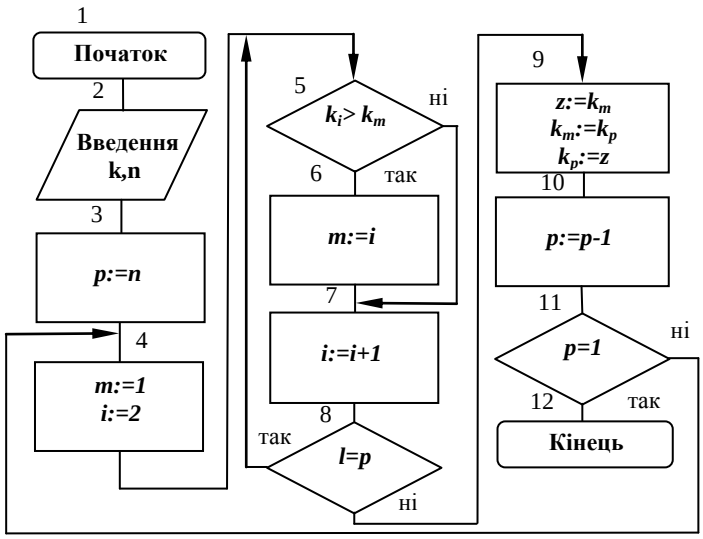


Рис. 16.3. Блок-схему алгоритму впорядкування методом вибору

У цьому алгоритмі *m* – поточний номер максимального елемента, *i* – номер розглядуваного елемента.

Виконання алгоритму починається із присвоєння змінної *p* номера крайнього правого елемента послідовності (блок 3). Перший елемент у будь-якій послідовності спочатку приймається за максимальний, і першого разу він завжди порівнюється з другим елементом. Тому в блоці 4 змінна *m* набуває значення, що дорівнює одиниці, а змінна *i* – значення, що дорівнює двом.

У блоці 5 перевіряється: чи більше значення поточного елемента послідовності (k_i) за максимальний елемент із числа розгляданих (k_m). Якщо $k_i > k_m$, то поточний номер максимального елемента набуває нового значення (блок 6). У блоці 7 здійснюється підготовка до вибору чергового елемента послідовності. Якщо номер розглянутого елемента менший за праву межу (блок 8), то продовжується пошук максимального елемента. Якщо ж *i>p,* то це означає, що вся послідовність розглянута, і в блоці 9 міняються місцями максимальний елемент із першим крайнім. У блоці 10 зменшується значення правої межі на одиницю. Ознакою закінчення виконання алгоритму є рівність правої межі одиниці (блок 11).

Розглянемо виконання алгоритму на прикладі послідовності 141, 2, 25, 30. Після першого перегляду послідовності вона набуде такого вигляду: 30, 2, 25, 141. Після другого – 25, 2, 30, 141, після останнього – 2, 25, 30, 141.

Перевага методу впорядкування вибором полягає в його простоті. Проте цей метод найповільніший, оскільки він не враховує наявності в заданій послідовності можливої впорядкованості деяких елементів. Навіть якщо елементи вихідної послідовності впорядковані, то кількість виконуваних операцій в алгоритмі не скоротиться.

Програму реалізації алгоритму впорядкування даних методом вибору зображено нижче.

```
program 16_03;
const n=6;                          {іменована константа}
var k:array[l..n] of integer;       {масив цілих чисел}
i,p,m,z:integer;                    {змінні цілого типу}
begin
  for i:=l to n do begin
    writeln ('уведіть k(',i,')=');  {повідомлення про введення елементів}
    readln (k[i]) end;              {введення елементів послідовності}
  p:=n;                             {початкове значення правої межі}
  while poї do begin
    m:=1;                           {початковий номер максимального елемента}
    for i:=l to p do
    if k[i]>k[m] then m:=i;         {порівняння значень елементів}
   z:=k[m]; k[m]:=k[p]; k[p]:=z;    {переміщення елементів послідовності}
   p:=p-1 end;                      {зменшення значення правої межі}
for i:=l to n do writeln ('k(',i,')=',k[i])   {виведення впорядкованої послідовності}
end.
```

16.4. Впорядкування даних методом попарної перестановки

Нехай задана послідовність $k_1, k_2, \ldots, k_n$, яку треба розмістити в порядку зростання її елементів. Суть методу попарної перестановки полягає у наступному. Послідовність $k_1, k_2, \ldots, k_n$ розглядається зліва направо. При цьому порівнюється елементи k_1 і k_2, k_2 і k_3, k_3 і k_4 і т. д. з елементами k_{n-1} і k_n. Кожен раз, коли попередній елемент більший за наступний (тобто коли $k_{i-1} > k_i$), вони міняються місцями. Після першого перегляду всієї послідовності на крайній правій позиції виявиться максимальний елемент. Після цього переглядається послідовність $k_1, k_2, \ldots, k_{n-1}$ і над її елементами виконуються аналогічні дії, внаслідок чого на позиції $n-1$ виявиться другий за величиною елемент. Далі виконуються аналогічні дії в послідовності $k_1, k_2, \ldots, k_{n-2}$, потім в послідовності $k_1, k_2, \ldots, k_{n-3}$ і т. д.

Послідовність вважається впорядкованою, якщо після будь-якого перегляду не робиться жодної перестановки. Крім того, послідовність буде впорядкованою, якщо у розгляданої послідовності залишиться лише один елемент.

На рис. 16.4 зображено блок-схему алгоритму впорядкування методом попарної перестановки.

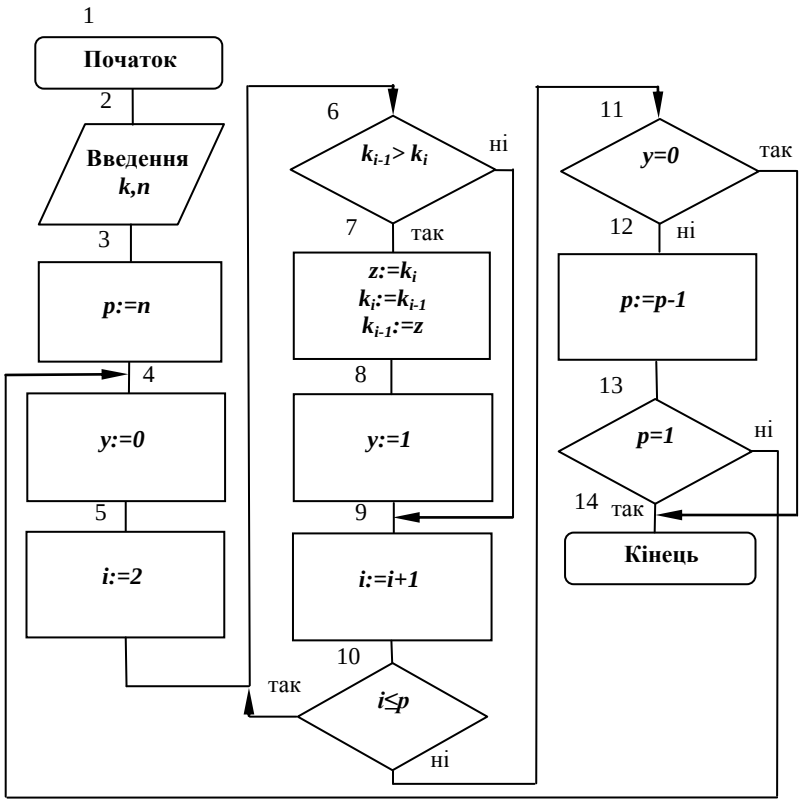


Рис. 16.4. Блок-схема алгоритму впорядкування методом попарної перестановки

В алгоритмі використані такі позначення: p – значення правої межі розгляданої частини послідовності; y – ознака наявності перестановки (якщо y =0, то це означає, що перестановки у поточному перегляді не було, якщо $y=1$, то була принаймні одна перестановка); i – номер елемента послідовності, який аналізується; z – змінна для тимчасового збереження елемента послідовності у процесі перестановки елементів.

У блоках 3,4,5 здійснюється підготовка до першого порівняння елементів. Порівняння елементів здійснюється в блоці 7, в блоці 8 формується ознака наявності перестановки. В блоці 9 здійснюється підготовка до порівняння чергового елемента послідовності.

Якщо послідовність переглянуто повністю (блок 10, випадок "ні"), то перевіряється ознака перестановки (блок 11). Якщо перестановки були (блок 11, випадок "ні"), то значення правої межі зменшується на одиницю (блок 12). У блоці 13 аналізується, чи не залишився в послідовності один елемент.

Проаналізуємо виконання алгоритму на прикладі розміщення в порядку збільшення значень елементів послідовності 8, 12, 3, 4, 2:

– розміщення елементів після першого перегляду: 8, 3, 4, 2, 12
– розміщення елементів після першого перегляду: 3,4,2,8,12
– розміщення елементів після першого перегляду: 3, 2, 4, 8, 12
– розміщення елементів після першого перегляду: 2, 3, 4, 8, 12

Метод попарної перестановки "швидший" за метод впорядкування вибором, оскільки він враховує наявність впорядкування елементів вихідної послідовності.

Програму реалізації алгоритму упорядкування числового масиву методом попарної перестановки наведено нижче.

```
program 16_04;
const n=6;                                  {іменована константа}
var k:array[l..n] of integer;               {оголошення масиву}
z,i,y,p:integer; label ml,m2;               {змінні та мітки}
begin
     for i:=l to n do begin
      writeln ('уведіть k(',i,')=');        {повідомлення про введення}
      readln (k[i]) end;                    {введення елементів масиву}
  p:=n;                                     {початкове значення правої межі}
  m1: for i:=2 to p do begin                {початок циклу}
  y:=0; if k[i-1]>k[i] then begin           {чи більший попередній елемент}
  z:=k[i]; k[i]:=k[i-1]; k[i-1]:=z;         {переміщення елементів послідовності}
  y:=1; end; end;                           {ознака, що переміщення елементів були}
if y=0 then goto m2 else p:=p-1;            {чи впорядкована вся послідовність}
if p<>1 then goto m1;                       {чи впорядкована вся послідовність}
m2: for i:=1 to n do writeln ('k(',i,')= ',k[i]); {виведення впорядкованої послідовності}
end.
```

Близьким до методу попарної перестановки є метод сортування „бульбашки", сутність його полягає у наступному. Перший елемент послідовності поступово порівнюється з елементами, що знаходяться справа від нього. Якщо елемент справа менший за перший, то вони міняються місцями. Потім порівнюється перший елемент з тими, що залишилися правіше розглянутого. Якщо знайдеться менший елемент, то вони також міняються місцями. Такий процес продовжується до досягнення правого крайнього елемента. Внаслідок цього на першому місці буде розташовано найменший елемент. Потім аналогічне порівняння здійснюється з 2-го елемента, далі – 3-го тощо.

Питання для самоконтролю

1. В чому полягає суть лінійного пошуку даних?
2. В чому полягає суть двійкового пошуку даних?
3. В чому полягає суть методу вибору?
4. В чому полягає суть методу попарної перестановки?

Література:

1. Інформатика. Комп'ютерна техніка. Комп'ютерні технології. Підручник за ред. ден проф. Пушкаря О.І. – Київ: Вид. центр. Академія, 2002.-704 с.
2. Брукшир Д. Г. Введение в компьютерные науки. Изд. Шестое. Изд. Дом «Вильямс» Киев 2001.- 688с.
3. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов: Пер. с англ.-М.: Мир,1979.
4. Ван Тассел Д. Стиль, разработка, эффективность, отладка и испытание программ.- М.: Мир, 1981.
5. Вирт Н. Алгоритмы и структуры данных.- М.Мир,1989.
6. Гейн А.Г. и др. Основы информатики и вычислительной техники.- М.Просвещение , 1992.
7. Гильберт Д., Аккерман В. Основы теоретической логики. – 2-е изд. – М.: URRS, 2010
8. Гильберт Д., Бернайс П. Основания математики. – М.: «Наука», 1979, 1982.
9. Гудман С., Хидетниели С. Введение в разработку и аналих алгоритмов. - М.: Мир, 1981.
10. Дайтибегов Д.М., Черноусов Е.А. Основы алгоритмизации и алгоритмические языки.- М.: Финансы и статистика, 1992.
11. Коллинз Г. Блэй Дж. Структурные методы разработки систем: от стратегического планирования до тестирования.Пер. с англ./ Под ред. В.М. Савинкова.- М.:Финансы и статистика, 1986.
12. Кузнецов А.А. и др. Основы информатики.- М.:Дрофа, 1998.
13. Кушниренко А.Г. и др. Информатика.- М.:Дрофа, 1998.
14. Ландо С.К. Алгоритмика: Методическое пособие. - М.: Дро фа,1997.
15. Марков А.А., Нагорный Н.М. Теория алгорифмов.-М.:Наука. Главная редакция физико-математической литературы, 1984.
16. Матросов В.Л. Теория алгоритмов. - М.: Прометей, 1989.
16. Могилев и др. Информатика: Учебное пособие для вузов / А.В.Могилев,Н.И.Пак, Е.К.Хеннер; Под ред. Е.К. Хеннера. - М.: Изд. центр "Академия", 2000.
17. Светозарова Г.Н. и др. Практикум по программированию на языке Бэйсик.- М.:Наука, 1988.
18. Успенский В.А., Семенов А.Л. Теория алгоритмов: основные открытия и приложения.- М.: Наука, 1987.
19. Хохлюк В.И. Параллельные алгоритмы целочисленной оптимизации.-М.: Радио и связь,1987.
20. Анастасия КОВАЛЕВА nastusha82@ua.fm
21. Константин Кноп [knop@computerra.ru] Ремесленник или ученый? Мир ПК.28.10.2002
22. Безбородов Ю.М. Индивидуальная отладка программ. - М.: Наука, 1982. - 192 с.
21. Бутаков Е.А. Методы создания качественного программного обеспечения ЭВМ. - М.: Энергоатомиздат, 1984.- 232 с.
23. Ван Тассел Д. Стиль, разработка, эффективность, отладка и использование программ: Пер. с англ.- М.: Мир, 1981. -320 с.
24. Вирт Н. Алгоритмы + структуры данных = программы: Пер. с англ. - М.: Мир, 1985. - 406 с.
25. Громов Г.Р. Национальные информационные ресурсы: проблемы промышленной эксплуатации. - М.: Наука, 1984. - 237 с.
26. Гудман С., Хидетниеми. Введение в разработку и анализ алгоритмов: Пер. с англ. - М.: Мир, 1981. - 368 с.
27. Дал У., Дейкстра Э., Хоор К. Структурное программирование: Пер. с англ. - М.: Мир, 1975. - 247 с.
28. Зельковиц М., Шоу А., Геннон Дж. Принципы разработки программного обеспечения: Пер. с англ. - М.: Мир, 1982. - 368 с.
29. Йодан Э. Структурное проектирование и конструирование программ: Пер. с англ. - М.: Мир, 1979. - 415 с.
30. Кнут Д. Искусство программирования для ЭВМ: В 7-ми т.:Пер. с англ. - М: Мир, 1978.- Т.3. Сортировка и поиск. - 1978. - 841 с.
31. Лингер Р., Миллс Х., Уитт Б. Теория и практика структурного программирования. - М.: Мир, 1982. - 406 с.

32. Майерс Г. Надежность программного обеспечения: Пер. с англ.- М.: Мир, 1980. - 360 с.
33. Майерс Г. Искусство тестирования программ: Пер. с англ. - М.: Финансы и статистика, 1982. - 176 с.
34. Фокс Д. Программное обеспечение и его разработка: Пер. с англ. - М.: Мир, 1980. - 278 с.
35. Хьюз Дж., Мичтон Дж. Структурный подход к программированию: Пер. с англ. - М.: Мир, 1980. - 278 с.
36. Шураков В.В. Надежность программного обеспечения систем обработки данных: Учебник для вузов. - М.: Статистика, 1981.- 272 с.
37. Любченко В.С. Новые песни о главном-II//Мир ПК. 1998. №7. с.112.
38. Успенский В.А. Машина Поста. 2-е изд., испр. М.: Наука, 1988. 96 с. (Популярные лекции по математике).
39. Катленд Н. Вычислимость. Введение в теорию рекурсивных функций: Пер. с англ. М.: Мир, 1983. 256 с.
40. Кузьминский М. Вышел Merced из тумана//Computerworld Россия.1997. №47. С.31.
41. Хамби Э. Программирование таблиц решений. М.: Мир, 1976. 86 с.
42. Глушков В.М. Введение в кибернетику. Киев: Изд-во АН Укр ССР. 1964. 324 с.
43. Питерсон Дж. Теория сетей Петри и моделирование систем: Пер. с англ. М.: Мир, 1984. 264 с.
44. Краснов С.А. Транспьютеры, транспьютерные вычислительные системы и Оккам//Вычислительные процессы и системы / Под ред. Г.И. Марчука. Вып. 7. М.: Наука., 1990. 352 с.
45. Любченко В.С. Эксперементы над абстрактными машинами //Мир ПК. 2002. №2.
46. Рассел Б. Введение в математическую философию. – Новосибирск: Сибирское университетское издательство, 2007.
47. Ландау Э. Основы анализа. Действия над целыми, рациональными, иррациональными, комплексными чисолами. – 3-е изд. – М.: URRS, 2010.
48. Нагель Э., Ньюмен Дж. Теорема Геделя. – 2-е изд. – М.: URRS, 2010.
49. Успенский В.А. Теорема Геделя о неполноте. – М.: «Наука», 1982.
50. Інформатика і комп'ютерна техніка: Навчальний посібник / За ред. М.Є. Рогози. - К.: Видавничий центр «Академія», 2006. – 368 с.
51. Інформатика. Базовий курс / Симонович С.В. і ін.– СПб: Питер, 2000. – 640 с.
52. Семашко Г.Л., Салтыков А.И. Програмування на мові Паскаль – М.: Наука, 1998. – 128 с.
53. Фаронов В.В. Турбо Паскаль 7.0 – М.: Нолидж, 2000. – 576 с.
54. Войтюшенко Н.М. Інформатика і комп'ютерна техніка: Навч. пос. з баз.підготовки для студ.екон. і техн.спеціальностей ден. і заочн. форм навчання. - К.: Центр навчальної літератури, 2006. – 568с.
55. Дибкова Л.М. Інформатика та комп'ютерна техніка: Навч.посіб. Вид. 2-ге, перероб., доп. - К.: Академвидав, 2007.- 416с.(Альма-матер).
56. Беспалов В.М. Інформатика для економістів: Навч.посібник.- К.: ЦУЛ, 2003.- 787с.
57. Інформатика та комп'ютерна техніка: Навч.-метод.посібник/ За заг. ред. О.Д.Шарапова.- К.: КНЕУ, 2002.- 534с.
58. Інформатика та комп'ютерна техніка: Навчально-методичний посібник для самост.вивчення дисципліни/ Д.О.Рзаєв, О.Д.Шарапов, В.М.Ігнатенко, Л.М.Дибкова.- К.: КНЕУ, 2003.- 485 с.
59. Інформатика та комп'ютерна техніка: програмне забезпечення ЕОМ: Навчальний посібник /П.А.Щербаков, О.В.Ульянченко, Н.М.Мартьянова, Т.А.Бутенко.- Харьков: ХДАУ ім.В.В.Докучаєва, 2001.-291с.-NT
60. Методичні вказівки і завдання для індивідуальної роботи студентів економічних спеціальностей з дисципліни "Інформатика та комп'ютерна техніка".-Тернополь: Економічна думка, 1999.-82 с.
61. Информатика для юристов и экономистов: Учебник для вузов/ Под ред. С.В.Симоновича.– С.-Пб.: Питер, 2001.– 687с.
62. Информатика. Базовый курс: Учебное пособие для студентов технических вузов/ Под ред.С.В.Симоновича.– С.-Пб.: Питер, 2000.– 638 с.

Зміст