МІНІСТЕРСТВО ОСВІТИ УКРАЇНИ
ІНСТИТУТ ЗМІСТУ ТА МЕТОДІВ НАВЧАННЯ
ХЕРСОНСЬКИЙ ДЕРЖАВНИЙ ПЕДАГОГІЧНИЙ ІНСТИТУТ

М.С.Львов, О.В.Співаковський

Основи алгоритмізації та програмування

частина 1

Затверджено Міністерством освіти України як посібник для студентів вищих навчальних закладів

Херсон - 1997

М.С.Львов, О.В.Співаковський
Основи алгоритмізації та
програмування
Навч. посібник - Херсон, 1997. - 122
с.

У даному навчальному посібнику викладається мова процедурного програмування Pascal і розглядаються основні методи проектування ефективних процедурних алгоритмів багатьох класичних задач програмування. Це - задачі сортування і пошуку, деякі задачі штучного інтелекту, задачі вибору. Методи розв'язання цих задач дають достатньо повне представлення про науку програмування і оволодіння ними послужить доброю основою для подальшого удосконалення у викладанні, практиці і теорії програмування.

У книзі описаний і використовується в основному стандарт мови. Виняток складають графічні засоби модуля GRAPH Turbo Pascal v.6 і деякі інші засоби (наприклад, рядки).

Відмінною особливістю книги є те, що мовні конструкції подаються паралельно з викладенням методів побудови і аналізу ефективних алгоритмів, що сприяє глибокому засвоєнню мови і стилю процедурного програмування.

Кожний параграф містить набір задач для самостійного розв'язання, або для розв'язання на практичних заняттях під керівництвом викладача.

Посібник призначено для майбутніх програмістів, вчителів інформатики, студентів вищих і середніх спеціальних навчальних закладів, професія яких пов'язана з індустрією інформатики. Він може використовуватись як основа факультативів і гуртків з програмування у школах. Книга також буде корисною широкому колу читачів, що бажають оволодіти основами програмування.

Рецензенти:
А.М.Гуржій, член-кореспондент АПН України, доктор технічних наук, професор
Є.П.Голобородько, член-кореспондент АПН України, академік Академії суспільних і педагогічних наук, доктор педагогічних наук, професор Херсонського державного педагогічного інституту
В.Г.Бутенко, член-кореспондент АПН України, доктор педагогічних наук, професор Херсонського державного педагогічного інституту

ISBN 5-7763-8332-3

АЛГОРИТМИ ТА ПРОГРАМИ

1. Поняття алгоритму.

Точне математичне визначення алгоритму та вивчення цього поняття - предмет спеціальної математичної дисципліни - теорії алгоритмів, яка, в свою чергу, спирається на апарат математичної логіки. Тут ми розглянемо на змістовному (неформальному) рівні лише основні характерні риси цього поняття.

Алгоритм - це правило перетворення інформації, застосування якого до заданої (вхідної) інформації приводить до результату - нової інформації.

Так, наприклад, застосування правила додавання дробів до 1/2 і 2/3 дає результат 5/6.

Основну увагу в теорії алгоритмів приділяється методам завдання (описання, конструювання) алгоритмів.

Алгоритм - це скінченний набір інструкцій по перетворенню інформації (команд), виконання яких приводить до результату. Кожна інструкція алгоритму містить точний опис деякої елементарної дії по перетворенню інформації, а також (в явному або неявному вигляді) вказівку на інструкцію, яку необхідно виконувати наступною.

Так, алгоритм додавання дробів можна задати такою послідовністю команд:

Приклад 1. Алгоритм додавання дробів.

Вхід: A/B, C/D;

1. Обчислити Y = B*D; {Перейти до наступної команди}
2. Обчислити X_1 = A*D; {Перейти до наступної команди}
3. Обчислити X_2 = B*C; {Перейти до наступної команди}
4. Обчислити X = X_1+X_2; {Перейти до наступної команди}
5. Обчислити Z = НСД(X,Y); {Перейти до наступної команди}
6. Обчислити E = X div Z; {Перейти до наступної команди}
7. Обчислити F = Y div Z; {Завершити роботу}.

Вихід: E/F

Наш алгоритм складається з 7-ми інструкцій, кожна з яких містить опис однієї з арифметичних дій над цілими числами: додавання, множення, обчислення НСД і цілочисленне ділення div. Крім того, кожна інструкція (у неявному вигляді) містить вказівку на наступну виконувану інструкцію. Таким чином, алгоритм описує деталізований по кроках процес перетворення інформації. Рівень деталізації опису визначається набором можливих команд. Цей набір називають набором команд Виконавця або Інтерпретатора.

При цьому мається на увазі, що алгоритми виконує Виконавець (Інтерпретатор) - деякий пристрій, що однозначно розрізнює і точно виконує (інтерпретує) кожну команду алгоритму.

Для виконання нашого алгоритму Виконавець повинен, очевидно, вміти оперувати з цілими числами, виділяти чисельник і знаменник дробів, а також складати з пари цілих чисел дріб. Крім того, Виконавець повинен вміти запам'ятовувати результати виконання операцій і переходити до виконання наступної команди.

Уявимо собі, що у нашому розпорядженні знаходиться Виконавець, який інтерпретує команди - операції цілочисельної арифметики - додавання віднімання, множення, обчислення неповної частки (div) та остачі (mod), обчислення НСД і НСК з запам'ятовуванням результатів і вмінням переходити до наступної команди. Тоді для цього Виконавця можна складати найрізноманітніші алгоритми арифметичних обчислень - тобто обчислень, заданих формулами типу

X = НСД ((A + B) div 100, (A * B - 7) mod 10)

використовуючи команди, аналогічні командам алгоритму з приклада 1.

Приклад 2. Алгоритм поділу відрізка навпіл за допомогою циркуля та лінійки.

Вхід: Відрізок AB;
Побудувати коло O_1 з центром A і радіусом AB;
Побудувати коло O_2 з центром B і радіусом AB;
Знайти точки C і D перетину кіл O_1 і O_2;
Побудувати відрізок CD;
Знайти точку E перетину AB і CD.
Вихід: Точка E - середина відрізку AB.

У прикладі 2 Виконавець має набір команд, за допомогою яких можна розв'язувати геометричні задачі на побудову за допомогою циркуля і лінійки.

Виконання алгоритму полягає у послідовному виконанні кожної побудови і переході до виконання наступної команди.

Відзначимо, що нумерувати команди алгоритму не треба, якщо Виконавець завжди переходить до наступної команди.

Приклад 3. Алгоритм розв'язування наведеного квадратного рівняння $x^2 + px + q = 0$;

Вхід: Коефіцієнти p і q рівняння $x^2 + px + q = 0$;
Обчислити $D = p^2 - 4q$;
Якщо D < 0 то (відповідь "Розв'язків немає"; Перейти до 1);
Якщо D = 0 то (обчислити x = -p/2; Перейти до 1);

Обчислити $x_1 = (-p + \sqrt{D}) / 2$;

Обчислити $x_2 = (-p - \sqrt{D}) / 2)$;

1: Завершити роботу.

Вихід відповідь “Розв’язків немає” або корінь х або корені x_1, x_2.

В цьому прикладі використовується команда виду

Якщо <умова>

то (<послідовність команд>)

Виконуючи цю команду, Виконавець перевіряє істинність умови. Якщо умова виконана, Виконавець переходить до виконання першої команди з послідовності команд, яка стоїть після слова то і команди, і виконання алгоритму йде тим чи іншим шляхом, що знаходиться в дужках. Якщо ж умова не виконана, Виконавець переходить до виконання наступної команди. Такі команди називають вибираючими, умовними або розгалуженнями. Вибираючі команди містять у собі інші команди, які виконуються в залежності від результатів перевірки умов.

Іншою характерною командою, яка використовується у прикладі, є команда переходу. Вона має вид Перейти до < N >, причому число N використовується в запису алгоритму як спеціальна відмітка деякої команди. У прикладі використовує команда переходу Перейти до 1, а числом 1 відмічена команда 1: Закінчити роботу.

Виконання команди переходу заключається в тому, що Виконавець переходить до виконання команди, позначеної відміткою N (порушуючи при цьому природну послідовність виконання команд).

2. Характерні властивості алгоритмів.

Поняттю алгоритму притаманні такі властивості:

1. Елементарність. Кожна команда з набору команд Виконавця містить вказівку виконати деяку елементарну (не деталізуєму більш повно) дію, яку Виконавець однозначно розуміє і виконує.

Поняття елементарності тому відносне. Так в алгоритмі прикладу 1 міститься команда обчислення НСД двох чисел. Це означає, що Виконавець вміє знаходити НСД, причому алгоритм обчислення (алгоритм Євкліда або якийсь інший) захований від людини, яка складає алгоритми для цього Виконавця. Якщо набір команд Виконавця не містить команди обчислення НСД, обчислення НСД певно визначено у виді алгоритму.

Приклад 4. Алгоритм Євкліда обчислення найбільшого спільного дільника цілих додатних чисел А і В: НСД (А,В).

Вхід A,B;
Обчислити U = A;
Обчислити V = B;
Поки U <> V виконувати
Якщо U < V
то Обчислити V = V - U
інакше Обчислити U = U - V;
Обчислити D = U.
Вихід: D - найбільший спільний дільник A і B.

У цьому прикладі використана команда повторення. Вона має вид
Поки <Умова> виконувати <Команда>
Виконуючи цю команду, Виконавець перевіряє істинність Умови. Якщо Умова істинна, Виконавець виконує Команду, вказану після слова виконувати і повторює перевірку Умови. Виконання Команди і перевірка Умови повторюються до тих пір, поки Умова істинна. Якщо Умова хибна, Виконавець переходить до виконання команди, наступної за командою повторення. В цьому ж прикладі використаний ще один різновид команди розгалуження - команда виду.
Якщо < Умова > то <Команда 1 > інакше < Команда 2 >
Виконуючи цю команду, Виконавець перевіряє Умову. Якщо умова виконана, виконується Команда 1, в іншому випадку виконується команда 2. Далі Виконавець переходить до наступної команди. Відмітимо, що команда повторення, як і команди розгалуження, містить у собі інші команди.

2.Визначеність. Виконання алгоритму суворо визначено. Це означає, що на кожному кроку Виконавець не тільки точно виконує команду, але й однозначно визначає наступну команду, що буде виконуватись. Тому повторне виконання алгоритму для одних і тих же вхідних даних у точності повторює перше його виконання.
Так, у виконанні алгоритму у прикладі 3 можливі розгалуження, які, однак, однозначно визначені умовами D < 0, D = 0.

3. Масовість. Алгоритми, як правило, описують хід рішення не одної-єдиної задачі, а цілого класу однотипних задач.

Так у прикладі 2 описаний алгоритм додавання будь-яких двох дробів. Одна-єдина задача, яка розв'язується Виконавцем у даний момент, визначена значеннями вихідних даних A, B, C, D. Зміна вхідних даних означає розв'язування іншої задачі з цього ж класу задач.

Аналогічно, алгоритм прикладу 2 будує середину будь-якого відрізка, заданого його кінцями, а у прикладі 3 за допомогою алгоритму розв'язується будь-яке наведене квадратне рівняння.

4. Результативність. Виконання будь-якого алгоритму повинно бути закінчено через скінченне число кроків (тобто виконання скінченного числа команд) з деяким результатом.

Так, у прикладі 2 результат - точка E - середина відрізку AB. Алгоритм прикладу 3 видає результат “Розв’язків немає” навіть у тому випадку, коли рівняння не має дійсних коренів, оскільки його дискримінант менше нуля.

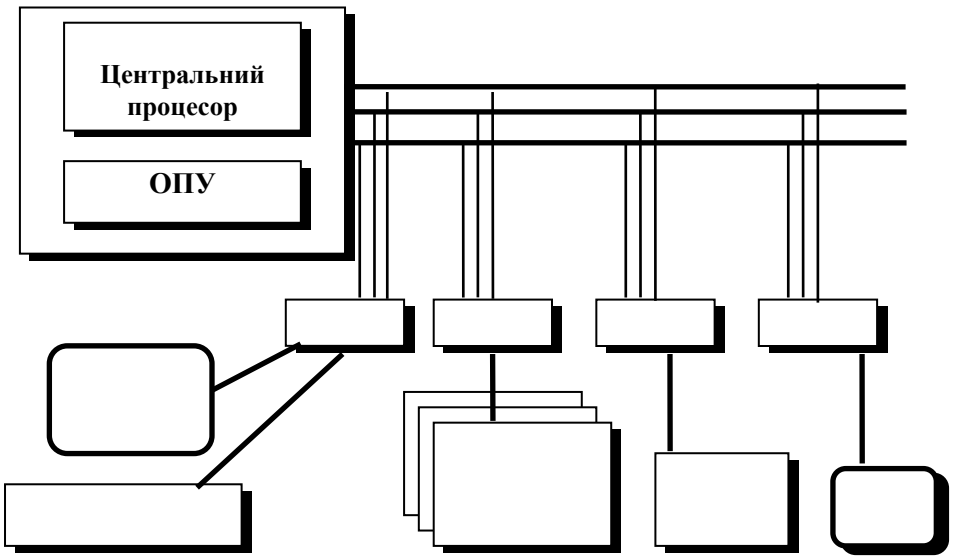
Відмітимо, однак, що кількість кроків алгоритму, який розв’язує деяку задачу, заздалегідь невідомо і може бути дуже великим, тому властивість результативності конкретного алгоритму часто потрібно спеціально доводити. Так, перевірка властивості результативності в прикладі 4 потребує доведення того, що виконання команди повторення закінчиться за скінчену кількість кроків.

Розглянуті приклади показують, що поняття Виконавця є однією з основних абстракцій, які використовують для вивчення алгоритмів. Саме система команд Виконавця і є уточнення набору засобів, за допомогою яких будуються алгоритми. У наших прикладах системи команд Виконавця предметно-орієнтовані. Так, у прикладі 1 Виконавець має справу з цілочисельною арифметикою, а у прикладі 3 - з арифметикою дійсних чисел. Особливо наочно це демонструє Виконавець прикладу 2, команди якого виконують геометричні побудови. Разом з цим очевидно, що існують команди, які повинні входити у систему команд будь-якого Виконавця, який претендує на універсальність у деякій предметній області. Це команди розгалуження, повторення, переходу. Ці команди безпосередньо не вказують на перетворення даних, а лише на керування цими перетвореннями.

Незважаючи на всю різноманітність форм представлення інформації та операцій її перетворення, які використовує людина у своїй діяльності, виявилось можливим створення універсального Виконавця, система команд якого дозволяє промоделювати будь-яку іншу систему команд. Таким Виконавцем є ЕОМ.

3. ЕОМ як універсальний Виконавець.

На рис. 1 представлена блок-схема ЕОМ.



Зовнішні пристрої ЕОМ.

Пристрої введення інформації призначені для введення інформації в ЕОМ. Пристрої введення перетворюють інформацію з форми, призначеної для користувача, у форму, призначену для збереження та обробки в ЕОМ - у двійковий код. Найбільш поширені пристрої введення - клавіатура, сканер, "миша", та інші.

Пристрої виведення інформації призначені для виведення інформації (результатів) у формі, призначеної для користувача - у виді чисел, текстів, малюнків і т.п.

Характерними для персональної ЕОМ пристроями виведення є монітор (дисплей), принтер, плотер.

Зовнішні запам'ятовуючі пристрої (ЗЗП) призначені для тривалого (при вимкненій ЕОМ) збереження інформації. У теперішній час найбільш поширені накопичувачі на гнучких магнітних дисках (НГМД), накопичувачі на твердих магнітних дисках (НМД), накопичувачі на магнітних стрічках (НМС).

Узгодження сигналів, якими обмінюються пристрої ЕОМ у процесі роботи, здійснюють інтерфейсні блоки (контролери). Інтерфейсний блок - це сукупність апаратних і програмних засобів, які підтримують процес обміну даними між пристроями ЕОМ, різними по швидкодії, рівню сигналів, засобу кодування інформації і таке інше.

Центральні пристрої ЕОМ.

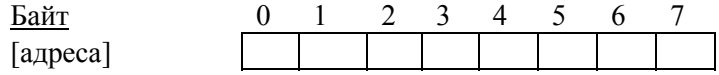
Ядро ЕОМ складають так звані центральні пристрої - центральний процесор і оперативний запам'ятовуючий пристрій (ЦП і ОЗП).

Центральні пристрої призначені для оперативного зберігання та перетворення інформації. Сукупність центральних пристроїв об'єднана в системний блок.

Вся інформація, необхідна для виконання алгоритму - початкові, проміжні і вихідні дані, а також сам алгоритм, зберігається в ОЗП у закодованому виді.

ОЗП представляє собою сукупність спеціальних комірок, кожна з яких призначена для збереження двійкового коду інформації фіксованого обсягу. Кожна комірка пам'яті має

свій номер, який називається адресою. У сучасних ПЕОМ комірка ОЗП зберігає 1 байт інформації - 8 двійкових цифр, які називаються бітами.



Характерною особливістю (ОЗП) є те, що доступ до даних, які там зберігаються, здійснюється за їх адресами. Час доступу до даного не залежить від адреси комірки, в якій воно зберігається. Тому ОЗП називають пам'яттю з прямим (випадковим) доступом. Розміром (об'ємом) ОЗП називають кількість її комірок. Розмір ОЗП виражають у байтах, кілобайтах (Kb) і мегабайтах (Mb). 1 кілобайт = 1024 байта, 1 мегабайт = 1024 кілобайта. І дані, і команди як правило, займають у ОЗП декілька підряд адресованих байтів.

Центральний процесор - пристрій, призначений для виконання алгоритмів, які зберігаються в ОЗП у виді набору команд. Кожний центральний процесор має свою систему команд Виконавця. Система команд реального процесора містить десятки команд, і її вивчення - предмет окремого курсу навчання. Ми розглянемо лише основні принципи побудови машинної мови.

Поняття про машинну мову.

Набір команд процесора містить:

арифметико-логічні команди - команди арифметичних дій над двійковими числами і логічних дій на двійковими векторами;

команди управління - команди переходу, розгалужень, повторень, і деякі інші команди;

команди пересилання даних - команди, за допомогою яких обмінюються даними ОЗП і ЦП;

команди введення-виведення даних - команди, за допомогою яких обмінюються даними ЦП і зовнішні пристрої.

Кожна команда містить код операції, яку вона виконує і інформацію про адреси даних, над якими ця операція виконується. Крім цього, команда (безпосередньо - команди керування і опосередковано - інші команди) містить інформацію про адресу команди, яка буде виконуватися наступною. Таким чином, будь-яка послідовність команд, яка розміщена в ОЗП, фактично представляє собою алгоритм, записаний в системі команд процесора - машинну програму.

Найбільш поширеною зараз є схема ЕОМ з загальною шиною. Загальна шина - це центральна інформаційна магістраль, яка зв'язує зовнішні пристрої з центральним процесором. Вона складається з шини даних, шини адреси і шини управління. Шина даних призначена для обміну даними між ОЗП і зовнішніми пристроями. По шині адреси передаються адреси даних. Ця шина однонаправлена. Шина управління служить каналом обміну управляючими сигналами між зовнішніми пристроями і центральним процесором.

Таким чином, мова процесора - це набір команд, кожна з яких описує деяку елементарну дію по перетворенню інформації, яка представлена у двійковому коді. Універсальне використовування двійкового коду представлення інформації самих різноманітних форм приводить до того, що програма рішення навіть достатньо простої задачі містить сотні машинних команд. Написати таку програму, використовуючи машинні команди, дуже непросто навіть кваліфікованому програмісту. Реальні програми складаються з десятків і сотень тисяч машинних команд. Тому будь-яка технологія проектування програми повинна опиратися на заходи, характерні для людського мислення, оперувати звичними для людини поняттями з тієї предметної області, якій належить задача.

Іншими словами, програміст (проектувальник алгоритмів) повинен мати можливість сформулювати свій алгоритм мовою звичних понять; потім спеціальна програма повинна виразити ці поняття за допомогою машинних засобів, - здійснити переклад (трансляцію) тексту алгоритму на мову машини.

Ця необхідність і привела до появи мов програмування високого рівня як мов запису алгоритмів, які призначені для виконання на ЕОМ.

МОВИ ПРОГРАМУВАННЯ І СИСТЕМИ ПРОГРАМУВАННЯ.

1. Мови програмування високого рівня.

Мови програмування високого рівня грають роль засобу зв'язку між програмістом і машиною, а також між програмістами. Ця обставина накладає на мову багато обов'язків:

1.Мова повинна бути близькою до тих фрагментів природничих мов, які забезпечують конкретну предметну область діяльності людини; (Мова, яка орієнтована на ділові сфери використань, повинна містити поняття, які використовуються у цьому виді діяльності: рахунок, база даних і т.п.).

2.Всі засоби мови повинні бути формалізовані у такому степені, щоб їх можна було реалізувати як машинні програми;

(наприклад, речення "Знайти документ X у базі Y" повинно генерувати програму в машинній мові, яка здійснює потрібний пошук).

3.Мова програмування не тільки підтримує предметно-орієнтовну діяльність , але і стимулює її розвиток (поняття бази даних, обчислювальної мережі привело до революції у діловій діяльності).

4.Мова програмування - дещо більш, чим засіб опису алгоритмів: він несе в собі систему понять, на основі яких людина може обдумувати свої задачі, і нотацію, за допомогою якої він може виразити свої розуміння з приводу рішення задачі.

Вивчаючи нову мову програмування, краще всього до неї відноситися, як до будь-якої іншої іноземної мови: засоби мови приймати як дані від Бога, навіть якщо вони нам здаються незрозумілими, поганими або непотрібними.

2. Коротка історія розвитку мов програмування.

Ідея мови програмування з'явилась так само давно, як і універсальні обчислювальні машини - на межі 40-50 років. Вже на перших кроках їх експлуатації виявилися недоліки використання машинного коду, визначились методи усунення або зменшення цих недоліків: використання бібліотек стандартних програм, імен замість адрес, попереднього розподілу пам'яті і т.п.

Великий вплив на наступні розробки виявила мова Fortran, створена у IBM під керівництвом Дж. Бекуса (1954-57 р.р.). У той же час М.Г.Хоппер (Ramington-Rand Univac) і її група розробили мову обробки комерційної інформації Flow-Matic. М.Г.Хоппер належить термін "компілятор". Таку назву мала її перша програма, яка транслювалась.

Перші виробничі мови програмування з'явились на межі 50-60 років, знаменуючи собою нову епоху в розвитку обчислювальних машин і методів обробки інформації. Ці мови високого рівня були реалізовані на перших ЕОМ 2-го покоління.

Ось деякі дати:

1957 р. Fortran США, IBM, Дж. Бекус: по суті перша широко застосовувана мова, орієнтована на науково-інженерні і чисельні задачі.

1960 р. Cobol США, Об'єднаний комітет виробників та користувачів ЕОМ: мова для комерційних задач.

1960 р. Algol-60. Поліпшений варіант мови Algol-58, Європа, США, міжнародна робоча група: універсальна мова, пращур Pascal -я і багатьох інших мов європейського стилю.

1965 р. BASIC Дж. Кемені, Т.Куртц, США, Дармутський коледж: мова для починаючих.

1969 р. Logo С.Пейперт, США, Массачусетський технологічний інститут: мова для дітей.

1966 р. PL-1 група IBM, США: Багатоцільова мова для систем колективного користування.

1968 р. Algol-68. Європа, міжнародна робоча група: європейська відповідь на PL - 1.

1970 р. Pascal Н. Вірт, Швейцарія, федеральний інститут технології, Цюрих: мова для навчання спеціалістів в області інформатики.

1959 р. Lisp Дж.Маккарті, США, Массачусетський технологічний інститут: мова функціонального програмування.

1972 р. Prolog А.Колмєрое і його колеги з лабораторії Штучного інтелекту, Марсельський університет, Франція: мова логічного програмування, що завоювала широку популярність як мова для задач обробки баз знань.

1972-75 р.р. С і його розвиток С++. Д.Креніган, Д.Річі, Б.Страуструп, AT & T Bell Lab.,: мови системного програмування, які отримали широке розповсюдження завдяки своїй ефективності і підтримці ведучих компаній, що розробляють програмне забезпечення.

1975 р. Modula - 2 Н. Вірт, Розвиток мов Pascal і Modula для системного програмування.

Перші мови програмування несли у собі явно виражені ознаки орієнтації на структуру ЕОМ. Вважалось, що програми, написані ними, призначені для виконання на ЕОМ. (Fortran - програми до цих пір пишуть на спеціальних бланках, орієнтованих на перфорацію. Ще одна яскраво виражена ознака машинної орієнтації - мітки і оператор GOTO.)

В результаті теоретичного осмислення процесів, які відбувалися у програмуванні, був вироблений так званий структурний підхід до написання програм, а для його реалізації розроблені такі мови, як Pascal, Modula - 2. Ідеологи структурного підходу вважають, що ЕОМ призначені для виконання програм, а не програми - для виконання на ЕОМ.

Перенесення акцентів з ЕОМ на програми ще більш яскраво виявилось у появі принципово нових стилів програмування - функціонального програмування (Lisp), логічного програмування (Prolog), алгебраїчного програмування (Reduce, APS).

У цих мовах центральну роль грають не процедури обробки даних, а співвідношення між даними, які повинні виконуватись у процесі виконання програми. Тому

ці мови, на відміну від процедурних (вказівних, імперативних), отримали назву декларативних (описових).

Сучасний етап у розвитку програмування характеризується наступними рисами:

- Розвиток мов програмування для мультіпроцесорних і мультімашинних систем;
- Розвиток декларативних мов програмування, орієнтованих на задачі штучного інтелекту;
- Розвиток об'єктно-орієнтованих мов, в яких ієрархія абстракцій дозволяє нарощувати засоби мов, одночасно змінюючи архітектуру ЕОМ стосовно до класу проблем, який розглядається.

3. Основні етапи проектування програми.

Процес проектування програми подібний процесам проектування складних систем. Ось його основні етапи:

- Постановка задачі і вибір її математичної моделі;
- Розробка алгоритму розв'язування задачі;
- Вибір апаратного обладнання і мови програмування;
- Написання програми;
- Налагодження і редагування;
- Контрольні випробування;
- Експлуатація.

Насправді проектування програми - не лінійний, а циклічний процес, на кожному кроку якого можливе повернення до будь-якого з попередніх кроків:

У теперішній час праця програміста у значній мірі регламентована спеціальними методиками, довідковими посібниками і стандартами. Розробники операційних систем і систем програмування велику увагу приділяють спеціальним пакетам програм, які автоматизують працю програміста. Але, не дивлячись на все, процес розробки програм все ще далекий від досконалості.

4.Технологія трансляції програм.

Взагалі ЕОМ не розрахована на те, щоб розуміти LOGO, Pascal або інші мови програмування високого рівня. Апаратура розпізнає і виконує тільки машинну мову, програма на якій являє собою не більш ніж послідовність двійкових чисел.

Поява мов програмування була пов'язана з осмисленням того факту, що переклад алгоритму, написаного "майже" природною мовою, на машинну мову може бути автоматизований і, отже, покладений на плечі машини. Тут важливо розрізняти мову і її реалізацію. Сама мова - це система запису, яка регламентується набором правил, що визначають його лексику і синтаксис. Реалізація мови - це програма, яка перетворює цей запис у послідовність машинних команд у відповідності з прагматичними і семантичними правилами, визначеними у мові.

Існують два основних засоби реалізації: компілятори і інтерпретатори.

Компілятори транслюють весь текст програми, написаний мовою програмування, у машинний код у ході одного безперервного процесу. При цьому створюється повна програма у машинних кодах, яку потім можна виконувати без участі компілятора.

Інтерпретатор у кожний момент часу розпізнає і виконує по одному реченню (оператору) програми, за ходом справи перетворюючи речення, яке оброблюється у машинну програму. Різниця між компілятором і інтерпретатором подібна різниці між синхронним перекладом усної мови і письмовим перекладом тексту.

У принципі будь-яка мова може бути і компилюємою, і інтерпретируємою, однак у більшості випадків кожна мова має засіб реалізації, якому віддають перевагу. Напевно, така перевага - дещо більше, ніж данина традиції. Вибір визначений самою мовою. Fortran, Pascal, Modula-2 в основному компілюють. Такі мови як Logo, Fort майже завжди інтерпретують. BASIC і Lisp широко використовується в обох формах. Кожний з цих засобів має як свої переваги, так і недоліки:

Основні переваги компіляції:

Швидкість виконання готової програми;

Незалежність програми від системи реалізації;

Основні недоліки компіляції:

Деякі незручності, які перевіряються програмістом при написанні і налагодженні великих програм;

Порівняно великий об'єм пам'яті, який займає компілятор в ОЗП;

5. Поняття про систему програмування.

Розглянемо послідовність змін, що відбуваються з програмою під час виконання (у процесі компіляції).

Текст програми, який написаний мовою програмування, називається вихідним модулем. Достатньо складні програми можуть складатися з кількох модулів, взаємодіючих друг з другом. Вихідний модуль - це вхідний потік для програми - компілятора, яка

- Здійснює лексичний аналіз вхідного потоку [блок лексичного аналізу];
- Здійснює синтаксичний аналіз вхідного потоку [блок синтаксичного аналізу];
- Генерує машинний код, який є перекладом вихідного модуля на мову ЕОМ в умовних адресах [генератор коду];

В результаті цих перетворень на виході отримується об'єктний модуль.

Навіть якщо ми маємо справу з одним вихідним модулем, для успішного виконання програми необхідно "зв'язати" її з деякими іншими програмами (наприклад, з стандартними процедурами введення-виведення, які реалізовані в мові). Ці функції виконує програма - редактор зв'язків. Вихідний потік цієї програми називають завантаженим модулем.

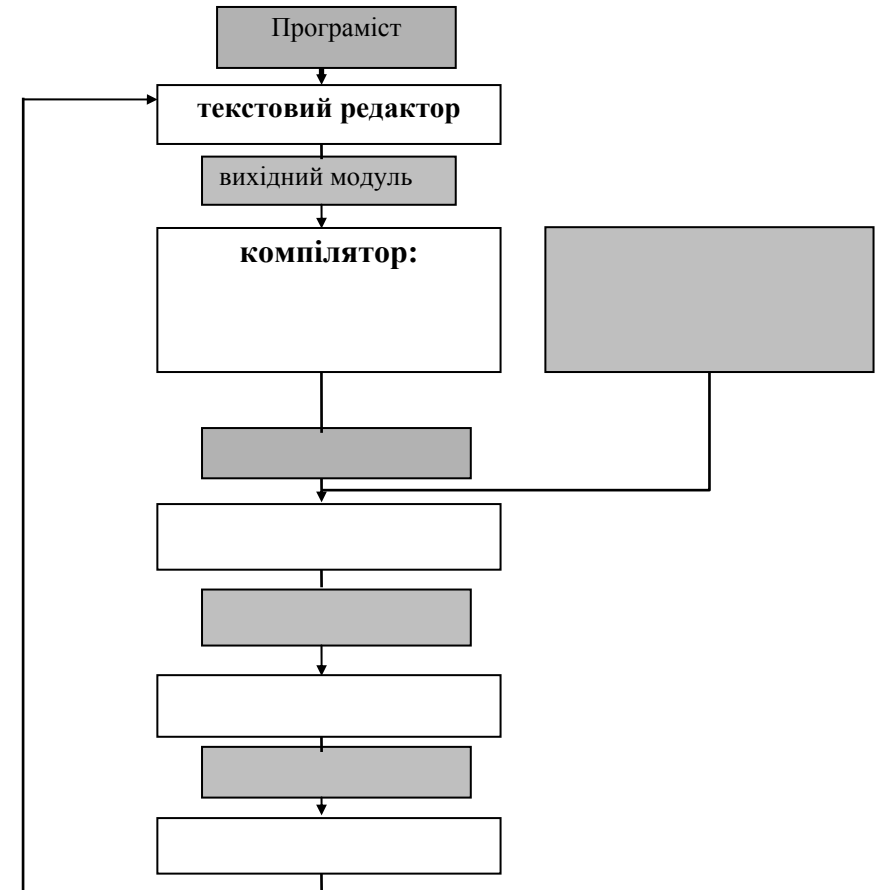
Сучасна технологія застосовування ЕОМ потребує, щоб виконуючу програму можна було розміщувати в довільному місці ОЗП. Тому і завантажений модуль написаний в умовних адресах. Розміщенням завантаженого модуля в пам'ять займається програма - завантажувач. Як правило, програми, які щойно написані, містять безліч помилок. Помилки бувають:

а) синтаксичні і лексичні (виявляються на етапах лексичного і синтаксичного аналізу). Наприклад, помилка y := sos(x) замість y:= cos(x) - лексична, а помилка в операторі if x < 0 then y:= 0; else y:= 1 - синтаксична.

б) семантичні (виявляються на етапі налагодження). Наприклад, помилка в операторі присвоювання - ділення на нуль: x:= y; z:= 1/ (x - y).

в) логічні (виявляються на етапі контрольних випробувань). До логічних ми відносимо такі помилки, в результаті яких програма не виконує того, що ми від неї чекаємо. Для автоматизації процесу пошуку і усунення семантичних і частково логічних помилок використовуються спеціальні програми, які називають налагоджувачами.

Процес перетворення тексту вихідного модуля в модуль, який виконується можна зобразити схематично:



Звичайно в склад системи програмування включають власний текстовий редактор, інші сервісні програми. Для керування роботою будь-яких частин системи програмування використовують керуючу програму, яку називають оболонкою.

Таким чином, для роботи в мові програмування використовуються спеціальні пакети програм, які називають системами програмування (СП). У склад СП входять:

- Оболонка
- Текстовий редактор
- Компілятор
- Редактор зв'язків
- Завантажувач
- Налагоджувач
- Бібліотеки стандартних процедур і функцій
- Сервісні програми

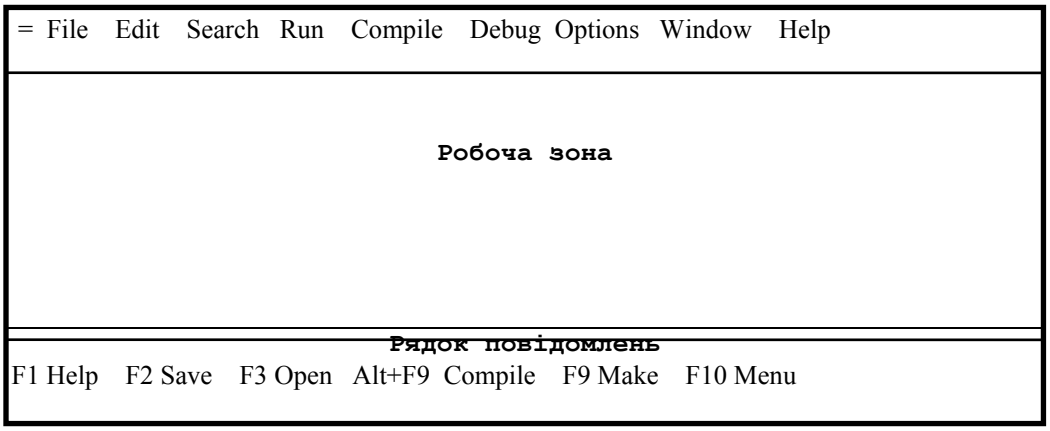
6. Короткий опис системи Turbo Pascal.

Система програмування Turbo-Pascal (TP) була створена фірмою Borland. Перша версія системи, яка з'явилась на ринку в 1985 році, (TP v. 3.0) швидко завоювала популярність завдяки добре реалізованій ідеї інтеграції всіх частин системи в єдиному середовищі програмування. У теперішній час як професіоналами, так і в навчанні активно використовується система TP v.6.0, можливості якої значно розширені. Нещодавно на ринку з'явилась наступна - 7-ма версія системи і треба очікувати, що цей ряд буде продовжуватись.

Інтегрована Інструментальна Оболонка (ІІО). Виклик системи TP виконується директивою Turbo, яка ініціює екран ІІО. Весь процес програмування виконується в середовищі ІІО.

Екран Інтегрованої Інструментальної Оболонки

Зона головного меню



Екран ІІО ТР розділений на 3 зони: - зона головного меню, робоча зона, рядок повідомлень.

Зона головного меню (верхній рядок екрану) містить набір груп команд, які використовуються програмістом при роботі в системі ТР.

=	етикетка системи, команди роботи з екраном;
File	команди роботи з файлами;
Edit	команди редагування файлу, які опрацьовують групи рядків;
Search	команди пошуку і заміни;
Run	команди запуску програми на компіляцію і наступне виконання;
Compile	команди запуску програми на компіляцію;
Debug	команди налагоджувача програм;
Options	команди конфігурування системи;
Window	команди роботи з різними вікнами;
Help	довідкова інформація про систему.

Рядок повідомлень (нижній рядок екрану) демонструє деякі команди, які найбільш часто використовують ІІО і комбінації клавіш для їх швидкого виконання (не звертаючись до головного меню). Наприклад, комбінація Alt - F9 запускає програму на компіляцію. В цей рядок система також вводить службові і довідково-інформаційні повідомлення.

Основне місце на екрані займає **Робоча зона**. В робочій зоні розташовано декілька вікон, в які виводяться тексти програм, вхід/ вихід програми, відладочна інформація і т.п. Будь-яке з вікон може бути активізоване (зроблено доступним для редагування). Кожне вікно можна закрити (за допомогою “мишки”, комбінацією Alt - F3 або відповідною командою головного меню).

На закінчення відмітимо, що робота в системі ТР потребує певних навичок, які вироблюються достатньо швидко в процесі практичної роботи в системі з використанням як довідкових матеріалів самої системи, так і спеціальних учбових посібників з роботи в ТР.

МОВА ПРОГРАМУВАННЯ PASCAL (ВСТУП).

Мова програмування Паскаль розроблена видатним швейцарським ученим і педагогом в області програмування Н.Віртом. Попереднє повідомлення з'явилось в 1968 р. В 1971 році запрацював перший компілятор переглянутої версії, яка набула статус стандарту.

Паскаль призначався для навчальних цілей - для викладання основ програмування студентам - майбутнім спеціалістам в області інформатики. Дуже швидко мова завоювала популярність не тільки в середовищі викладачів і студентів, але й серед професіоналів завдяки своїй компактності, гнучкості, старанній розробці концепції.

Існуючі сьогодні реалізації мови, зберігши його стандарт в якості ядра, володіють дуже потужними додатковими засобами, що сприяє широкому використанню мови.

1. Алфавіт мови.

В мові використовуються:

1.Латинські букви (великі і маленькі), знак підкреслення "_"

2.Цифри 0,...,9

3.Математичні символи +, -, *, /, <, >, =

4.Роздільники: ; " ' . : ^

5.Дужки ( ) [ ] { }

6.Інші символи (що використовуються для друку): букви національних алфавітів, !, ?, \, ...

У різних версіях можуть використовуватися різні набори символів. Зараз широко використовується набір символів коду ASCII (American Standard Code for Information Interchange).

Цей код передбачає розширення для національних алфавітів, символів псевдографіки, які можуть змінюватись від версії до версії.

2. Концепція даних.

Дані є загальне поняття для всього того, з чим оперує ЕОМ. Мови програмування дозволяють нам абстрагуватись від деталей представлення даних на машинному рівні перш за все за рахунок введення поняття типу даних.

У мові Pascal представляються числа і рядки.

Цілі числа записуються в десятковій системі числення: 137, -56, +26.

Дійсні числа використовують також десяткову нотацію, причому ціла частина відокремлюється від дробової не комою, а крапкою. Для позначення порядку числа в якості роздільника використовується буква E. Наприклад, -5.1E14 означає -5.1, помножене на 10 у степені 14 ($-5{,}1*10^{14}$). Порядки можуть бути і від'ємними: 6.74E-8, -56.89E-10.

Послідовності символів в лапках, називаються рядками. Якщо у рядок треба включити лапки, то замість неї записують дві лапки:

' рядок з символів ', ' апостроф ' ' у слові '

В мові дуже старанно в відповідності з концепцією структурного програмування пророблено поняття типу даних. Існують так звані прості (елементарні) типи, які або визначені як стандартні, або визначаються програмістом. Визначений також механізм описання і використання складних типів з більш простих як з базових.

3. Імена та їх застосування.

Іменем в мові називається послідовність (латинських) букв, знака підкреслення "_" і цифр, що починається з букви або знака підкреслення. Хоча імена можуть бути будь-якої довжини, в реалізації кількість значущих символів в імені може бути обмежено. У стандарті мові імена розрізняються по першим восьми символам. Це означає, що імена VeryLongNumber, VeryLongCardinal у стандарті мови позначають (іменують) один і той же об'єкт. Крім цього, мова не розрізняє великих і маленьких букв. Тому імена Sin, SIN, sin не розрізняються.

Імена використовуються для позначення констант, типів, змінних, процедур і функцій. Наприклад:

Pi, Constant - імена констант; x, y1, y2, Counter - імена змінних;

Integral, MaxMin - імена процедур; Man, Color, WeekDay - імена типів;

Деякі імена визначені наперед. Наприклад:

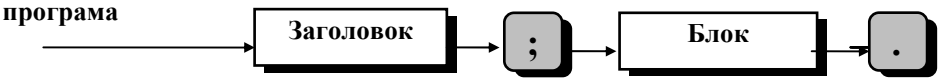
Sin - ім'я для позначення функції синус; Read - ім'я для позначення процедури читання;

Вони називаються стандартними. Всі інші імена вибираються програмістом на його погляд. Однак, для того, щоб програма краще читалась, рекомендується вибирати імена, що несуть інформацію про названий об'єкт.

Нажаль, практично всі реалізації мови допускають використання тільки латинських букв в іменах, тому програми не так зрозумілі неангломовним користувачам, як цього хотілось би.

4. Структура Pascal-програми.

Програма на мові Pascal складається з заголовку і блоку. Блок називають тілом програми. Заголовок програми відділений від тіла крапкою з комою. Крапка, що стоїть після блоку, служить ознакою завершення програми. Таким чином, програма має вид:



Відзначимо, що малюнок, приведений вище, більш точно, наочно і коротко визначає поняття програми, ніж попередній текст. Тому в подальшому ми, наслідуючи традицію, що йде від автора мови, будемо застосовувати саме такий спосіб означень мовних конструкцій, який називають мовою синтаксичних діаграм.

5. Поняття про лексику, прагматику, синтаксис і семантику мови.

Будь-яка мова програмування схожа з природними мовами. Як і природна мова, вона має свій алфавіт, словник, знаки пунктуації (роздільники), за допомогою яких можна утворювати більш складні мовні конструкції, подібні реченням природної мови. Словник мови програмування складається з чисел, слів і деяких інших символів. Елементи цього словника називають лексемами. Приклади лексем:

394, -5678, 12.456, 67.5e8 - числа; Integer, Cos, MaxInt - імена; (,) - дужки.

Мова програмування містить набір правил побудови лексем. Сукупність цих правил називається лексикою мови. Лексичні правила для чисел, строк і імен наведені вище.

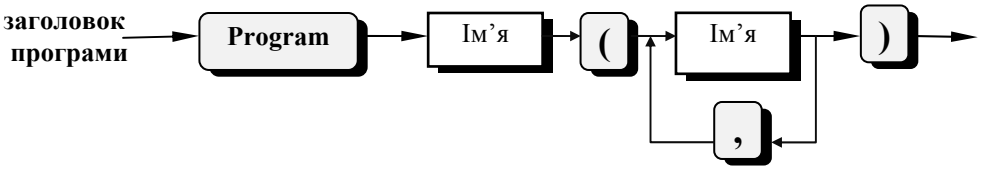
Оскільки текст будь-якої програми є послідовність лексем, основна задача лексичного аналізу - перевірка правильності написання і ідентифікація лексем в цьому тексті. Кожна лексема має свою інтерпретацію (смисл). Так, послідовність цифр, розмежована крапкою, інтерпретується як дійсне число в десятинній нотації, а Cos - як ім'я функції. Сукупність інтерпретацій лексики мови називається його прагматикою.

Правила утворення більш складних конструкцій мови називаються синтаксичними. Сукупність синтаксичних правил утворює синтаксис мови програмування. Одно з синтаксичних правил - правило описання заголовку програми - наведено вище.

Також , як і лексеми, інші конструкції мови інтерпретуються як дії або описання. Наприклад, оператор присвоювання x := x+2 має смисл "скласти значення змінної x з числом 2 і результат інтерпретирувати як (нове) значення цієї ж змінної". Сукупність інтерпретацій синтаксичних правил називається семантикою мови. Можна сказати, що вивчення мови програмування полягає у вивченні його синтаксису і семантики.

6. Синтаксичні діаграми як засіб визначення мови.

Правила побудови синтаксичних діаграм пояснимо на прикладі діаграми заголовку програми:



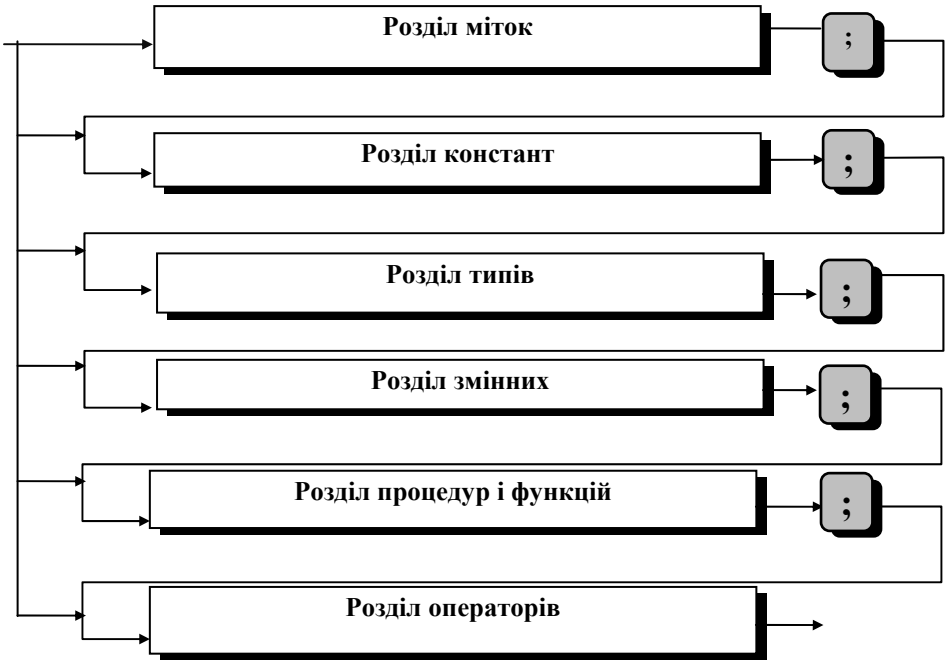
Мовна конструкція, яка визначається діаграмою, указана на початку діаграми (у прикладі це - заголовок програми). Власно діаграма має вид схеми, рух уздовж якої визначає синтаксично правильну мовну конструкцію.

В діаграмі виділені два види об'єктів: термінальні символи або ланцюжки (лексеми) і мовні конструкції, що визначаються іншими (зокрема, цією ж) діаграмами. Вони називаються нетермінальними об'єктами. Термінальні об'єкти обмежуються в овали, а нетермінальні - у прямокутники. Направлення руху уздовж діаграми (обходу) вказують стрілки, які з'єднують об'єкти.

Треба відмітити, що існують і інші метамови (мови описання мов), наприклад - мова нормальних форм Бекуса-Наура.

На завершення наведемо визначення блока мовою синтаксичних діаграм:

Блок



Таким чином, блок програми складається з шести розділів, кожний з яких, за винятком розділу операторів, може бути відсутнім. Розділи блоку відокремлені один від одного крапкою з комою.

ПРОСТІ ТИПИ ДАНИХ. ЛІНІЙНІ ПРОГРАМИ.

Лінійними (нерозгалуженими) називають програми, в яких відсутні обчислення, зв'язані з перевіркою деякої умови і вибором того чи іншого продовження обчислень у залежності від значення цієї умови (розгалуження). Будь-яка складна програма містить розгалуження. Разом з тим будь-яка програма містить лінійні фрагменти.

Для того, щоб складати найпростіші лінійні програми, необхідно вивчити поняття заголовку програми, розділу констант, розділу змінних, розділу операторів.

1. Заголовок програми.

Синтаксична діаграма заголовку програми зображена на сторінці 15.
У стандарті мови використовуються стандартні файли з іменами:
Input - вхідний стандартний файл (ім'я стандартного пристрою введення).
Output - вихідний стандартний файл (ім'я стандартного пристрою виведення).

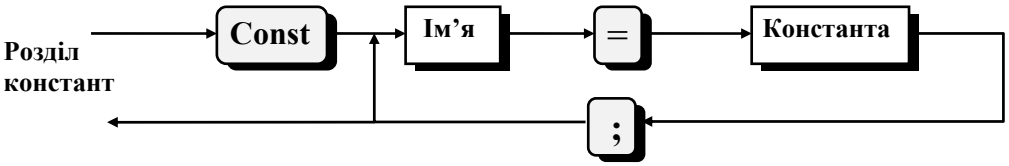
Приклади заголовка :

program LinearUnequation (Input, Output);
program GrafTrans;

Однак практично у всіх реалізаціях мови на ПЕОМ ці імена можна опускати, оскільки вони визначені за замовченням (клавіатура і алфавітно-цифровий екран користувача). Тому далі використовуються заголовки програм без цих параметрів.

2. Константи і їх використання. Розділ констант.

Розділ констант визначається наступною діаграмою:



В розділі констант визначаються імена як синоніми констант. Під константою розуміється або деяке число, або ім'я константи, можливо з знаком, або рядок.

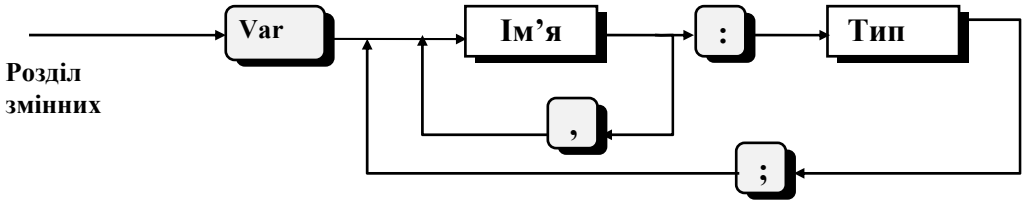
Приклад розділу констант:

```
const Pi = 3.1415926; alfa = 7.1219;
MinInt = -MaxInt;
Line = '_____';
FirstLine = '_____ Список групи _____';
```

Використання імен констант робить програму більш зрозумілою. Крім цього, програміст може згрупувати константи - параметри програми у розділі констант: тут вони легше піддаються контролю і зміні.

3. Змінні програми. Розділ змінних.

Розділ змінних визначений діаграмою:



Змінні використовуються в програмі для позначення даних. На відміну від констант, значення змінних можуть змінюватись у процесі виконання програми.

Будь-яка змінна, яка зустрічається в якому-небудь операторі з розділу операторів повинна бути описана в розділі змінних. Опис змінної зв'язує з новою змінною її ім'я і тип. Інформація, яка знаходиться в розділі змінних, використовується компілятором для:

1.Розподілу пам'яті. Розподіл (резервування) пам'яті для змінних, що описані в розділі змінних, робить компілятор на етапі генерації коду. Для кожної змінної в ОЗП відводиться певне місце. Розмір цієї частини пам'яті визначається типом змінної.

2.Правильної інтерпретації дій над даними. Наприклад, складання цілих чисел інтерпретується не так, як складання дійсних чисел або рядків.

3.Контролю правильності використання змінних. Помилка, яка допущена при написанні змінної в розділі операторів, приведе до повідомлення про синтаксичну помилку, так як ця змінна не описана в розділі змінних.

Приклад розділу змінних:

```
var     Root1, Root2, Discriminant : Real;
        Index, Counter : Integer;
        A,B,C : Real;
        Letter : Char;
        IsSolution : Boolean;
```

4. Стандартні прості типи даних

У мові Паскаль визначені 4 стандартних простих даних:
Integer (цілий);
Real (дійсний);
Char (символьний).
Boolean (логічний);

Для повного опису кожного типу даних, які використовуються в мові програмування, необхідно знати:

- множину допустимих значень для даних цього типу;
- допустимі операції над даними цього типу;
- функції, що визначені на даних цього типу або приймають значення в цьому типі;
- допустимі відношення на даних цього типу.

Тип даних Integer .

Значеннями цілого типу Integer є елементи підмножини (відрізка) цілих чисел, яка залежить від реалізації. Це означає, що існує стандартна константа з ім'ям MaxInt, така що для будь-якого даного X типу Integer

MaxInt < X < MaxInt

Найбільш поширене для 16 розрядних ПЕОМ значення MaxInt = 2^{15} - 1 = 32767.

Операції:

* - множення; div - цілочисельне ділення; mod - остача від цілочисельного ділення;

+ -додавання; - -віднімання;

Функції:

Abs(x) - | x |;
Sqr(x) - x^2;
Trunc(x) - відкидання дробової частини від дійсного x;
Round(x) - округлення дійсного x;
Succ(x) - x + 1;
Pred(x) - x - 1;

З деякими іншими функціями ми познайомимось пізніше - при визначення інших типів даних.

Відношення:

< - менше <= - менше або дорівнює
> - більше >= - більше або дорівнює
= - дорівнює <> - не дорівнює

Тип даних Real.

Значеннями дійсного типу є елементи підмножини дійсних чисел, яка залежить від реалізації. В TP-6 діапазон типу Real [$2.9*10^{-39}$ ... $1.7*10^{38}$]

Операції:

* -множення; / - ділення;
\+ - додавання; - - віднімання;

Функції:

Abs(x) - модуль x;
Sqr(x) - x у квадраті;
Sqrt(x) - корінь з x.
Sin(x) - sin x;
Cos(x)- cos x;
Arctan(x)- arctg x;
Ln(x) - ln x;
Exp(x) - e^x;
Відношення: такі самі, як і для типу Integer.

Числові типи Integer і Real сумісні. Це означає, що дані типу Integer можуть оброблятися, як дійсні числа і результат буде мати тип Real.

Тип даних Char.

Значеннями символьного типу є елементи скінченої і упорядкованої множини символів. Символи цієї множини визначаються реалізацією. Вони повинні підтримуватись на пристроями введення-виведення. Більшість ПЕОМ підтримує ASCII - код, тому в реалізації мови множина значень типу Char співпадає з деяким розширенням ASCII - символів. ASCII код кожному символу, що визначений в ньому, ставить у відповідність один байт.

Незалежно від реалізації множина символів включає:
A, B, C, ... , Z, _ (знак підкреслення)
1, ... , 9 - (десяткові цифри)
Символ “пробіл”.

Функції:

Ord(x) - порядковий номер x.
Chr(n)- символ з порядковим номером N.
Pred(x)- символ, який передує x.
Succ(x) - символ, який слідує за x.

Відношення.

Як вже повідомлялось, тип даних Char упорядкований. Це означає, що дані типу Char можна порівнювати, як і дані числових типів, за допомогою відношень:

= , <> , > , < , >= , <= .

Порядок на множені букв латинського алфавіту погоджений з алфавітним, а на множині цифр - з числовим.

Логічний тип даних Boolean буде описаний нижче - коли ми будемо вивчати поняття умови.

5. Поняття виразу. Значення виразу. Тип виразу.

Вирази складаються з змінних, констант, функцій, знаків операцій у відповідності з загально-прийнятими математичними правилами. Точне поняття виразу у мові може бути визначено за допомогою синтаксичних діаграм.

Вираз задає порядок обчислення його значення, оснований на загальноприйнятих правилах. Ці правила визначають семантику виразу за допомогою поняття старшинства (пріоритету) операцій. Найбільший пріоритет мають функції і логічна операція not, далі слідують мультиплікативні операції, адитивні операції і відношення. Операції, які мають більший пріоритет, виконуються раніше, ніж операції з меншим пріоритетом.

Таблиця пріоритетів.

Функції, not.

Мультиплікативні операції: * , / ,div , mod , and

Аддитивні операції: : + , - , or

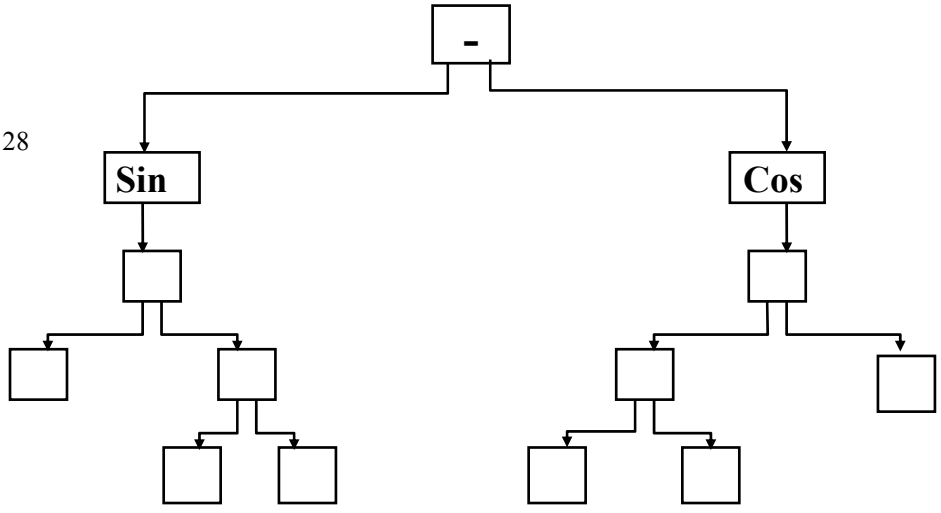
Відношення: = , <> , > , < , >= , <= , in

Операції одного пріоритету обчислюються зліва направо. Це відповідає групуванню дужок у бездужковому виразі уліво.

a + b + c = (a + b) + c, a * b * c = (a * b) * c

Вирази, що стоять у дужках, обчислюються незалежно один від одного.

Важливо розуміти, що в ході обчислення значення виразу кожний проміжний результат - дане деякого типу, точно визначеного знаком операції або функції і типами операндів. Будь-яка невідповідність типу значення операнда приведе до помилки, яка виявляється компілятором при синтаксичному аналізі. Наочне уявлення про структуру виразу дає так зване дерево виразу. Наприклад, вираз sin(x+pi/2) - cos(2*y-pi) може бути представлений у виді дерева:

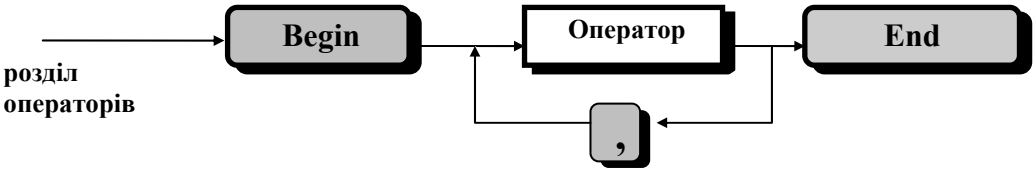


Обчислення значення виразу здійснюється у відповідності з рухом по гілках від листів до кореня - знизу уверх.

6. Розділ операторів. Оператор присвоювання.

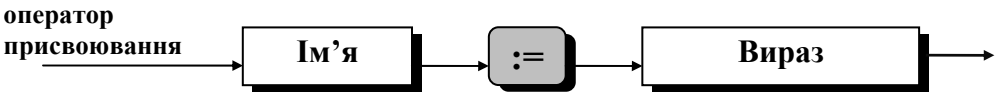
Дії, що роблять над даними, описуються у розділі операторів.

Синтаксична діаграма розділу операторів має вид:



Самим основним, фундаментальним оператором мови є оператор присвоювання. За допомогою оператора присвоювання здійснюється перетворення інформації.

Він має вид: **< ім'я > := < вираз >**



Ім'я ліворуч від символу присвоювання := є ім'я змінної, якій присвоюється значення виразу, який стоїть праворуч. Тому поряд з значенням виразу важливим атрибутом є його тип. Тип виразу у правій частині оператора присвоювання повинен співпадати або бути сумісним з типом змінної з лівої частини. Компілятор на етапі синтаксичного аналізу програми здійснює цю перевірку - так званий контроль типів. Допустиме присвоювання змінних будь-яких типів, за винятком типу File.

Root1 := Pi*(x - y)
Solution := Discriminant >=0
Discriminant := Sqrt(b*b-4*a*c)/2/A
Index := Index + 1
Letter := Succ(Succ(Letter))

7. Оператори введення - виведення.

Для організації введення - виведення даних у мові Pascal використовуються оператори - процедури Write, Read, Writeln, Readln. За допомогою цих операторів організується введення - виведення даних в/з файли Input ,Output.

Текстові файли Input ,Output представляються користувачу як текст, що поділений на рядки і забезпечений ознакою кінця тексту і ознаками кінців рядків. Кожний рядок може містити числа або символьні дані (тобто рядок складається з декількох даних типів Integer, Real, Char). Читання / запис здійснюється через т.н. буфер файла. В момент звернення до файла його буфер встановлений на деяке дане - елемент файла. Буфер файла може бути переміщений або до наступного даного, або до першого даного наступного рядка.

Оператор Read(x) читає дане з Input у змінну x і переміщує буфер до наступного даного.

Оператор Write(x) переміщує буфер у наступну позицію і пише дане в Output з змінної x.

Оператор Readln(x) читає дане з нового рядка з файла Input у змінну x.

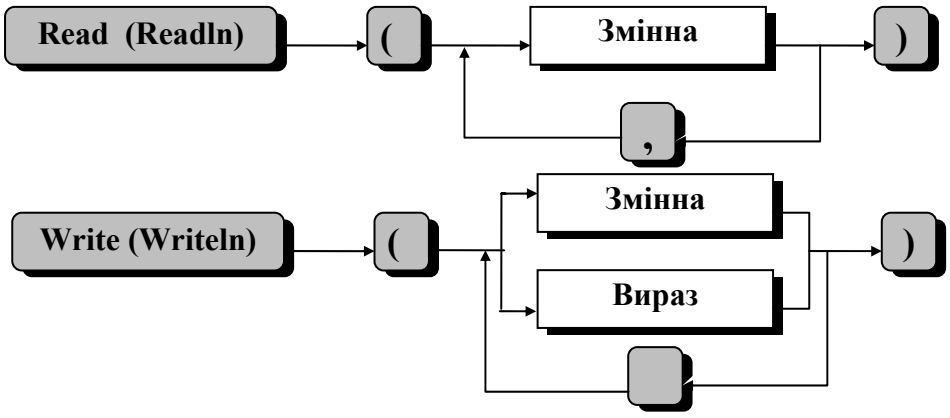
Оператор Writeln(x) пише дане з нового рядка в Output з змінної x.

Оператори введення/виведення можуть використовуватись у більш загальній формі:

Read(<список змінних>), Readln(<список змінних>)

Write(<список виразів або рядків>) Writeln(<список виразів або рядків>)

Ця форма визначається синтаксичними діаграмами:



Тепер ми володіємо знаннями, яких достатньо для написання лінійних (які не розгалужуються) програм.

8. Приклад лінійної програми.

Приклад 1. Програма обчислення площі кола, вписаного у трикутник і площі кола, описаного навколо трикутника.

```
Program Triangle; {програма обчислює площі, вписаного в трикутник і описаного навколо трикутника кіл}
Const  Pi = 3.1415926;
        Line = '_____';
Var a, b, c : Real;
     Sint, Sout : Real;
     S, p : Real;
Begin
 { введення даних}
  Write(' введіть сторони трикутника a b c ');
  Readln(a, b, c);
  {обчислення }
  p := (a + b + c)/2;
  S := Sqrt(p*(p - a)*(p - b)*(p - c));
  Sout := Pi*sqrt(a*b*c/4/S);
  Sint := Pi*sqrt(2*S/p);
  {друк відповіді}
  Writeln; Writeln (Line);
  Writeln('Sвпис = ', Sout); Writeln(Line);
  Writeln('Sопис = ', Sint); Writeln(Line)
End .
```

Оператори Readln і Writeln можуть застосовуватись без параметрів (тобто без тексту у дужках разом з дужками), для переходу до введення/ виведення з наступного рядка або для пропускання рядка.

Для оформлення введення в операторах Write, Writeln можна використовувати рядок. Наприклад: writeln('***', ' площа уписаного кола = ', sint, '***'). Текст програми може містити коментарії. Коментарій - це деякий текст, який стоїть у фігурних дужках. Коментарії

використовують для пояснень, необхідних для кращого розуміння програми. Добре прокоментована програма - признак кваліфікації і сумлінності програміста.

Особливу увагу треба звертати на проектування введення/ виведення. У професіональних програмах це - одна з найбільш важливих проблем. Тут треба виділити наступні аспекти:

- Захист програми від помилок користувача при введенні даних;
- Орієнтація на користувача, який неознайомлений з програмою - введення/ виведення в режимі дружелюбного діалогу.

Обміркування цих аспектів, однак, виходить за рамки цієї книги.

9. Поняття складності виразу. Оптимізація обчислень.

Основний критерій якості програми - її правильність. Строге математичне поняття правильності програми виходить за рамки нашого розгляду. Припустимо, що наші програми виконують саме ті дії, які ми від них чекаємо.

У цьому припущенні критеріями якості програми є, наприклад, її розмір, об'єм пам'яті, що відведений під дані, швидкість виконання, і т.п.

Швидкість виконання програми, що не розгалужується, визначається кількістю обчислень, які необхідні для отримання значень виразів, що містяться в операторах присвоювання і операторах виведення.

Кількість обчислень, необхідних для отримання значення виразу, будемо називати його складністю. Розглянемо приклад:

U := X*Cos(Alfa) + Y*Sin(Alfa) (1)

V := -X*Sin(Alfa) + Y*Cos(Alfa) (2)

Для обчислення значення U необхідно виконати два обчислення значення тригонометричної функції, два множення й одне додавання. Це і складність правої частини оператора (1). Права частина оператора (2) має ту саму складність, що і оператора (1). Зрозуміло, швидкість обчислень значень елементарних функцій, операцій множення і додавання різні і залежать від реалізації мови програмування. Однак реально припускати, що швидкості обчислень всіх елементарних трансцендентних функцій тотожні між собою, швидкості виконання мультиплікативних операцій тотожні, і швидкості виконання адитивних операцій також тотожні.

Хай T_f, T_m і T_a - час обчислення відповідно функцій, множення і додавання, а T(U) - час обчислення значення U. Тоді

$T(U) = 2T_f + 2T_m + T_a$, T(V) = T(U) (3)

Час обчислення значення виразу і є міра його складності. Зрозуміло, що програміст повинен турбуватись про зменшення складності виразів, що входить в програму. Для цього використовують:

а)Тотожні перетворення, що зменшують складність;

б) Попередні обчислення загальних підвиразів;

Приклади:

U := 4*Sin(X)*Cos(X) + 3 <==> U := 2*Sin(2*X) + 3
U := Sin(X)^2 - 3*Sin(X) + 2 <==> U := (Sin(X) - 2)*(Sin(X) - 1) <==>
Y := Sin(X); U := (Y - 2)*(Y - 1)

Між величинами T_f, T_m і T_a існують співвідношення

$$T_f >> T_m > T_a \quad (4)$$

які при програмуванні обчислень не можна ігнорувати. Тому особливу увагу треба приділяти функціональній і мультиплікативній складності за рахунок адитивної.

Наступний приклад по суті визначає оптимальну форму запису полінома для задачі обчислення його значення.

Приклад 2 (Схема Горнера)

Обчислити значення $Y = a_0x^3 + a_1x^2 + a_2x + a_3$;

Оскільки операції піднесення до степеня у мові немає, можна замінити його множеннями:

$Y = a_0*x*x*x + a_1*x*x + a_2*x + a_3$;

У цьому варіанті $T(Y) = 6T_m + 3T_a$;

Якщо виносити x за дужки там, де це можливо, отримаємо:

$Y = ((a_0x + a_1)x + a_2)x + a_3$;

$T(Y) = 3T_m + 3T_a$ {Схема Горнера}.

Можна показати, що схема Горнера обчислення багаточлена є оптимальною.

10. Оптимізація лінійних програм.

Задача зменшення складності програми, що містить декілька виразів, носить більш складний характер. Тут оптимізації підлягають одночасно декілька виразів, які обчислюються послідовно.

Приклад 3. Програма обчислення координат вектора, повернутого на кут Alfa.

```
Program Vector;
Var X, Y : Real;
Alfa : Real;
U, V : Real;
Begin
{ Введення X, Y, Alfa }
U := X*Cos(Alfa) + Y*Sin(Alfa);
V := -X*Sin(Alfa) + Y*Cos(Alfa);
{ Виведення U, V }
End.
```

У цьому варіанті складність програми T(U, V) є $T(U, V) = 4T_f + 4T_m + 2T_a$

Здійснимо попереднє обчислення функції Sin, Cos:

```
{ Описати допоміжні змінні Fsin, Fcos }
Begin
{ Введення X, Y, Alfa }
Fsin := Sin(Alfa); Fcos := Cos(Alfa);
U := X*Fcos + Y*Fsin;
V := -X*FSin + Y*Fcos;
{ ВиведенняU, V }
End.
```

Отримаємо: $T(U, V) = 2T_f + 4T_m + 2T_a$

В результаті перетворення складність зменшилась приблизно вдвічі. Можна ще зменшити мультиплікативну складність, якщо обчислити U і V наступним способом:

```
A := (Fcos + Fsin)*(X + Y);
B := X*Fsin; C := Y*Fcos;
U := A - B - C;
V := C - B;
```

Тоді $T(U, V) = 2T_f + 3T_m + 5T_a$

Рішення питання про те, який з двох варіантів перетворень програми краще по швидкодії, залежить від реалізації множення у комп'ютері.

Приклад показує, що прийоми оптимізації оператора присвоювання ще в більшій степені застосовувані для оптимізації лінійних програм, що складається з декількох операторів. Менш очевидним є прийом оптимізації, що заключається в використанні співвідношень між виразами - правими частинами операторів. Розглянемо його на наступному прикладі:

Приклад 4. Відомо, що рівняння $x^2 - px + q$ має два рішення. Знайти їх.

Варіант 1:

```
Discriminant := Sqrt(p*p - 4*q);
Root1 := ( p + Discriminant )/2;
Root2 := ( p - Discriminant )/2;
```

Варіант 2:

```
Discriminant := Sqrt(p*p - 4*q);
Root1 := (p + Discriminant)/2;
Root2 := p - Root1;
```

Обчислення у другому варіанті містять на одну ділення менше. Тут оптимізація досягнута завдяки наявністю співвідношення між Root1 і Root2: Root1+ Root2 = p.

Відмітимо на закінчення, що розглянуті оптимізаційні прийоми мають смисл застосовувати тоді, коли обчислення повторюються в програмі достатньо часто і час виконання програми - критичний параметр.

11. Задачі і вправи.

Скласти програму розв'язку задачі:

1. Знайти гіпотенузу, площа і гострі кути прямокутного трикутника, заданого катетами.

2. Змішано V_1 літрів води температурою t_1 з V_2 літрами води температури t_2. Знайти об'єм V і температуру t утвореної суміши.

3. Знайти радіус кола з центром в (X_0,Y_0), дотичною до якого є пряма y = kx + b.

4.Обчислити центр ваги системи з трьох матеріальних точок на площині з масами M_1, M_2, M_3 і координатами (X_1,Y_1), (X_2,Y_2), (X_3,Y_3).

5.Розв'язати систему лінійних рівнянь методом Крамера.

$$\begin{cases} a_{11}x_1 + a_{12}x_2 = b_1 \\ a_{21}x_1 + a_{22}x_2 = b_2 \end{cases}$$

{Вважати, що її визначник не дорівнює нулю }

6.Обчислити координати точки A(X,Y) при повороті системи координат на кут Alfa і паралельному переносі на вектор a = (u, v).

7.Знайти корінь степені n і n-ту степінь позитивного дійсного числа a.

8.Обчислити цілі коефіцієнти A, B, C квадратного рівняння по його раціональним кореням $x_1 = n_1 / m_1$, $x_2 = n_2 / m_2$.

9. Обчислити внутрішні кути трикутника, заданого довжинами сторін.

10.Перерахувати координати точки з полярної системи в декартову систему координат.

11.Перерахувати координати точки з декартової системи в полярну систему координат.

12. Розрахувати координати матеріальної точки, пущеної з початковою швидкістю V_0 під кутом Alfa до горизонту в напрямі вектора a = (X_0,Y_0) в момент часу t.

13. Обчислити суму, добуток і частку двох комплексних чисел $z_1 = a+bi$, $z_2 = c+di$.

14. Багаточлени F(x) = ax + b і G(x) = cx + d задані своїми коефіцієнтами. Знайти коефіцієнти багаточлена H(x) = F(x)*G(x).

15.Багаточлени F(x) = ax + b і G(x) = cx + d задані своїми коефіцієнтами. Знайти коефіцієнти багаточленів $H_1(x) = F(G(x))$ і $H_2(x) = G(F(x))$.

Задачі

1.Знайти розв'язок вправи 13, яке використовує 3 множення.

2.Знайти розв'язок вправи 14, яке використовує 3 множення.

3.Знайти розв'язок приклада 3, який використовує лише одну операцію обчислення тригонометричної функції.

4.Використовуючи розв'язок задачі 2, знайти схему множення двох квадратних трьохчленів, який використовує 6 множень.

ПРОГРАМИ, ЩО РОЗГАЛУЖУЮТЬСЯ.

1. Поняття умови. Тип даних Boolean (логічний).

Умови використовуються в програмах для організації розгалужень і дій, що повторюються. Умовою в мові є логічний вираз - вираз типу Boolean. Булевські значення - це логічні істинні значення: : True (істина) і False (хибність).

Цей тип даних, як і інші прості типи даних, упорядкований. На ньому визначені функції Ord, Succ, Pred.

Таким чином, мають місце наступні співвідношення:

False < True ,

Ord (False)=0, Ord (True)=1,

Succ (False)=True, Pred (True)=False.

На множені < True, False > визначені логічні операції.

Операції:

And - логічна кон'юнкція (і)

Or - логічна диз'юнкція (або)

Not - логічне заперечення (ні)

Ці операції визначаються наступними таблицями істинності:

And	False	True
False	False	False
True	False	True

Or	False	True
False	False	True
True	True	True

x	Not x
False	True
True	False

Відношення, що були визначені раніше для простих стандартних типів є операціями, результат яких має логічний тип. Іншими словами, булевське значення дає будь-яка з операцій відношень : =, < > , <= , < , > , >= , in.

Для типу Boolean визначені стандартні функції, які приймають значення цього типу (логічні значення):

Odd(X)

{ Odd(X) = True, якщо X - ціле непарне число
Odd(X) = False, якщо X - ціле парне число}

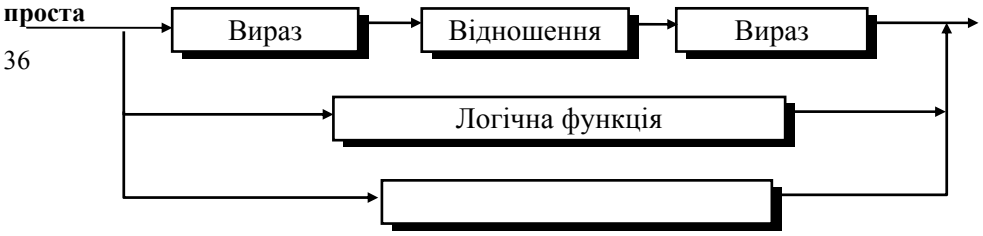
Eoln(F) { кінець рядка в текстовому файлі}

Eof(F) { кінець файла}

Функції Eoln(F) і Eof(F)

Умови можна класифікувати як прості і складні.

Прості умови визначені діаграмою:



умова

Складні умови конструюються з простих за допомогою логічних операцій.

Приведемо приклади простих і складних виразів типу Boolean (умов).

Прості вирази типу Boolean (умови):

Sin(2*x) > S, (X + Y) mod Prime = 0,

b*b > 4*a*c ,

Number div Modulo = 2, Odd(A*P + B),

Flag ;

Складні вирази типу Boolean (умови) :

а) (a + i > в) or (x [Index] = c)

{Тут a, в, c - змінні типу Real, x - масив дійсних чисел , Index - змінна типу Integer }

б) Odd (n) And (n < 10e4)

в) Eof(f) Or (f^.data = 0) {f - файл дійсних чисел}

г) Not(beta) And (gamma) {beta і gamma - змінні типу Boolean}

д) (A > B) = (C > D)

Логічні вирази перетворюються за законами логіки висловлювань. Наприклад,

Not((A > 0) And (B <> 0)) <==> Not(A > 0) Or Not(B <> 0) <==> (A <= 0) Or (B = 0)

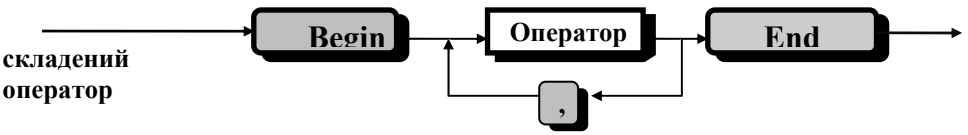
Перетворення логічних виразів часто приводять до зменшення їх складності і, тим самим, оптимізації програми за часом.

2. Складений оператор.

Декілька операторів, що виконуються послідовно, можна об'єднувати в один складений оператор.

Складений оператор передбачає виконання операторів, які в нього входять - компонент у порядку їх написання. Службові слова Begin і End грають роль дужок операторів - вони виділяють тіло складного оператора.

Складений оператор визначається діаграмою:



Зверніть увагу на те, що розділ програми представлений як один складений оператор.

Приклади складених операторів:

а)

```
Begin
 Write(‘ Введіть координати вектора: ‘);
 Readln(a, b, c);
 Length := sqrt(a*a + b*b+ c*c);
 Write(‘ довжина (a,b,c) дорівнює ‘, Length)
end
```

б)

```
Begin
 u := u*x/n;
 s := s+u
End
```

в) Begin writeln (‘ рівняння коренів не має ’) End

3. Оператори вибору: умовний оператор.

Оператори вибору призначені для виділення операторів, які їх представляють - компонент одного - єдиного, який і виконується. Таким чином, оператори вибору реалізують управляючу структуру “розгалуження”. У якості оператори вибору в мові визначені умовний оператор і оператор варіанта.

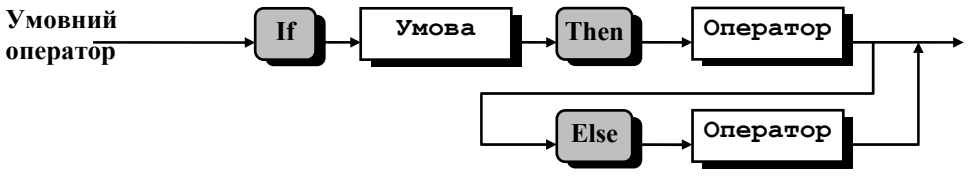
Існують дві форми умовного оператора:

If < умова > then < оператор >

If < умова > then < оператор > else < оператор >

Вони відповідають базовим управляючим структурам короткого і повного розгалуження. Умова - це вираз типу Boolean.

Синтаксична діаграма умовного оператора має вид:



Приклади умовних операторів:

а) If a >= b then Max := a else Max := b

б) If IntFun(i) mod 3 = 0 then write(i)

в) If (a11*a22 = a12*a21) And
((a11*b2 <> a12*b1) Or
(b1*a22 <> b2*a21))
then Write(‘ система розв’язків не має ’)
else Write(‘ система має розв’язки ’)

г) If x <= 0
then begin
u :=x*x - 2*x + 3;
v :=1/2*x + 1
end
else begin
u :=1/3*x+2;
v :=x*x+3*x-2
end

Зверніть увагу на те, що в тілі умовного оператора може використовуватись і інший умовний оператор. Це створює можливість реалізовувати багатозначне розгалуження. Наприклад:

```
If Discriminant < 0
  then If LeadCoef < 0
    then Write(‘ Розв’язків немає ’)
     else Write(‘ Розв’язки - вся числова вісь ’)
   else If LeadCoef < 0
     then Write(‘ Розв’язки - між коренями рівняння ’)
      else Write(‘ Розв’язки - поза коренями рівняння ’)
```

Відмітьте однак, що наступна конструкція мови, що складається з вкладених розгалужень синтаксично двозначна (допускає два різних варіанта синтаксичного аналізу).

If <умова 1> then If <умова 2> then <оператор 1> else <оператор 2>

1 варіант:

```
If < умова 1>
 then begin
          I f < умова 2> then <оператор 1> else <оператор 2>
         end
```

2 варіант:

```
If< умова 1>
 then begin
  If < умова 2> then <оператор 1>
 end
else <оператор 2>
```

Для усунення двозначності в мові обрано 1-ий варіант інтерпретації у відповідності з правилом: роздільнику else відповідає найближчий попередній роздільник then.

4. Програми, що розгалужуються. Приклад.

Програмами, що розгалужуються, називаються програми, в яких використовуються оператори розгалуження.

Приклад 1.

```
Program SquareEquation;
 Const Line = '----------------------------';
 Var a, b, c, Root1, Root2: real;
 Solution: Integer;
 Discriminant: Real;
 Begin
  {Введення даних}
  Write(' Введіть коефіцієнти рівняння (через пробіл):');
  Readln(a, b, c) ; Writeln(Line);
            {Обчислення}
 Discriminant := Sqr(b) - 4*a*c;
 If Discriminant < 0
  then Solution := 0 { Немає коренів }
  else If Discriminant = 0 { Один корінь }
    then begin
      Solution := 1;
      Root1 := -b/a;
      Writeln ('x1= ' ,Root1)
    end
   else { Два кореня } begin
     Solution := 2;
     Root1 := (-b + Sqrt(Discriminant))/(2*a);
     Root2 := -b/a - Root1;
     Writeln('x1= ', Root1, ' x2= ', Root2)
   end ;
     Writeln(Line);
     Writeln(' Кількість розв'язків дорівнює: ', Solution)
End.
```

5. Оптимізація програм, що розгалужуються за часом.

Складність програми, що розгалужується, визначається складністю умови і складністю обчислень гілок. На відміну від програми, що не розгалужується, час виконання

тут залежить від гілки, якою слідує процес виконання. Тому для таких програм має смисл поняття складності програми в гіршому випадку і складності програми в середньому.

Хай $T_п$ - складність програми за часом в гіршому випадку, $T_у$ - складність за часом умови і $T_{в1}$, $T_{в2}$ - складності гілок за часом програми. Тоді має місце співвідношення:

$T_п = T_у + Max(T_{в1}, T_{в2})$ (1)

Складність за часом у середньому визначається формулою

$T_п = T_у + P_у T_{в1} + (1 - P_у) T_{в2}$ (2)

де $P_у$ - ймовірність виконання умови.

Оскільки умовою є логічний вираз, загальні прийоми оптимізації виразів застосовуються і для логічних виразів. Час Тл виконання логічних операцій And, Or, Not, =, <> значно менше часу виконання аддитивних операцій, а час виконання операцій <, >, <=, >= дорівнює часу виконання аддитивних операцій.

$T_f >> T_m > T_a > T_л$ (3)

Розглянемо приклад: потрібно з'ясувати, являється чи дорівнює нулю одне з двох чисел A, B .

1 варіант умови: A * B = 0

2 варіант умови: (A = 0) Or (B = 0)

У першому варіанті використані множення і порівняння, у другому - 3 логічних операції. Складність 2-го варіанта менша.

У програмах з багатозначним розгалуженням, коли в управлінні використовується декілька умов або обчислень логічних виразів, існує можливість оптимізації управління обчисленнями. Наприклад, наступні обчислення еквівалентні:

```
If x < 0 then                    <==>   If (x < 0) And (Y < 0)
If y < 0 then z := 1                      then z := 1
If x < 0
then Flag := False               <==>   Flag := (x >= 0)
else Flag := True
```

У наступному прикладі умови розгалужень спрощені за рахунок використання співвідношень, що виконуються після обчислення попередньої умови:

Приклад 2. Програма обчислення значення кусково-визначеної функції:

$$y = \begin{cases} 2x - 1 & при\ x < -1 \\ x^2 + 1 & при\ -1 <= x < 1 \\ 2x + 1 & при\ x >= 1 \end{cases}$$

1 варіант (який часто зустрічається у починаючих)

```
If x < -1 then y := 2*x - 1;
If (-1 <= x)And(x < 1) then y := Sqr(x) + 1;
If x >= 1 then y := 2*x + 1;
```

2 варіант (оптимальний)

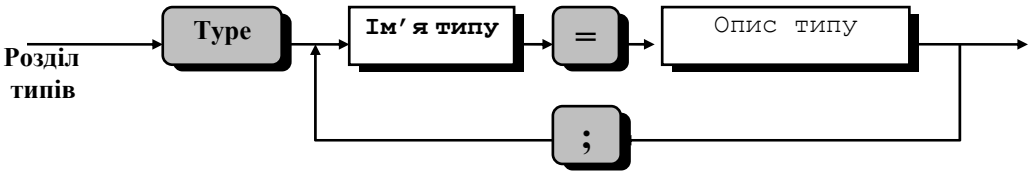
```
If x < -1 then y := 2*x - 1
  else If x < 1 then y := Sqr(x) + 1
  else y := 2*x + 1;
```

Для отримання 2-го варіанта відмітимо, що умови < -1, (-1 <= x)And(x < 1), x >= 1 взаємно протилежні і у сукупності тотожно істинні. Тому (-1 <= x) і x >= 1 можна замінити на else.

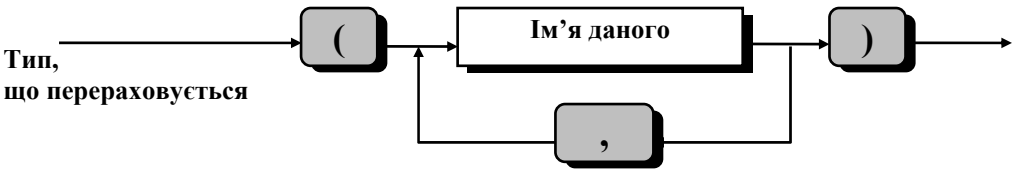
6. Тип, що перераховується. Розділ типів.

Поряд з стандартними типами даних у мові Pascal широко використовуються типи, що визначаються програмістом. Один з таких типів - це тип, що перераховується. Визначення цього типу задає упорядковану множину значень шляхом перерахування імен, що позначають ці значення.

Типи даних, що визначає програміст, описуються у спеціальному розділі - розділі типів. Розділ типів визначений синтаксичною діаграмою:



Тип даних, що перераховується, визначається наступною діаграмою:



Приклади визначень типів, що перераховуються:

а) Type Weekday = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);

Colour = (Red, Orange, Yellow, Green, Blue, Black);

Operation = (Plus, Minus, Times, Divide)

Відмітимо, що стандартний тип Boolean, якщо б його треба було описати, виглядав би як : type Boolean = (False, True);

Для аргументів типу, що перераховується, визначені такі стандартні функції:

Succ(x) - значення, наступне за x.

Pred(x) - значення, попереднє x.

Ord(x) - порядковий номер x.

До значення типу, що перераховується, можуть бути застосовані відношення:
=, <>, <, <=, >=, > .

Упорядкованість значень визначається порядком перераховування констант в описанні типу. Наприклад:

Red < Orange < Yellow < Green < Blue < Black ;

Описання типу змінної може бути дано і в розділі змінних. Наприклад, описання:
Type Figure = (Triangle, Circle, Rhombus, Square);

Var f: Figure; еквівалентне описанню Var f: (Triangle, Circle, Rhombus, Square); однак у другому випадку описання типу становиться анонімним: тип описаний, але не має імені. Використання цього типу обмежене. Тому 1-ий варіант більш відповідає стилю мови.

7.Оператори вибору: оператор варіанта

Оператор варіанта складається з виразу, який називається селектором, і списку операторів, кожний з яких відмічений константою того ж типу, що й селектор. Селектор повинен бути скалярного типу, але не дійсного.

Оператор варіанта обчислює значення селектора і вибирає для виконання оператор, одна з міток якого дорівнює цьому значенню. По закінченню виконання вибраного оператора управління передається на виконання наступного за оператором варіанта оператора.

Якщо значення селектора не співпадає ні з однією з міток, то вибирається оператор, помічений ключовим словом else. Цей оператор повинен бути останнім у списку варіантів. Якщо значення селектора не співпадає ні з однією з міток і else відсутнє, то оператор варіанта ігнорується.

Оператор варіанта має вид:

Case < вираз {селектор}> of <список міток варіанта > : < оператор >;

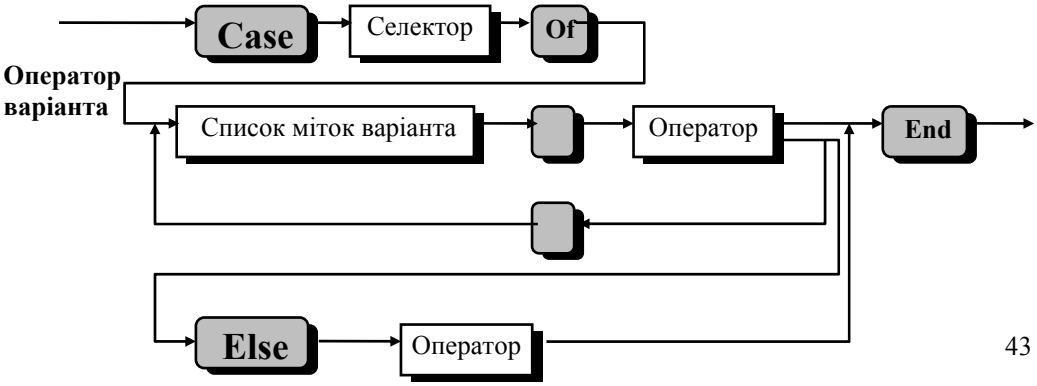
. . . . . .

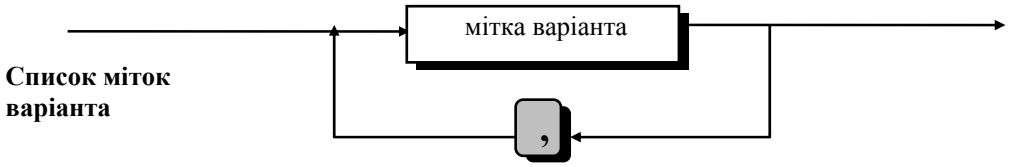
< список міток варіанта > : < оператор >

[else < оператор >]

end

Мовою синтаксичних діаграм це виглядає так:





Приклади операторів варіанта:

а) Select : = Index mod 4;
```
                    case Select of
                      0 : x := y*y + 1;
                      1 : x := y*y - 2*y;
                      2,3 : x := 0
                    end;
```
В цьому прикладі Select приймає значення 0, 1, 2, 3. Це досягнуто обчисленням Select := Index mod 4.

Таким чином, замість імені Select можна використовувати вираз Index mod 4:

а)
```
case Index mod 4 of
  0 : x := y*y + 1;
  1 : x := y*y - 2*y;
  2,3 : x := 0
end;
```
б)
```
case ch of
  ‘a’,’b’,’c’ : ch := succ(ch);
  ‘y’,’z’ : ch := pred(ch);
  ‘f’,’g’ : {порожній варіант};
else ch := pred(pred(ch)
end;
```
Програма в наступному прикладі обчислює знак однієї з тригонометричних функцій у залежності від квадранта декартової площини.

Пример 3.
```
program Sign_of_Function;
Type Fun = (Unknown, FSin, FCos, Ftg, Fctg);
 var FunNumber, Quoter: Integer;
```

```
TrigFun : Fun;
Begin
  Write(‘ Введіть номер тригонометричної функції ‘);
  Readln(FunNumber);
{ Обчислення імені функції }
Case FunNumber of
  1: TrigFun := FSin;
  2: TrigFun := FCos;
  3: TrigFun := FTg ;
  4: TrigFun := FCtg
else begin
  TrigFun := Unknown;
  Writeln(‘ Невідома функція ’)
end
end;
Write(‘ Введіть номер квадранта ‘); Readln(Quoter);
{ Обчислення знака функції }
case TrigFun of
FSin: case Quoter of
  1, 2: Writeln (‘ знак синуса +’);
  3, 4: Writeln (‘ знак синуса -‘)
  end;
  FCos: case Quoter of
1, 3: Writeln (‘ знак косинуса +’);
2, 4: Writeln (‘ знак косинуса  -‘)
end;
FTg, FCtg: case Quoter of
            1, 4: Writeln (‘ знак тангенса і котангенса +’);
  2, 3: Writeln (‘ знак тангенса і котангенса -‘) end;
Unknown: Writeln(‘ Функція не визначена ’)
end
end.
```

8. Вправи.

І. Сформулюйте умови для оператора розгалуження:

1.Білий кінь розташований на полі (x, n). Чорний пішак розташований на полі (y, m). Чи знаходиться пішак під боєм коня?

2.Білий слон розташований на полі (x, n).Чорний пішак розташований на полі (y, m). Інших фігур на полі немає. Чи знаходиться пішак під боєм слона?

3.Біла шашка розташована на полі (x, n). Чорна шашка розташована на полі (y, m). Чи знаходиться чорна шашка під боєм білої?

4.Біла дамка розташована на полі (x, n).Чорна шашка розташована на полі (y, m). Чи знаходиться чорна шашка під боєм білої дамки? (Інших фігур на дошці немає)

5.Вектор a = (x, y), вектор b = (u, v). Чи являються вектори паралельними або перпендикулярними?

6.Вектор a = (x, y), вектор b = (u, v). Чи можна повернути вектор a проти годинної стрілки на деякий кут, менший 180^0 так, щоб вектори стали співнаправленими?

7.Чи пересікаються коло O_1 з центром (x, y) і радіусом R_1 з колом O_2 з центром (u, v) і радіусом R_2?

8.Чи є трикутник з вершинами $A(x_1, y_1)$, $B(x_2, y_2)$, $C(x_3, y_3)$ рівнобедреним?

9.Чи можна з відрізків a, b, c скласти трикутник і чи можна цей трикутник помістити у коло радіуса R?

10.Пряма задана рівнянням Y = kX + b. Чи лежить точка A(u, v) над цією прямою?

II. Напишіть програму розв'язування задачі:

1.Розв'яжіть квадратну нерівність $ax^2 + bx + c < 0$.

2.Розв'яжіть систему лінійних рівнянь:

$$\begin{cases} a_{11}x + a_{12}y = b_1 \\ a_{21}x + a_{22}y = b_2 \end{cases}$$

3.Обчислити внутрішні кути трикутника, з вершинами $A(x_1, y_1)$, $B(x_2, y_2)$, $C(x_3, y_3)$

4.Знайти найкоротшу сторону трикутника з вершинами $A(x_1, y_1)$, $B(x_2, y_2)$, $C(x_3, y_3)$

5.Знайти максимум і мінімум з трьох цілих чисел.

6.Обчислити значення функції:

1, якщо x > 0

Sign(x) = 0, якщо x = 0

-1, якщо x < 0 .

7.Обчислити найбільше і найменше значення функції $Y=ax^2 +bx+c$ на відрізку [p ; q].

8.Знайти область визначення функції: $y=1/(x^2 + px + q)$.

9.Розв'язати біквадратне рівняння $ax^4 + bx^2 + c = 0$

10.Знайти квадрат найменшої площини з сторонами, паралельними осям координат, який містить три точки площини $A(x_1,y_1)$, $B(x_2,y_2)$, $C(x_3,y_3)$.

11.Здійснити переклад не більш чим трьохзначного цілого додатного числа у відповідний йому складений числівник українською, російською або англійською мовою.

12.За номером місяця і номером дня знайти день тижня, що припадає на цю дату.

ОПЕРАТОРИ ПОВТОРЕННЯ З ПАРАМЕТРОМ І МАСИВИ.

1. Оператор циклу з параметром.

Цикли - основний засіб у програмуванні, що дозволяє коротко записувати алгоритм, який здійснює велику кількість дій.

Для реалізації циклічних алгоритмів у мові Паскаль використовуються оператори повторення:

- оператор циклу з параметром;
- оператор циклу з передумовою;
- оператор циклу з постумовою.

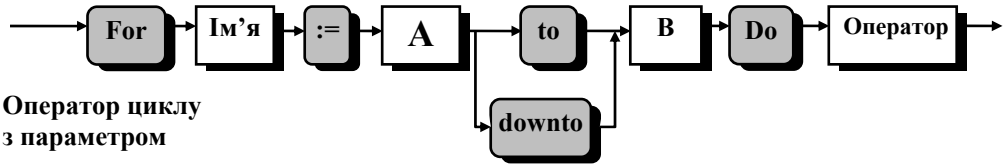
У цьому параграфі вивчається оператор циклу з параметром. Такий оператор передбачає повторне виконання деякого оператора з одночасною зміною значення, що присвоюється керуючою змінною (параметру цього циклу). Він має вид:

For < параметр > := <початкове значення > to <кінцеве значення > do <оператор >

або

For < параметр >:=< початкове значення > downto <кінцеве значення > do <оператор>

Синтаксична діаграма оператора циклу з параметром:



Оператор циклу з параметром

Тут

Ім'я - це ім'я змінної - параметра циклу;

A - початкове значення параметра циклу;

B - кінцеве значення параметра циклу;

Оператор - тіло циклу.

Параметр циклу, початкове і кінцеве значення повинні бути одного й того ж скалярного типу (крім дійсного). Початкове і кінцеве значення обчислюються лише один раз - при вході в цикл, і, отже, повинні бути визначені до входу в цикл і не можуть бути змінені в тілі циклу.

Якщо початкове і кінцеве значення розділяє службове слово to, то після виконання оператора (тіло циклу) параметр циклу v приймає значення Succ(v), якщо ж дільником початкового і кінцевого значень служить слово downto, то параметр циклу v після виконання тіла циклу приймає значення Pred(v). Зокрема, якщо параметр v має тип Integer, то одночасно з виконанням тіла циклу здійснюється або присвоювання v := v + 1 (to), або v := v - 1 (downto). Додатково (примусово) змінювати значення параметра в тілі циклу не рекомендується, оскільки контроль за правильністю виконання такого циклу дуже затруднений. Якщо в циклі з to початкове значення більше, ніж кінцеве, то цикл не виконується взагалі. Аналогічно, якщо в циклі з downto початкове значення менше, ніж кінцеве, цикл також не виконується.

Стандарт мови локалізує дію параметра тільки в самому циклі, тому при нормальному виході з циклу, тобто коли параметр циклу вже прийняв всі можливі значення між початковим і кінцевим значеннями, значення параметра вважається невизначеним (В реалізації мови це правило часто не виконується).

Приклад 1.

```
Program NFactorial; var Factorial, Argument: Integer; i : Integer;
  Begin
    Write(‘ введіть аргумент факторіала ‘);
     Readln(Argument) ;
     Factorial := 1;
     For i := 2 to Argument do Factorial := i*Factorial;
     Writeln(Argument,’! = ‘, Factorial)
  End.
```

У цьому прикладі
i - параметр циклу;
2 - початкове значення параметра циклу;
Argument - кінцеве значення параметра циклу;
Factorial := i*Factorial - тіло циклу.

Відмітимо, що на вході Argument > 7 значення Factorial вийде за межі типу Integer і результат буде неправильним.

В наступному прикладі обчислюються суми $\sum_{n=0}^{\infty} x^n/n^2$, $\sum_{n=0}^{\infty} x^n/n^3$

Приклад 2.

```
Program SeriesSumma;
  Var Summa1, Summa2, x : Real;
  u1, u2 : Real;
  v : Real;
  Index, n : Integer;
Begin
```

```
 Write(‘ введіть x і n ‘); Readln(x, n);   {Ініціалізація змінних, що використовуються в
    циклі}
 Summa1 := 1; Summa2 := 1;                { суми рядів }
  v := 1; { x^n }
  For Index := 1 to n do begin
    v := v*x;
    u1 := v/Sqr(Index);  u2 := u1/Index;
    Summa1 := Summa1 + u1;   Summa2 := Summa2 + u2;
   end;
  Writeln( ‘ S1 = ‘, Summa1,’ ‘,’ S2 = ‘, Summa2)
End.
```

Зверніть увагу на те, що всі оператори, які необхідно виконувати в циклі, об’єднані у складений оператор.

Часто крок зміни змінної, яка управляє циклом, відрізняється від 1, -1. Наступний приклад демонструє використання циклу з параметром у таких обчисленнях.

Приклад 3. Табулювання функції дійсної змінної.

```
Program Tabulation;
   const Pi=3.14159262;
   var MinBound, MaxBound, Step: Real;
   x, y : Real;   Coef : Real;
   i, n : Integer;
 Begin
  Write(‘Введіть межі табулювання ‘); Readln(MinBound, MaxBound);
  Write(‘Введіть крок табулювання ‘); Readln(Step);
  n := Round((MinBound - MaxBound)/Step);
  x := MinBound;   Coef := 1/Sqrt(Pi);
  for i := 0 to n do  begin
    y := Coef * exp(-Sqr(x)/2);
    writeln(‘ x = ‘,x,’ y = ‘,y);
    x := x + Step
  end;
End.
```

Програма табулює функцію $y=1/\pi\ e^{-x2/2}$ в інтервалі значень [MinBound, MaxBound] з кроком Step. Зверніть увагу на те, що перед входом у цикл обчислюється N - верхня межа параметра циклу, а в тілі циклу обчислюється наступне значення x.

Приклад 4.

```
program Alfabet;
  Var Letter : char;
```

```
Begin
 For Letter := 'z' downto 'a' do Write(Letter, ',')
End.
```

2. Циклічні програми. Складність циклічної програми. Оптимізація циклічних програм.

Циклічними називають програми, що містять оператори циклів. Циклічна програма може містити декілька операторів циклу, що виконуються послідовно або входять в інші оператори . Найпростіші циклічні програми містять один оператор циклу і оператори, що керують введенням-виведенням. До найпростіших відносяться циклічні програми розглянутих прикладів.

Складність $T_ц$ найпростішої циклічної програми, що містить арифметичний цикл, визначається формулою $T_ц = NT_b$ де N - кількість повторень циклу, T_b - складність тіла циклу. Якщо арифметичний цикл завершується нормально, то N = |Ord(MaxVal) - Ord(MinVal)|, де MaxVal, MinVal - нижня і верхня межі параметра. Таким чином, оптимізація програми за часом полягає у зменшенні складності тіла циклу і (якщо це можливо) зменшенні кількості повторень циклу.

Для оптимізації тіла циклу використовують наступні прийоми:

Попереднє (поза циклу) обчислення підвиразів, значення яких не змінюється в циклі. Наприклад, у програмі Tabulation до входу в цикл обчислено значення Coef=1/Sqrt(Pi);

Використання співвідношень, що зв'язують змінні, які змінюються в циклі, для обчислень однієї з них через інші. У програмі SeriesSumma змінні u1 і u2 - члени ряду - визначені як функції від x, i:

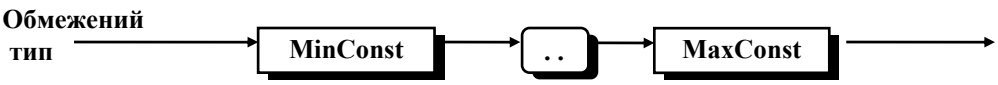
$$u1(x, i) = x^i / i^2; \quad u2(x, i) = x^i / i^3;$$

Тому має місце рівність u2*i = u1, яка використовується для обчислення u2.

3. Обмежені типи.

Обмежений тип у мові Паскаль можна визначити, накладаючи обмеження на вже визначений скалярний тип - його називають базовим скалярним типом. Обмеження визначаються діапазоном - мінімальним і максимальним значеннями констант базового скалярного типу. Обмеження стандартного типу Real не допускається.

Синтаксична діаграма обмеження має вид:



Наприклад:

а) Type Day = 1..30; - обмеження на тип integer;

б) Digit = ‘0’..’9’; - обмеження на тип char;

в) Type Weekday = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);

Workday = Monday .. Friday; - обмеження на скалярний тип Weekday.

Базовий скалярний тип визначає допустимість всіх операцій і відношень над значеннями обмеженого типу. Обмежені типи дають можливість описувати алгоритм у більш наочній формі. Крім цього, у процесі трансляції і виконання програми з’являється можливість економити пам’ять і здійснювати контроль присвоювань.

Приклад 5.

```
Program Time_Table;
  Type Weekday = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);
       Workday = Monday .. Friday;
Var i : Workday;
 Begin
   For i := Monday to Friday do
    Case i of
     Monday: Writeln(‘фізика‘, ’інформатика‘, ’історія’);
    Tuesday, Friday: Writeln(‘мат. аналіз‘,’педагогіка‘, ‘англійська’);
    Wednesday, Thursday: Writeln(‘фізика ‘,’алгебра‘,’інформатика’)
   end
End.
```

4. Складні (складені) типи.

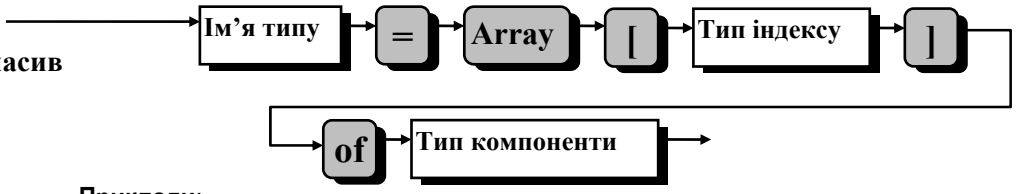
У мові Pascal реалізований механізм визначення складних (складених) типів даних. Новий тип даних визначається як структурована сукупність даних-компонент стандартних або раніше визначених типів. Оскільки типи компонент можуть також складеними, можна будувати складні ієрархії типів. Методи структурування даних у мові дозволяють будувати масиви, записи, множини і файли. Ці методи називають статичними, оскільки їх опис здійснюється попередньо. Більш складні структури можна створити динамічно - у процесі виконання програми - за допомогою засилок. При вивченні складних типів основна увага приділяється способам конструювання даного і способам доступу до компонентів даного.

5. Регулярний тип. Масиви.

Значеннями регулярного типу являються масиви. Масив - це найбільш поширена структура даних. У багатьох мовах програмування, що були одними з перших мов високого рівня, (Fortran, Algol-60, Basic) цей єдиний явно визначений складний тип.

Масив - це послідовність однотипних даних, що об'єднана загальним іменем, елементи (компоненти) якої відрізняються (ідентифіцируються) індексами. Індекс елемента вказує місце (номер) елемента в масиві. Кількість елементів масиву фіксовано і визначено в його описі. Доступ до елемента масиву здійснюється обчисленням значення його індексу. Тому масиви - це структури даних з прямим (випадковим) доступом. Всі компоненти масиву є однаково доступними. При визначенні регулярного типу задається і тип компонент, і тип індексів. Саме визначення має вид:

<ім'я типу > = Array [< тип індексу >] of <тип компоненти >;



Приклади:

а) Type LinearTable = Array [0..100] of Integer;
б) type Word = Array [1..20] of Letter;
Letter = 'a'..'z';
Order = Array [Letter] of Integer;
в) type Matrix = array [1..N] of array [1..M] of Real;
г) Tenzor = array [-10..10] of array [0..20] of array [0..3] of Real;

У прикладі в) M і N - константи цілого типу. Зверніть увагу на те, що значення типу Matrix - M*N матриці - визначаються як масиви, компонентами яких, в свою чергу, є масиви з дійсних чисел.

Регулярний тип, значеннями якого є багатомірні масиви (наприклад, в) і г)), можна визначати в скороченому виді:

Type <ім'я> = Array [<Тип> {,<Тип>}] of <тип компоненти>;

Наприклад:

а) Type matrica = array [1..N,1..M] of real;
б) Type Index1 = -10..10;
Index2 = 0..20;
Index3 = 0..3;
Tenzor = Array [Index1, Index2, Index3] of Real;
в) Type Labyrinth = array [1..100,1..100] of Boolean;
Типи Real і Integer не можуть бути типами індексів!

Компонента змінної регулярного типу - компонента масиву явно позначається іменем змінної, за яким у квадратних дужках слідують індекси; індекси являються виразами типу індексу. Наприклад, Table[1, i+j], T[2*i+1, (j*j) mod i], S[Monday, Friday]. Зверніть увагу на те, що на відміну від індексних виразів, межі індексів у змінних - масивах повинні бути константами. Значення індексних виразів повинні бути значеннями типу індексу, тобто знаходитись в межах, що визначені межами індексів.

Розглянемо приклади:

Приклад 6. Програма обчислює скалярний добуток вектора V і вектора V`, отриманого з V перестановкою координат у зворотному порядку.

```
Program ScalarMult;
 Const n = 10;
  Type Vector =  array[1..n] of Real;
   Var V : Vector;   Summa : Real;   i : Integer;
 Begin
  For i := 1 to n do begin { блок читання вихідного вектора }
   Write('Введіть координату вектора : ');  Readln(V[i]);
  end;
  Summa := 0; { блок обчислень}
  For i := 1 to n do
   Summa := Summa + V[i]*V[n-i+1];
   write(' Результат : ', Summa)
 End.
```

Блок обчислень можна оптимізувати за часом. Зазначимо, що Summa обчислюється за формулою: : Summa = V[1]*V[n] + V[2]*V[n-1] +...+ V[2]*V[n-1] + V[1]*V[n]. Отже, її доданки, рівновіддалені від кінців, рівні. Тому кількість повторень циклу можна зменшити вдвоє. При цьому необхідно враховувати парність числа n:

```
{Program ScalarMult1;}
For i := 1 to n div 2 do   Summa := Summa + V[i]*V[n-i+1];
 If Odd(n)
  then Summa := 2*Summa
  else Summa := 2*(Summa + V[n div 2 + 1];
```

Приклад 7. Програма розкладу матриці в суму симетричної і кососиметричної матриць.

Позначимо через Mat, Sym, Assym відповідно вхідну, симетричну і кососиметричну матриці. Тоді

Sym[i, j] = (Mat[i, j] + Mat[j, i])/2

Assym[i, j] = (Mat[i, j] - Mat[j, i])/2

Блоки введення - виведення даних пропонуємо читачеві оформити самостійно. Використовуючи уроки попередньої програми, врахуємо властивості симетрії матриць Sym, Assym:

Sym[i, j] = Sym[j, i], Assym[i, j] = - Assym[j, i];

```
          Program MatrixSymmetry;
  Const n = 10;
   Type Msize = 1 .. n ;
        Matrix = Array [Msize, Msize] of Real;
    Var Mat, Sym, Assym : Matrix;
        i, j : Msize;
Begin
{блок читання вихідної матриці}
For i := 1 to n do
 For j := 1 to i do
{ перебор усіх елементів під головною діагоналлю }
 begin
  Sym[i, j] := (Mat[i, j] + Mat[j, i])/2;
  Assym[i, j] := (Mat[i, j] - Mat[j, i])/2;
  Sym[j, i] := Sym[i,j];                {симетрія}
  Assym[j, i] := -Assym[i, j]           {симетрія}
 end;
{блок друкування отриманих матриць}
End.
```

Ще одне джерело зайвих дій - пересилання даних в масиви, що супроводжуються обчисленнями індексних виразів. Ми двічі звертаємось до Mat[i,j], Mat[j,i], Sym[i,j], Assym[i,j]. Введенням декількох простих допоміжних змінних усунемо цей недолік:

```
{Program MatrixSymmetry1;}
{Var X, Y, U, V : Real;}
For i := 1 to n do
 For j := 1 to i do
{перебор усіх елементів під головною діагоналлю}
 begin { тіло внутрішнього циклу }
  X := Mat[i, j], Y := Mat[j, i];
  U := (X + Y)/2; V := U - X ;    {співвідношення U + V = X}
  Sym[i, j] := U; Sym[j, i] := U;    {симетрія}
  Assym[i, j] := V; Assym[j, i] := -V {симетрія}
 end;
```

На цьому прикладі продемонстровано прийом оптимізації обчислень, що використовують змінні з індексами - усунення зайвих до них звернень. Змінні з індексами подібні функціям: для обчислення їх значень необхідно обчислювати значення індексів!

6. Пошук елемента в масиві.

Задача пошуку елемента в послідовності - одна з важливих задач програмування як з теоретичної, так і практичної точок зору. Ось її формулювання:

Нехай A = {a_1, a_2, ...} - послідовність однотипних елементів і b - деякий елемент, який володіє властивістю P. Знайти місце елемента b в послідовності A. Оскільки представлення послідовності в пам'яті може бути здійснено в виді масиву, задачі можуть бути уточнені як задачі пошуку елемента в масиві A:

1.Знайти максимальний елемент масиву;

2.Знайти даний елемент масиву;

3.Знайти k-тий за величиною елемент масиву;

Найбільш прості і часто оптимальні алгоритми основані на послідовному перегляді масиву A з перевіркою властивості P на кожному елементі.

Приклад 8. Пошук мінімального елемента в масиві.

```
Program MinItem;
  Const n = 23;
  Var A : Array [1..n] of Real;
        Min, Item : Real;   Index, i : Integer;
  Begin
   {Блок введення масиву}
   Index := 1; Min := A[1];
   For i := 1 to n do begin
     Item := A[i];
     If Min > Item
      then begin Index := i; Min := Item end
   end;
   {Виведення значень Index, Min}
  End.
```

Змінна Item введена для того, щоб уникнути повторних обчислень індексного виразу в A[i]. Ми пропонуємо читачу розглянути поведінку цього алгоритму і його модифікації у випадку, коли мінімальних елементів у масиві декілька і треба знайти перший, останній, а також всі мінімальні елементи.

7. Ефективність алгоритму за часом.

Раніш, ніж розглянути задачу пошуку даного елемента, відзначимо важливу відмінну особливість алгоритмів обробки масивів: їх складність визначається характерними параметрами - розмірами вхідних масивів. У розглянутих вище прикладах розмір входу - константа n.

Програма	Складність	Оцінка складності
ScalarMult	$T(n) = T_b*n$	$T(n) = O(n)$
ScalarMult1	$T(n) = T_b*n/2$	$T(n) = O(n)$
MatrixSymmetry	$T(n) = T_b*n*(n + 1)/2$	$T(n) = O(n^2)$
MatrixSymmetry1	$T(n) = T_b*n*(n + 1)/2$	$T(n) = O(n^2)$
MinElement	$T(n) = T_b*n$	$T(n) = O(n)$

Тут T (b) - складність тіла (внутрішнього) циклу. В реальних програмах величина T_b залежить від багатьох факторів, зв'язаних з апаратурою, операційною системою і системою програмування. Тому якість алгоритму, реалізованого у виді програми і не залежного від зовнішніх обставин можна оцінити функцією параметра задачі. При цьому основна властивість функції оцінки складності - швидкість її росту. Для алгоритму ScalarMult ця функція зростає лінійно. Цей факт відображений формулою $T(n) = O(n)$. Складність алгоритму MatrixSymmetry оцінюється квадратичною функцією. Тому $T(n) = O(n^2)$. (Точне визначення запису O(n) дано в математичному аналізі.)

Абстрагування від деталей реалізації програми дає можливість оцінювати алгоритми розв'язування задач і їх реалізацію у виді програми. Мірою ефективності алгоритму є оцінка його складності.

Розглянемо задачу пошуку даного елементу в масиві. Очевидний алгоритм її розв'язування, як і в попередній задачі - послідовний перегляд масиву і порівняння кожного елемента масиву з даним. Відміна полягає в тому, що коли елемент знайдений, перегляд можна припинити. Це означає, що виконання циклу переривається. У мові є засіб переривання - оператор переходу.

8. Мітки. Оператор переходу. Застосування оператора переходу для дострокового виходу з циклу.

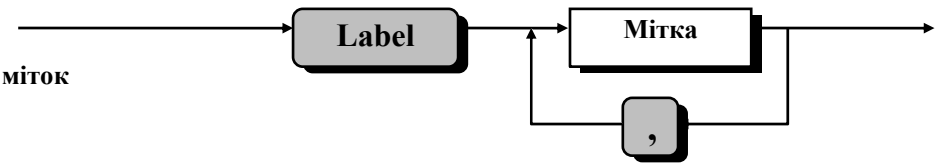
Оператор переходу вказує, що подальша робота (виконання програми) повинна продовжуватись з іншої точки програми, а саме, з оператора, відміченого міткою.

Оператор має вид:

Goto < мітка >

Мітка представляє собою ціле число без знака, що складається не більш ніж з 4 цифр. У розділі операторів кожна мітка може зустрічатися тільки перед одним оператором. Кожна мітка, що зустрічається в розділі операторів, повинна бути описана в розділі міток.

Розділ міток визначений наступною діаграмою:



Розділ міток

Після мітки, що відмічає оператор, треба ставити двокрапку. Наприклад:
1995 : x := x +I; 1 : read(Y);

Увага! Дія оператора переходу усередину складного оператора ззовні не визначено.

Оператор переходу слід використати у незвичайних, виняткових ситуаціях, коли доводиться порушувати природну структуру алгоритму. Треба пам'ятати, що будь-який алгоритм може бути реалізований без застосування Goto без втрати ефективності. Цей факт має принциповий характер. Саме тому структурний стиль програмування іноді називають "Програмування без Goto".

У якості єдиного приклада програми з Goto розглянемо задачу пошуку елемента в одномірному масиві.

Приклад 9.

```
Program Search_in_Array;
  Label 1;
  Const n = 100;
    Var A : Array[1..n] of Real;
        b : Real;
        Flag : Boolean;
           i : Integer;
  Begin
   {Блок читання масиву A і елемента b}
   Flag := true;
   For i := 1 to n do
     If A[i] = b then begin
       Flag := false; goto1
   end; { переривання циклу }
   1:If Flag
     then Writeln(' Елемент ',b,' у  масиві відсутній ')
     else Writeln(' елемент ',b,' стоїть на ',i,'-тому  місці ');
  End.
```

Вишуканий (елегантний) розв'язок цієї задачі без застосування Goto буде розглянуто нижче.

9. Постановка задачі сортування.

Під сортуванням послідовності розуміють процес перестановки елементів послідовності у визначеному порядку. Мета такої впорядкованості - полегшення подальшої обробки даних (зокрема, задачі пошуку). Тому задача сортування - одна з найбільш важливих внутрішніх задач програмування.

Цікаво, що задача сортування є ідеальним прикладом великої кількості різноманітних алгоритмів, розв'язування одної і тої ж задачі. Розглядаючи різні методи сортування, ми побачимо, як зміна алгоритму приводить до нових, більш ефективних у порівнянні з простими, розв'язувань задачі сортування.

Крім цього, послідовності можна представити (реалізувати) в пам'яті різними структурами даних. Як і слід очікувати, ефективність алгоритмів стає дуже залежною від реалізації послідовності.

Нехай дана послідовність елементів a_1, a_2, ... , a_n. Елементи цієї послідовності - дані довільного типу, на якому визначено відношення порядку "<<" (менше) таке, що будь-які два різні елементи можна порівняти.

Сортування означає перестановку елементів послідовності a_{k1}, a_{k2}, ... , a_{kn} таку, що $a_{k1} << a_{k2} << ... << a_{kn}$.

Приклад: послідовність документів, кожний з яких містить інформацію про людину, включаючи його вік. Потрібно розмістити документи цієї послідовності у порядку за віком, починаючи з старшого.

Сортування масивів.

Якщо послідовність a_1, a_2, ... , a_n реалізована як масив a[1..n], вся вона розміщена в адресованій пам'яті. Тому наряду з вимогами ефективності за часом основна вимога - економне використання пам'яті. Це означає, що алгоритм не повинен використовувати додаткові масиви і пересилки з масиву a в ці масиви.

Постановка задачі сортування в загальному виді передбачає, що існують тільки два типи дій з даними сортованого типу : порівняння двох елементів (x << y) і пересилання елемента (x := y). Тому зручна міра складності алгоритму сортування масиву a[1..n] за часом - кількість порівнянь C(n) і кількість пересилань M(n).

Прості алгоритми сортування.

Прості алгоритми сортування можна класифікувати як сортування обміном, сортування вибором і сортування включеннями.

Сортування обмінами.

Основна дія сортування обмінами - порівняння двох елементів і, якщо результат порівняння від'ємний, перестановка їх місцями:

Якщо при i < j a[i] > a[j] то переставити a[i] і a[j].

В найбільш простому варіанті порівнюються елементи a[j] і a[j+1], що стоять поряд:

Приклад 10.

```
Program BublSort;
  Const n = 100;
    Var a : array[1..n] of Real;
        i, j : Integer;
       TempMem : Real;
 Begin
  { Блок введення масиву }
  For i := n - 1 downto 1 do
  For j := 1 to i do
  If a[j] > a[j+1]
    then begin
      TempMem := a[j+1]; a[j+1] := a[j]; a[j] := TempMem end;
  { Блок виведення масиву }
 End.
```

Аналіз алгоритму сортування обмінами.

Аналіз алгоритму полягає в обгрунтуванні його властивостей. Важливішими властивостями алгоритму є: коректність (правильність), оцінка складності за часом і пам'яттю, а також і деякі інші властивості.

Для обгрунтування алгоритму сортування необхідно довести, що алгоритм завжди (незалежно від входу) завершує свою роботу і його результат - зростаюче впорядкований масив. Оцінка складності (ефективності) алгоритму за часом полягає у находженні C(n), M(n) у гіршому випадку, у середньому.

Оскільки алгоритм сортує масив "на місці", його складність по пам'яті - константа.

До інших властивостей алгоритму можна віднести властивість стійкості. (Алгоритм сортування називається стійким, якщо він не переставляє місцями тотожні елементи.) Здійснимо аналіз алгоритму сортування обмінами.

Завершуваність.

Алгоритм BublSort завжди завершує свою роботу, оскільки він використовує тільки цикли з параметром, і в тілі циклів параметри примусово не змінюються.

Коректність.

Внутрішній цикл алгоритму (з параметром j) здійснює послідовний перегляд перших і елементів масиву. На кожному кроці перегляду порівнюються і, якщо це необхідно, переставляються два сусідніх елемента. Таким чином, найбільший серед перших і +1 елементів " спливає " на і +1-е місце. Тому після виконання оператора If має місце співвідношення:

$$a[j+1] = Max(a[1], ..., a[i], a[j+1]) \quad (1)$$

а після завершення циклу (i = j) має місце співвідношення

$$a[i+1] = Max(a[1], ..., a[i], a[i+1]) \quad (2)$$

Зовнішній цикл (з параметром i) керує довжиною частини масиву, що переглядається: вона зменшується на 1. Тому після завершення внутрішнього циклу має місце співвідношення:

$$a[i+1] \le a[i+2] \le \ ... \le a[n] \quad (3)$$

Після завершення зовнішнього циклу отримаємо:

$$a[1] \le a[2], a[2] \le a[3] \le \ ... \le a[n] \quad (4)$$

тобто масив відсортований.

Відзначимо, що більш докладне обгрунтування складається перш за все в доказі співвідношень (1), (3). Це можна зробити методом математичної індукції.

Ефективність за часом.

Зовнішній цикл виконався n-1 разів. Внутрішній цикл виконується i разів (i = n-1, n-2, ..., 1). Кожне виконання тіла внутрішнього циклу складається одно порівняння і, можливо, з одної перестановки. Тому $C(n) = 1+2+ ... +n-1 = n*(n - 1)/2$, $M(n) \le n*(n - 1)/2$. У гіршому випадку (коли елементи вихідного масиву розміщені в порядку спадання)

$C(n) = n*(n - 1)/2 = O(n^2)$, $M(n) = n*(n - 1)/2 = O(n^2)$

Стійкість.

Для доказу стійкості достатньо відмітити, що переставляються тільки сусідні нерівні елементи.

Сортування вибором.

Приклад 11.

Основна дія сортування вибором - пошук найменшого елемента в частині масиву, що переглядається і перестановка з першим елементом частини, що переглядається:

```
For i := 1 to n - 1 do begin
 k := Індекс( Min(a[i], ..., a[n]));
 Переставити a[i], a[k]
end;
Program SelectSort;
  Const n = 10;
    Var a : array[1..n] of Real;
       i, j, MinIndex : Integer;
       TempMem : Real;
  Begin
   {Блок введення масиву}
   For i := 1 to n - 1 do  begin
```

```
  { пошук мінімального елемента }
   MinIndex := i;
   for j := i + 1 to n do
   If a[j] < a[MinIndex] then MinIndex := j;
   {перестановка елементів}
   TempMem := a[MinIndex]; a[MinIndex] := a[i]; a[i] := TempMem
  end;
 {Блок виведення масиву}
End.
```

Аналіз алгоритму сортування вибором.

Внутрішній цикл здійснює пошук мінімального елемента. Після виконання оператора If має місце співвідношення Min = Min(a[i], a[i+1], ... , a[j]), а після завершення циклу Min = Min(a[i], a[i+1], ... , a[n]). Після перестановки маємо a[i] = Min(a[i], a[i+1], ... , a[n]).

Зовнішній цикл керує довжиною частини масиву, що переглядається. Після виконання тіла зовнішнього циклу початок масиву вже відсортований: $a[1] \le a[2] \le .... \le a[i]$

Після завершення зовнішнього циклу отримаємо:

$a[1] \le a[2] \le ... \le a[n-1]$, a[n-1] = Min(a[n-1], a[n]), тобто масив відсортований.

Зовнішній цикл виконався n-1 разів. Внутрішній цикл виконується i-1 разів (i = n-1, n-2, ...,1). Кожне виконання тіла внутрішнього циклу складається в одному порівнянні. Тому

$C(n) = 1 + 2 + ... + n - 1 = n(n - 1)/2$,

Перестановка елементів здійснюється у зовнішньому циклі. Тому

$M(n) = 3(n - 1)$

Також, як і алгоритм сортування обмінами, алгоритм, що аналізується, стійкий. Дійсно, блок пошуку мінімального елемента в хвості масиву шукає мінімальний елемент з найменшим індексом, тому перестановки будуть стійкими. Можна зробити висновок, що просте сортування вибором ефективніше сортування простими обмінами за критерієм M(n). Якщо послідовність, що сортується, складається з даних великого розміру, цей критерій може мати вирішальне значення.

10.Задачі і вправи.

І.Скласти програму табулювання функції z = f(x, y) у прямокутнику [a, b]*[c, d] з кроком табуляції h і точністю ε

	Функція z = f(x,y)	[a, b]	[c, d]	h	ε
1	$z = \ln(1 + x^2 + y^2)$	[-2, 3]	[-1, 3]	0.1	10^{-5}
2	$z = \sin^2(2x+y)$	[$-\pi$, $\pi/2$]	[-2π, π]	0.2	10^{-6}
3	$z = \exp(x^2+y^2)$	[-2, 2]	[-2, 2]	0.1	10^{-5}

4	$z = 1/(tg^2(x+y)+1)$	$[-\pi/2, \pi/2]$	$[-\pi, \pi]$	0.2	10^{-4}
5	$z = \ln(x+sqrt(x^2+y^2))$	[-2, 3]	[-2, 3]	0.1	10^{-5}

II. Скласти програму розв'язування задачі і оцінити її складність:

1.Дано масив A[1..n]. Знайти суму всіх елементів A, що стоять між A[1] і A[n].

2.Дано масив A[1..n]. Підрахувати середнє арифметичне всіх від'ємних і всіх додатних його елементів.

3.Послідовність з n точок площини задана масивами X[1..n] і Y[1..n] координат. Знайти точку, найменш віддалену від початку координат.

4.Дано масив A[1..n]. Знайти всі його елементи, менші, ніж всі попередні.

5.Дано масив A[1..n]. Знайти в цьому масиві найбільшу за кількістю елементів зростаючу підпослідовність елементів, що йдуть підряд.

6.Дано масив A[1..n], що складається з непарного (n = 2k+1) числа попарно нерівних елементів. Знайти середній по величині елемент у масиві.

7.Дано масиви A[1..n] і B[1..m]. Знайти всі елементи масиву A, що не входять у B.

8.Дано масиви A[1..n] і B[1..m]. Знайти всі елементи масиву A, що входять у B.

9.Дано масив A[1..n]. Величину S_{ij} визначимо наступним чином:

$S_{ii} = A[i]$,

$S_{ij} = A[i] + A[i+1] + ... + A[j]$ при $i < j$

$S_{ij} = S_{ji}$ при $i > j$

Знайти Max S_{ij} при $1 \le i, j \le n$

III. Скласти програму розв'язування задачі і оцінити її складність:

1.Помножити матрицю A розміром m x n на вектор b.

2.Перемножити матриці A і A'. (A' - A транспонована)

3.Перемножити матриці A і B.

4.Дано масив A[1..n, 1..m]. Знайти сідлову точку масиву або встановити її відсутність. (Елемент двомірного масиву називається сідловою точкою, якщо він максимальний у своєму стовпці і мінімальний у своєму рядку.)

5.Дано масив A[1..n, 1..m]. Знайти стовпець, сума квадратів елементів якого мінімальна.

6.Дано масив A[1..n, 1..m]. Знайти всі елементи A менші, ніж усі сусідні. (Сусідніми називаються елементи, індекси яких відрізняються на 1).

ІТЕРАЦІЙНІ ЦИКЛИ.

1. Оператори повторення While і Repeat.

У попередньому параграфі ми вивчили оператор повторення з параметром (For).

Це оператор використовується лише у випадку, коли заздалегідь відома кількість повторень тіла циклу. У більш загальному випадку, коли кількість повторень заздалегідь невідома, а задана деяка умова закінчення (або продовження) циклу, у мові Pascal використовують інші оператори повторення: оператор циклу з передумовою While і оператор циклу з постумовою Repeat.

Оператор циклу з передумовою визначений діаграмою:

Оператор циклу з передумовою

Оператор (тіло циклу) виконується до тих пір, поки умова істинна. Якщо при першій перевірці умова виявилась хибною, оператор не виконується ні разу.

Приклад 1. Знайти найменший натуральний розв'язок нерівності $x^3 + ax^2 + bx + c > 0$ з цілими коефіцієнтами.

Очевидний алгоритм пошуку розв'язку складається у послідовному обчисленні значень $Y = x^3 + ax^2 + bx + c$ для x = 1, 2, 3, ... до тих пір, поки $Y \le 0$.

```
Program UneqvSolution;
        Var a, b, c : Integer;
                X : Integer;  Y : Real;
      Begin
        Write(‘ введіть коефіцієнти a, b, c : ‘); Readln(a, b, c);
         X := 1; Y := a + b + c + 1; { Ініціалізація циклу }
        While Y <= 0 do begin
         X := Succ(X); { Наступне значення X }
         Y := ((X + a)*X + b)*X + c { Наступне значення Y }
      end;
      Writeln(‘X = ‘, X, ‘ Y = ‘, Y )
  End.
```

Для обгрунтування цього методу вимагається єдине: впевнитись у тому, що цикл коли-небудь завершиться, тобто що розв'язок нерівності існує! Відмітимо, що

$\lim (x^3 + ax^2 + bx + c) = +\infty$

$x \to +\infty$

Тому при деякому x_0 $x_0^3 + ax_0^2 + bx_0 + c > 0$ Покладемо $x = |a| + |b| + |c|$. Тоді $x^3 = (|a| + |b| + |c|)x^2 > |a|x^2 + |b|x + c$ Отже, $x^3 + ax^2 + bx + c > x^3 - (|a|x^2 + |b|x + c) > 0$ Тому кількість повторень циклу не більше ніж величини $|a| + |b| + |c|$, що дає можливість не тільки обгрунтовувати завершення, але й оцінити складність програми у гіршому випадку.

Оператор циклу з постумовою визначений діаграмою:

Тіло циклу Repeat виконується до тих пір, поки умова приймає значення False. Дії, що містяться в тілі циклу, будуть виконані у крайньому випадку один раз. Таким чином, умова є умовою закінчення циклу.

Приклад 2. Знайти номер найменшого числа Фібоначчі, що ділиться на 10. Послідовність Фібоначчі { F(n) } визначається рекуррентно:

F(1) = F(2) = 1, F(n+2) = F(n+1) + F(n)

Як і у попередньому прикладі, обчислюємо F_n для n = 3, 4, ... до тих пір, поки не знайдемо елемент послідовності, що ділиться на 10. Проблема обгрунтування методу залишається тією ж: чи існує потрібний елемент?

```
Program Fibbonachy;
  Var Fib1, Fib2 : Integer;
        Index : Integer;  Buf : Integer;
    Begin
Fib1 := 1; Fib2 := 1; Index := 2; { Ініціалізація циклу }
Repeat
  Buf := Fib2;
  Fib2 := Fib2 + Fib1;                  { Наступне число Фібоначчі }
  Fib1 := Buf;                          { Попереднє число Фібоначчі }
  Index := Succ(Index)                  { Номер числа Фібоначчі }
 until Fib2 mod 10 = 0;
 Writeln('Номер = ', Index, ' Число = ', Fib2)
End.
```

Цикли While і Repeat називають ще ітераційними циклами, оскільки за їх допомогою легко реалізувати різного роду ітераційні обчислення (обчислення, в яких кожний наступний результат є уточненням попереднього). Умова закінчення циклу - досягнення відхилення результату Y_n от Y_{n-1} деякої допустимої похибки ε:

$$|Y_n - Y_{n-1}| < \varepsilon$$

Приклад 3: Обчислити з заданою точністю значення кореня рівняння x = cos(x) на відрізку [0 ; 1] методом ділення відрізка навпіл .

Метод наближеного розв'язування рівняння діленням відрізка навпіл - один з найбільш простих і поширених чисельних методів. Суть його полягає у послідовному уточненні меж відрізка, на якому існує корінь. На кожному кроці методу цей відрізок ділиться навпіл, а потім вирішується питання про те, в якій половині (лівій чи правій) знаходиться корінь. Зменшення відрізка вдвоє теоретично гарантує завершення циклу пошуку з заданою точністю.

```
Program TransEquation;
  Const a = 0; b = 1;
        Epsilon = 1e-5;
   Var u, v, Root : Real;
        Fu, Fr : Real;
  Begin
   u := a; v := b; { ініціалізація циклу}
   Fu := Cos(u) - u;
   Repeat
     Root := (u + v)/2 ; { середня точка відрізка }
     Fr := Cos(Root) - Root;
     if Fu*Fr > 0 { вибір правої або лівої половини }
      then begin u := Root; Fu := Fr end
      else v := Root
   until Abs(v - u) < Epsilon;
  writeln (‘ корінь= ‘, Root, ‘ з точністю ‘,Epsilon)
 End.
```

Для того, щоб знайти кількість N повторень у циклі, відмітимо, що на k-тому кроку циклу виконана нерівність $| v - u | = (b - a)/2^k$. Тому при найменшому k такому, що $(b - a)/2^k < \varepsilon$ цикл завершиться. Неважко обчислити N:

$$N = [\log_2(b - a)/ \varepsilon] \qquad (1)$$

Як показують розглянуті приклади, одна з основних проблем аналізу програм, що містять ітераційні цикли, складається в обгрунтуванні завершення циклу (і програми). На відміну від алгоритмів з арифметичними циклами, ця проблема не є тривіальною!

Оскільки кількість повторень ітераційного циклу не визначена, задача оцінки складності програми часто також не проста.

2. Алгоритми пошуку і сортування. Лінійний пошук у масиві.

Приклад 4. Розглянемо красивий розв'язок задачі пошуку даного елемента в масиві, що використовує цикл While. Насправді, замість використання циклу For з достроковим виходом при допомозі Goto можна застосувати цикл While:

```
Program WhileSearch;
  Const n = 100;
    Var A : Array[1..n] of Real;
        b : Real; i : Integer;
  Begin
  {Блок читання масиву A і елемента b}
   i := 1;
   While (i <= n) and (A[i] <> b) do i := Succ(i);
     If i = n + 1
       then Writeln('Елемент ',b,'у масиві відсутній')
       else Writeln('елемент ',b, розміщений на',i,'-тому місці');
     End.
```

Недоліком такої реалізації є по-перше, використання складної умови в циклі, по-друге - можливо некоректне обчислення підвиразів A[i] <> b при i = n +1. Не дивлячись на те, що умова завідомо хибна (i <= n = False), вираз A[n+1] <> b не визначено!

Доповнимо масив A ще одним елементом: A[n+1] = b, і всі недоліки легко ліквідуються!

```
Program LinearSearch;
  Const n = 101; { Розмір масиву збільшений на 1 }
    Var A : Array[1..n] of Real;
        b : Real;  i : Integer;
  Begin
   { Блок читання масиву A і елемента b }
   i := 1;
   A[n] := b; { Доповнимо масив "бар'єрним" елементом }
   While A[i] <> b do i := Succ(i);
    If i = n
      then Writeln('Елемент ',b,'в масиві відсутній')
      else Writeln('елемент ',b,'розміщений на',i,'-тому місці');
   End.
```

Поліпшений алгоритм сортування обмінами.

Приклад 5. У програмі BublSort цикл For використовується в якості зовнішнього. Це приводить до того, що він виконується рівно n - 1 разів, навіть якщо масив вже упорядкований після декількох перших проходів. Від зайвих проходів можна позбавитися, застосувавши цикл Repeat і перевіряючи у внутрішньому циклі масив на упорядкованість:

```
Program BublSort1;
        Const n = 100;
  Var a : array[1..n] of Real;
      i, j : Integer;
     TempMem : Real; isOrd : Boolean;
 Begin
  {Блок введення масиву}
  i := n - 1;
  Repeat
   isOrd := True; { ознака упорядкованості}
   For j := 1 to i do
    If a[j] > a[j+1]
     then begin
      TempMem := a[j+1]; a[j+1] := a[j]; a[j] := TempMem;
       isOrd := False { виявлено“засортування” }
      end;
    i := Pred(i)
   until isOrd;
  {Блок виведення масиву}
 End.
```

Цей алгоритм можна ще покращити, якщо кожний наступний прохід починати не з початку (j = 1), а з того місця, де на попередньому проході відбувся перший обмін:

```
{LowIndex : Integer;}
Repeat
isOrd := True; { ознака упорядкованості}
LowIndex := 1; { відрізок A[1..LowIndex] упорядкований}
For j := LowIndex to i do
 If a[j] > a[j+1]
 then begin
  TempMem := a[j+1]; a[j+1] := a[j]; a[j] := TempMem;
   If isOrd then begin LowIndex := j; isOrd := False
  end
 end;
 i := Pred(i)
until isOrd;
```

Відзначимо однак, що наш алгоритм працює не симетрично: якщо найбільший елемент “спливає” на своє місце за один прохід, то найменший елемент “потоне” на 1-е місце, знаходячись початково на n-тому місці, за n-1 прохід. При цьому здійсниться максимально можливе число порівнянь. Усунути асиметрію можна, чергуючи проходи

вперед і назад. Реалізацію цієї ідеї, як і подальші поліпшення алгоритму сортування простими обмінами, ми надаємо читачеві в якості вправи.

Всі поліпшення методу, що розглядається відносяться до ефективності в середньому. Жодне з них не зменшує оцінки складності C (n) у гіршому випадку. (Доведіть це самостійно!)

Бінарний пошук в упорядкованому масиві.

Приклад 6. Задача пошуку суттєво спрощується, якщо елементи масиву упорядковані. Стандартний метод пошуку в упорядкованому масиві - це ділення відрізка навпіл, причому відрізком є відрізок індексів 1..n. Насправді, нехай m (k < m < l) - деякий індекс. Тоді якщо A[m] > b, далі елемент треба шукати на відрізку k..m-1, а якщо A[m] < b - на відрізку m+1..l.

Для того, щоб збалансувати кількість обчислень в тому і іншому випадках, індекс m треба вибирати так, щоб довжини відрізків k..m, m..l були (приблизно) рівними. Описану стратегію пошуку називають бінарним пошуком.

```
Program BinarySearch;
  Const n = 100;
   Var A : Array[1..n] of Real;
       b : Real; Buffer : Real;
       k, l, m : Integer;

 Begin
   { Блок читання упорядкованого масиву A і елемента b }
   k := 1; l := n;
   Repeat
   m := (k + l) div 2; Buffer := A[m];
   If Buffer > b
    then l := Pred(m)
    else k := Succ(m)
   until (Buffer = b) or (l < k);
   If A[m] = b
    then Writeln('Індекс = ', m)
    else Writeln( 'Елемент не знайдений')
  End.
```

Неважко отримати оцінку числа C (n) порівнянь методу, застосувавши формулу (1) прикладу 3. Нехай M - кількість повторень циклу. Тоді

$$M \leq [\log_2 (n)]$$

Оскільки на кожному кроку здійснюється 2 порівняння, у гіршому випадку

$$C(n) = 2[\log_2 n] = O(\log_2 n) \qquad (2)$$

3. Алгоритми сортування масивів (продовження). Сортування вставками.

Ще один простий алгоритм сортування - сортування вставками базується на наступній ідеї: припустимо, що перші k елементи масиву A[1..n] вже упорядковані:

$A[1] \le A[2] \le ... \le A[k], A[k+1], ..., A[n]$

Знайдемо місце елемента A[k+1] в початковому відрізку A[1],...,A[k] і вставимо елемент на своє місце, отримавши упорядковану послідовність довжини k+1. Оскільки початковий відрізок масиву упорядкований, пошук треба реалізувати як бінарний. Вставці елемента на своє місце повинна передувати процедура зсуву “хвоста” початкового відрізка для звільнення місця.

```
Program InsSort;
  Const n = 100;
    Var A : array[1..n] of Integer;
        k : Integer;
        l, r, i, j : Integer;
        b : Integer;
  Begin
  {Блок читання масиву A}
   For k := 1 to n-1 do begin
    b := A[k+1];
    If b < A[k]
     then begin
      l := 1; r := k; {Бінарний пошук}
     Repeat
       j := (l + r) div 2;
       If b < A[j]  then r := j  else  l := j + 1;
      until (l = j);{ Зсув “хвоста” масиву на 1 позицію праворуч}
      For i := k downto j do A[i+1] := A[i];
     A[j] := b ; { Пересилання елемента на своє місце}
    end
   end;
  {Блок виведення масиву A}
  End.
```

Оцінимо ефективність алгоритму. Пошук міста елемента A[k+1] потребує, як показано вище, $O(\log_2 k)$ порівнянь. Тому у гіршому випадку кількість порівнянь C (n) є

$C(n) = O(\log_2 2) + ... + O(\log_2(n-1)) + O(\log_2 n) = O(n \log_2 n)$

Зсув “хвоста” на кожному кроку зовнішнього циклу у гіршому випадку потребує k перестановок. Тому у гіршому випадку

$M(n) = 1 + ... + (n-2) + (n-1) = n*(n-1)/2 = O(n^2)$

Таким чином, алгоритм сортування вставками значно ефективніший, ніж всі розглянуті раніше алгоритми за числом порівнянь. Однак число перестановок у гіршому випадку також буде значним, як і у самому неефективному алгоритмі - сортуванню простими обмінами.

4. Задачі і вправи.

Вправа 1. Визначити число членів нескінченного ряду чисел, необхідне для обчислення його суми з точністю ε.

1. $\sum_{n=0}^{\infty} [(-1)^n /(2n+1)] = \pi/4$
2. $\sum_{n=0}^{\infty} [(-1)^n/(2n)!] = \cos 1$
3. $\sum_{n=0}^{\infty} [(-1)^n/(2n+1)!] = \sin 1$

Вправа 2. Знайти суму членів функціонального ряду з заданою похибкою ε.

1. $\sin x = \sum_{n=0}^{\infty} (-1)^n(x^{2n+1})/(2n+1)!$
2. $\ln x = \sum_{n=1}^{\infty} (-1)^n * x^n / n;\ x>1/2$
3. $\operatorname{arctg} x = \sum_{n=0}^{\infty} (-1)^n\, x^{2n+1}/(2n+1);\ |x|<1$

Вправа 3. Знайти всі трьохзначні члени послідовності F_n, визначеної рекурентними співвідношеннями:

1. $F_0 = 1,\ F_1 = 1,\ \ F_{n+2} = F_{n+1} + F_n$ при $n > 0$
2. $F_0 = 5,\ F_1 = 1,\ F_2 = 1,\ \ F_{n+3} = F_{n+2} + F_{n+1} - F_n$ при $n > 0$

Вправа 4.

1.Дано раціональне число p/q. Побудувати його розклад у ланцюговий дріб.

2.Побудувати розклад натурального числа N у добуток степенів його простих дільників.

3.Перевести ціле число N у 2-ічну систему числення. Узагальнити алгоритм на випадок p-ічної системи числення.

4.Відомо, що якщо d = НСД(a,b), то існують такі числа u, v, що d = au + bv. По даним a, b знайти d, u, v.

5.Число N називається досконалим, якщо воно співпадає з сумою своїх власних дільників. (Наприклад, 6=1+2+3). Знайти всі досконалі числа, які не перевищують M.

6.Знайти номер найменшого числа Фібоначі, що закінчується 2-ма нулями. Знайти само це число.

Вправа 5.

1.Дано масив A[1..n] дійсних чисел. Знайти у цьому масиві найменше додатне число.

2.Дано масив A[1..n] цілих чисел. Знайти в цьому масиві найбільшу кількість нулів, що йдуть підряд.

3.Дано масив A[1..n] цілих чисел. Відрізок індексів k .. m називається відрізком росту масиву, якщо A[k] < A[k+1] < ... < A[m]. Знайти відрізок росту максимальної довжини.

Задача 1. Масив A[1..n] складається з даних типу Color = (white, black). Скласти програму сортування масиву A, яка перевіряє значення (колір) кожного елемента тільки один раз.

Задача 2. Масив A[1..n] складається з даних типу Color = (white, red, black). Скласти програму сортування масиву A, яка перевіряє значення (колір) кожного елемента тільки один раз.

Задача 3. Масив A[1..n] складається з дійсних чисел, причому для будь-якого $2 \le i \le n-1$ виконана нерівність $A[i] \le (A[i-1] + A[i+1])/2$ Скласти програму пошуку найменшого і найбільшого елементів масиву, складність якої $C(n) = O(\log_2(n))$

Задача 4.Задані упорядковані масиви A[1..n] і B[1..m] цілих чисел і ціле число C. Скласти програму пошуку таких індексів i і j, що C = A[i] + B[j], складність якої $C(n) = O(\log_2(n\ m))$

Задача 5. Множина точок площини $\{A_i = (X_i, Y_i)\}$ задана масивами координат X[1..n] і Y[1..n]. Випуклою лінійною оболонкою цієї множини називається така його підмножина B, що багатокутник з вершинами в точках з B є випуклими і містить всю множину A. Скласти програму пошуку випуклої лінійної оболонки множини A. Оцінити її складність.

Задача 6.

а) Реалізувати арифметичний цикл за допомогою циклу While;

б) Реалізувати арифметичний цикл за допомогою циклу Repeat;
в) Реалізувати цикл While за допомогою циклу Repeat;
г) Реалізувати цикл Repeat за допомогою циклу While;
д) Реалізувати цикл While за допомогою операторів If і Goto;
є) Реалізувати цикл Repeat за допомогою операторів If і Goto;

ПРОЦЕДУРИ І ФУНКЦІЇ

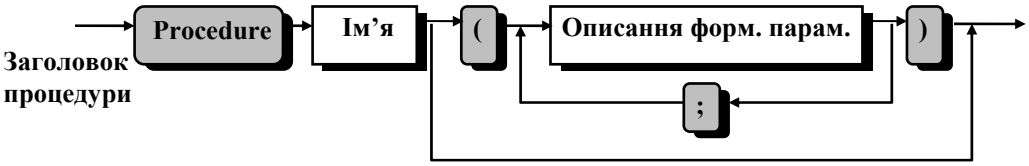
1. Опис процедур.

Технологія програмування мовою програмування типу мови Pascal пропонує проектувати програми методом послідовних уточнень.

На кожному етапі - кроку уточнення програміст розбиває задачу на деяке число підзадач, визначаючи тим самим деяку кількість окремих підпрограм. Концепція процедур (підпрограм) дозволяє виділити підзадачу як явну підпрограму.

У мові Паскаль процедури визначаються в розділі процедур і функцій за допомогою описань процедур. Звернення до процедури здійснюється оператором процедури.

Описання процедури таке ж, як і описання програми, але замість заголовка програми фігурує заголовок процедури. Заголовок має вид:



<u>Приклади заголовка процедури:</u>

procedure Picture;

procedure Power(X: real; n: Integer; var u, v: Real);

procedure Integral(a, b, epsilon: Real; var S: Real);

2.Формальні параметри. Локальні і глобальні об'єкти.

Як бачимо з прикладів, у розділі формальних параметрів перелічуються імена формальних параметрів, а потім вказується їхній тип. Таким чином, кожне описання формальних параметрів з точки зору синтаксису має такий же вигляд, як і описання змінних у розділі змінних. Перед деякими описаннями ставиться службове слово Var. Такі параметри називаються параметрами-змінними. Якщо перед описанням службове слово Var не стоїть, це - параметри-значення. Різницю між цими типами параметрів описано нижче.

Мітки, імена констант, типів, змінних, процедур і функцій, що описуються в тілі процедури, а також всі імена, що вводяться в розділи формальних параметрів є локальними в описанні процедури, яка називається областю дії для цих об'єктів. За межами цієї області вони не відомі.

Резервування пам'яті під локальні об'єкти здійснюється таким чином, що вона "захоплюється" процедурою при її виклику оператором і звільняється при виході з процедури. Такий механізм розподілу пам'яті називається динамічним. Динамічний розподіл дозволяє економити пам'ять, що адресується.

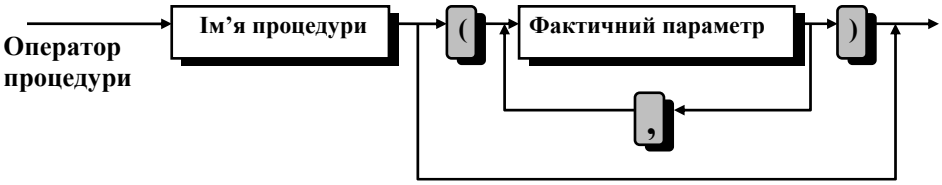
Об'єкти, що описані в основній програмі, доступні для використовування в цій процедурі. Вони називаються глобальними. Відмітимо, що локальний і глобальний об'єкти цілковито різні, але можуть мати одне й те ж ім'я. Якщо, наприклад, змінна X описана як в основній програмі, так і в процедурі, то в процедурі вона використовується як локальна. Якщо ж в процедурі змінна X не описана, то в процедурі вона використовується як глобальна.

3. Оператор процедури. Фактичні параметри.

Оператор процедури має вид:

< ім'я > або < ім'я > (< список фактичних параметрів >)

Синтаксична діаграма оператора процедури:



<u>Приклади операторів процедури:</u>

Picture

Power((a + b)/2, 3, degree, root)

Integral (0, P/2, 1E-6, SUMMA)

Зверніть увагу на відповідність між заголовком процедури і оператором процедури. Між списками формальних і фактичних параметрів встановлена взаємно-однозначна відповідність, що визначена їх місцями в списках. Ця відповідність ілюструється наступним прикладом:

Приклад 1. Розглянемо заголовок процедури і оператор цієї процедури:

```
Procedure Integral ( a, b, eps: real; var s: real );
   Integral ( -Pi/2, Pi/2, 1E-6, summa );
```

Відповідність:

Формальний параметр		Фактичний параметр
Значення	a	Вираз -Pi/2
Значення	b	Вираз Pi/2
Значення	eps	Дане 1E-6
Змінна	s	Змінна Summa

Як було вказано вище, параметри бувають 2-х видів: параметри-значення і параметри-змінні. Якщо перед описанням параметрів ніякого службового слова немає, мова йде про параметри-значення. Перед описанням параметрів-змінних ставиться службове слово var. При зверненні до процедури (в процесі виконання оператора процедури) формальним параметрам-значенням присвоюються значення відповідних фактичних параметрів, а замість імен формальних параметрів-змінних підставляються відповідні фактичні параметри - імена змінних, а потім виконується підпрограма, що описана процедурою.

Якщо x1, x2, ..., xn - фактичні параметри-змінні, відповідні формальним параметрам-змінним v1, ... , vn, то x1, x2, ..., xn повинні бути різними. Фактичними параметрами-значеннями можуть бути вирази або дані відповідних типів.

Розглянемо приклад:

Приклад 2. Програма обчислює координати точки (x_0, y_0) при послідовних поворотах і паралельних переносах системи координат.

```
Program Coordinates;
  Const Pi = 3.141592;
    Var Alfa, Beta : Real;
        x0, y0, x1, y1, x2, y2 : Real;
        x, y : Real;
 Procedure Rotate(x, y, Fi: Real; var u, v: Real );
       var cosFi, sinFi : Real; { локальні змінні }
      begin
        Fi := Fi*Pi/180 ;
        cosFi := Cos(Fi); sinFi := Sin(Fi);
```

```
        { параметри x, y захищені від глобальних змінних x, y }
         u := x * cosFi - y * sinFi ;
         v := x * sinFi + y * cosFi
        end ;
Procedure Move(x, y, a, b : Real; var u, v: Real);
    begin
      u := x + a ; v := y + b
     end;
  begin
    Read (x0, y0);  Read (Alfa);  Rotate(x0, f0, alfa, x, y);
    Read (x1, y1);     Move(x, y, x1, y1, x, y);
    Read (Beta);       Rotate(x, y, Beta, x, y);
    Read ( x2, y2 );  Move(x, y, x2, y2, x, y);
    Writeln ('================================');
    Writeln ('абсциса точки : ', x);
    Writeln ('ордината точки : ', y);
  end.
```

Параметри-значення використовуються для передачі даних в процедуру. Це значить, що для параметра-значення на час виконання процедури резервується пам'ять, розмір якої визначений типом параметра і яка заповнюється при виклику процедури. Таким чином, використання параметрів-значень при передачі даних великого об'єму може привести до неоправданих витрат часу процесора і пам'яті, що адресується.

Нехай, наприклад, змінна A типу Sequence - масив з 1000 дійсних чисел і Procedure MaxMin(X : Sequence; var Max, Min : Real) - пошук максимального і мінімального елементів масиву X. Тоді при зверненні до процедури MaxMin за допомогою оператора MaxMin(A, Sup, Inf) компілятор виділить пам'ять (6 - 10 байт на кожний елемент масиву - всього 6000 - 10000 байт) і здійснить 1000 циклів пересилання чисел з A в X. Якщо ж параметр X визначити як параметр-змінну: Procedure MaxMin(var X : Sequence; var Max, Min : Real) ні пам'яті, ні пересилань не знадобиться.

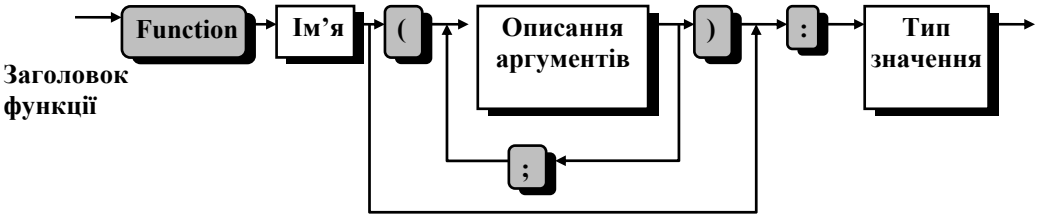
4. Функції.

Поряд із стандартними функціями, в мові можна визначити і інші необхідні програмі функції. Функція - це підпрограма, що визначає одне - єдине скалярне або засилкове значення, що використовується при обчисленні виразу. Описання функції має, по суті, такий самий вид, як і описання процедури. Різниця тільки у заголовку, який має вид:

Function < ім'я > : < тип результату > ;

або **Function < ім'я > (<список описань формальних параметрів >): < тип результату >;**

Синтаксична діаграма заголовка функції:



Зверніть увагу на описання результату, який визначає тип значення функції.

Таким чином, для функції визначені всі ті поняття, які були сформульовані для процедур.

Ім'я, що задане в заголовку функції, іменує цю функцію. В середині описання функції - в розділі операторів - повинно бути присвоювання, що виконується, в лівій частині якого стоїть ім'я функції, а в правій - вираз що має тип значення функції.

Приклади опису функцій.

Приклад 3. Функція GCD (алгоритм Евкліда) обчислює найбільший спільний дільник двох натуральних чисел x і y.

```
Function GCD (x, y : Integer) : Integer ;
  Begin
    While x <> y do
      If x < y
       then y := y - x
       else x := x - y ;
       GCD := x
   End;
```

Приклад 4. Функція IntPow підносить дійсне число x до степеня N. ($Y = x^N$)

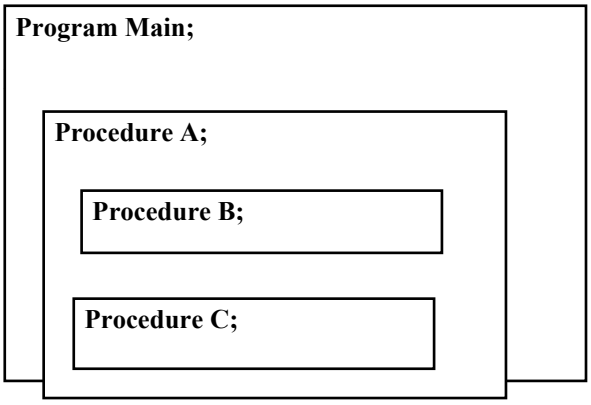
```
Function IntPow(x: Real; N: Integer) : Real;
    Var i: Integer;
  Begin
    IntPow := 1;
    For i:=1 to Abs(N) do IntPow := IntPow * x;
    If N < 0 then IntPow := 1/IntPow
   End;
```

Приклад 5. Програма обчислює найбільший спільний дільник послідовності натуральних чисел, яка представлена масивом.

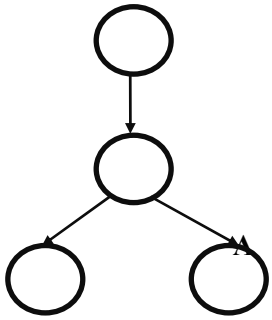
```
Program GCD_of_Array;
```

```
   Const n = 100 ;
    Var i, D : Integer;
        A : Array[1..n] of Integer;
Function GCD (x, y : Integer) : Integer ;
  Begin
   While x <> y do
    If x < y
     then y := y - x
     else x := x - y;
     GCD := x
   End;
Begin { основна програма }
{Процедура читання масиву натуральних чисел}
  D := GCD (A[1], A[2]);
  For i := 3 to n do D := GCD(D, A[i]);
  writeln ( ‘ НОД послідовності = ‘ , D )
 End.
```

Кожна процедура або функція може, в свою чергу, містити розділ процедур і функцій, в якому визначені одна або декілька процедур і функцій. В цьому випадку кажуть про вкладення процедур. Кількість рівнів вкладень може бути довільним. Структура вкладення ілюструється малюнком:



C



B

Поняття локальних і глобальних об’єктів поширюються і на вкладені процедури. Наприклад, змінна, описана в процедурі А локальна по відношенню до основної програми і глобальна для процедур В і С, вкладених в А.

В деяких випадках необхідно з процедури здійснити виклик іншої процедури, описаної в тому ж розділі процедур і функцій. Наприклад, процедура C може містити оператор виклику процедури B. В цьому випадку компілятор правильно обробить текст програми, оскільки процедура B описана до процедури C. Якщо ж з процедури B необхідно звернутись до C, для правильної обробки виклику C необхідно використовувати механізм так званого попереднього описання C. Описання, що опереджає процедури (функції) - це її заголовок, услід за яким через ";" стоїть службове слово Forward. У тексті програми описання, що опереджає, повинно передувати процедурі, в якій процедура, що попередньо описана, викликається.

Якщо процедура або функція описані попередньо, описанню її тіла передує скорочений заголовок, що складається тільки з відповідного службового слова і імені - без списку описань параметрів.

```
Procedure A (x : TypeX; Var y : TypeY); Forward;
Procedure B (z : TypeZ) ;
Begin
... A( p, q); ...
End;
Procedure A;
Begin
 ...
End;
```

5. Рекурсивно-визначені процедури і функції.

Описання процедури A, в розділі операторів якої використовується оператор цієї процедури, називається рекурсивним. Таким чином, рекурсивне описання має вид

```
Procedure A(u, v : ParType);
 ...
Begin
...; A(x, y); ...
End;
```

Аналогічно, описання функції F, в розділі операторів якої використовується виклик функції F, називається рекурсивним. Рекурсивне описання функції має вид

```
Function F(u, v : ArgType) : FunType;
 ...
Begin
...; z := g(F(x, y)); ...
 End;
```

Використання рекурсивного описання процедури (функції) приводить до рекурсивного виконання цієї процедури (обчисленню цієї функції). Задачі, що формулюються природнім чином як рекурсивні, часто приводять до рекурсивних розв'язків.

Приклад 6. Факторіал.

Розглянемо рекурсивне визначення функції n!=1*2*...*n (n-факторіал). Нехай F(n) = n! Тоді

1.F(0) = 1

2.F(n) = n*F(n - 1) при n > 0

Засобами мови це визначення можна сформулювати як обчислення:

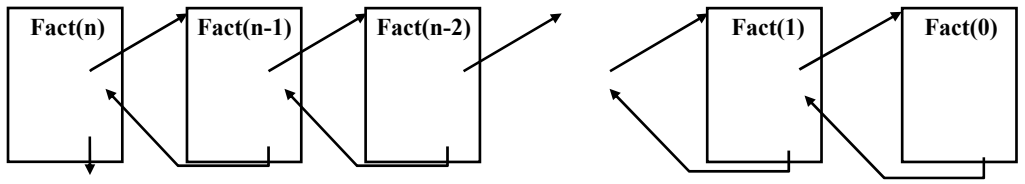
```
If n = 0
 then F := 1
 else F := F(n - 1) * n
```

Оформивши це обчислення як функцію і змінивши ім'я, отримаємо:

```
Function Fact(n: Integer): Integer;
  Begin
   If n = 0
    then Fact := 1
    else Fact := Fact(n - 1) * n
   End;
```

Обчислення функції Fact можна представити як ланцюжок викликів нових копій цієї функції з передачею нових значень аргументу і повернень значень функції в попередню копію.

n-1 n-2 n-3 1 0



n! (n-1)! (n-2)! 1 1

Ланцюжок викликів обривається при передачі нуля в нову копію функції. Рух у прямому напрямку (розгортання рекурсії) супроводжується тільки обчисленням умови і викликом. Значення функції обчислюється при згортанні ланцюжка викликів. Складність обчислення $T_{fact}(n)$ функції Fact можна оцінити, виписавши рекурентне співвідношення:

$T_{fact}(n) = T_{fact}(n-1) + T_m + T_l + T_c$

Для того, щоб обчислити Tfact(n), треба здійснити одну перевірку, одне множення і один виклик $T_{fact}(n-1)$. Виклик T_{fact} потребує затрат часу T_c на "адміністративні" обчислення: передачу параметра, запам'ятовування адреса повернень і т.п. Поклавши $C = T_m+T_l+T_c$, отримаємо $T_{fac}t(n) = T_{fact}(n-1) + C$. Неважко тепер показати, що $T_{fact}(n) = Cn$.

Приклад 7. Перестановки. Згенерувати всі перестановки елементів скінченої послідовності, що складається з букв.

Спробуємо звести задачу до декількох підзадач, більш простих, ніж вихідна.

Нехай $S = [s_1, s_2, ..., s_n]$ - набір символів.

Через Permut(S) позначимо множину всіх перестановок S, а через Permut(S,i) - множину всіх перестановок, в яких на останньому місці стоїть елемент s_i. Тоді

$Permut(S) = Permut(S,n) \cup Permut(S, n-1) \cup ... \cup Permut(S,1)$

Елемент множини Permut(S, i) має вид $[s_{j2},..., s_{jn},s_i]$ де j2,..., jn - всі можливі перестановки індексів, не рівних I. Тому Permut(S, i) = (Permut(S \ s_i), s_i) і $Permut(S)=(Permut(S \setminus s_1),s_1)+...+ (Permut(S \setminus s_n), s_n)$.

Отримане співвідношення виражає множину Permut(S) через множини перестановок наборів з (n-1) символу. Доповнивши це співвідношення визначенням Permut(S) на одноелементній множині, отримаємо:

1. Permut({s}) = {s}
2. $Permut(S) = (Permut(S\setminus s_1), s_1) + ... + (Permut(S\setminus s_n), s_n)$

Уточнимо алгоритм, опираючись на представлення набору S в виді масиву S[1..n] of char.

По перше, визначимо параметри процедури Permut:

k - кількість елементів в наборі символів;

S - набір символів, що переставляються.

Алгоритм починає роботу на вхідному наборі і генерує всі його перестановки, що залишають на місці елемент s[k]. Якщо множина перестановок, в яких на останньому місці стоїть s[j], уже породжена, міняємо місцями s[j-1] і s[k], виводимо на друк отриманий набір і застосовуємо алгоритм до цього набору. Параметр k керує рекурсивними обчисленнями: ланцюжок викликів процедури Permut обривається при k = 1.

```
Procedure Permut(k : Integer; S : Sequence);
Var j : integer;
   Begin
     if k <> 1 then Permut(k - 1, S);
     For j := k - 1 downto 1 do begin
     Buf := S[j]; S[j] := S[k]; S[k] := Buf;
     WriteSequence(S); Permut(k - 1, S)
     end
   End;
   Begin { Розділ операторів програми}
   {Генерація вихідного набору S}
```

```
    WriteSequence(S); {Виведення першого набору на друк}
     Permut(n, S)
   End.
```

Оцінимо складність алгоритму за часом у термінах C(n): Кожний виклик процедури Permut(k) містить k викликів процедури Permut(k-1) і 3(k-1) пересилання. Кожний виклик Permut(k-1) супроводжується передачею масиву S як параметра-значення, що за часом еквівалентне n пересиланням. Тому мають місце співвідношення

$$C(k) = kC(k-1) + nk + 3(k-1),\ C(1) = 0,\ \text{звідки}\ C(n) = (n+3)n!$$

Оцінимо тепер розмір пам'яті, необхідної для алгоритму. Оскільки S - параметр-значення, при кожному виклику Permut резервується n комірок (байтів) для S, а при виході з цієї процедури пам'ять звільнюється. Рекурсивне застосування Permut призводить до того, що ланцюжок вкладених викликів має максимальну довжину (n - 1):

Permut(n) --> Permut(n-1) --> ... --> Permut(2) --> Permut(1)

Тому дані цього ланцюжка потребують n^2 - n комірок пам'яті, тобто алгоритм потребує пам'ять розміром $O(n^2)$. Кількість перестановок - елементів множини Permut(S) дорівнює n!. Тому наш алгоритм "витрачає" C(n)/n! = O(n) дій для породження кожної перестановки. Зрозуміло, такий алгоритм неможна називати ефективним. (Інтуїція нам підказує, що ефективний алгоритм повинен генерувати кожну перестановку за фіксовану, незалежну від n кількість дій.) Джерело неефективності очевидне: використання S як параметра-значення. Це дозволило нам зберегти незмінним масив S при поверненні з рекурсивного виклику для того, щоб правильно переставити елементи, готуючи наступний виклик.

Розглянемо інший варіант цього алгоритму. Використаємо S як глобальну змінну. Тоді при виклику Permut S буде змінюватись. Отже, при виході з рекурсії масив S треба поновлювати, використовуючи обернену перестановку!

```
Program All_permutations;
  Const n = 4;
   Type Sequence = array[1..n] of char;
    Var S : Sequence;
        Buf : char;
          i : Integer;
Procedure WriteSequence;
     begin
      For i := 1 to n do Write(S[i]); Writeln
      end;
Procedure Permut(k : Integer);
      Var j : integer;
Procedure Swap(i, j : Integer);
     begin  Buf := S[j]; S[j] := S[i]; S[i] := Buf  end;
    Begin
      if k > 1 then Permut(k - 1);
```

```
      For j := k - 1 downto 1 do begin
       Swap(j, k); {Пряма перестановка}
       WriteSequence; Permut(k - 1);
       Swap(k, j) {Обернена перестановка}
      end
    End;
  Begin
    {Генерація вихідного набору S}
    WriteSequence;  Permut(n)
   End.
```

Тепер оцінка C(n) виходить з співвідношень

$$C(k) = kC(k-1) + 3(k-1),\ C(1) = 0 ,\ \text{т.е. } C(n) = O(n!)$$

Цікаво, що цей варіант не потребує і пам'яті розміру $O(n^2)$ для збереження масивів. Необхідна тільки пам'ять розміру O(n) для збереження значення параметра k.

6.Приклади рекурсивних описів процедур і функцій.

Декілька прикладів рекурсивних процедур, розглянутих у цьому пункті, допоможуть кращому засвоєнню техніки застосування рекурсії. Ми також побачимо переваги і недоліки рекурсивних описань.

Приклад 8. Повернені послідовності.

Розглянемо послідовність V(n), що задана рекурентно:

1.$V(0) = V_0$, $V(1) = V_1$, ..., $V(k) = V_k$;

2.V(n+k) = F(V(n+k-1), ... V(n+1), V(n), n)

Таке визначення задає послідовність V(n) фіксуванням перших її декількох членів і функції F, що обчислює кожний наступний член, виходячи з попередніх. Рекурентне визначення "дослівно переводиться" в функцію, що обчислюється рекурсивно:

```
Function V(n: Integer) : Integer;
  Begin
   Case n of
     0:  V := V0;
      . . . . .
     k:  V := Vk
    else V := F(V(n+k-1), ... V(n+1), V(n), n)
   end
End;
```

Один з прикладів повернених послідовностей - послідовність Фібоначчі, програму обчислення якої ми вже розглядали. Ось її рекурсивна версія:

```
Function RF(n: Integer) : Integer;
  Begin
   Case n of
```

```
  0,1: RF := 1
  else RF := RF(n-1) + RF(n-2)
 end
End;
```

Нажаль, ця красива і прозора версія вкрай неефективна: її складність визначається з співвідношення $T_{rf}(n) = T_{rf}(n-1) + T_{rf}(n-2) + 1$.

Функція $T_{rf}(n)$ дорівнює функції RF(n). Неважко показати, що $T_{rf}(n) = RF(n) > 2^{n-2}$ при n>5.

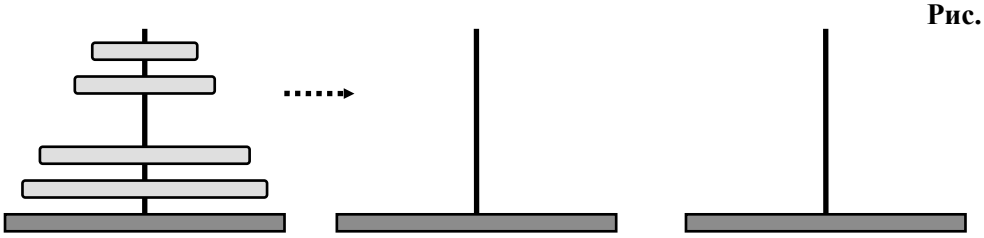
Таким чином, складність обчислення RF(n) експоненціальна. Складність же ітеративного алгоритму обчислення Fib лінійна. У загальному випадку ситуація точно така ж: можна побудувати ітеративний алгоритм обчислення поверненої послідовності лінійної складності, рекурсивна ж версія при k>=1 має експоненціальну складність.

Приклад 9. Ханойські башти.

Класичний приклад застосування рекурсії для описання ефективного алгоритму - задача про ханойські башти. На перший з трьох стержнів, закріплених вертикально на підставці, насаджено декілька кілець різних діаметрів так, що кільце меншого діаметра лежить на кільці великого діаметра. Два інших стержня порожні (рис. А).

У задачі треба переставити всі кільця з 1-го стержня на 2-ий за декілька кроків. Кожний крок - це перестановка верхнього кільця одного стержня на верхнє кільце іншого стержня з дотриманням правила: діаметр кільця, що переставляється повинен бути менше, ніж діаметр кільця, на яке здійснюється перестановка.

Рис. А



1 стержень **2 стержень** **3 стержень**

Нехай N - кількість кілець на стержні, I - номер кільця, з якого здійснюється перестановка і J - номер кільця, на який кільця треба переставити. Змінні N,I,J - параметри процедури HanojTower, керуючої перестановками. Крок розв'язку - процедура Step з параметрами I, J.

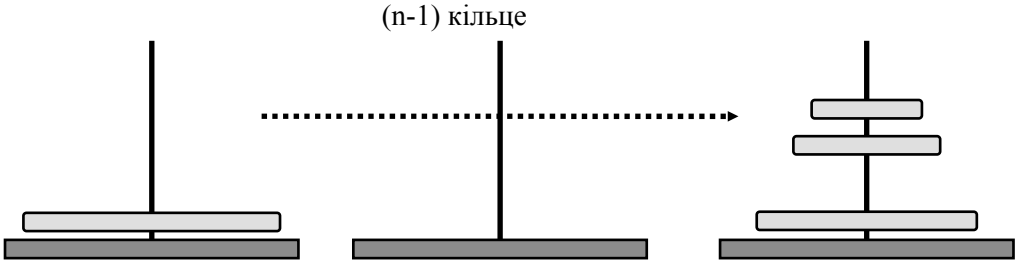
HanojTower (N,I,J) - процедура, що переставляє N кілець з I-того стержня на J-тий.

Step (I, J) - процедура , що переставляє 1-не кільце з I-того стержня на J-тий.

Відмітимо, що якщо I і J - номера 2-стержней, то 6-I-J - номер третього стержня.

Припустимо, що ми перемістили N-1 кільце з I-того стержня на 6-I-J стержень. (рис. B)

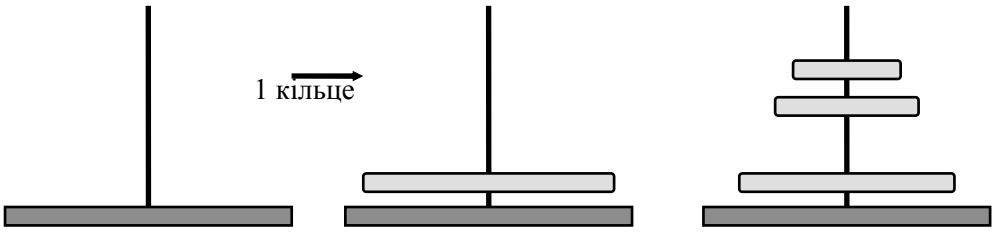
Рис. B



I стержень **J стержень** **6-I-J стержень**

Тоді можна перемістити кільце з стержня I на J. (рис. C)

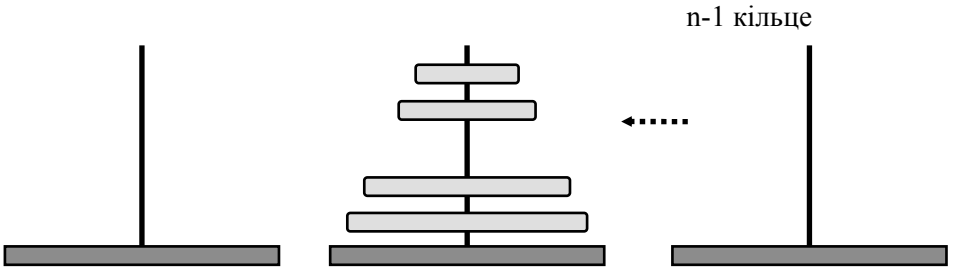
Рис. C



I стержень **J стержень** **6-I-J стержень**

Відмітимо тепер, що на стержні J лежить кільце з найбільшим діаметром, тобто цей стержень можна використовувати без порушення обмежень, пов'язаних з величинами діаметрів. Тому можна тепер переставити всю піраміду з N-1 кільця з стержня 6-I-J на J, (рис. D) і задача розв'язана!

Рис. D



I стержень **J стержень** **6-I-J стержень**

Відмітивши, що при N = 1 задача розв'язується за допомогою процедури Step (I, J), опишемо процедуру HanojTower (N,I,J):

```
Procedure HanojTower(N, I, J: Integer);
  Begin
    If N = 1
     then Step(I, J)
     else begin
      HanojTower(N-1, I, 6-I-J);
      Step(I, J);
      HanojTower(N-1, 6-I-J, J)
     end
    End;
```

Процедуру Step(I, J) можна реалізувати, використовуючи представлення даних у масиві Rings[1..N, 1..3] і графічну візуалізацію переміщення кілець.

Визначимо складність алгоритму за часом C(n) у термінах кількості кроків (викликів Step), вписавши рекурентне співвідношення: $C(n) = 2C(n-1) + 1$

Легко тепер показати, що $C(n) = 2^n - 1$. Доведено, що ця кількість кроків є мінімально можливою, тому наш алгоритм оптимальний.

Приклад 10. Лінійні діафантові рівняння. Перерахувати всі невід'ємні цілі розв'язки лінійного рівняння $a_1x_1 + a_2x_2 + ... + a_nx_n = b$ з цілими додатними коефіцієнтами.

Як і в попередніх прикладах, опишемо алгоритм рекурсивно, здійснивши зведення вихідної задачі до задачі меншого розміру.

Перепишемо вихідне рівняння в виді: $a_1 x_1 + a_2 x_2 + ... + a_{n-1} x_{n-1} = b - a_n x_n$

Організуємо перебор всіляких значень x_n, при яких права частина $b - a_n x_n > 0$. $x_n = 0, 1, ... y$, де $y = b \text{ div } a_n$. Тоді перші n-1 компонент розв'язка $(x_1, ..., x_{n-1}, x_n)$ вихідного рівняння - розв'язок рівняння $a_1 x_1 + a_2 x_2 + ... + a_{n-1} x_{n-1} = b - a_n x_n$.

Таким чином ми звели розв'язок вихідного рівняння до розв'язка y+1 рівняння з n-1 невідомим, і, отже, можемо реалізувати алгоритм рекурсивно. Визначимо умови виходу з рекурсії:

при b = 0 існує єдиний розв'язок - (0, 0, ..., 0)

при n = 1 якщо b mod a_1 = 0 то x_1 = b div a_1 інакше розв'язків немає.

Таким чином, виходити з рекурсивних обчислень треба в двох (крайніх) випадках. Ми встановили і параметри процедури: n і b.

```
Procedure Solution(n, b : integer);
    Var i, j, y, z : Integer;
  Begin
   If  b = 0
    then begin
     For j := 1 to n do X[j] := 0; WriteSolution end
     else If (n = 1) and (b mod a[1] = 0)
       then begin X[1]:= b div a[1]; WriteSolution end
      else If n > 1
        then begin
          z := a[n]; y := b div z;
          For i := 0 to y do begin
          X[n] := i; Solution(n - 1, b - z*i)
       end
   end
 End;
 Program AllSolutions;
    Const n = 4;
     Type Vector = array[1..n] of Integer;
      Var a, X : Vector; b : Integer;  i : Integer;
   {Procedure WriteSolution друкує  розв'язок X[1..n] }
   {Procedure Solution}
   Begin
    {Введення масиву коефіцієнтів a[1..n] і св. члена b}
    Solution(n, b)
   End;
```

Приклад 11. Підведення числа до натурального степеня. Піднести дійсне число a у натуральний степінь n.

Раніше ця задача була розв'язана методом послідовного домноження результату на a. Однак її можна розв'язати ефективніше, якщо застосувати метод половинного ділення степеня n. Саме: $a^n = (a^{n/2})^2$. Оскільки число n не обов'язково парне, формулу треба уточнити:

a^n = $(a^{n \text{ div } 2})^2$ * $a^{n \text{ mod } 2}$. Доповнивши визначення a^n визначенням a^1 = a і замінивши домноження на $a^{n \bmod 2}$ розбором випадків парний-непарний, отримаємо:

```
Function CardPower(a : Real; n : Integer) : Real;
var b : Real;
```

```
Begin
  If n = 1
   then CardPower := a
    else begin
      b := Sqr(CardPower(a, n div 2));
      If n mod 2 = 0
       then CardPower := b
       else CardPower := a*b
    end
  End;
```

Доведіть, що функція CardPower використовує не більш $O(\log_2 n)$ множень і піднесень у квадрат.

Зрозуміло, функцію CardPower можна реалізувати без циклів і рекурсій, за допомогою формули: при $a > 0$ $a^n = e^{n * \ln a}$, але цей метод не придатний наприклад, якщо в натуральний степінь підноситься багаточлен.

Приклад 12. Перетин зростаючих послідовностей.

Нехай $A = [a_1, a_2,...,a_n]$ і $B = [b_1, b_2, ... , b_m]$ - дві зростаючі числові послідовності. Перетином цих послідовностей називається зростаюча послідовність $C = [c_1, c_2,..., c_k]$, що складається з тих і тільки тих чисел, які належать обом послідовностям. Треба знайти $C = A \cap B$.

Зведемо задачу до кількох підзадач, більш простих, ніж вихідна. Поділимо для цього вихідні послідовності на частини

$A = [a_1] \cup A_2$; $B = [b_1] \cup B_2$, де $A_2 = [a_2,...,a_n]$, $B_2 = [b_2,...,b_n]$

Тоді $A \cap B = (a_1 \cup A_2) \cap (b_1 + B_2) = a_1 \cap b_1 \cup a_1 \cap B_2 \cup A_2 \cap b_1 \cup A_2 \cap B_2$

(Дужки в позначеннях одноелементних множин опущені)

Так як послідовності A і B зростають, маємо:

Якщо $a_1 = b_1$ то $A \cap B = a1 \cap b1 \cup A2 \cap B2 = a1 \cup A2 \cap B2$

Якщо $a1 < b1$ то $A \cap B = b1 \cap A2 \cup A2 \cap B2 = A2 \cap B$

Якщо $a1 > b1$ то $A \cap B = a1 \cap B2 \cup A2 \cap B2 = A \cap B2$

Ці відношення зводять вихідну задачу до задач менших розмірів, тому їх можна використовувати для рекурсивного описання обчислень.

Процедура Intersect пошуку перетину залежать від параметрів і та j - номерів початкових елементів підпослідовностей A[i..m] і B[j..n], представлених масивами A[1..m] і B[1..n]. Знайдений елемент перетину роздруковується.

```
Procedure Intersect(i, j : Integer);
   Begin
    If (i <= m) and (j <= n)
     then If a[i] = b[j]
      then begin Write(a[i]); Intersect(i+1, j+1) end
```

```
  else If a[i] < b[j]
    then Intersect(i+1, j)
    else Intersect(i, j+1)
End;
```

7. Переваги і недоліки рекурсивних алгоритмів.

Розглянуті приклади показують, що застосування рекурсивних означень само по собі не приводить до хороших розв'язків задач. Так, задачі прикладів 6, 8, 12 допускають більш ефективні і достатньо прості ітеративні розв'язки. У прикладі 12 це продемонстровано особливо наочно. У прикладах 6 і 8 рекурсія моделює простий цикл, який витрачає час на обчислення, що зв'язані з керуванням рекурсивними викликами і пам'ять на динамічне збереження даних. Те ж саме можна сказати і про приклад 11. Ми вже розглядали ітеративну версію метода ділення навпіл в інших прикладах.

Безсумнівним достоїнством алгоритмів з прикладів 8, 11, 12 є їх простота і обгрунтованість. При програмуванні ми по-суті одночасно обгрунтовували правильність алгоритму. У прикладах 7, 9,10 розглядались комбінаторні задачі, розв'язки яких не зводяться до простого застосування циклу, а потребують перебору варіантів, природнім чином, що керується рекурсивно. Ітеративні версії розв'язків цих задач складніші по управлінню. Можна сказати, що основна обчислювальна складність комбінаторної задачі полягає не в реалізації алгоритму її розв'язку, а в самому методі розв'язку.

Пізніше ми побачимо, як рекурсія в поєднанні з іншими методами використовується для побудови ефективних програм.

8.Задачі і вправи.

Вправа І.

1.Опишіть процедури додавання і множення n*n матриць, що складаються з дійсних чисел. За допомогою цих процедур складіть програму обчислення матриці B =AX + XA + E за даними матрицями A і X.

2.Опишіть процедуру пошуку найбільшого і найменшого елементів у масиві. За допомогою цієї процедури складіть програму, що визначає, чи співпадають найбільший і найменший елементи двох масивів A [1..n] і B [1..n].

3.Опишіть процедури пошуку середнього за величиною елемента у масиві A [1..2n+1] і середнє арифметичне елементів цього масиву. За допомогою цих процедур складіть програму, яка порівнює середнє за величиною і середнє арифметичне у масиві A [1..99].

4.Опишіть процедуру пошуку найбільшого і найменшого елементів у масиві A[1..n]. За допомогою цієї процедури складіть програму, що визначає координати вершин найменшого прямокутника зі сторонами, паралельними осям координат, який містить в собі множину точок $T_1(X_1,Y_1)$, ... , $T_n(X_n,Y_n)$.

5.Опишіть процедуру перевірки розкладення натурального числа в суму двох квадратів. Складіть програму, яка вибирає з даного масиву ті і тільки ті числа, які розкладаються в суму двох квадратів.

6.Опишіть функцію перевірки простоти числа. Опишіть функцію перевірки числа на розкладність в суму виду $1+2^a$. Складіть програму, яка вибирає з даного масиву чисел ті і тільки ті його елементи, які задовольняють обом властивостям.

7.Опишіть процедуру перетворення цілого числа з десяткової системи в р-ічну систему числення. Опишіть процедуру перетворення правильного десяткового дробу в р-ічний з заданою кількістю розрядів. Складіть програму перетворення довільного дійсного числа з 10-вої в р-ічну систему числення.

8.Опишіть процедуру пошуку найменшого елемента в рядку матриці. Опишіть функцію перевірки на максимальність даного елемента в стовпці матриці. Складіть програму пошуку сідлової точки матриці.

Вправи II.

1.Опишіть функцію f(x) - найбільший власний дільник числа x. Складіть програму пошуку всіх власних дільників числа N.

2.Опишіть функцію y = arcsin(x). Складіть програму розв'язку рівняння sin(ax+b) = c.

3.Опишіть функцію f(x) = max (t*sin(t)).

$$t \in [0,x]$$

Складіть Програму табулювання функції f(x) на відрізку [a,b].

4.Опишіть функцію f(x) - число, яке отримується з натурального числа x перестановкою цифр у зворотному порядку. Складіть програму, яка роздруковує симетричні числа з масиву натуральних чисел.

5.Опишіть функції L(x,y) і Φ(x,y), які визначені формулами $L(x,y) = |x + iy|$, $\Phi(x, y) = \arg(x + iy)$. Складіть програму переведення послідовності комплексних чисел $z_1, ..., z_n$, $z_j = x_j + iy_j$, в тригонометричну форму.

Вправи III.

1.Реалізуйте алгебру раціональних чисел як бібліотеку процедур і функцій, що містить арифметичні дії і відношення порядку на множині раціональних чисел.

2.Реалізуйте алгебру комплексних чисел як бібліотеку процедур і функцій, що містить арифметичні дії над комплексними числами в алгебраїчній формі.

3.Реалізуйте алгебру - скінчене поле GF(p) лишків за простим модулем p як бібліотеку процедур і функцій, що містить арифметичні дії над лишками. GF(p) = (0, 1, ..., p-1)

4.Реалізуйте алгебру 3 x 3 матриць над полем дійсних чисел як бібліотеку процедур і функцій, що містить додавання, віднімання, множення матриць, обертання невироджених матриць, ділення матриці на невироджену матрицю, множення матриці на число, обчислення визначника матриці.

Задачі.

1.Перечислити всі вибірки по K елементів у масиві A з N елементів. Масив містить перші N натуральних чисел: A[i] = i. Знайти їх кількість і оцінити складність алгоритму.

2.Перечислити всі підмножини множини (масиву A) з задачі 1. Знайти їх кількість і оцінити складність алгоритму.

3.Дано масив A[1..n] цілих додатних чисел і ціле число b. Перечислити всі набори індексів j1, j2, ..., jk такі, що A[j1] + A[j2] + ... + A[jk] = b. Оцінити складність алгоритму.

4.Дано масив A[1..n] цілих чисел і ціле число b. Перечислити всі набори індексів j1, j2, ... , jk такі, що A[j1] + A[j2] + ... + A[jk] = b. Оцінити складність алгоритму.

5.Відома в теорії алгоритмів функція Акермана A(x, y) визначена на множні натуральних чисел рекурсивно:

A(0, y) = y + 1,

A(x, 0) = A(x - 1, 1),

A(x, y) = A(x - 1, A(x, y - 1));

а)Опишіть функції A(1, y), A(2, y), A(3, y), A(4, y) явним чином;

б)Реалізуйте рекурсивну програму обчислення A(x, y);

в) Реалізуйте програму обчислення A(x, y) без рекурсії.

ШВИДКІ АЛГОРИТМИ СОРТУВАННЯ І ПОШУКУ.

У теперішньому параграфі ми встановимо нижню грань обчислювальної складності задачі сортування масиву і завершимо вивчення алгоритмів сортування і пошуку в масивах, розглянув так звані швидкі сортування.

1. Нижня оцінка часу задачі сортування масиву за числом порівнянь.

Теорема. Будь-який алгоритм сортування масиву, що використовує тільки порівняння і перестановки, у гіршому випадку потребує мінімум $O(n \log_2 n)$ порівнянь.

Доведення. Нехай $a_1, a_2, ..., a_n$ - підмножина елементів, вибраних з абстрактної множини U, на якій визначено відношення лінійного порядку. (Це означає, що будь-які два різних елемента a і b з U порівняльні: a < b або b < a.) Ця підмножина породжує n! перестановок, і кожна перестановка може бути вихідною послідовністю (входом) для застосування алгоритму сортування. Іншими словами, задача сортування розміру n має n! різних входів. Позначимо цю множину входів через Per(A). Наприклад, при n = 3 підмножина {a, b, c} породжує 3! = 6 різних входів: [a b c], [a c b], [b a c], [b c a], [c a b], [c b a].

Оскільки множина U абстрактна, алгоритм сортування здійснює перестановки, що сортують вхід B, виходячи тільки з перевірки умов, що містять порівняння двох елементів A, прямо або опосередковано вибраних з масиву B по їх індексах. Нехай S - деяка підмножина з Per(A).

Умова, що містить порівняння a[i] ? a[j], на одній частині множини S дасть результат - True, для іншої - False. Нехай T(S), F(S) - ці підмножини. Тоді |S| = |T(S)| + |F(S)|. Отже, кількість елементів принаймні в одній з підмножин T(S), F(S) не менше |S|/2. Нехай C(1,B), C(2,B), ..., C(k, B) - послідовність значень умов, що перевіряються алгоритмом послідовно на вході B довжини k і ConSeq(k, B) - множина входів, які генерують ту ж послідовність умов. Тоді

$\cup$(ConSeq(k, B), B $\in$Per(A)) = Per(A)

і, отже, існує такий вхід B, для якого

$|ConSeq(k, B)| \geq |Per(A)|/2^k$ (*)

Нехай B і B' - два (різних) входи і P - деяка перестановка. Легко показати, що з $B \neq B'$ слідує $P(B) \neq P(B')$. Тому послідовності перестановок, що сортирують різні входи, будуть різними. Але кожна послідовність перестановок однозначно визначена відповідною послідовністю умов. Отже, кожному входу B відповідає своя послідовність значень умов C(1,B), C(2,B), ..., C(m, B),що перевіряється в процесі виконання програми сортування на цьому вході. Таким чином,

ConSeq(m, B) = {B}. (**)

З (*), (**) випливає існування такого входу B, для якого виконана нерівність $|Per(A)|/2^m \leq 1$, тобто $n!/2^m \leq 1$ або $m \geq \log(n!)$. За формулою Стірлінга

$\log(n!) \approx \log_2((n/e)^n) = n(\log_2 n - 1) = O(n \log_2 n)$.

Ми показали, що для будь-якого алгоритму сортування за допомогою порівнянь і перестановок існує такий вхід, на якому $m >= O(n \log_2 n)$

Таким чином, функція складності в гіршому випадку алгоритму сортування C(n) обмежена знизу величиною $O(n \log_2 n)$. Кінець доведення.

Спам'ятаймо, що алгоритм сортування вставками має таку ж саму за порядком величини функцію C(n). Таким чином, ця оцінка може бути досягнена.

Питання про оцінку функції кількості перестановок M(n) залишилося відкритим. Можна припустити, що кожній перестановці передує деяке порівняння. Тоді загальна кількість перестановок, що здійснюється алгоритмом, не перевищує C(n) і, отже, ефективні алгоритми сортування повинні мати складність $O(n \log_2 n)$ як за порівняннями, так і по перестановках. Нижче ми розглянемо деякі такі алгоритми.

2. Швидкі алгоритми сортування: Сортування деревом.

Алгоритм сортування деревом TreeSort по суті є поліпшенням алгоритму сортування вибором. Процедура вибору найменшого елементу вдосконалена як процедура побудови т.н. дерева, що сортує. Дерево, що сортує - це структура даних, у якій представлений процес пошуку найменшого елемента методом попарного порівняння елементів, що стоять поряд. Алгоритм сортує масив у два етапи.

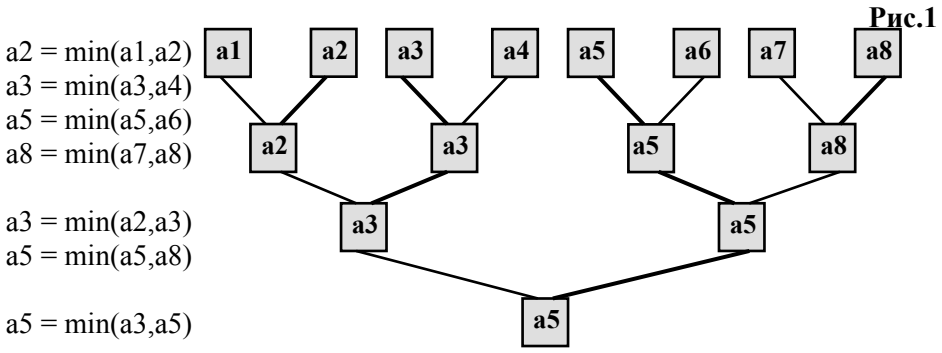
I етап: побудова дерева, що сортує;

II етап: просівання елементів по дереву, що сортується.

Розглянемо приклад:

Нехай масив A складається з 8 елементів (мал. 1, 1-ий рядок). Другий рядок складається з мінімумів елементів першого рядка, що стоять поряд. Кожний наступний рядок складається з мінімумів елементів попереднього, що стоять поряд.

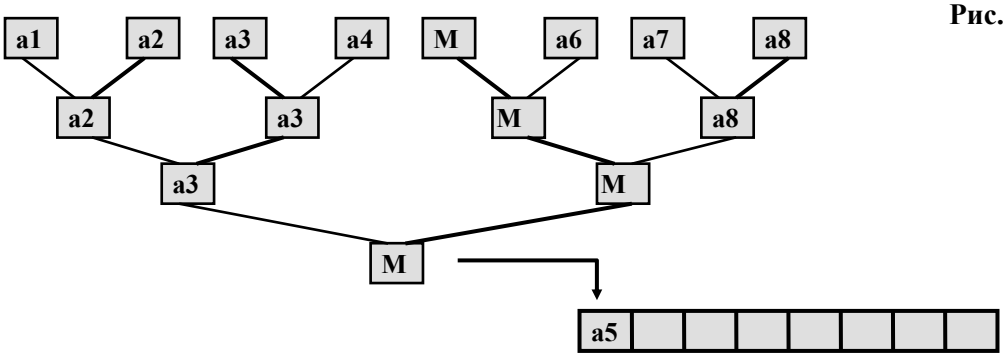
Рис.1



Ця структура даних називається сортуючим деревом. У корені дерева, що сортує, знаходиться найменший елемент. Крім цього, в дереві побудовані шляхи елементів масиву від листя до відповідного за величиною елемента вузла - розгалуження. (На рис. 1 шлях мінімального елемента a5 - від листа a5 до кореня відмічений товстою лінією.)

Коли дерево побудовано, починається етап просівання елементів масиву по дереву. Мінімальний елемент пересилається у вихідний масив B і всі входження цього елемента в дереві замінюються на спеціальний символ M.

Рис. 2



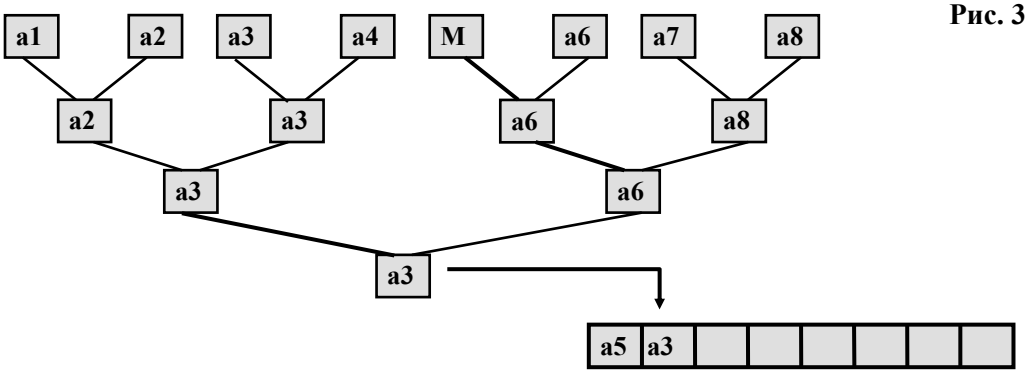
B

1 2 ...

8

Потім здійснюється просівання елемента уздовж шляху, відміченого символом M, починаючи з листка, сусіднього з M (На рис. 2 зверху-вниз) і до кореня. Крок просівання - це вибір найменшого з двох елементів, що зустрілись на шляху до кореня дерева і його пересилання у вузол, відмічений M. Просівання 2-го елемента показано на мал. 3 (Символ M більше, ніж будь-який елемент масиву.)

Рис. 3



a6 = min(M, a6)
a6 = min(a6, a8)
a3 = min(a3, a6) B
b2 := a3 1 2 ...
8

Просівання елементів відбувається до тих пір, поки весь вихідний масив не буде заповнений символами M, тобто n разів:

For I := 1 to n do begin
Відмітити шлях від кореня до листка символом M;
Просіяти елемент уздовж відміченого шляху;
B[I] := корінь дерева
end;

Обгрунтування правильності алгоритму очевидне, оскільки кожне чергове просівання викидає у масив B найменший з елементів масиву A, що залишився.

Дерево, що сортує можна реалізувати, використовуючи або двомірний масив, або одномірний масив ST[1..N], де N = 2n-1 (див. наступний розділ). Оцінимо складність алгоритму в термінах M(n), C(n). Перш за все відмітимо, що алгоритм TreeSort працює однаково на всіх входах, так що його складність у гіршому випадку співпадає зі складністю в середньому.

Припустимо, що n - степінь 2 ($n = 2^l$). Тоді дерево, сортує має l + 1 рівень (глибину l). Побудова рівня I потребує $n / 2^I$ рівнянь і пересилань. Таким чином, I-ий етап має складність

$C_1(n) = n/2 + n/4 + ... + 2 + 1 = n - 1$, $M_1(n) = C_1(n) = n - 1$

Для того, щоб оцінити складність II-го етапу $C_2(n)$ і $M_2(n)$ відмітимо, що кожний шлях просівання має довжину l, тому кількість порівнянь пересилань при просіванні одного елемента пропорційна l. Таким чином, $M_2(n) = O(l\ n)$, $C_2(n) = O(l\ n)$.

Оскільки $l = \log_2 n$, $M_2(n)=O(n \log_2 n))$, $C_2(n)=O(n \log_2 n)$, але $C(n) = C_1(n) + C_2(n)$, $M(n) = M_1(n) + M_2(n)$. Так як $C_1(n) < C_2(n)$, $M_1(n) < M_2(n)$, остаточно отримаємо оцінки складності алгоритму TreeSort за часом:

$M(n) = O(n \log_2 n)$, $C(n) = O(n \log_2 n)$,

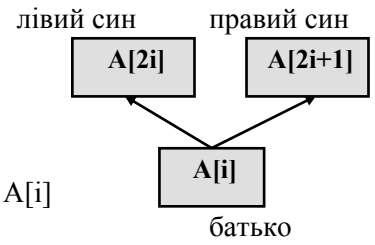
У загальному випадку, коли n не є степенем 2, дерево, що сортує, будується дещо по іншому. “Зайвий” елемент (елемент, для якого немає пари) переноситься на наступний рівень. Легко бачити, що при цьому глибина дерева, що сортує, дорівнює $[\log_2 n] + 1$. Удосконалення алгоритму II етапу очевидне. Оцінки при цьому міняють лише мультиплікативні множники. Алгоритм TreeSort має суттєвий недолік: для нього потрібна додаткова пам’ять розміром 2n - 1.

Пірамідальне сортування.

Алгоритм пірамідального сортування HeapSort також використовує представлення масиву у виді дерева. Цей алгоритм не потребує додаткових масивів; він сортує "на місці". Розглянемо спочатку метод представлення масиву у виді дерева:

Нехай A [1 .. n] - деякий масив. Співставимо йому дерево, використовуючи наступні правила:

1.A[1] - корінь дерева ;
2.Якщо A[i] - вузол дерева і $2i \le n$,
то A[2*i] - вузол - "лівий син" вузла A[i]
3.Якщо A[i] - вузол дерева і $2i + 1 \le n$,
то A[2*i+1] - вузол - "правий син" вузла A[i]

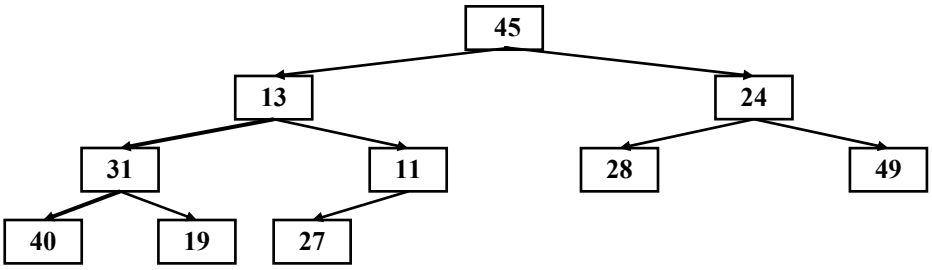


Правила 1 - 3 визначають у масиві структуру дерева, причому глибина дерева не перевищує $[\log_2 n] + 1$. Вони ж задають спосіб руху по дереву від кореня до листя. Рух уверх задається правилом 4:

4.Якщо A[i] - вузол дерева і i > 1
то A[i mod 2] - вузол - "батько" вузла A[i];

Приклад1. Нехай A = [45 13 24 31 11 28 49 40 19 27] - масив. Відповідне йому дерево має вид:

Рис. 5



Зверніть увагу на те, що всі рівні дерева, за винятком останнього, повністю заповнені, останній рівень заповнений зліва і індексація елементів масиву здійснюється зверху-вниз і зліва-направо. Дерево впорядкованого масиву задовольняє наступним властивостям:

$$A[i] \le A[2*i],\ A[i] \le A[2*i+1],\ A[2*i] \le A[2*i+1].$$

Як це не дивно, алгоритм HeapSort спочатку будує дерево, яке задовольняє прямо протилежним співвідношенням

$$A[i] \ge A[2*i],\ A[i] \ge A[2*i+1] \qquad (6)$$

а потім міняє місцями A[1] (найбільший елемент) і A[n].

Як і TreeSort, алгоритм HeapSort працює в два етапи:

I.Побудова дерева, що сортує;

I I.Просівання елементів по дереву, що сортує.

Дерево, що представляє масив, називається сортуючим, якщо виконуються умови (6). Якщо для деякого і ця умова не виконується, будемо говорити, що має місце (сімейний) конфлікт у трикутнику і (мал.4).

І на І-ому, і на ІІ-ому етапах елементарна дія алгоритму полягає у розв'язку сімейного конфлікту: якщо найбільший з синів більше, ніж батько, то переставляються батько і його син (процедура ConSwap).

У результаті перестановки може виникнути новий конфлікт в тому трикутнику, куди переставлений батько. Таким чином можна говорити про конфлікт (роду) у піддереві з коренем у вершині і. Конфлікт роду розв'язується послідовним розв'язком сімейних конфліктів - проходом по дереву зверху-вниз. (На мал.5 шлях розв'язку конфлікту роду у вершині 2 відмічений.) Конфлікт роду розв'язаний, якщо прохід завершився (i > n div 2) або у результаті перестановки не виникнув новий сімейний конфлікт. (процедура Conflict)

```
Procedure ConSwap(i, j : Integer);
    Var b : Real;
    Begin
     If a[i] < a[j] then begin
       b := a[i]; a[i] := a[j]; a[j] := b
     end
    End;
Procedure Conflict(i, k : Integer);
      Var j : Integer;
    Begin
      j := 2*i;
      If j = k
       then ConSwap(i, j)
       else if j < k then begin
          if a[j+1] > a[j] then j := j + 1;
           ConSwap(i, j); Conflict(j, k)
        end
      End;
```

І етап - побудови дерева, що сортує - оформимо в виді рекурсивної процедури, використовуючи визначення:

Якщо ліве і праве піддерева T(2i) і T(2i+1) дерева T(i) , що сортують, то для перебудови T(i) необхідно розрішити конфлікт роду в цьому дереві.

Procedure SortTree(i : Integer);

```
begin
 If i <= n div 2 then begin
   SortTree(2*i);
   SortTree(2*i+1);
    Conflict(i, n)
  end
end;
```

На II-ому етапі - етапі просівання - для k от n до 2 повторюються наступні дії:

1.Переставити A[1] і A[k];

2.Побудувати дерево, що сортує початкового відрізка масиву A[1..k-1], усунувши конфлікт роду в корені A[1]. Відмітимо, що 2-а дія потребує введення в процедуру Conflict параметра k.

```
Program HeapSort;
  Const n = 100;
    Var A : Array[1..n] of real;
        k : Integer;
   {процедури ConSwap, Conflict, SortTree, введення, виведення}
   Begin
    { введення }
    SortTree(1);
    For k := n downto 2 do begin
     ConSwap(k, 1); Conflict(1, k - 1)
    end;
   { виведення }
   End.
```

Оцінимо складність алгоритму у термінах C(n), M(n). Оскільки кожна перестановка пов'язана з порівняннями в процедурі ConSwap, $M(n) \le C(n)$. Далі, в процедурі Conflict порівняння передує виклику ConSwap, тому загальне число порівнянь не перевищує подвоєного числа викликів ConSwap. Отже, C(n) можна отримати, оцінивши кількість звернень до ConSwap.

Складність I-го етапу $C_1(n)$ задовольняє співвідношенню $C_1(n) \le 2C_1(n / 2) + C_0(n)$ де $C_0(n)$ - складність процедури Conflict(1, n). Для $C_0(n)$ маємо: $C_0(n) \le C_0(n/2) + 1$, оскільки виклик Conflict(i, n) містить один (рекурсивний) виклик Conflict(2i, n) і ланцюжок викликів обривається при i > n/2. Тому $C_0(n) \le c_0 \log_2 n$. Таким чином,

$C_1(n) \le 2\, C_1(n / 2) + c_0 \log_2 n$.

Нехай k - деяке число. Тоді

$C_1(n) <= 2^k C_1(n/2^k) + c_0 (2^{k-1} \log_2(n/2^{k-1}) + ... + \log_2(n/2) \log_2 n)$

Оскільки $C_1(1) = 0$, маємо $C_1(n) <= c_0 (\log_2 n + 2 \log_2(n/2) + ... (n/2) \log_2(2))$. Замінимо $\log(n/2^i)$ на $\log_2 n$, посиливши нерівність. Отримаємо $C_1(n) <= (c_0 \log_2 n)(1 + 2 + 4 + ... n/2)$

Сума в дужках не перевищує n. Тому

$$C_1(n) <= c_0 n \log_2 n \quad (7)$$

Арифметичний цикл II-го етапу виконується n - 2 разів і кожне повторення містить ConSwap(k, 1) і Conflict(1, k - 1). Тому його складність $C_2(n)$ можна оцінити нерівністю $C_2(n) <= (1+C_0(n-1)) + (1+C_0(n-2)) + ... + (1+C_0(2))$ Замінимо $C_0(i)$ на $C_0(n-1)$, посиливши нерівність. Отримаємо

$$C_2(n) <= n + n\, C_0(n-1) = c_1 n \log_2 n \quad (8)$$

Остаточно $C(n) = C_1(n) + C_2(n) = (c_0+c_1)\, n \log_2 n$, або

$$C(n) = O(n \log_2 n) \quad (9)$$

Ми показали, що пірамідальне сортування (з точністю до мультиплікативної константи) оптимальне: його складність співпадає (з тією ж точністю) з нижньою оцінкою задачі.

Швидке сортування Хоара.

Удосконаливши метод сортування, оснований на обмінах, К.Хоар запропонував алгоритм QuickSort сортування масивів, що дає на практиці відмінні результати і дуже просто програмується. Автор назвав свій алгоритм швидким сортуванням.

Ідея К.Хоара полягає у наступному:

Виберемо деякий елемент x масиву A випадковим способом;

Розглядаємо масив у прямому напрямку (i= 1, 2, ...), шукаючи у ньому елемент A[i] не менший, ніж x;

3.Розглядаємо масив у зворотному напрямку(j = n, n-1,..), шукаючи у ньому елемент A[j] не більший, ніж x;

4.Меняємо місцями A[i] і A[j];

Пункти 2-4 повторюємо до тих пір, поки i < j;

В результаті такого зустрічного проходу початок масиву A[1..i] і кінець масиву A[j..n] опиняються розділеними “бар’єром” x: $A[k] \le x$ при $k < i$, $A[k] \ge x$ при $k > j$, причому на розділ ми затратимо не більш n/2 перестановок. Тепер залишилось зробити ті ж дії з початком і кінцем масиву, тобто застосувати їх рекурсивно!

Таким чином, процедура Hoare, що описана нами, залежить від параметрів k і m - початкового і кінцевого індексів відрізку масиву, що оброблюється.

```
Procedure Swap(i, j : Integer);
     Var b : Real;
   Begin
     b := a[i]; a[i] := a[j]; a[j] := b
   End;
Procedure Hoare(L, R : Integer);
     Var left, right : Integer;
         x : Integer;
   Begin
    If L < R then begin
```

```
      x := A[(L + R) div 2];                    {вибір бар'єра x}
      left := L; right := R ;
      Repeat                                          {зустрічний прохід}
       While A[left] < x do left := Succ(left);   {перегляд уперед}
       While A[right] > x do right := Pred(right); {перегляд назад}
        If left <= right then begin
         Swap(left, right);                       {перестановка}
         left := Succ(left);  right := Pred(right);
        end
       until left > right;
       Hoare(L, right);                                {сортуємо початок}
       Hoare(left, R)                                  {сортуємо кінець}
     end
    End;
Program QuickSort;
  Const n = 100;
    Var A : array[1..n] of Integer;
{ процедури Swap, Hoare, введення і виведення }
Begin
  Inp;  Hoare(1, n);  Out
 End.
```

Найбільш тонке місце алгоритму Хоара - правильне опрацювання моменту закінчення руху покажчиків left, right. Помітьте, що в нашій версії переставляються місцями елементи, що дорівнюють x. Якщо в циклах While замінити умови (A[left] < x) і (A[right] > x) на (A[left] <= x) і (A[right] >=x), при x = Max(A) індекси left і right пробіжать весь масив і побіжать далі! Ускладнення ж умов продовження перегляду погіршить ефективність програми.

Аналіз складності алгоритму в середньому, що використовує гіпотезу про рівну ймовірність всіх входів, показує, що

$C(n) = O(n \log_2 n)$, $M(n) = O(n \log_2 n)$.

У гіршому випадку, коли в якості бар'єрного вибирається, наприклад, максимальний елемент підмасиву, складність алгоритму квадратична.

3. Пошук k-того в масиві. Пошук медіани масиву.

З задачею сортування тісно пов'язана задача находження k-того за величиною елемента у масиві. З її частковим випадком - пошук 1-го (мінімального) і n-того елемента (максимального) - ми вже знайомі. Аналогічно можна розв'язати задачу пошуку 2-го і (n - 1)-го за величиною елементів, однак кількість порівнянь зростає вдвічі. Перш, ніж розглянути задачу у загальному виді, приведемо необхідне визначення.

К-тим за величиною елементом послідовності називається такий елемент b цієї послідовності, для якого в послідовності існує не більш, ніж k-1 елемент, менший, ніж b і не менше k елементів, менше або рівних b.

Наприклад, у послідовності {2, 1, 3, 5, 4, 2, 2, 3 } четвертим за величиною буде 2.

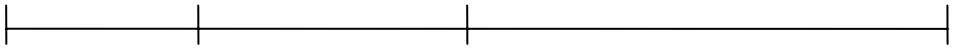
Іншими словами, якщо відсортувати масив, то k-тий елемент опиниться на k-тому місці (b = A[k]). Можна очкувати, що найбільш складним буде алгоритм пошуку елемента при k = n div 2 - тобто елемента, рівновіддаленого від кінців масиву. Цей елемент назвемо медіаною масиву.

Очевидний розв'язок задачі пошуку k-того елемента - повне або часткове (до k) сортування масиву A. Наприклад, алгоритм TreeSort можна перервати після просівання k елементів. Тоді $C(n) = O(n + k \log_2 n)$

Однак ефективний на практиці розв'язок можна отримати, якщо застосувати ідею К.Хоара розділення масиву бар'єром. Насправді, аналіз метода розділення елементів по бар'єру в процедурі Hoare показує, що після її закінчення мають місце співвідношення

Right < Left, при i < Left A[i] <= x, при i > Right A[i] >= x

Таким чином, відрізок First .. Last розбиває на 3 частини:



First	Right	Left	Last
A[i] <= x	A[i] = x	A[i] >= x	

Подальший пошук, отже, можна організувати так:

Якщо k <= right
 то шукати k-тий елемент в A[First .. right]
 інакше якщо k <= left
  то k-тий елемент дорівнює x
  інакше шукати k - тий елемент в A[left + 1 .. Last]

```
Procedure HoareSearch(First, Last : Integer);
      Var l, r : Integer; x : Integer;
     Begin
      If First = Last
        then Write(A[k]) { залишився 1 елемент }
        else If First < Last then begin
         x := A[k]; {вибір бар'єра x}
          l := First; r := Last;
          Repeat {зустрічний прохід}
           While A[l] < x do l := Succ(l);
           While A[r] > x do r := Pred( r );
            If l <= r then begin
             Swap(l, r);
```

```
        l := Succ(l); r := Pred( r );
       end
      until l > r;
  If k <= r
  then HoareSearch(First, r) {1-а частина масиву}
  else If k <= l
   then Write(x) {елемент знайдений}
   else HoareSearch(l+1, Last) {3-ая частина масиву}
  end
End;
```

Автор методу пропонує у якості бар'єрного елемента вибирати A[k], хоча це не дає ніяких переваг перед вибором середнього або випадкового елемента.

Як і алгоритм швидкого сортування, цей алгоритм у гіршому випадку неефективний. Якщо при кожному розділу 1-а частина масиву буде містити 1 елемент, алгоритм витратить O(kn) кроків. Однак в середньому його складність лінійна (незалежно від k). Зокрема, навіть при пошуку медіани A[n div 2]

$$C_{cp}(n) = O(n),\ M_{cp}(n) = O(n)$$

В завершення цього розділу відзначимо, що існує лінійний за складністю у гіршому випадку алгоритм пошуку k-того елемента, однак його теоретичні переваги переростають у практичні при дуже великих значеннях n.

4. Метод "розділяй і володій".

Один з найбільш відомих методів проектування ефективних алгоритмів - метод "розділяй і володій" полягає у зведенні задачі, що розв'язується, до рішення однієї або кількох більш простих задач.

Якщо спрощення задачі полягає в зменшенні її параметрів, таке зведення дає рекурсивний опис алгоритму. Зокрема, зменшення параметра може означати зменшення кількості даних, з якими працює алгоритм.

Характерна ознака методу - зведення до декількох задач рівних за складністю підзадач. Ефективність отриманого алгоритму залежить від, по-перше, від кількості дій, затрачених на зведення задачі до підзадач, по-друге - від балансу складностей підзадач.

Класичний приклад застосування методу - бінарний пошук у впорядкованому масиві. Масив розбивається на дві рівні частини. Використовуючи потім два порівняння, алгоритм або знаходить елемент, або визначає ту половину, в якій його треба шукати - тобто зводить задачу розміром n до задачі розміром n/2. Цей же прийом використовується в алгоритмах Хоара сортування і пошуку, в алгоритмі піднесення числа в натуральну степінь ($8, приклад 11), у деяких інших раніше сформульованих задачах. Розглянемо відомий розв'язок задачі пошуку мінімального і максимального елемента в масиві.

Нехай A[1..n] - числовий масив. Треба знайти Max(A) і Min(A). Припустимо, що $n = 2^k$. Таке припущення дає можливість завжди ділити масиви, отримані навпіл.

Опишемо основну процедуру MaxMin. Нехай S - деяка множина і |S| = m. Розіб'ємо S на рівні за числом елементів частини: $S = S_1 \cup S_2$, $S_1 \cap S_2 = \varnothing$, $|S_1| = m/2$, $|S_2| = m/2$. Знайдемо Max1, Min1 як результат MaxMin(S_1) і Max2, Min2 як результат MaxMin(S_2).

Потім Max := Max(Max1, Max2); Min := Min(Min1, Min2);

```
procedure MaxMin(L, R : Integer Var Max, Min : Real);
     Var Max1, Min1, Max2, Min2 : Real;
    Begin { L, R - межі індексів масиву A }
     If R - L = 1
      then If A[L] > A[R] { вибір Max, Min за одне порівняння}
       then begin
         Max := A[l]; Min := A[R] end
       else begin
         Max := A[R]; Min := A[L] end
       else begin
        C := (R + L - 1) div 2;
        MaxMin(L, C, Max1, Min1);
        MaxMin(C+1, R, Max2, Min2);
        If Max1 > Max2 then Max := Max1 else Max := Max2;
        If Min1 < Min2 then Min := Min1 else Min := Min2
       end
    End;
```

Нехай C(n) - оцінка складності процедури MaxMin в термінах порівнянь. Тоді, очевидно

C(2) = 1,

C(n) = 2 C(n/2) + 2 при n > 2

Звідси C(n) = 2 (n/4 + n/8 + ... + 1) + n/2 = 2(n/2 - 1) + n/2;

C(n) = 3/2 n - 2

Для порівняння відзначимо, що алгоритм пошуку Max і Min методом послідовного перегляду масиву потребує 2n -2 порівнянь. Таким чином, застосування метода "розділяй і володій" зменшило часову складність задачі в фіксоване число раз. Зрозуміло, істинна причина поліпшення - не в застосуванні рекурсії, а в тому, що максимуми і мінімуми двох елементів масиву, що стоять поруч, відшукуються за одне порівняння! Метод дозволив просто і природно описати обчислення.

Довжина ланцюжка рекурсивних викликів в алгоритмі дорівнює $\log_2 n$ -1, і в кожній копії використовується 4 локальних змінних. Тому алгоритм використовує допоміжну пам'ять об'єму O($\log_2 n$), що розподіляється динамічно. Це - плата за економію часу.

5. Метод цифрового сортування.

Іноді при розв'язані задачі типу задачі сортування можна використовувати особливості типу даних, що перетворюються, для отримання ефективного алгоритму. Розглянемо одну з таких задач - задачу про обернення підстановки.

Підстановкою множини 1..n назвемо двомірний масив A[1..2, 1..n] виду

1	2	.....	n-1	n
j1	j2	.....	j n-1	j n

в якому 2-ий рядок містить всі елементи відрізка 1. .n. Підстановка B називається оберненою до підстановки A, якщо B отримується з A сортуванням стовпців A у порядку зростання елементів 2-ого рядка з наступною перестановкою рядків. Треба побудувати алгоритм обчислення оберненої підстановки. З визначення випливає, що стовпець [i, j_i] підстановки A треба перетворити в стовпець [j_i , i] і поставити на j_i-те місце в підстановці B.

```
{Type NumSet = 1..n;
Substitution = array[ 1..2, NumSet] of NumSet; }
Procedure Reverse ( Var A, B : Substitution);
   Begin
    For i := 1 to n do begin
      B[A[2, i], 2] := i; B[A[2, i], 1] := A[2, i]
    end
   End;
```

Складність процедури Reverse лінійна, оскільки тіло арифметичного циклу складається з двох операторів присвоювання. Між тим стовпці підстановки відсортовані.

Аналогічна ідея використовується в так званому сортуванні вичерпуванням (або цифровому сортуванні), що застосовується для сортування послідовностей слів (багаторозрядних чисел). Для кожної букви алфавіту алгоритм створює структуру даних "черпак", в яку пересилаються слова, що починаються на цю букву. В результаті перегляду послідовність виявляється відсортованою по першій букві. Далі алгоритм застосовують рекурсивно до кожного "черпака", сортуючи його по наступним буквах.

6. Здачі і вправи.

1.Дано масив A[1..900] of 1..999. Знайти трьохзначне число, що не знаходиться у цьому масиві. Розглянути два варіанта задачі:

а) Допоміжні масиви в алгоритмі не використовуються;

б) Використовування допоміжних масивів дозволено.

2.Реалізувати алгоритм TreeSort, застосувавши метод вкладення дерева в масив. Для цього використати допоміжний масив B.

3.Реалізувати ітеративну версію алгоритму HeapSort:

а)Замінити рекурсію в процедурі SortTree арифметичним циклом For...downto, що оброблює дерево за рівнями, починаючи з нижнього;

б)Замінити рекурсію в процедурі Conflict ітераційним циклом, керуючим змінною i. { i := 2i або i := 2i + 1 }.

4.Реалізувати ітеративну версію алгоритму процедури HoareSeach, замінивши рекурсію ітераційним циклом.

5-6. Застосувати для розв'язку задач 1-2 параграфа 8 ідею Хоара.

7.Реалізувати алгоритм "тернарного" пошуку елемента в упорядкованому масиві, поділяючи ділянку пошуку на 3 приблизно рівні частини. Оцінити складність алгоритму у термінах C (n). Порівняти ефективності бінарного і тернарного пошуку.

8.Реалізувати алгоритм пошуку максимального і мінімального елементів у масиві (процедура MaxMin) для довільного n.

9.Нехай A[1..n] - масив цілих чисел, упорядкований за зростанням (A[i] < A[i+1]). Реалізувати алгоритм пошуку такого номера i, що A[i]= i методом "розділяй і володій". Оцінити складність алгоритму.

10-12. Застосувати для розв'язку задач 3-5 параграфа 8 метод "розділяй і володій".

13.Реалізувати алгоритм процедури Reverse "на місці", формуючи обернену підстановку в масиві A.

ГРАФІКА В СИСТЕМІ ПРОГРАМУВАННЯ ТР-6.

Характерна особливість систем програмування, реалізованих на персональних комп'ютерах - використання розширень мови засобами обробки графічної інформації. В системі програмування Turbo Pascal ці засоби зосереджені в модулі GRAPH.

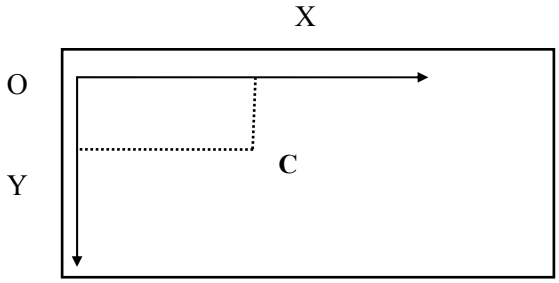
Модуль (UNIT) - це сукупність (бібліотека) описань констант, типів даних, змінних, процедур і функцій. Кожний модуль представляє собою по суті самостійну програму на Паскалі: він може містити декілька операторів, що виконуються перед викликом програми, і здійснюють всю необхідну ініціалізацію. Засоби, що описані в модулі, можна використовувати в програмі. Модуль можна компілювати окремо. Для того, щоб включити модулі в програму, необхідно описати їх у спеціальному розділі описань Uses. Синтаксичне описання цього розділу має вид:

Uses <список модулів >; {необов'язковий параметр}

Всі описання усередині модуля як правило пов'язані. Наприклад, стандартний зовнішній модуль CRT містить засоби, що підтримують взаємодію програми з екраном монітора в алфавітно-цифровому режимі (текстовий екран). Модуль GRAPH призначений для взаємодії програми з екраном у графічному режимі (графічний екран).

1. Графічний екран. Види графічних адаптерів.

Елементарною одиницею інформації, що виводиться в графічному режимі на екран, є точка, покрашена в деякий колір. Ця точка називається пікселом. Позиція піксела на екрані визначається в декартовій системі координат (мал. 1). Відмітимо незвичний напрямок осі OY: вона направлена з лівого верхнього кута екрана униз.



Таким чином, кожний піксел описується трьома параметрами:
(X, Y, C), де C - колір піксела.
X - ціле число з інтервалу [0 ... GetMaxX];
Y - ціле число з інтервалу [0 ... GetMaxY];
C - ціле число з інтервалу [0 ... GetMaxColors]

де GetMaxX, GetMaxY, GetMaxColors - константи, що описані в модулі GRAPH.

Діапазони допустимих значень X, Y, C визначаються технічними характеристиками апаратури комп'ютера, операційною системою і наладкою програми на один з можливих режимів використовування модуля GRAPH - ініціалізацією графіки. Детальні відомості про ініціалізацію графіки знаходяться в документації системи програмування.

Система програмування Turbo-Pascal в MS-DOS підтримує, зокрема, наступні графічні режими:

Відеоадаптер	GetMaxX	GetMaxY	GetMaxColor	Драйвер
CGA	320	200	4 з 16	CGA.BGI
MCGA	320	200	16 з 256	CGA.BGI
EGA	640	200	16 з 256	EGAVGA.BGI
EGA	640	350	16 з 256	EGAVGA.BGI
VGA	640	480	16 з 256	EGAVGA.BGI

Настройка програми на один з графічних режимів виконується автоматично оператором InitGraph, якщо відповідний драйвер установлений на комп'ютері.

InitGraph(< гр. адаптер >, < режим гр.др >, < путь >);

Перехід у текстовий режим здійснюється оператором CloseGraph.

Таким чином, програма, що використовує графіку, має вид:

```
Program MyGraph;
    Uses Graph;
     Var grDriver, grMode, errCode: Integer;
     .............
   Begin
     grDriver := Detect;
     InitGraph(grDriver, grMode, '< шлях до драйвера >'); { приклад шляху: D:\PasSys6\BGI }
     .............
   End.
```

Константа Detect визначена в модулі GRAPH. Її значення дорівнює 0. Detect вказує системі на необхідність підключення драйвера графіки, відповідного типу адаптера дисплея.

Графічний модуль GRAPH містить декілька десятків (більш 50) процедур і функцій, кожна з яких описана в документації системи і доступна програмісту з розділу HELP головного меню.

Тому ми приведемо тільки деякі відомості про GRAPH.

Константи:

процедури Bar3D;

процедури PutImage;
відсікання, кольорів, стиля ліній і ширини;
стиля тексту;
графічні драйвери і графічні режими драйверів;
шаблонів заповнення;
.....

Типи:

ArcCoordsType
FillPatternType
FillSettingsType
LineSettingsType
PaletteType
PointType
TextSettingsTtype
.....

Процедури і функції:

Графічні примітиви:

PutPixel, GetPixel, GetX, GetY,
Line, LineTo, Move, Bar, DrawPoly, Rectangle, ...
Arc, Circle, Ellipse, ...

Обробка кольорів:

GetColor; GetPixel, GetBkColor, ...
SetColor, SetBkColor, SetPalette,...

Установка стиля малювання: SetLineStyle, SetFillStile, ...

Обробка тексту: OutText, OutTextXY, TextWidth, SetTextStyle, ...

Обробка графічного екрана: SetActivePage, PutImage, GetImage, SetViewPort,...

Процедура PutPixel(X, Y, C), наприклад, фарбує точку (X,Y) екрана в колір C. Одночасно графічний курсор поміщається в цю точку.

Процедура Line(X1,Y1, X2,Y2) малює відрізок AB кольором, що встановлений попередньо A = (X1, Y1), B = (X2, Y2).

2. Задача побудови графіка функції.

Як приклад застосування графічних засобів розглянемо задачу побудови графіка функції y = f(x). Побудова графіка функції потребує як мінімум:

Побудову вікна з оформленою системою координат;
Побудову кривої y = f(x) у вікні;
Оформлення необхідних надписів.

а) Оформлення системи координат.

Вікно системи координат визначається параметрами:

(Un, Vn) - координати лівого верхнього кута вікна;

(Uk, Vk) - координати правого нижнього кута вікна;

(Uo, Vo) - координати початку координат;

Mx, My - масштаби системи координат зо x і y.

Всі параметри повинні бути визначені в комп'ютерній системі координат (в координатах пікселів). При цьому(Un, Vn), (Uk, Vk) - константи (нерухоме вікно), а (Uo, Vo), M обчислюються, виходячи з даних, що вводяться користувачем. На практиці користувач частіше всього вказує межі змін змінних x, y - (Xn, Xk), (Yn, Yk). Значення M, Uo, Vo необхідно обчислювати перед побудовою системи координат.

Для простоти ми будемо вважати, що Xn<0<Xk, Yn<0<Yk. Тоді початок системи координат попадає у вікно графіка. У протилежному випадку треба програмувати декілька спеціальних варіантів розміщення осей координат.

Mx = [(Uk - Un)/(Xk - Xn)],

My = [(Vk - Vn)/(Yk - Yn)],

Uo = Uk - [Mx*Xk], Vo = Vn + [My*Yk];

Використовуючи ці формули, напишемо потрібну процедуру:

```
Procedure WinGraf(Xn, Xk, Yn, Yk: Real;       Var Mx, My, Uo, Vo : Integer);
   Begin
    { обчислення параметрів }
    Mx := Round((Uk - Un)/(Xk - Xn));    My := Round((Vk - Vn)/(Yk - Yn));
    Uo := Uk - Round(Mx*Xk);             Vo := Vn + Round(My*Yk);
    { малювання системи координат }
     SetColor(LightGray);
     Bar(Un, Vn, Uk, Vk); { вікно графіка }
     SetColor(Green); { малюємо зеленим кольором}
     SetLineStyle(0, 0, ThickWidth);    {Малюємо товсті суцільні осі}
     Line(Un, Vo, Uk, Vo); { ось OX }
     Line(Uo, Vn, Uo, Vk); { ось OY }
     SetColor(Red);
     Circle(Uo, Vo, 2); {початок координат - невелике коло}
     SetLineStyle(0, 0, 1); {Малюємо тонкі суцільні відмітки1 }
     Line(Uo+Mx, Vo-4, Uo+Mx, Vo+4); { відмітка 1 на OX }
     Line(Uo-4, Vo-My, Uo+4, Vo-My); { відмітка 1 на OY }
     SetTextStyle(0, 0, 2);
     OutTextXY(Uo + Mx - 7, Vo + 16, '1'); { малюємо 1 }
     OutTextXY(Uo - 24, Vo - My - 7 , '1'); { малюємо1 }
      { Подальші прикрашення вікна графіка }
    End;
```

б) Побудова кривої y = f(x) у вікні;

При поточечній побудові кривої на кожному кроці необхідно обчислювати наступні значення (x, y), обчислювати істинні координати (u, v) піксела - образа точки (x, y) на екрані і виводить піксел на екран.

На побудову графіка витрачається більша частина часу. Тому доцільно апроксимувати дугу кривої y = f(x) хордою, вибираючи вузли апроксимації з деяким кроком DrawStep (наприклад, DrawStep = 10 (пікселів)).

Таким чином, процедура малювання графіка має вид:

```
Procedure FuncDraw;
    Const DrawStep = 10;
      Var i: Integer;
          x, y: Real;
          u, v: Integer;
          ArgStep: Real;
          StepNum: Integer;
     Begin
      SetColor(Blue); {колір кривої}
      SetLineStyle(0, 0, NormWidth); {товщина нормальна}
      ArgStep := DrawStep / Mx; {крок зміни x}
      StepNum := Round((Uk - Un) / DrawStep); {кількість вузлів}
      { обчислення початкової точки графіка }
      x := Xn ; u := Un;
      y := sqrt(Abs(x))*sin(2*x);  v := Round(Vo - My*y);
       MoveTo(u, v);
       { цикл малювання апроксимуючою ламаною }
       For i := 1 to StepNum do begin
          x := x + ArgStep; u := u + DrawStep;
          y := sqrt(Abs(x))*sin(2*x); v := Round(Vo - My*y);
          LineTo(u,v);
        end
     End;
```

Робота з текстами і введення-виведення в графічному екрані.

Введення-виведення інформації в режимі графіки суттєво відрізняються від відповідних процедур в алфавітно-цифровому режимі. Виведення тексту на екран здійснюють процедури OutText, OutTextXY.

OutText(TxtStr: String) виводить строку на пристрій виведення (графічний екран) в точку - позицію графічного курсора.

OutTextXY(X, Y: Integer; TxtStr: String) виводить строку на пристрій виведення (графічний екран) в точку (X, Y).

Крім цих процедур, в модулі GRAPH описані процедури і функції, підтримуючі різні стилі виведення. Так, наприклад:

Функції TextHeight(TxtStr:String), TextWidth(TxtStr:String) повертають відповідно висоту і ширину рядка в пікселах.

Процедура SetTextStyle(Font, Direction:Word; CharSize:Word) установлює стиль виведення тексту в графічному режимі, тобто визначає шрифт, стиль тексту і коефіцієнт збільшення символу. Як приклад опишемо процедуру WinText виведення надписів-коментарів у вікно графіка функції.

```
Procedure WinText;
    Begin
      SetColor(LightMagenta);
      SetTextStyle(0, 1, 1);
      OutTextXY(15,15, 'Function Graphics Window');
      SetColor(Cyan);
      SetTextStyle(0, 0, 1);
      OutTextXY(460,15, 'Production of SL.XXI');
      SetTextStyle(0, 0, 2);
      OutTextXY(28, 390, 'Y = Sqrt(Sin(|x|)');
     End;
```

Введення інформації в графічному режимі засобами модуля GRAPH не підтримується. Тому для організації введення необхідно скористуватися або стандартними процедурами, або процедурами модуля CRT (бібліотека алфавітно-цифрового режиму). На практиці це означає посимвольне введення інформації, формування рядка введених символів і перетворення типу рядка до типа даного, необхідного програмі.

Для введення символу використовується функція ReadKey - читання символу з клавіатури. Для відображення введеного символу на екрані (відлуння) використовується одна з графічних процедур виведення. Суперпозиція цих дій має вид Symbol := ReadKey; OutTextXY(x0, y0, Symbol). Для більш повної імітації процедури Read точка введення інформації на екрані звичайно відмічається зображенням курсора, який здвигається на одну позицію вправо при кожному введенні символу.

```
Procedure ReadGrStr(x, y: Integer; var Str: String);
    Const CursW = 8; CursH = 6; {ширина і висота курсора}
      Var Symbol: Char; x1, y1: Integer;
    Begin
      x1 := x + CursW; y1 := y + CursH;
      SetColor(Green); {колір курсора }
      Rectangle(x, y, x1, y1);
      Str := ''; Symbol := ReadKey; {ініціалізація }
      While Ord(Symbol) <> 13 do begin
      SetColor(LightGray); {Колір вікна графіка}
      Bar(x, y, x1, y1); {Віддалити курсор}
```

```
      OutTextXY(x, y, Symbol); {Вивести символ}
       x := x + CursW; x1 := x + CursW;
       Str:= Str + Symbol; {Формування вихідного рядка}
       SetColor(Green); {Вивести курсор}
       Rectangle(x, y, x1, y1);
       Symbol := ReadKey {Рахувати символ}
      end
    End;
```

Відмітимо, що ніяких редагуючих дій процедура ReadGrStr не підтримує. Для реальних програм її треба удосконалити.

Числа вводяться як рядки, перетворюються потім системною процедурою Val. Val(Str: String; var v {компонент Text}; var Code: Integer);

Змінна Code містить номер помилки формату. За її допомогою можна контролювати правильність числового формату введеного рядка.

3. Рекурсивні описи в графіку.

Використовуючи графічні можливості комп'ютера і рекурсивні описи малюнків, можна будувати на екрані зображення красивих мозаїк і кривих.

Розглянемо метод отримання одного з таких малюнків - зображення квадрата Кантора. (Квадрат Кантора відомий в математиці як приклад ніде не щільної множини додатної площі (міри)).

Квадрат Кантора - це фігура, що отримана шляхом вирізання деяких ділянок з початкового суцільного квадрату. Для того, щоб побудувати квадрат Кантора, необхідно:

1.Розбити суцільний (пофарбований в один колір) квадрат на 9 рівних квадратів вертикальними і горизонтальними лініями (3 на 3);

2.Вирізати (пофарбувати в інший колір) середній з 9-ти маленьких квадратів.

3.Застосувати (рекурсивно) дії 1, 2, 3, до 8-ми маленьких суцільних квадратів, що залишилися.

```
Program KantorSet;
   Uses Graph, Crt;
  Const Xmin = 0; Ymin = 0; { розміри квадрата }
        Xmax = 486; Ymax = 486;
        N = 6; { глибина рекурсії }
   Var grDriver, GrMode: Integer; ch: Char;
 Procedure Picture(k: Integer; x0, y0, x3, y3: Integer);
     Var x, y, dx, dy: Integer; i, j: Integer;
   Begin
    If k <> N
     then begin
       dx := (x3-x0) div 3; dy := (y3-y0) div 3;
```

```
        x := x0;
       For i := 0 to 2 do begin
       y := y0;
       For j := 0 to 2 do begin
         If (i = 1) and (j = 1)
           then Bar(x, y, x+dx, y+dy) {вирізаємо квадрат}
           else Picture(k+1, x, y, x+dx, y+dy);
         y := y + dy  end;
      x := x + dx  end;
     k := k + 1
    end
End;
  Begin
    grDriver := Detect;
    InitGraph(grDriver, grMode, 'E:\lvov\pascal\bgi');
     SetBkColor(Blue);
     Picture(1, Xmin, Ymin, Xmax, Ymax);
     ch := ReadKey
   End.
```

Аналогічно можна побудувати і інші симпатичні мозаїки.

4. Робота з сторінками і фрагментами.

Графічне зображення, що виводиться на екран, зберігається в спеціальній області оперативної пам'яті комп'ютера, яка називається відео пам'яттю. Розмір пам'яті, що відводиться під зображення одного графічного екрана, залежить від графічного режиму. Ця частина пам'яті називається сторінкою відеопам'яті (відеосторінкою). Так, у режимі EGA (640 * 350, 16 кольорів) сторінка відеопам'яті займає приблизно 110 Kb. У цьому режимі, якщо відеопам'ять має 256 Kb, визначені дві сторінки - з номерами 0 і 1.

Модуль Graph містить засоби, що підтримують роботу з декількома відеосторінками. Наявність декількох сторінок дозволяє оперувати заздалегідь заготовленими малюнками і їх фрагментами. З цією метою введені поняття видимої сторінки і активної сторінки. Видима сторінка - це сторінка, яка виводиться на екран. На активній сторінці формуються зображення, що виконуються операторами графіки.

Найбільш використовувані засоби, підтримуючі роботу з сторінками і їх фрагментами, коротко описані нижче:

SetVisualPage(Page:Word)- візуалізує сторінку з номером Page;

SetActivePage(Page: Word) - активізує сторінку з номером Page;

GetMem(P, Size) - резервує пам'ять розміру Size і встановлює на неї покажчик P; {Поняття покажчика розглядається в розділі 12}

ImageSize(x1, y1, x2, y2: Integer):Word - повертає розмір відеопам'яті прямокутної області екрана;

GetImage(x1, y1, x2, y2:Integer; var BitMap) - зберігає образ вказаної області в буфері;

PutImage(x, y: Integer; var BitMap; BitBit: Word) - поміщає (двійковий) образ із буфера на екран. Параметр BitBit визначає спосіб накладення образа на активну сторінку;

```
Program ImageExample;
    Uses Graph, Crt;
     Var grDriver, GrMode: Integer;
         Ch :Char;
          P :Pointer; {покажчик на образ у пам'яті}
          Size :Word; {розмір образа}
        Begin
         grDriver := Detect;
          InitGraph(grDriver, grMode, 'E:\lvov\pascal\bgi');
          SetGraphMode(1); { Розмір екрана = 640 * 350 }
          {Приклад для SetVisualPage, SetActivePage}
          SetBkColor(Blue);
          SetActivePage(0); {Малюємо на сторінці 0. На екрані сторінка 0}
          Rectangle(40, 40, 140, 140);  OutTextXY(50, 90, 'First Page');
          Ch := ReadKey;  SetActivePage(1); { Малюємо на сторінці 1. На екрані сторінка
           0}
          SetColor(Magenta);
          Circle(90, 90, 50);
          OutTextXY(45, 90, 'Second Page');
          SetVisualPage(1); {Виводимо на екран стор. 1}
          Ch := ReadKey;    SetVisualPage(0); {Виводимо на екран стор. 0}
          Ch := ReadKey;    SetVisualPage(1); {Виводимо на екран стор. 1}
          Ch := ReadKey;
           {Приклад для GetImage, ImageSize, і PutImage}
           Size := ImageSize(40, 40, 140, 140); {Установили розмір малюнка}
           GetMem(P, Size); {Резервуємо пам'ять, на яку установлений покажчик P }
           SetActivePage(0);
           GetImage(40, 40, 140, 140, P^); {Пересилаємо малюнок в область пам'яті -
значення P}
           Ch := ReadKey;
           SetActivePage(1); {на сторінці 1}
           PutImage(400, 200, P^, NormalPut); {Виводимо на стор.1 малюнок}
           Ch := ReadKey;
           ClearDevice;
           CloseGraph;
```

```
end.
```

5.Задачі і вправи.

1.Описати процедуру оформлення системи координат, у якій точка початку координат (Uo,Vo) може виходити за межі вікна графіка. Осі координат у цьому випадку повинні співпадати з відповідними межами вікна графіка.

2.Удосконалити процедуру FuncDraw виведення графіка функції y = f(x) обробкою випадків, коли крива виходить за межі вікна системи координат.

3.Змінити процедуру FuncDraw для виведення на екран графіка функції, заданої параметрично: x = fi(t), y = psi(t), A <= t <= B.

4.Написати програму побудови графіка функції в полярній системі координат.

5.Реалізувати процедури лінійних геометричних перетворень (перенос, поворот, осьова і центральна симетрії, гомотетія).

6.Використовуючи процедури впр.5, написати програму, що демонструє геометричні перетворення плоских геометричних фігур (зокрема - трикутника).

7.Удосконалити процедуру ReadGrStr, перетворивши її в редактор рядка з діями Insert, Delete, BackSpace за допомогою відповідних клавіш.

8.Змінити процедуру ReadGrStr, перетворивши її в редактор дійсного числа.

9.Змінити процедуру ReadGrStr, перетворивши її в редактор цілого числа.

10.Написати програму зображення мозаїки, яка визначена правилами:

а. У центрі квадрата розміром A*A малюється коло радіусом A/2;

б. Квадрат розбивається на 4 рівних (під)квадрата горизонтальною і вертикальною лініями.

в. Дії 1-3 застосовуються рекурсивно до кожного з 4-х квадратів.

Задачі обчислювальної геометрії.

11. Дано набір з n точок площини, заданих координатами $A_1(X_1, Y_1)$, $A_2(X_2, Y_2)$, ..., $A_n(X_n, Y_n)$. Знайти таку точку S(X, Y) на площині, сума відстаней від якої до даних точок набору найменша. Проілюструвати розв'язок на малюнку.

12. Дано набір з n точок площини, заданих координатами: $A_1(X_1, Y_1)$, $A_2(X_2, Y_2)$, ..., $A_n(X_n, Y_n)$. Вибрати з цього набору точки, що знаходяться в вершинах випуклого багатокутника найменшої площини і який містить всі точки набору. Проілюструвати розв'язок на малюнку.

13. Дано набір з n точок площини, заданих координатами: $A_1(X_1, Y_1)$, $A_2(X_2, Y_2)$, ..., $A_n(X_n, Y_n)$. Побудувати коло найменшого діаметра, який містить всі точки набору. Проілюструвати розв'язок на малюнку.

14. Дано набір з n точок площини, заданих координатами: $A_1(X_1, Y_1)$, $A_2(X_2, Y_2)$, ..., $A_n(X_n, Y_n)$. Побудувати прямокутник найменшої площини, який містить всі точки набору. Проілюструвати розв'язок на малюнку.

15. Випуклий n-кутник заданий набором вершин $A_1(X_1, Y_1)$, $A_2(X_2, Y_2)$, ..., $A_n(X_n, Y_n)$.

Розбиття цього багатокутника на трикутники діагоналями, що не перетинаються називається тріангуляція. Вартістю тріангуляції називається сума довжин діагоналей розбиття. Знайти тріангуляцію багатокутника найменшої вартості. Проілюструвати розв'язок на малюнку.

СКЛАДНІ ТИПИ ДАНИХ: ЗАПИСИ І ФАЙЛИ.

1. Складні типи даних у мові Pascal.

Ми вже отримали уявлення про деякі типи даних, які використовуються при програмуванні на Паскалі. Прості типи даних або заздалегідь визначені як стандартні, або визначаються в програмі.

Стандартні типи	Типи, що визначаються програмістом
Integer;	Тип, що перераховується;
Real;	Скалярний тип.
Char:	
Boolean.	

З поняттям простого типу пов'язані:

- ім'я типу;
- множина допустимих значень типу;
- набір операцій, що визначені на типі;
- набір відношень, що визначені на типі;
- набір функцій, що визначені або набувають значення на типі.

З даних простих типів можна конструювати дані складних типів. Єдиним (поки) прикладом складних типів є регулярні типи (значення - масиви). З поняттям складного типу пов'язані:

- ім'я типу;
- спосіб об'єднання даних у дане складного типу;
- спосіб доступу до даних, що утворюють складне дане типу;

Так, визначення

```
Type Vector = array [1..N] of Integer;
Var X : Vector;
```

задає N цілих чисел, що об'єднані в єдине ціле - масив з іменем X. Доступ до компоненти масиву здійснюється за його іменем X і значенням індексного виразу: X[i].

Крім регулярних типів, у мові визначені ще деякі складні типи. Кожне таке визначення задає свій, характерний механізм об'єднання даних у єдине ціле і механізм доступу до даних - компонент складного даного.

Це - комбінований тип (записи), файловий тип (файли), множинний тип (множини) і посилальний тип (посилання).

2. Записи.

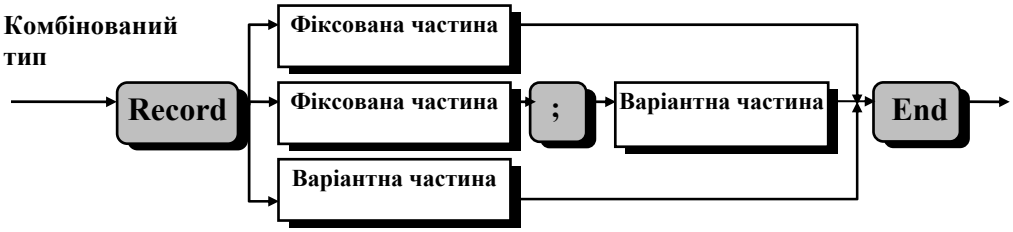
Значеннями так званого комбінованого типу даних є записи. Комбінований тип задає образ структури об'єкта - даного цього типу, кожна частина якого (поле) може мати зовсім різні характеристики.

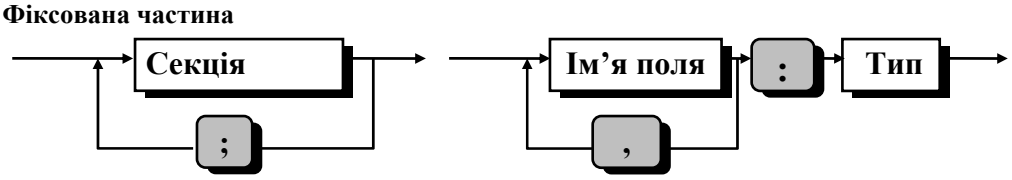
Таким чином, запис - це набір різнотипних даних, що об'єднані спільним іменем. Більш формально, запис містить визначене число компонент, що називаються полями.

У визначенні типу запису задається ім'я і тип кожного поля запису:

<комбінований тип >::= Record < список описань полів > End
<список полів>::= <фіксов. част.> | <фіксов. част.>;<варіант. част.> | <варіант. част.>
<фіксована частина >::= <секція запису > {,<секція запису >}
< секція запису >::= <ім'я поля >{,<ім'я поля >}:< тип > < пусто >

Синтаксис записів, що містять варіантну частину - записів з варіантами - ми визначимо нижче. Відповідні синтаксичні діаграми записів з варіантами:





Приклади.

Приклад 1

Type Complex = Record

```
  Re, Im : Real
 end;
Var z1, z2 : Complex;
```

Приклад 2

```
Type Name = array [1..15] of Char;
          Student = Record
                      F1,F2,F3 : Name;
                      Day : 1..31;
                      Month : 1..12;
                      Year : integer;
                      StudDoc : integer
                    end;
Var Group : array [1..25] of student;
                S : Student;
```

При позначенні компоненти запису в програмі слідом за іменем запису ставиться точка, а потім ім'я відповідного поля. Таким чином здійснюється доступ до цієї компоненти. Наприклад:

```
1)  z1.Re := 2;  z1.Im := 3;
    M := sqrt(sqr(z1.Re) + sqr(z1.Im));
2)  S.F1 := Group[i].F1;
    S.Year := Group[i + 1].Year;
    writeln( Group[i].StudDoc);
```

Запис може входити у склад даних більш складної структури. Можна говорити, наприклад, про масиви і файли, що складаються з записів. Запис може бути полем іншого запису.

Приклад 3

```
Type Name = array[1..20] of Char;
     FullName = Record
       Name1, Name2, Name3 : Name
     end;
     Date = Record
      Day : 1..31;
      Month : 1..12;
      Year : integer
     end;
    Student = Record
     StudName: FullName;
      BirthDay: Date;
```

```
  StudDoc: integer
end;
Var StudGroup : Array [1..30] of Stugent;
    A, B : Student;
```

Наприклад, доступ до поля day змінної A можливий по імені A.BirthDay.Day, а до першої букви поля Name2 імені студента з номером 13 змінної StudGroup - по імені StudGroup[13].StudName.Name2[1]

3.Записи з варіантами.

Синтаксис комбінованого типу містить і варіантну частину, що припускає можливість визначення типу, який містить визначення декількох варіантів структури. Наприклад, запис у комп'ютерному каталозі бібліотеки може мати наступну структуру:

Фіксована частина

Прізвище Ім'я по батькові	{автора}
Назва	{книги}
Видавництво	{його атрибути}
Шифр	{бібліотеки}
Стан	(**видана, у фонді, в архіві**)

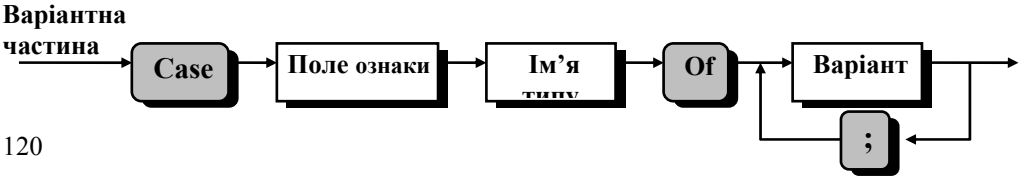
Варіантна частина

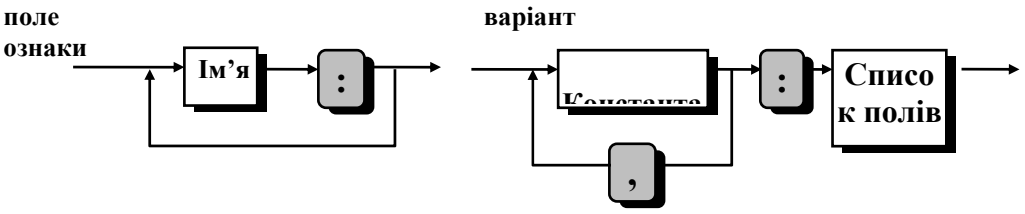
Значення признака	**видана**	**у фонді**	**в архіві**
Поля Варіантної части	Прізвище ім'я, по батькові N% {чит.квитка} Дата {видачі}	адрес {схову}	ім'я {архіву} адрес {схову} дата {архівації}

Синтаксис варіантної частини:

<варіантна частина >::= case <поле ознаки > <ім'я типа > of < варіант >{;<варіант >}
< варіант >::=<список міток варіанта >:(<список полів >) _ < пусто >
<список міток варіанта >::=<мітка варіанта >{;<мітка варіанта >}
<мітка варіанта >::=< константа >
<поле ознаки >::=<ім'я >:< пусто >

Відповідні синтаксичні діаграми:





Описання типу запису в розглянутому прикладі може мати вид:

Приклад 4

```
Book = record
  Author : FullName; {фіксована частина}
BookName: String;
BookCode: Code;
Station : (Readed, inFile, inArchive);
case Station of {поле ознаки}
        Readed: Reader : FullName; {варіантна частина}
        ReadCode : Integer;
        ReadDate : Date;
        inFile: FilAdress : Adress;
        inArc : ArcName : Srting;
        ArcAdress: Adress
    end
end;
```

У нашому прикладі на варіанти вказує поле Station. У залежності від значення цього поля запис має ту чи іншу структуру. Це часта ситуація. Звичайно на варіант запису вказує одне з полів фіксованої частини цього запису. Тому синтаксисом допускається скорочення: опис компоненти, що визначає варіант, (яка називається полем ознаки - дискримінантом), включається в заголовок варіанта. У нашому прикладі 4 це виглядить так:

```
Type BookStation = (Readed, inFile, inArc);
Book = record
        Author : FullName;
        BookName : String;
        BookCode : Code;
        case Station : BookStation of
```

```
            Readed : Reader : FullName;
            ReadCode : Integer;
            ReadDate : Date;
            inFile : FilAdress: Adress;
            inArc : ArcName : String;
            ArcAdress: Adress
  end
end;
```

Всі імена полів повинні бути різними, навіть якщо вони зустрічаються в різних варіантах. (Наприклад, Author, Reader - імена людей, а FilAdress і ArcAdress - адреси, що вказують на місцезнаходження книги на полках сховища). У випадку, коли один з варіантів не містить варіантної частини, він повинен бути оформлений наступним чином:

EmptyVariant :( ) {EmptyVariant - мітка порожнього варіанта}

4.Оператор приєднання.

Якщо A - змінна типу Student із приклада 1, її значення можна змінити групою операторів:

```
A.F1 := ' Іванов '; A.F2 := ' Ілля '; A.F3 := 'Інокентійович ';
A.Day := 14; A.Month := 9; A.Year := 1976;
A.StudDoc := 123;
```

Приведені вище позначення можна скоротити за допомогою оператора приєднання. Заголовок цього оператора відкриває область дії “внутрішніх” імен полів запису, які можуть бути використані як імена змінних. Оператор приєднання має вид:

With <змінна-запис > {,<змінна-запис >} do < оператор >

Приклад 5

```
with A do begin
   F1 := ' Іванов ';  F2 := ' Ілля '; F3 := ' Інокентійович ';
   Day := 14; Month := 9; Year := 1976;
   StudDoc := 123;
 end { оператора with }
```

Більш того, в групі операторів обробки змінної StudGroup[i] (приклад 3) запис може бути сформований наступним чином:

```
With StudGroup[3], StudName do begin
 Name1 := ' Інокентіївна '; Name2 := ' Інна '; Name3 := ' Іванівна ';
 BirthDay.Day := 14; BirthDay.Month := 9; BirthDay.Year := 1976;
 StudDoc := 123
```

```
 end;
```

Таким чином, оператор виду

With r1,...,rn do S

еквівалентний оператору

With r1 do with r2 ... with rn do S.

Зауваження: В операторі With R do S вираз R не повинен містити змінні, що змінюються в операторі S. Наприклад, оператор With S[j] do j := j + 1 недопустимий!

Як приклад розглянемо програму, яка оброблює масив записів, кожен з яких містить відомості про людину, необхідні для зв'язка з ним по пошті. Обробка запису полягає у пошуку всіх осіб, які народилися в даний день.

Запис має вид:

Повне ім'я (Прізвище Ім'я по батькові)
День народження (День, Місяць, Рік)
Адреса (Індекс, Країна, Місто, Вулиця, № будинку, № квартири)

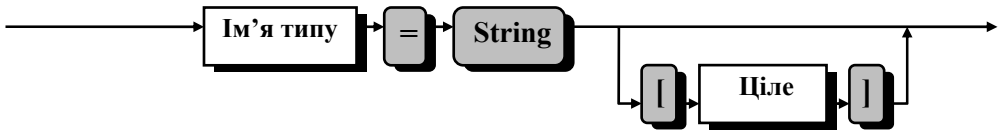
```
Program Adress;
  Const n = 100;
  Type
{ описання типів Name, FullName, Date із приклада 3 }
   Mail = Record Index : LongInt; { довге (4 байта) ціле число }
             Country, Sity, Street : Name;
             House, Flat : Integer
           end;
Person = Record
              Who : FullName;
              When : Date;
              Where : Mail
            end;
  Var
    Today : Date;
    Department : array [1..N] of Person;
    i : Integer;

    Function theSame(var X : Date, var Y : Person): Boolean;
      Begin
        With Y, When do
         theSame := (X.Day = Day) and (X.Month = Month)
       End;
  {Procedure ReadDepartment - введення масиву записів}
```

```
Procedure ReadDate(var Z : Date);
   Begin
     With Z do Read(Day, Month)
    End;
 Procedure WriteMail(var X : Person);
   Begin
     With X, Who do Writeln(Name1, ' ', Name2, ' ', Name3);
     With X, Where do begin
       Writeln(Street, ' ',House, ' ', Flat);
       Writeln(Index, ' ', Sity, ' ', Country);
     end
   End;
 Begin
   ReadDepartment; { введення масиву записів}
   ReadDate(Today);
   For i := 1 to N do
     if theSame(Today, Department[i])
       then WriteMail(Department[i])
   End;
```

5. Рядки і засоби їх обробки.

Значенням рядкового типу даних є рядки. Стандарт мови передбачає використання рядків тільки як констант, що використовуються в операторах виведення Write, Writeln. У розширенні мови Turbo-Pascal рядковий тип даних визначений більш повно.

Визначення рядкового типу слідує діаграмі:



Тут ціле належить діапазону 1..255 і означає максимальну кількість символів у рядку цього типу. Якщо описання типу String використовується без вказівки максимальної кількості символів, це (за замовченням) означає, що під цей тип резервується 255 символів.

Наприклад:

```
Type Name = String[20]; { рядки з 20-ти символів }
  Post = String; { рядки з 255-ти символів }
```

Процедури і функції типу String.

Над рядками визначена операція конкатенації “+”, результат якої - рядок, в якому операнди з’єднанні в порядку їх слідування у виразі. Наприклад:

'Turbo' + 'Pascal' = 'TurboPascal'; 'Turbo_' + 'Pascal ' + 'System' = 'Turbo_Pascal System';

Тому результатом виконання серії операторів

```
X := ' Приклад'; У := ' додавання '; Z := ' рядків'  ;
Writeln(X + Y + Z); Writeln(Y + ' ' + Z + ' ' + X)
```

будуть виведені на екран рядки

Прикладдодаваннярядків
додавання рядків Приклад

Тип String допускає і пустий рядок - рядок, який не містить символів: EmptyStr := '' {лапки, що йдуть підряд}. Вона грає роль нуля (нейтрального елемента) операції конкатенації: EmptyStr + X = X + EmptyStr = X

Над рядками визначені також відношення (операції логічного типу)

" = ", " <> ", " < ", " > ", " <= ", " >= ".

Таким чином, кожний із рядкових типів упорядкований, причому лексикографічно. Це означає, що

а) порядок на рядках погоджений з порядком, заданим на символьному типі (Char);

б) порівняння двох рядків здійснюється посимвольно, починаючи з перших символів;

в) якщо рядок А є початком рядка В, то А < В;

г) пустий рядок - найменший елемент типу.

Наприклад:

а) 'c' < 'k', так як Ord(‘c’) < Ord(‘k’);

б) 'abc' < 'abk', так як перші два символи рядків співпадають, а порівняння третіх дає Ord(‘c’) < Ord(‘k’);

в) 'abc' < 'abkd', так як перші два символи рядків співпадають, а порівняння третіх дає Ord(‘c’) < Ord(k);

г) 'ab' < 'abcd', так як рядок 'ab'- початок рядка 'abcd'.

На рядковому типі даних визначені:

Функції:

а) Length(X: String): Byte; - довжина рядка X; { Length(EmptyStr) = 0 }

б) Pos(Y:String; X:String):Byte; - позиція першого символу першого зліва входження підрядка Y у рядок X. Якщо X не містить Y, Pos(Y, X) = 0.

в) Copy(X:String; Index, Count: Integer):String - підрядок рядка X, що починається з позиції Index і містить Count символів.

г) Concat(X1, X2, .., Xk: String): String; - конкатенація рядків X1, X2, .., Xk. Інша форма запису суми X1+X2+ .. +Xk.

Процедури:

д) Delete(var X: String; Index, Count: Integer); З рядка X видаляється Count символів, починаючи з позиції Index. Результат поміщується в змінну X.

е) Insert(Y:string; var X: String; Index: Integer); В строку X вставляється рядок Y, причому вставка здійснюється починаючи з позиції Index.

Стандартні процедури введення-виведення Паскаля розширені для введення-виведення рядків. Відмітимо, однак, що для введення декількох рядкових даних треба користуватись оператором Readln. Оператор Read у цих випадках може вести себе непередбачено.

Приклад 2. Дано масив A[1..n] of string[20]. Скласти програму заміни всіх перших входжень підрядка L в A[i] на підрядок R. Рядки L і R вводяться з клавіатури в виді рівності L = R. Результат замін відобразити у масив, елементи якого - рівності виду:

A[i]=результат заміни L на R в A[i].

```
Program RewriteArray;
  Const n = 100; Single = 20; Double = 41;
  Type
          Sitem = string[Single];    Ditem = string[Double];
       SWordArray = array[1..n] of Sitem;   DWordArray = array[1..n] of Ditem;
  Var
          A: SWordArray;    B: DWordArray;
          L, R: Sitem;   X : Sitem;
          i, Index : Integer;
 Procedure InpWord(var U, V : Sitem);
   Var X : Ditem;
       j : Integer;
 Begin
  Writeln('_________ Введення рівності L = R ___________');
  Read(X); j := Pos('=', X);
  U := Copy(X, 1, j - 1);
  V := Copy(X, j + 1, Length(X))
  End;


Procedure InpArray;
 begin
  Writeln('====== Введення масиву слів ======');
  For i:=1 to n do Readln(A[i])
 end;
Procedure OutArray;
 begin
  Writeln('====== Виведення масиву слів ====');
  For i:=1 to n do Writeln(B[i])
end;
Begin
 InpArray;                    {введення масиву слів з клавіатури}
 InpWord(L, R);     {введення і аналіз рівності L = R}
```

```
 For i := 1 to n do begin
  X := A[i]; Index := Pos(L, X);
  If Index <> 0
    then begin
      Delete(X, Index, Length(L));
      Insert(R, X, Index)
    end;
   B[i] := A[i] + '=' + X
 end;
 OutArray;                    {виведення масиву слів до друку}
End.
```

6.Файли. Управління файлами.

Програма, яка написана мовою Pascal, повинна якимось чином обмінюватись даними з зовнішніми пристроями (отримувати дані з клавіатури, магнітного диска, виводити дані на екран, принтер і т.д.) Для роботи з зовнішніми пристроями використовуються файли. Файли - це значення файлового типу даних - ще одного стандартного складного типу в мові.

(Послідовний) файл - це послідовність однотипних компонент, яка має ознаку кінця і оброблюється послідовно - від початку до кінця.

Порядок компонент визначається самою послідовністю, подібно до того, як порядок слідування чергового кадру кінофільму визначається його розташуванням на кіноплівці. У будь-який момент часу доступний тільки один елемент файла (кадр кінофільму). Інші компоненти доступні тільки шляхом послідовного просування по файлу.

У результаті виконання програми відбувається перетворення одного текстового файла (який називається Input) в інший текстовий файл (який називається Output). Обидва ці файли є стандартними і використовуються для введення /виведення даних.

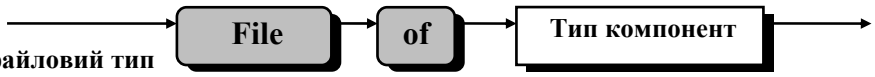
Над файлами можна виконувати два явних виду дій:

1.Перегляд (читання) файла.

2.Створення (запис) файла - виконується шляхом приєднання нових компонент у кінець початково порожнього файла.

Файли, з якими працює програма, повинні бути описані в програмі. Частина файлів (що уявляють собою фізичні пристрої) має в операційній системі стандартні імена. Наприклад, для читання даних з клавіатури і виведення результатів на екран монітора ми користуємось стандартними файлами Input і Output. Файл - принтер має ім'я Prn:

Імена нестандартних файлів, що використовуються в програмі, необхідно описувати у розділі змінних. Описання файлового типу відповідає діаграмі:



Файл, компоненти якого є символами, називається текстовим. Він має стандартний тип Text:

```
Type Text = File of Char;
```

Приклади:

```
   Type
ExtClass = File of Person;CardList = File of Integer;
    Var
F : Text;
Data : File of real;
List1, List2 : CardList;
Class10A, Class10B : ExtClass;
```

Для роботи з нестандартними файлами ім'я файла повинно бути зв'язане з реально існуючим об'єктом - зовнішнім пристроєм. Саме, якщо нам необхідно обробити дані з файла, що знаходиться на магнітному диску і який має (зовнішнє) ім'я D:\ExtName.dat, ми повинні повідомити системі, що працюючи з файлом IntName (читаючи з нього дані або записуючи у нього дані), ми працюємо з файлом ExtName.dat, що знаходиться на диску D:.

Для ототожнення внутрішнього імені файла з зовнішнім іменем використовується процедура **Assign**(< внутрішнє ім'я >, < 'зовнішнє ім'я ' >).

Після того, як ім'я файла описано і визначено, можна приступити до роботи з файлом. При використанні нестандартних файлів треба пам'ятати, що перед роботою необхідно відкрити їх, тобто зробити доступними з програми. Для цього треба застосувати одну з двох наступних процедур:

Процедура **Rewrite**(<ім'я файла >) - відкриває файл для запису. Якщо файл раніше існував, всі дані, що зберігались у ньому, знищуються. Файл готовий до запису першої компоненти.

Процедура **Reset**(<ім'я файла >) - відкриває файл для читання. Файл готовий для читання з нього першої компоненти.

По закінченню роботи з файлом (на запис) він повинен бути закритий. Для цього використовується процедура **Close**(<ім'я файла >). Ця процедура виконує всі необхідні машинні маніпуляції, що забезпечують збереження даних у файлі.

Для обміну даними з файлами використовують процедури Read і Write.

Процедура **Read** (<ім'я файла >,<список введення >), читає дані з файла (по замовченню ім'я файла - Input). Список введення - це список змінних.

Процедура **Write** (<ім'я файла >,<список виведення >), записує дані у файл (по замовченню ім'я файла - Output). Список виведення - це список виразів.

Якщо F - файл типу Text, то у списку введення/виведення допустимі змінні/вирази типу Integer, Real, Char, String[N]. В інших випадках типи всіх компонент списку повинні співпадати з типом компоненти файла.

При роботі з файлами застосовують стандартні логічні функції:

Eof(F) (end of file) - стандартна функція - признак кінця файла. Якщо файл F вичерпаний, то Eof(F) = True, в протилежному випадку Eof(F) = False.

Eoln(F) (End of line) - стандартна функція - признак кінця рядка текстового файла. Якщо рядок текстового файла F вичерпаний, то Eoln(F) = True, в протилежному випадку Eoln(F) = False. Функція Eoln визначена тільки для файлів типа Text. Звичайно в програмах використовують або текстові файли, або файли, компонентами яких є структуровані дані (наприклад, записи).

Треба пам'ятати, що дані файлових типів неможна використовувати в якості компонент інших структур даних. Наприклад, неможна визначити масив, компонентами якого є файли, запис, полем якої є файл.

Приклад 3. Програма формування файла як вибірки з компонент іншого файла. Нехай F - файл записів виду (поле ключа, поле даних). Треба сформувати файл G, який містить ті компоненти F, ключі яких задовольняють умові: значення ключа - ціле число з інтервалу]Max, Min [.

```
Program FileForm;
   Type Component = Record
          Key: Integer; { поле ключа }
          Data: String[20] { поле даних}
         End;
          OurFile = File of Component;
               Var F, G : OurFile;  X: Component;
          Max, Min : Integer;
  Function Correct(var U: Component): Boolean;
   begin
    Correct := (U.Key < Max) and (U.Key > Min)
   end;

  Begin
   Read(Max, Min);
   Assign(F, 'D:InpFile.dat');                {визначення значення F }
   Assign(G, 'D:OutFile.dat');        {визначення значення G }
   Reset(F);                                  {файл F відкритий для читання}
   Rewrite(G);                        {файл G відкритий для запису}
   While not(Eof(F)) do begin         {цикл читання F - запису G}
    Read(F, X);
    If Correct(X) then Write(G, X)
   end;
   Close(G)                                   {файл G закритий}
  End.
```

7.Основні задачі обробки файлів.

Специфіка файлового типу, яка пов'язана з послідовним доступом до компонент і розташуванням файлів на зовнішніх носіях, накладає жорсткі обмеження на засоби розв'язку задач обробки файлів. Розглянемо деякі такі задачі. Для визначеності будемо вважати, що всі файли мають тип OurFile із приклада 3 і впорядковані за значенням ключового поля Key (ключу).

Задача 1. Доповнення елемента до файла. Дано файл F і елемент X типу Component. Розширити F, включивши в нього компоненту X з збереженням упорядкованості.

Ось, напевно, єдиний можливий розв'язок:

- Переписувати покомпонентно F у новий файл G до тих пір, поки F^.Key < X.Key ;
- Записати X у G;
- Переписати "хвіст" файла F і G;
- Перейменувати G у F .

Задача 2. Вилучення елемента з файла. Дано файл F і число K - значення ключа елементів, що вилучаються. Скоротити F, вилучивши із нього всі компоненти з ключем K.

Розв'язок аналогічний розв'язку задачі 1:

- Переписувати покомпонентно F у новий файл G до тих пір, поки F^.Key < X.Key;
- Поки F^.Key= K читати F;
- Переписати "хвіст" файла F у G;
- Перейменувати G у F.

Відмітимо, що:

$\Rightarrow$ Розв'язки цих задач потребують послідовного пошуку міста елемента X як компоненти файла. Ефективний розв'язок задачі пошуку (наприклад, бінарний пошук) неможливий.

$\Rightarrow$ У якості вихідного використовується новий файл, оскільки читання/запис в один і той же файл неможливі!

Наступні задачі присвячені обробці двох файлів.

Задача 3. Злиття (об'єднання) файлів. Дано файли F і G. Треба сформувати файл H, який містить всі компоненти як F, так і G.

Алгоритм полягає у послідовному і почерговому перегляді файлів F і G і запису чергової компоненти в H. Почерговість визначається порівнянням значень ключів компонент F і G. Оформимо алгоритм у виді процедури:

```
Procedure FileMerge(var F, G, H: OurFile);
    Var X, Y : Component;
        Flag : Boolean;
```

```
Procedure Step(var U:OurFile; var A, B:Component);
    begin
     Write(H, A);
       If Eof(U)
         then begin Write(H, B); Flag := False end
         else Read(U, A)
    end;

Procedure AppendTail(var U: Ourfile);
     Var A: Component;
     Begin
       While not(Eof(U)) do begin
       Read(U, A); Write(H, A)
      end
     end;

  Begin
    Reset(F); Reset(G); Rewrite(H);
    Flag := True;
    Read(F, X); Read(G, Y);
    While Flag do
     If X.Key < Y.Key
       then Step(F, X, Y)
       else Step(G, Y, X);
    AppendTail(F);
    AppendTail(G);
    Close(H)
  End;
```

Задача 4. Перетин файлів. Дано файли F і G. Треба сформувати файл H, який містить всі компоненти, що входять як у F, так і в G.

Задача 5. Віднімання файлів. Дано файли F і G. Треба сформувати файл H, який містить всі компоненти, що входять у F, але не входять у G.
Розв'язок цих задач аналогічний розв'язку задачі злиття файлів.

8. Сортування файлів.

Упорядкованість компонент файла за одним або кількома ключовими полями - одна з основних умов ефективної реалізації задач обробки файлів. Так, задача роздруку

файла у визначеному порядку слідування компонент, якщо файл не впорядкований відповідним чином, розв'язується за допомогою багатократних переглядів (прогонів) файла. Кількість прогонів при цьому пропорційна кількості компонент.

Відсутність прямого доступу до компонент приводить до того, що розглянуті вище алгоритми сортувань масиву неможливо ефективно адаптувати для сортування файла. На відміну від масивів, основні критерії ефективності алгоритму сортування файла - кількість прогонів файлів і кількість проміжних файлів.

Так, наприклад, алгоритм сортування простими обмінами потребує N прогонів файла, що сортується (N - кількість компонент файла). Алгоритм швидкого сортування взагалі не має сенсу розглядати, оскільки при його реалізації необхідно було би читати файл від кінця до початку!

Розглянемо тому новий для нас алгоритм - алгоритм сортування злиттям, який найбільш ефективний при сортуванні файлів і відноситься до швидких алгоритмів при сортуванні масивів, хоча і потребує додаткової пам'яті.

Алгоритм сортування злиттям.

Нехай послідовність, що сортується, $F = \{ f_1, ..., f_N \}$ представлена в виді двох уже відсортованих половин - F_1 і F_2. Тоді для сортування F достатньо злити (тобто застосувати аналог алгоритму злиття FileMerge) послідовності F_1 і F_2 у вихідну послідовність G. Оскільки упорядковані половинки F_1 і F_2 можна отримати за допомогою тих же дій, ми можемо описати алгоритм сортування злиттям рекурсивно. При цьому, очевидно, необхідно використовувати оптимальну кількість проміжних файлів. Оскільки злиття треба починати з однокомпонентних файлів, точна реалізація описаного алгоритму потребує 2N додаткових файлів, що, звичайно, неприйнятно. Тому ми повинні використовувати один файл для збереження декількох послідовностей, що зливаються. Уточнимо цю ідею:

Нехай файл F, що сортується, містить k-елементні упорядковані підпослідовності $A_1, A_2, ..., A_{2l}$, $k = 2l$.

1.Розділ F полягає у переписі послідовностей Aj з парними номерами в файл F_1, а з непарними номерами - в файл F_2.

Нехай файл F_1 містить k-елементні упорядковані підпослідовності $B_1, B_3, ..., B_{2l-1}$, а F_2 - k-елементні упорядковані підпослідовності $B_2, B_4, ..., B_{2l}$.

2.Злиття F_1 і F_2 полягає в злиттях пар $<B_i, B_{i+1}>$ у підпослідовності $A_{(i+1) \text{ div } 2}$ у файл F. Після обробки F буде містити l 2k-елементних підпослідовностей A_j.

Якщо N - степінь 2-х ($N = 2^m$), то після m-кратного повторення розділення-злиття, починаючи з $A_1=\{f_1\}, ..., A_N=\{f_N\}$, файл F буде містити відсортовану послідовність. У загальному випадку, коли $N \neq 2^m$, керування злиттями-розділеннями трохи ускладнюється:

- при розділі кількість підпослідовностей може опинитися непарною;
- при злитті остання за рахунком підпослідовність одного з файлів може мати меншу, ніж усі інші, кількість елементів.

```
Program MergeSort;
```

```
    Type Component = Record
            Key: Integer; { поле ключа }
            Data: String[20] { поле даних}
          End;
        OurFile = File of Component;
     Var F, F1, F2 : OurFile;
       k, L, t, SizeF: Integer;
{Procedure FileOut(var F :OurFile);}
{Procedure FileInp(var F: Ourfile);}
      Procedure CopyBlock(Var U, V: OurFile; k: Integer);
           Var i: Integer;
               X: Component;
         Begin
           For i := 1 to k do begin
            Read(U, X); Write(V, X)
           end
         End;
{ розділення блоками по k компонент F ==> F1, F2 }
      Procedure Partition(k: Integer);
           Var i: Integer;
         Begin
           Reset(F); Rewrite(F1); Rewrite(F2);
           For i:= 1 to L do
            If Odd(i)
              then CopyBlock(F, F1, k)
              else CopyBlock(F, F2, k);
           If Odd(L)
              then CopyBlock(F, F2, t)
              else CopyBlock(F, F1, t);
              Close(F1); Close(F2)
         End;
{ злиття блоками по k компонент F1, F2 ==> F }
      Procedure Merge(k: Integer);
          Var i: Integer;
      Procedure MergeBlock(t: Integer);
           Var count1, count2: Integer;
               X1, X2: Component;
              Flag: Boolean;
       Procedure AppendTail(var U: Ourfile; var p: Integer; s: Integer);
           Var A: Component;
         Begin
```

```
            While p <= s do begin
            Read(U, A); Write(F, A); p := p + 1
          end;
        End { AppendTail };
Procedure Step(var U: OurFile; Var A, B: Component; var p, q: Integer; s: Integer);
          begin
            Write(F, A); p := p + 1;
            If p <= s
              then Read(U, A)
               else begin  Write(F, B); q := q + 1; Flag := False end
         end { Step };
      Begin { MergeBlock }
        count1 := 1; count2 := 1; Flag := True;
        Read(F1, X1); Read(F2, X2);
        While Flag do
          If X1.Key < X2.Key
            then Step(F1, X1, X2, count1, count2, k)
            else Step(F2, X2, X1, count2, count1, t);
        AppendTail(F1, count1, k); AppendTail(F2, count2, t);
      end { MergeBlock };
      Begin { Merge }
        Rewrite(F); Reset(F1); Reset(F2);
        For i := 1 to (L div 2) do MergeBlock(k);
          If t<> 0
            then if Odd(L)
              then MergeBlock(t)
              else CopyBlock(F1, F, t)
            else if Odd(L)
              then CopyBlock(F1, F, k)
    End { Merge };
    Procedure FirstPartition;
          Var X : Component;
       Begin
        Reset(F); Rewrite(F1); Rewrite(F2);
        SizeF := 0;
        While not(Eof(F)) do begin
          Read(F, X); SizeF := Succ(SizeF);
          If Odd(SizeF)
             then Write(F1, X)
            else Write(F2, X)
         end;
```

```
    Close(F1); Close(F2)
  End { FirstPartition };
 Begin                          {Визначення зовнішніх імен файлів F, F1, F2}
    FileInp(F);
  FirstPartition;      {перше розділення підрахунком SizeF}
  L := SizeF; t := 0;
  Merge(1);                     {перше злиття1-блоків}
  k := 2;
  While k < SizeF do begin
    L := SizeF div k;           {кількість повних k-блоків у F}
    t := SizeF mod k;           { розмір неповного k-блока }
      Partition(k);
    Merge(k);
    k := 2*k
  end;
 FileOut(F);
End.
```

Оцінимо складність алгоритму в термінах C(n), M(n), L(n), де L(n) - число прогонів файла F і n = SizeF.

1.Оцінка L(n).

L(n)= число розділень + число злить. Кожне розділення - виклик процедури Partition, а злиття - виклик Merge. Тому L(n) - подвоєне число повторень тіла циклу While. Звідси, оскільки змінна k, що керує циклом, кожний раз збільшується вдвічі, на L-тому кроку $k = 2^L$, і, отже, L - найбільше число таке, що $2^L < n$, тобто $L = [\log_2 n]$.

$$L(n) = 2[\log_2 n]$$

2.Оцінка C(n).

Порівняння компонент за ключем відбуваються при злитті. Після кожного порівняння виконується процедура Step, яка записує одну компоненту в F. Тому при кожному злитті кількість порівнянь не перевищує n. Звідси $C(n) \le L(n)*n/2$, тобто

$$C(n) \le n [\log_2 n]$$

3.Оцінка M(n).

Процедури Partition і Merge пересилають компоненти файлів. Незалежно від значень ключів, виклик кожної з них або читає F, або записує F. Тому M(n) = nL(n), тобто

$$M(n) = 2n[\log_2 n]$$

Отримані оцінки дозволяють класифікувати алгоритм як ефективний алгоритм сортування послідовних файлів. Його також можна адаптувати до сортування масивів.

Алгоритм сортування злиттям може бути поліпшений кількома способами. Розглянемо лише деякі з них:

а. Помітимо (зауважимо), що процедура Partition носить допоміжний характер. Не аналізуючи ключів, вона просто формує файли F_1 і F_2 для злиття. Тому її можна вилучити, якщо процедуру Merge примусити формувати не F, а одразу F_1 і F_2. При цьому, звичайно, кількість файлів у програмі збільшується. Саме:

1.F розділимо на (F_1, F_2).
2.Визначимо допоміжні файли G_1, G_2
3.Основний цикл алгоритму:
Злиття (F_1, F_2) ===> (G_1, G_2);
Злиття (G_1, G_2) ===> (F_1, F_2);
(Непарні пари блоків зливаємо на 1-ий файл, парні - на 2-ий).
Часова складність алгоритму поліпшується майже вдвічі.

б. Зауважимо, що на початковій стадії роботи алгоритму розміри блоків малі. Їх можна сортувати в оперативній пам'яті, використовуючи представлення в виді масиву і швидкі алгоритми сортування масивів. Таким чином, треба змінити процедуру FirstPartition, визначив її як процедуру внутрішнього сортування k_0-блоків при деякому (максимально можливому) значенні k_0. Цикл While основної програми тепер можна починати з $k = k_0$.

в. В реальних файлах часто зустрічаються вже упорядковані ділянки компонент. Тому файл можна початково розглядати як послідовність упорядкованих ділянок, і зливати не блоки фіксованого розміру, а упорядковані ділянки. Такі сортування називають природними.

9.Задача корегування файла.

Задача корегування файла є одною з основних задач обробки файлів. Розглянемо її формулювання:

Вихідні дані задачі - файл F, що корегується, файл корегувань G. Результат - відкорегований файл H. Будемо вважати, що файл F має тип OurFile. Тоді файл H має той же тип. Файл корегувань G складається з записів з варіантами:

```
Type
    CorComponent = record
                       Key: Integer;
                       job: (Include, Change, Exclude); {включити, змінити,
                  вилучити}
                       Case job of
                         Include, Change: Data: String[20];  Exclude: ()
                        end
                       End;
     CorFile = File of CorComponent;
Var G : CorFile;
```

Це значить, що файл корегувань містить компоненти, які треба включити в основний файл, компоненти, які треба вилучити з файла і компоненти, зміст яких треба змінити (з врахуванням змісту як основного файла, так і файла корегувань).

Файл F вважаємо впорядкованим (за ключем), а файл G, взагалі кажучи, ні. Результатом повинен бути упорядкований відкорегований файл H.

Розв'язок задачі звичайно містить два етапи:

а) Сортування файла корегувань G;

б) Узагальнене злиття файлів F, G у H.

На практиці файл G може бути невеликим. У цьому випадку застосовують внутрішнє сортування. Узагальнений алгоритм злиття робить всі три варіанта обробки файла F за один прогін.

На завершення відзначимо, що сучасні ЕОМ рідко активно використовують зовнішні носії з послідовним доступом у розв'язках задач керування базами даних, тому структури даних, у яких зберігається інформація, як правило, більш складні, ніж послідовні файли.

10. Задачі і вправи.

Файли.

1.Розробіть програму Добавлення елемента до файла.

2.Розробіть програму Вилучення елемента з файла.

3.Розробіть програму Перетин файлів.

4.Розробіть програму Віднімання файлів.

5.Розробіть програму Корегування файлів.

6.Розробіть програму корегування упорядкованих файлів, у яких проводиться зміна значень ключових полів. Файл корегувань містить пари

< старий ключ, новий ключ >.

7.Реалізуйте поліпшення а) алгоритму сортування злиттям.

8.Реалізуйте поліпшення в) алгоритму сортування - сортування природним злиттям.

Записи.

1.Опишіть тип запису - клітинки розкладу занять на факультеті для своєї спеціальності і курсу. Сформуйте файл двохтижневого розкладу для своєї підгрупи. Розробіть програму, яка визначає кількість лекційних, практичних і лабораторних занять у двохтижневому циклі для своєї підгрупи за вказаною дисципліною.

2.Опишіть тип запису - відомості про студента групи, необхідні декану факультету. Сформуйте файл студентів своєї підгрупи. Розробіть програму, яка визначає стан успішності в підгрупі.

3.Опишіть тип запису - відомості про батьків учнів класу, необхідні класному керівнику. Сформуйте файл, що складається не менш, ніж з восьми учнів “Вашого” класу. Розробіть програму, яка за прізвищем і іменем учня друкує відомості про його батьків.

4.Опишіть тип запису - відомості про успішність учня, необхідні для вчителя-предметника зі свого предмета. Сформуйте файл, що складається не менш, ніж з восьми учнів “Вашого” класу. Розробіть програму, яка визначає самого слабшого і самого сильного учня класу.

5.Опишіть тип запису - рядок залікової книжки (екзаменаційна частина). Сформуйте файл іспитів, зданих Вами. Розробіть програму, яка визначає середній бал, складає список екзаменаторів і по номеру семестру роздруковує результати Вашої сесії.

6.Опишіть тип запису - рядок залікової книжки (залікова частина). Сформуйте файл заліків, зданих Вами. Розробіть програму, яка визначає дні, коли Ви здавали по два і більш заліків.

7.Опишіть тип запису - відомості про вік, зріст і вагу учня. Сформуйте файл, що складається не менш, ніж з восьми учнів “Вашого класу”. Розробіть програму, яка визначає всіх учнів, що народилися в даний проміжок часу, вказаний датами початку і кінця і визначає середній зріст і середню вагу цієї групи учнів.

8.Опишіть тип запису - відомості про книгу (наприклад, з інформатики). Сформуйте файл книг, необхідних учителю інформатики. Складіть програму, яка підбирає книги для курсу, номер якого вводиться, друкує імена їх авторів і рік видання.

9.Опишіть тип запису - відомості про товар у магазині. Сформуйте файл товарів, що є в магазині. Розробіть програму корегування масиву товарів і визначення виручки магазину на даний момент часу.

10.Опишіть тип запису - рядок у телефонній книзі. Сформуйте файл записів - вашу телефонну записну книжку. Розробіть програму пошуку номера телефона за прізвищем і пошуку адреси за номером телефона.

Файли рядків (слів).

1. Знайти в файлі F всі оборотні слова і скласти з них файл G.
2. Знайти в файлі F входження слова p, замінити їх на слово q, отримавши новий файл G.
3. Знайти в файлі F всі слова, що представляють числа (у десятковому запису) і отримати числовий файл G, що містить знайдені числа.
4. Знайти в файлі F всі однобуквенні слова і зайві пробіли, і, вилучивши їх, отримати новий файл G.

5. Знайти в файлі F всі слова, що зустрічаються більш одного разу, і скласти файл G, вилучивши з F знайдені слова.
6. У файлі F знайти всі слова, що містять здвоєні букви і скласти з них файл G.
7. Знайти в файлі F всі слова, що містять підслово p і скласти з них файл G.
8. Дано файл F. Відсортувати його в алфавітному порядку.
9. Дано слово p і файл F. Знайти в F всі слова, які можна скласти з букв слова p.

МНОЖИНИ.

1.Множинний тип.

Ще одним складним стандартним типом даних, визначеним у мові Pascal, є множинний тип. Значенням множинного типу даних є множина, що складається з однотипних елементів. Тип елемента множини називається базовим типом. Базовим типом може бути скалярний або обмежений тип. Таким чином, множина значень множинного типу - це множина всіх підмножин базового типу, враховуючи і порожню множину. Якщо базовий тип містить N елементів, відповідний множинний тип буде містити 2^N елементів.

Характерна відміна множинного типу - визначення на ньому найбільш поширених теоретико-множинних операцій і відношень. Це робить множинний тип схожим на прості типи даних. Множинні типи описуються в розділі типів наступним чином:

Type < ім'я типу > = Set of < базовий тип >

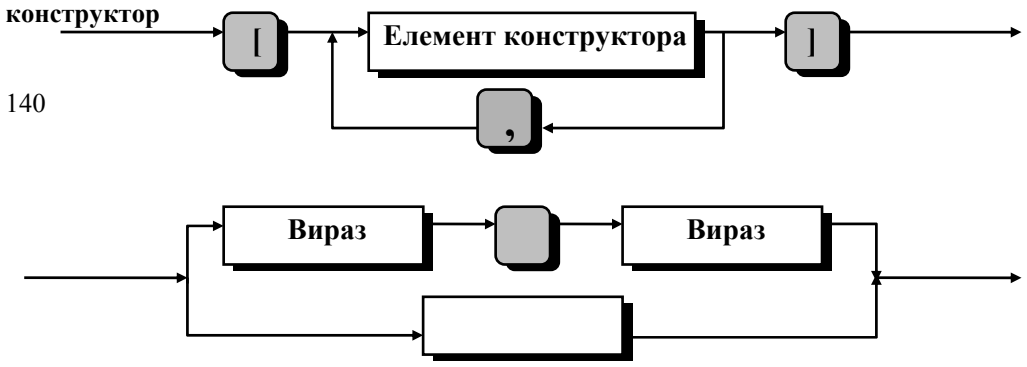
Наприклад,

а) Type Beta = Set of 100..200;
б) Type Glas = Set of char ; {Vowel}
в) Type Color = (red, orange, yellow, green, light_blue, blue, violet);
Paint = Set of Color;
г) Type TwoDigNum = Set of 10..99;

Var A, B: Beta;
llet, flet: Glas;
last, first: Paint;
Sinit: TwoDigNum;

2.Конструктор множини.

Множини будуються з своїх елементів за допомогою конструктора множин. Конструктор представляє собою перелік через кому елементів множини або відрізків базового типу, взятий у дужки [,]. Порожня множина позначається через [ ].



Елемент конструктора

Наприклад:

[ ] - порожня множина
[2, 5 ..7] - множина {2, 5, 6, 7}
['A'..'Z', 'O'..'9'] - множина, що складається з всіх великих латинських букв і цифр
[i + j .. i + 2*j] - множина, що складається з всіх цілих чисел між i + j і i + 2j
Відмітимо, що якщо у виразі [v1..v2] v1 > v2, множина [v1 .. v2] - порожня.

3. Операції і відношення.

До операндів - однотипних множин A і B можна застосувати такі дії:

A + B - об'єднання $A \cup B$

A * B - перетин $A \cap B$

A - B - різниця $A \setminus B$

Між A і B визначені також відношення порядку і рівності

A = B, A <> B, A < B, A <= B, A > B, A >= B;

Відношення порядку інтерпретуються як теоретико-множинні включення.

Якщо A - множина і x - елемент базового типу, то визначено відношення належності **x in A** - x належить A ($x \in A$).

Кожне з відношень, описаних вище, по суті є операцією, результат якої має тип Boolean. Таким чином, якщо Init - змінна типу Boolean, можливе присвоювання Init := A < B. Можливі такі порівняння (A = B) = (C = D).

Наявність операцій над множинами дозволяє застосовувати в програмах оператори присвоювання, в лівій частині яких стоїть змінна типу множини, а в правій - вираз того ж типу. Наприклад:

A := A * [1 .. 10] + B ; B := (A + B)*['A' .. 'Z'] ;

4. Застосування множин у програмуванні.

При реалізації мови розміри множин завжди обмежені константою, що залежить від реалізації. Звичайно ця константа кратна довжині машинного слова. Це відбувається тому, що множини реалізовані в виді логічних (двійкових) векторів наступним чином: кожній координаті двійкового вектора однозначно відповідає один з елементів базового типу.

Якщо елемент a належить множині A, що представляється, то значення координати вектора, відповідне a, дорівнює 1. У протилежному випадку значення відповідної координати дорівнює 0.

Наприклад, якщо множина A описана як Set of 0..15, то його представляє 16-ти мірний двійковий вектор, координати якого перенумеровані від 0 до 15, і i-тій координаті відповідає елемент i базового типу.

```
Базовий тип :          0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Двійковий вектор :     0 0 1 1 0 1 0 1 0 0  0  1  0  1  0  0
Представлена множина : A = [2, 3, 5, 7, 11, 13]
```

Такий спосіб реалізації дозволяє швидко виконувати операції над множинами і перевірки теоретико-множинних відношень. Тому, наприклад, замість

```
For X := 'A' to 'Z' do
        If (X ='A') or (X ='E') or  (X ='I') or (X ='O') or (X='U')
          then Statement1
          else Statement2
```

краще написати

```
        For X := 'A' to 'Z' do
          If X in ['A','E','I','O','U']
          then Statement1
          else Statement2
```

Остання форма запису не тільки краще читається, але й значно швидше обчислюється.

У системі Turbo-Pascal максимальна кількість елементів у множині дорівнює 256. Таким чином, у якості базового типу можна вибирати, наприклад, Char або відрізок 0..255. У завершення розділу наведемо приклад програми, що використовує множинні типи даних.

Приклад. Побудувати множину всіх простих чисел з відрізка 2..n ($n \leq 255$).

Метод, за допомогою якого ми це зробимо, відомий як “Решето Ератосфена”. Суть цього метода у наступному: Нехай Prime - множина простих чисел, що будується, і Grating - множина, що називається решетом. Алгоритм починає роботу з Prime = [ ]; Grating = [2..n].

Крок основного циклу:

а. Найменший елемент Grating помістити у Prime;

б. Вилучити з Grating всі числа, кратні цьому елементу;

Алгоритм закінчує роботу при Grating = [ ]

```
Program EratosfenGrating;
   Const n = 255;
     Var Grating, Prime: set of 2 .. n ;
         i, Min : integer ;
```

```
Begin
  Grating := [2 .. n] ; Prime := [] ; Min := 2;    {ініціалізація}
  While Grating <> [] do begin        {основний цикл}
    While not(Min in Grating) do      {пошук найменшого елемента в решеті}
      Min := Min + 1;
      Prime := Prime + [Min] ;        {поповнення множин простих чисел}
      For i := 1 to n div Min do      {вилучення кратних із решета}
        Grating := Grating - [i*Min];
  end;
  Writeln('Primes: ');                          {виведення множин простих чисел}
   For i := 1 to n do
     If i in Prime then write(i, ', ')
End.
```

Відмітимо, що доступ до елемента множини у мові не передбачений. У цьому - ще одна якісна відміна множинного типу від інших типів даних. Тому наприклад, для введення множини Prime доводиться перебирати всі елементи базового типу і кожний із них перевіряти на належність Prime.

5. Задачі і вправи.

1.Записати за допомогою конструктора множину X, яка складена з латинських букв a, b, c, d, i, j, k, x, y, z.

2.Записати за допомогою конструктора множину з трьох основних кольорів множинного типу Paint.

3.Записати за допомогою конструктора множину цілих розв'язків квадратної нерівності $x^{\wedge 2} + p*x + q < 0$ у припущенні, що корні відповідного квадратного рівняння лежать в інтервалі [0; 255].

4.Записати за допомогою конструктора множину простих чисел-близнюків з інтервалу 1..30.

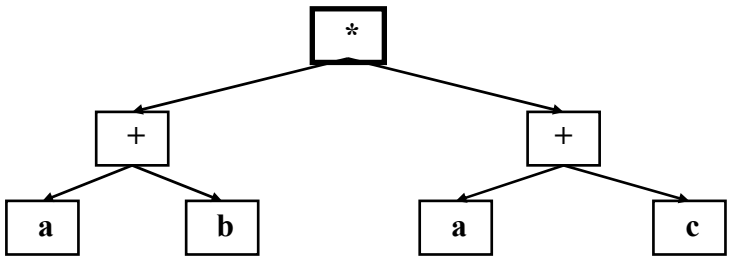
ДИНАМІЧНІ СТРУКТУРИ ДАНИХ.

1. Динамічні інформаційні структури.

У попередніх параграфах були визначені фундаментальні структури даних, що використовуються у процедурному програмуванні: масиви, записи, файли і множини. Фундаментальність цих структур означає, що вони, по-перше, частіше всього використовуються в практиці програмування, і, по-друге, визначають методи структурування даних - тобто методи утворення складних структур із більш простих. Наприклад, можна визначити масив із записів, запис, що складається з множин, файл із масивів, компоненти яких - записи, і т.д. Для кожної з таких структур даних характерна та обставина, що розмір пам'яті, що відводиться для неї, визначається компілятором під час компіляції розділів типів і змінних і залишається незмінним під час виконання програми. Тому такі змінні-структури називаються статичними. Наряду з статичним розподілом пам'яті ми вже використовували в програмах і динамічний розподіл пам'яті - під локальні змінні процедур і функцій. Особливо випукло динамізм тут проявляється при використанні рекурсії.

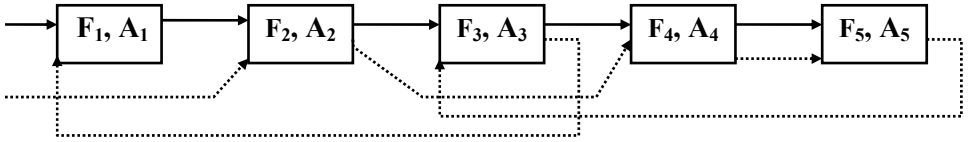
Однак багато задач для своєї ефективної реалізації потребують явних методів динамічного використання пам'яті, тобто описання таких структур даних, розмір і конфігурація яких змінюються під час виконання програм. Такі структури даних називаються динамічними.

Приклад 1. Уявимо собі, що наша програма повинна деяким чином обробляти послідовність символів, яка представляє математичну формулу (арифметичний вираз). Якщо обробка пов'язана з обчисленням значення цієї формули, то представлення формули в виді рядка символів неприродно. Зручніше представити, наприклад, формулу f = (a + b)*(a - c) у виді наступної структури:



Тепер обчислення значення f можна організувати "знизу-уверх", підставляючи результати операції замість знаків операцій. Легко бачити, що такий метод обчислення є універсальним.

Приклад 2. Нам треба обробити набір відомостей про людей (прізвище - F, вік - A), причому обробка включає процедури включення людини у список, вилучення із списку, виведення списку як у алфавітному порядку по прізвищам, так у порядку зменшення віку. Дані для цієї задачі зручно уявити у виді структури:



в якій F_i - прізвища, A_i - віки людей, суцільні стрілки вказують на людей що йдуть в алфавітному порядку., а пунктирні - на людей, що йдуть по росту.

Тоді включення - вилучення елементів можна робити переорієнтацією стрілок, а порядок виведення легко отримати, слідуючи по відповідним стрілкам. Нижче ми розглянемо і інші приклади задач, у програмуванні яких зручно використовувати динамічні структури.

Розглянуті приклади показують, що динамічні структури даних представляють із себе сукупність елементів, кожний з яких містить як деяку значущу інформацію, так і інформацію про зв'язки з іншими елементами структури. Інформацію про зв'язки називають посиланнями або покажчиками.

Динамічні структури даних, що реалізуються засобами мови Паскаль, представляються у виді сукупності записів, кожна з яких містить інформаційні поля і поля посилань (покажчиків) на інші записи структури.

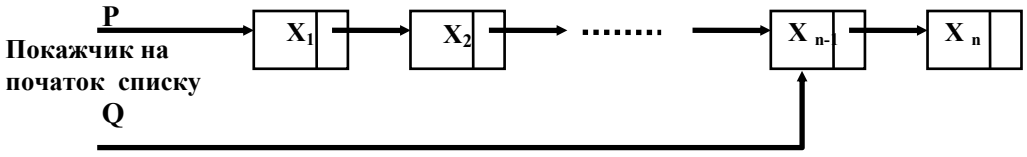
Елемент динамічної структури даних

Посилання на деякий елемент - це по суті адреса першого (молодшого байта) фрагмента оперативної пам'яті, відведеної під цей елемент.

Для реалізації ефективних алгоритмів розв'язків задач вирішальну роль грають способи об'єднання елементів у структури даних. Для кожного такого способу характерна як топологія структури, так і методи її обробки. Нижче ми розглянемо деякі з таких динамічних структур, які по суті є стандартними.

У програмуванні часто використовують наступні динамічні структури: списки, стеки, черги, дерева, графи і т.д. Точні математичні визначення цих структур, як ми побачимо нижче, використовують рекурсивні описання. Для попередніх пояснень краще всього використовувати графічні зображення.

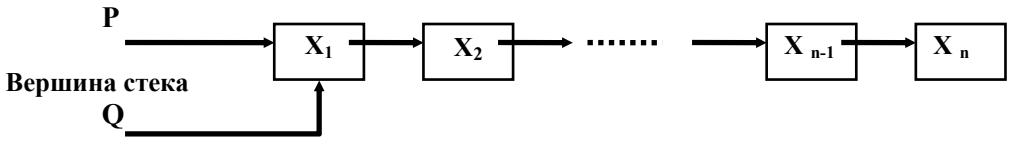
Список



Покажчик на елемент, що оброблюється

Список - це такий спосіб представлення послідовності однотипних елементів, при якому вставка і вилучення елемента допускаються у довільному її місці, а доступ до елемента, що оброблюється, здійснюється послідовним переглядом від початку списку до його кінця.

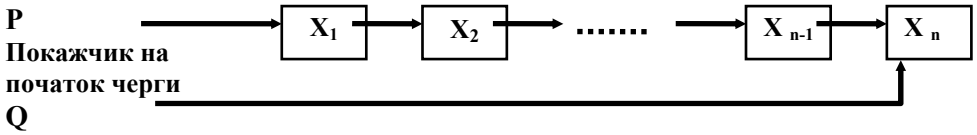
Стек



Покажчик на елемент, що оброблюється

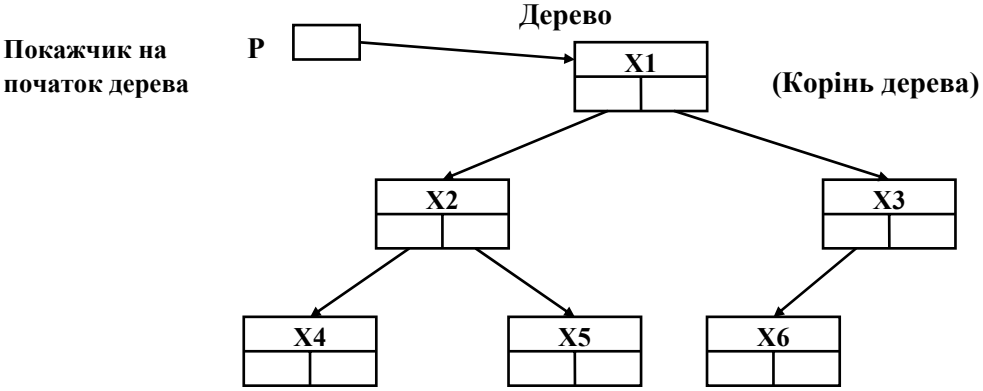
Стек - це такий спосіб представлення послідовності однотипних елементів, при якому вставка і вилучення елемента допускаються тільки на її початку - вершині стека. Доступ до інших елементів стека не підтримується методами обробки стека.

Черга

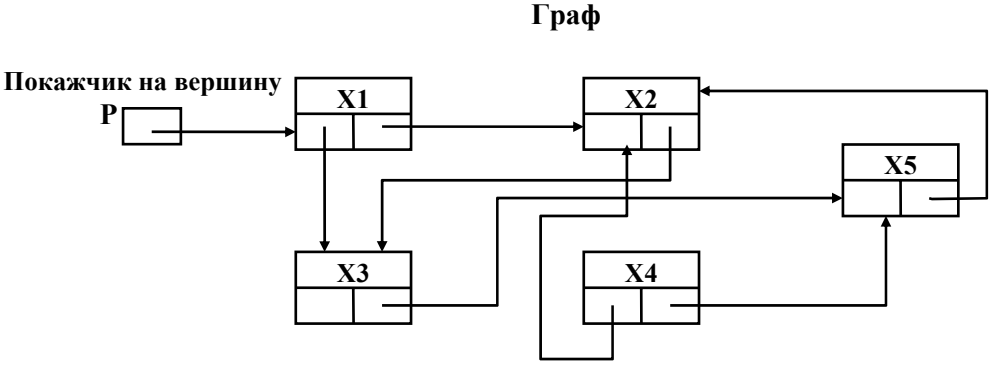


Черга - це такий спосіб представлення послідовності однотипних елементів, при якому вставка елемента допускається тільки в її хвості, а вилучення - тільки на початку. Доступ до інших елементів черги не підтримується методами обробки черги.

Списки, черги і стеки - це т.д. лінійні динамічні структури. Вони представляють з себе так звані послідовності записів, у кожній з яких, за винятком останньої, покажчик виставлений на наступну. Відрізняються вони друг від друга способом обробки: зміна списку здійснюється вставкою або вилученням елемента у довільному місці. У стеку елементи добавляють або вилучають в одному й тому ж місці - вершині стека. У черзі запису добавляються у хвіст, а вилучаються з голови.



Дерева - це структури даних, що розгалужуються. Кожний запис у дереві, за винятком одного - кореневого - має одного попередника. Кожний елемент дерева, за винятком так званого листя, вказує на декілька “спадкоємців”. Точне визначення дерева як динамічної структури даних буде дане нижче.



Граф утворюють декілька записів, покажчики яких виставлені довільним чином. У програмуванні використовуються і інші типи динамічних інформаційних структур (двохзв'язні списки, кільця, і т.д.).

Ще раз відмітимо, що використання посилань частіше всього - чисто технічний прийом. Пам'ять для розміщення змінної може виділятись автоматично при виклику процедури і звільнятись при виході з неї. Однак явне використання посилань дозволяє використовувати при програмуванні більш різноманітні структури. Тому у сучасних мовах програмування (зокрема, в Паскалі) введені засоби, що маніпулюють як даними, так і посиланнями на них.

2. Посилальний тип даних. Посилання.

Значеннями посилального типу даних є посилання (покажчики) на інші дані. Посилальні типи описуються у розділі типів наступним чином:

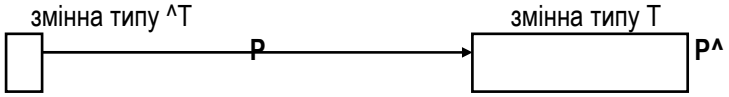
Type <ім'я посилального типу > = ^ <ім'я базового типу >

Ім'я змінної може бути зв'язано з описанням її типу безпосередньо в розділі змінних, або з іменем типу, вже описаного в розділі типів. Стиль мови віддає перевагу явному використанню імен типів.

Приклади описань:

```
1. Type Vector = Array[1..100] of Integer;
                 Place = ^ Vector;
       Var a, b: Place;
або
       Var a, b: ^ Array[1..100] of Integer;
   2. Type Item = Record
 Name: String[20];
                          Age: Integer
end;
Point = ^ Item;
   Var x, y: Point;
```

Якщо p - покажчик на змінну типу T, то p^ - позначення самої цієї змінної. Ця система позначень може бути продемонстрована так:



Ще раз укажемо на те, що при описанні посилальної змінної p пам'ять резервується тільки для неї. Значенням p по суті є початкова адреса розміщення p^ у пам'яті.

Для резервування пам'яті під дане типу T**^** у мові використовується процедура New.

New (< змінна посилального типу >)

Виконання процедури New(p) полягає у заповненні комірки пам'яті, відведеної під змінну p початковою адресою ділянки пам'яті, в якій буде розміщено значення p^. Розмір цієї ділянки визначається типом змінної p.

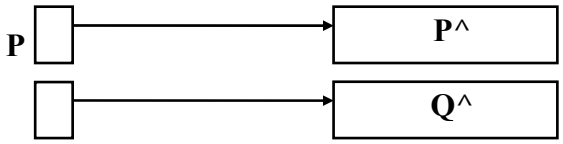
Для визволення пам'яті, відведеної раніш під T^, використовується процедура Dispose.

Dispose(<змінна посилального типу>)

Приклад 3.

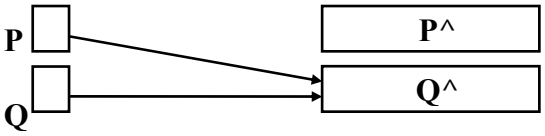
```
Program ExamNewDis;
    Type Vector = Array[1..100] of Integer;
           Place = ^Vector;
             Item = Record
                      Name: String[20];
                        Age: Integer
                     end;
            Point = ^Item;
     Var a, b: Place;
         x, y: Point;
             i: Integer;
Begin
  New(x); New(y);
  Read(x^.Name);  Read(x^.Age);
  y^ := x^;
 Dispose(x);
 New(a);
 For I:= 1 to 100 do  a^[i] := Sqr(i)+1;
 b := a;
 Dispose(a);
End.
```

Тут при зверненні до процедури New(x) буде відведено 22 байта під x^, при виконанні New(y) - 22 байта під y^, а при виконанні New(a) - 200 байт під a^. (Ми вважаємо, що під ціле число відводиться 2 байта). Оператор y^ := x^ пересилає дані з запису x^ у запис y^. Пам'ять, яку займає x^, звільняється оператором Dispose(x). Для переводу покажчика з одного даного на інше використовується оператор присвоювання. Нехай, наприклад, p і q - змінні одного посилального типу і має місце ситуація



Q

Тоді після виконання оператора p := q схема зміниться:



Зверніть увагу на те, що дані, на які до присвоювання посилалась p, стають недоступними! У прикладі, що розглядається, посилання b встановлено оператором b := a на масив a^. Тому оператор Dispose(a), звільняючи пам'ять, позбавить захисту не тільки масив a^, але й масив b^! Тому наступний оператор New може розмістити дані на місці масиву b.

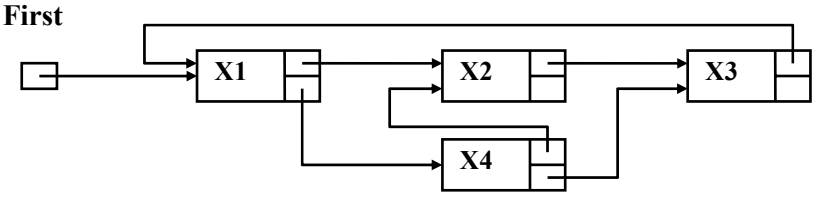
На прикладах з інформаційними динамічними структурами ми бачили, що деякі поля посилальних типів можуть бути не заповнені посиланнями на інші записи, причому програміст повинен явним чином це вказувати. Для цього використовується стандартне ім'я Nil. (Після виконання операторів p := Nil; q := Nil покажчики p і q вказують на одне й те ж дане - у нікуди!)

3. Програмування інформаційних динамічних структур.

Найпростіші характерні прийоми обробки динамічних структур даних розглянемо на наступному прикладі:

Приклад 4.

а) Побудувати динамічну структуру даних, що зображена на малюнку:



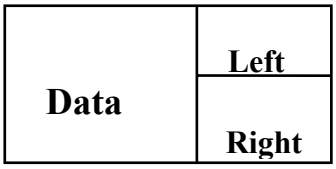
(Дані xi - цілі числа.)

б) Прочитати дані в наступній послідовності x1 x4 x3 x1, x1 x2 x3.

Розв'язок.

Тип елемента структури визначмо наступним чином:

```
Type Point = ^ Item;
             Item = record
                        Data: Integer;
                        Right, Left: Point
                    End;
```



Для побудови структури використовуємо дві змінні типу Point - p і First.

```
Program Structure;
  Type Point = ^ Item;
       Item = Record
                Data: Integer;
                Right, Left: Point
                End;
   Var p, First: Point;
 Begin
  New(p); First := p; Read(p^.Data);          {побудований перший елемент структури}
  New(p^.Left); Read(p^.Left^.Data);          {побудований другий елемент структури}
  New(p^.Right); Read(p^.Right^.Data);        {побудований четвертий елемент структури}
  p := p^.Left;
  New(p^.Left); p := p^.Left; Read(p^.Data);
  p^.Right := Nil; p^.left := First;          {побудований третій елемент структури}
  p := First^.Right;                                {p установлений на 4-ий елемент }
  p^.Left := First^.Left;
  p^.Right := First^.Left^.Left;              {покажчики виставлені. Побудова закінчена}
                                                    {початок блока читання}
  Writeln('x1,x4,x3,x1:');
  p := First;
  Write(p^.Data,''); p := p^.Right;
  Write(p^.Data,''); p := p^.Right;
  Write(p^.Data,''); p := p^.Left;
  Write(p^.Data,'');
  p := First;
  Writeln; Writeln('x1,x2,x3:');
  Write(p^.Data,''); p := p^.Left;
  Write(p^.Data,''); p := p^.Left;
  Write(p^.Data,'');
  Writeln('Кінець роботи')
 end.
```

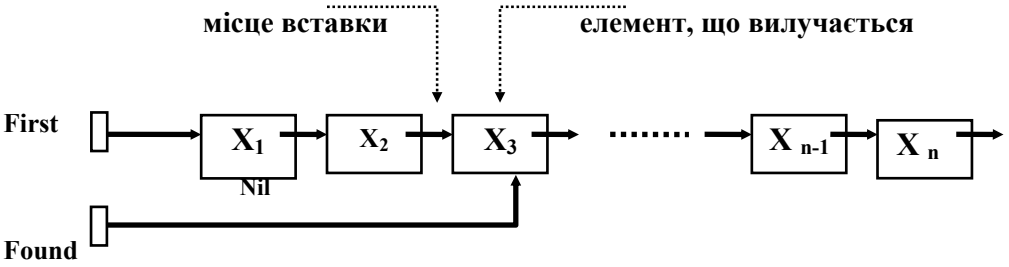
Посилання p використовувалось для обходів структури, а First - як покажчик на початковий елемент структури.

4. Списки.

У цьому розділі розглядаються процедури, що реалізують стандартні засоби роботи зі списками: пошук елемента, вилучення елемента, вставка елемента. Аналогічними засобами користуються і при обробці інших інформаційних структур. Елемент списку описується наступним чином:

```
Type Point = ^ Item;
     Item = Record
              Data: Integer; Next: Point
            End;
```

Відмітимо характерну деталь: описання елемента динамічної структури рекурсивне! Таким чином, описання динамічних структур неможливе без явного описання типів елементів цих структур.



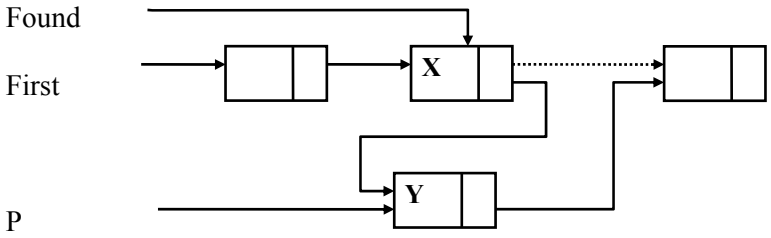
а) Пошук. Процедура Search здійснює пошук елемента списку з числом x у якості значення поля Data і повертає посилання Found на цей елемент. Якщо такий елемент у списку відсутній, Found = Nil.

```
Procedure Search(var Found, First: Point; x: Integer);
   Begin
     Found := First;
     While (Found <> Nil) and (Found^.Data <> x)
       do Found := Found^.Next
   End;
```

б) Вставка. Процедура InsList добавляє елемент у список на місце, що передує Found^ (див. малюнок). Посилання Found встановлюється на вставлений елемент.

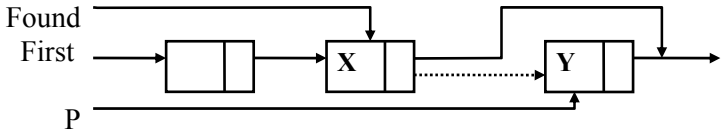
```
Procedure InsList(var Found: Point, x:Integer);
      Var p: Point;
    Begin
```

```
    New(p);
    p^.Data := Found^.data;
    Found^.Data := x;
   p^.Next := Found^.Next;
   Found^.Next := p
End;
```



в) Вилучення. Процедура DelList вилучає з списку елемент Found^. Посилання Found встановлюється на елемент, що йде за вилученим.

```
Procedure DelList(Found: Point);
      Var p: Point;
          y: Integer;
   Begin
     y := Found^.Next^.Data;
     p:= Found^.Next;
    Found^.Next := Found^.Next^.Next;
    Found^.Data := y;
    Dispose(p) {збирання сміття}
 End;
```



Відмітимо один явний недолік процедур InsList і DelList: вони непридатні для обробки останнього елемента списку! InsList для правильної роботи потребує наявності елемента Found^.Next^, а DelList - елемента Found^.Next^.Next. Тому, якщо треба вставити в список останній елемент, покажчик Found повинен бути виставлений на Nil, але тоді Found^.Next не визначений! Аналогічна ситуація має місце і при вилученні останнього елемента.

У нашій постановці задачі цей недолік виправити непросто. Суть ускладнень у тому, що до елементів списку, покажчики яких треба перекинути, немає прямого доступу: посилання Found виявляється встановленим на елемент, наступний за потрібним. Укажемо два виходи з цієї ситуації:

а) Останній елемент списку можна вважати ознакою кінця, поле Data якого не містить значущої інформації і покажчик Found на нього за умовою не може бути встановлений.

б) Можна змінити умови вставки-вилучення: посилання Found за умовою встановлюється на елемент, що передує місцю вставки або елементу, що вилучається. Тоді окремо (поза процедур) треба розглядати випадок, коли оброблюється 1-ий елемент. Самі ж процедури в цьому варіанті спрощуються.

```
Procedure Ins(var Found: Point, x:Integer);
    Var p: Point;
  Begin
     New(p);
     p^.Data := x;
     p^.Next := Found^.Next;
     Found^.Next := p;
  End;
Procedure Del(Found: Point);
    Var p: Point;
  Begin
    p:= Found^.Next;
    Found^.Next = Found^.Next^.Next;
    Dispose(p) {збирання сміття}
  End;
```

Часто вставці/вилученню передує пошук місця зміни списку. Для правильної роботи процедур Ins і Del процедуру Search необхідно модифікувати. Переглядати список треба "на крок уперед". Розглянемо варіант процедури пошуку місця елемента з значенням поля Data = x у списку First із упорядкованими за зростаючими значеннями полів Data.

```
Procedure ForwardSearch(var Found, First: Point;  var isFirst: Boolean; x: Integer);
   Begin
     Found := First;
      If First^.Data >= x
         then isFirst := True
         else begin
           isFirst := False:
           While (Found^.Next <> Nil) and  (Found^.Next^.Data < x)
             do Found := Found^.Next;
         end
   End;
```

Якщо isFirst = True, то місце - перше, інакше на місце вказує Found^.Next.

Задачі на списки.

Елемент списку описується наступним чином:

```
Type Point = ^ Item;
     Item = Record
              Key: Integer;
              Data: Real;
              Next: Point
            End;
```

1. Реалізувати процедуру злиття двох списків, елементи яких розташовані за зростанням значень полів Key. При рівних значеннях полів Key значення полів Data додаються. Оцінити складність алгоритму за часом у термінах довжин списків.
2. Реалізувати процедуру зчеплення (конкатенації) двох списків. Оцінити складність алгоритму за часом у термінах довжин списків.
3. Реалізувати процедуру сортування списку злиттям. Оцінити складність алгоритму за часом і пам'яттю (у гіршому випадку).
4. Реалізувати процедуру побудови списку, значущі елементи якого зберігаються у файлі F
   а) F: File of Record Key: Integer; Data: Real End; із збереженням порядку розташування елементів у файлі. Оцінити складність алгоритму за часом.
   б) F: File of Record Key: Integer; Data: Real End; Список повинен бути впорядкований за значеннями полів Key. Оцінити складність алгоритму за часом.
5. Реалізувати процедури Push і Pop відповідно добавлення і вилучення елемента для стека.
6. Реалізувати процедури Push і Pop відповідно добавлення і вилучення елемента для черги.

5. Дерева.

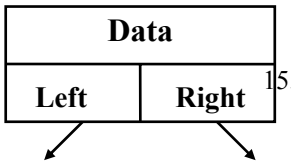
Динамічні структури даних зручно визначати за допомогою рекурсивних визначень. Список, наприклад, визначається наступним чином:

<список > ::= Nil | <елемент списку > ⟶ < список >

Тут символ "::=" означає "є за визначенням", "|" - "або", "⟶" - покажчик (посилання).

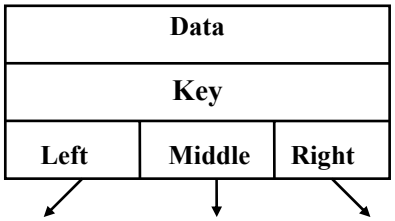
Аналогічно можна визначити і структури, що розгалужуються - так звані дерева. Елемент такої структури (вузол дерева) визначається як запис, що містить декілька полів посилань. Наприклад:

```
Type Point = ^ Item;
```

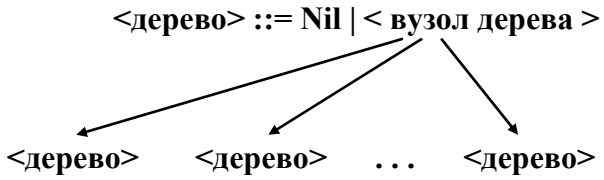


```
Item = Record
         Data: Integer;
         Right, Left: Point
       End;
```

```
Type
    Link = ^ TreeVert;
    TreeVert = Record
                 Data: Real;
                 Key : Integer;
                 Left,  Middle,  Right : Link
               End;
```



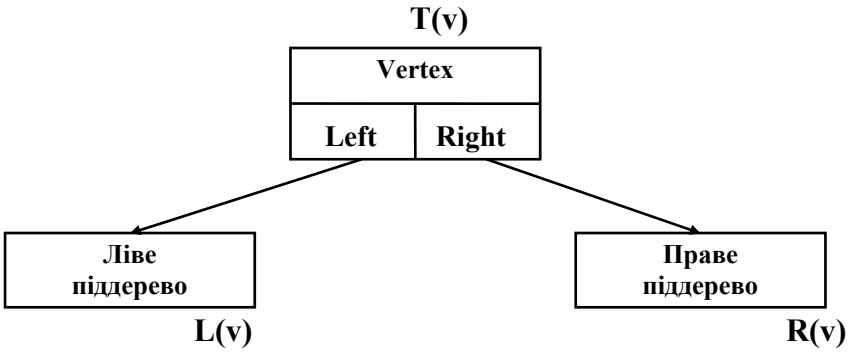
Кількість посилальних полів вузла дерева називається степенем розгалуження (арністю) вузла. Рекурсивне визначення дерева тоді має вид:



Таким чином, дерево може бути або виродженим (Nil), або складено з вузла дерева, всі покажчики якого виставлені на дерева. У цьому випадку вузол називають коренем дерева, а дерева, на які виставлені покажчики - піддеревами. Якщо піддерево складається з одного вузла, всі покажчики якого встановлені на Nil, його називають листом. Інші вузли дерева називають проміжними. Сукупність посилальних полів може бути оформлена як запис або як декілька полів запису (як у наших прикладах). Часто сукупність посилальних полів визначається в виді масиву посилань, або організується у виді списку посилань. У цих випадках говорять про впорядковані дерева, так як піддерева одного кореня виявляються впорядкованими або індексами масиву, або за порядком доступу. Якщо всі вузли дерева мають один і той же степінь розгалуження, можна говорити про степінь розгалуження дерева. Дерева, степінь розгалуження яких дорівнює двом, називають бінарними. Бінарні дерева - одна з найбільш поширених структур, що розгалужуються, які застосовуються у програмуванні.

Бінарні дерева.

Бінарне дерево має вид



Як і для списків, основними процедурами обробки дерев є пошук, вставка і вилучення вузла дерева. Ці процедури ми розглянемо на прикладі, в якому сортування послідовності реалізована за допомогою побудови і наступної обробки бінарного дерева.

Приклад 5: Нехай A = a_1, a_2,..., ,a_n - послідовність цілих чисел. Для того, щоб відсортувати послідовність A, що складається з елементів a_i

а) побудуємо бінарне дерево, що задовольняє наступній властивості: для будь-якого вузла дерева v будь-який елемент L(v) ≤ v ≤ будь-який елемент R(v)

б) побудуємо послідовність із вузлів дерева, в якій елементи розташовані у відповідності з цим же принципом: послідовність {L(v)}, v, послідовність {R(v)}.

Легко бачити, що вихідна послідовність буде впорядкованою.

Розв'язок.

Послідовність A ми реалізуємо у виді списку. Нам знадобляться процедури:

- побудови списку з чисел, що вводяться з клавіатури;
- побудови дерева, що задовольняє властивості п. а);
- побудови списку, що задовольняє властивості п. б);
- виведення списку.

Типи елементів динамічних структур даних задачі:

```
Type Point = ^ Item;
        Item = Record
                Key: Integer; Next: Point
                End;
Link = ^ Vertex;
```

```
        Vertex = Record
                        Key: Integer;
                Left, Right: Link;
                        End;
```

Програма розв'язку задачі:

```
 Program ListSort;
{описання типів даних}
      Var A, B: Point;
           LastB:Point; {кінець списку B}
             Tree: Link;
          {описання процедур}
          {процедура введення списку}
          {процедура побудови дерева з списку}
          {процедура побудови списку з дерева}
          {процедура виведення списку}

          Begin {розділ операторів}
          InpList(A);                                  {процедура введення списку}
          Tree := Nil;
          TreeBild(Tree, A);                  {процедура побудови  дерева Tree з списку A}
          B := Nil;
          ListBild(Tree, B);          {процедура побудови списку B з дерева Tree}
          OutList(B);                         {процедура виведення списку}
        End.                                  {кінець програми}

  Procedure InpList(var P: Point);    {процедура введення списку}
        Var Ch: Char;
            Q: Point;
    Begin
      P := Nil;                               {порожній список}
      Writeln('Для введення числа нажміть y');
      ch := Readkey;
      While ch = 'y' do begin
      New(Q);                                 {формування нового елемента списку}
      Writeln('Input Item:');
      Read(Q^.Key);
      Q^.Next := P;                           {включення нового елемента в список}
      P := Q;                                 {покажчик списку - на початок списку}
      Writeln('Продовжити введення? Y/N ');
```

```
      ch:= Readkey
    end
  End;

 Procedure TreeBild(var T: Link; P: Point);     {процедура побудови дерева з списку}
       Var x: Integer;
   Procedure Find_Ins(var Q: Link; x: Integer);
     Procedure Ins(var S: Link);
        Begin                                          {процедури вставки елемента}
          New(S);
          S^.Key := x;
          S^.Left := Nil; S^.Right := Nil
        End;                                           {процедури вставки елемента}

Begin                                                  {процедур пошуку і вставки елемента}
         x := P^.Key;
         If Q = Nil
           then Ins(Q)
           else if x < Q^.Key
             then Find_Ins(Q^.Left, x)
             else Find_Ins(Q^.Right, x)
       End;                                            {процедури пошуку і вставки елемента}

      Begin                                            {процедури побудови дерева з списку}
        If P <> Nil
         then begin
          Find_Ins(T, P^.Key);
          TreeBild(T, P^.Next)
        end
      End;                                             {процедури побудови дерева з списку}
```

Тонким місцем організації управління в процедурі побудови дерева є передача посилань на вершини структур, що оброблюються в виді параметрів-змінних. Це дозволяє породжувати вузли дерева без використання додаткових покажчиків.

```
 {процедура побудови списку з дерева}
    Procedure ListBild(var T: Link; var F: Point);
          Var Temp: Point;
        Begin
          If T <> Nil
            then begin
```

```
            ListBild(T^.Left, F);
            New(Temp);                    {породження нового елемента}
            Temp^.Key := T^.Key;
            Temp^.Next := Nil;
             If F = Nil
               then begin
                 F := Temp; LastB := Temp
               end
          else begin
            LastB^.Next := Temp; LastB := Temp
           end;
           {змінна LastB - покажчик на останній елемент списку, що будується}
          ListBild(T^.Right, LastB^.Next)
        end
    End;                                          {процедури побудови списку з дерева}

      Procedure OutList(var P: Point);    {процедура виведення списку}
    Var x: Integer;
  Begin
    If P <> Nil
     then begin
       Write(P^.Key, ' ');
       OutList(P^.Next)
     end
  End;                                            {процедури виведення списку}


      {процедура виведення дерева - використовувалась при налагодженні}
Procedure OutTree(var T: Link);
  Begin
    If T <> Nil
      then begin
        OutTree(T^.Left);                 {ліве піддерево}
        Write('V = ', T^.Key);            {корінь дерева}
        OutTree(T^.Right);                      {праве піддерево}
      end
  End;                                            {процедури виведення дерева}
```

У процедурах ListBild і OutTree використаний загальний принцип обробки дерева - так званий обхід дерева зліва - направо:
L(v) - обробка лівого піддерева; v - обробка кореня; R(v) - обробка правого піддерева.

Обробка L(v) і R(v) реалізована як рекурсивний виклик процедури обробки дерева. Крім обходу зліва направо, в задачах використовуються також і інші порядки переліку вузлів. Для бінарних дерев це:

L(v) - v - R(v) - зліва направо;
R(v) - v - L(v) - справа наліво;
v - L(v) - R(v) - зверху вниз;
L(v) - R(v) - v - знизу вверх.

Якщо в процедурі ListBild застосувати обхід справа наліво, елементи списку B будуть упорядкованими за зменшенням. У процедурі Find_Ins по суті застосований обхід зверху вниз у модифікації: v - L(v) або R(v) (у залежності від результату порівняння x < Q^.Key).

Якщо в виді дерева представлено арифметичний вираз (приклад 1), процедуру обчислення його значення треба програмувати обходом знизу-вверх: обчислити значення лівого операнда, правого операнда, результат бінарної операції. Неважко побачити, що якщо при побудові дерева Tree будь-яке ліве піддерево буде містити стільки же вузлів, скільки і відповідне праве, для його побудови треба $O(n*\log_2 n)$ порівнянь. Однак у гіршому випадку (якщо, наприклад, вихідний список упорядкований), для вставки кожного наступного вузла алгоритм буде слідувати по одній і тій же гілці (наприклад, лівій). Тому алгоритм тратить $O(n^2)$ порівнянь, тобто не є ефективним. Однак його можна модифікувати таким чином, щоб процедура TreeBild будувала т.н. збалансоване дерево. Тоді алгоритм сортування стане ефективним.

Задачі на деревах.

1.Клас арифметичних виразів, який містить однобуквені змінні, операції +, -, *, / і круглі дужки, назвемо класом найпростіших виразів. Реалізувати:

процедуру побудови дерева найпростішого арифметичного виразу, який заданий у виді рядка.

процедуру обчислення значення найпростішого арифметичного виразу, який заданий деревом при значеннях змінних, що вводяться з клавіатури.

процедуру перетворення дерева в рядок.

2. У параграфі 9, алгоритм пірамідального сортування, викладений метод представлення масиву в виді бінарного дерева. Реалізувати процедуру побудови цього дерева по масиву V[1..n].

3. Книга складається з 3-розділів. Кожний розділ містить 3 параграфа, а кожний параграф - 3 пункти. Всі вказані структурні частини книги мають назви. Реалізувати :

процедуру побудови змісту книги в виді тернарного дерева;

діалогову процедуру пошуку назви кожної частини книги за її номером;

виведення змісту на екран за допомогою різних обходів;

пошук розділу по його назві.

4. Реалізувати процедури порівняння двох однотипних бінарних дерев T_1 і T_2 на рівність ($T_1 = T_2$) і вкладення ($T_1 \le T_2$), використовуючи наступні рекурсивні визначення:

Через root(T) позначимо корінь дерева T. Тоді

$T_1 = T_2$, якщо $root(T_1) = root(T_2)$ і $L(root(T_1)) = L(root(T_2))$, $R(root(T_1)) = R(root(T_2))$

$T_1 \le T_2$, якщо $root(T_1)=root(T_2)$ або T_1=Nil і $L(root(T_1))\le L(root(T_2))$, $R(root(T_1))\le R(root(T_2))$

5. Реалізувати процедуру пошуку в дереві T_1 піддерева, який дорівнює T_2. Використати процедуру порівняння з попередньої задачі.

6. Реалізувати процедури, які ілюструють графічно:

побудову бінарного дерева;

пошук вузла з даним ключем різними обходами.

СТРУКТУРНЕ ПРОГРАМУВАННЯ.

Структурне програмування.

З виникненням мов високого рівня (кінець 50-х років) з'явилась можливість проектування великих програмних систем. Програмування з мистецтва комбінування машинних команд, таємницями якого володіли вибрані, перетворилося в індустрію виробництва програмного обладнання ЕОМ. До початку 70-их років витрати на виробництво програм перевищили витрати на виробництво апаратури. Тому проблема розробки ефективних і надійних програмних систем стала центральною задачею інформатики. Використання мов високого рівня зняло лише частину проблем (таких, наприклад, як проблема розподілу обчислювальних ресурсів), породивши одночасно нові проблеми (наприклад, у зв'язку з неефективністю машинного коду, що генерується транслятором у порівнянні з кодом "ручної роботи", виникла задача оптимізації). Дослідження цих проблем привели, зокрема, до формування науково-обгрунтованого стилю програмування - структурного програмування.

Структурне програмування - це технологія проектування програм, яка базується на суворому визначенні засобів мови і методів їх використання. До засобів мови відносяться стандартні типи даних і оператори управління обчисленнями.

1. Основні структури управління.

Розглянемо алгоритми розв'язку трьох задач, що відносяться до різних предметних областей:

а) Алгоритм Евкліда:

```
Program Evclid;
    Var a, b, u, v, w: Integer;
  Begin
    Read(a,b);
    u := a; v := b;
    While u <> v do begin
      w := u - v;
      If w > 0 then u := w else v := -w;
    end;
   Write(u)
 end.
```

б) Наближений розв'язок рівняння методом ділення навпіл:

```
Program EquationSol;
   Const Eps = 1e-4;
     Var a, b, u, v, w: Real;
  Begin
```

```
            Read(a, b);
            u := a; v := b;
            While u - v >= Eps do begin
              w := (u + v)/2;
              If f(u)*f(w) > 0  then u := w  else v := w;
             end;
Write(u)
End.
```

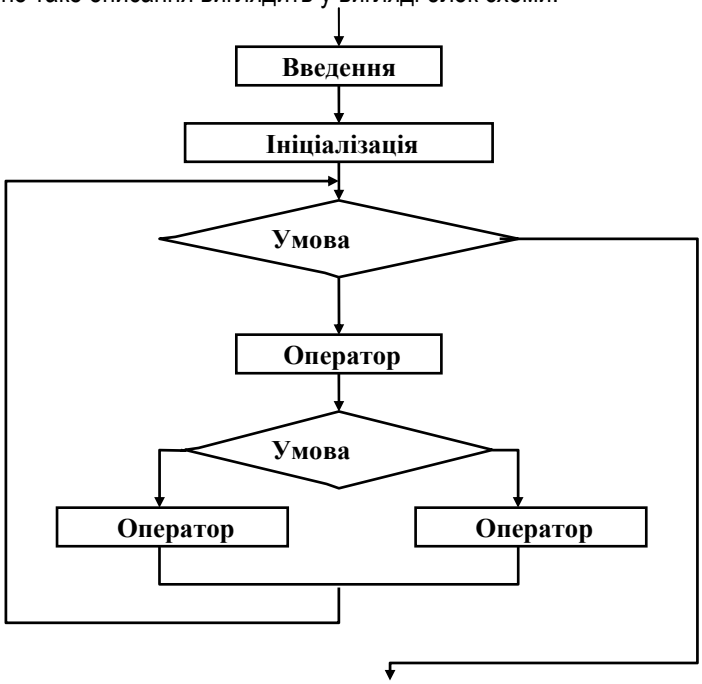
в) Розподіл масиву на два масиви за бар'єрним елементом 0:

```
Program Partition ;
          Const n = 16;
             Var f, g, h : array [1..n] of Integer;
                   x, b: Integer;
                   i, j, k: Integer;
         Begin
           { введення масиву f }
           i := 1; j := 1; k := 1;
           While f[i] <> 0 do begin
             x := f[i];
            If x > 0
              then begin
                g[j] := x; j := j + 1; i := i + 1 end
              else begin h[k] := x; k := k + 1; i := i + 1 end;
           end;
           Write(j, k)
         End.
```

Програма Evclid працює з цілими числами. Предметна область програми EquationSol - область дійсних чисел. Третя програма - Partition - оброблює масиви, причому тип компонент масиву не має суттєвого значення. Не зважаючи на різницю в предметних областях, між цими програмами існує схожість, що грає велику роль для розуміння суті програмування. Насправді, розділ операторів кожної з цих програм може бути описаний так:

```
        Begin
          < процедура введення >;
          < послідовність простих операторів >;
          while <умова> do begin
           <оператор>;
           if <умова> then < оператор > else < оператор > ;
          end;
         < процедура виведення >
       End.
```

Ще більш наочно таке описання виглядить у вигляді блок схеми:

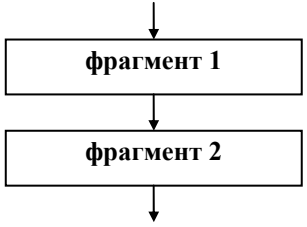


Те загальне, що об'єднує розглянуті програми, називають структурою управління. Структура управління програми грає роль її несучої конструкції, скелета. Зрозуміло, що правильне проектування програми неможливе без правильної побудови її структури управління.

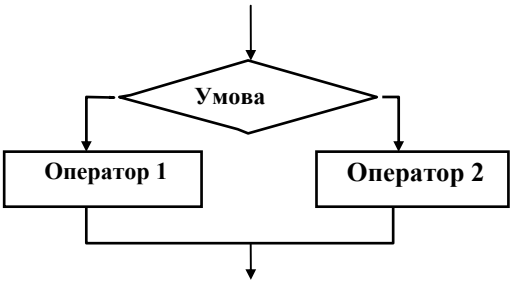
Основним досягненням у теорії програмування 60-х років є усвідомлення і теоретичне осмислення того факту, що існують декілька основних структур управління, оперування якими приводить до створення як завгодно складних за управлінням програм, причому ефективність програм при цьому не погіршується, а такі властивості, як читабельність, надійність суттєво поліпшуються. Процес програмування набуває наукових рис системного проектування.

До основних структур управління відносяться:

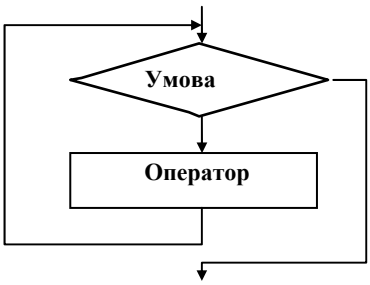
- **послідовне виконання**



•розгалуження

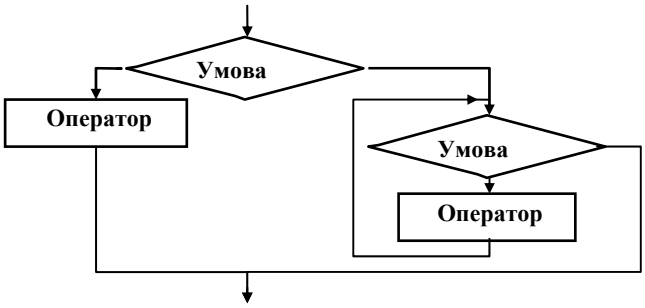


•повторення



Особливу роль у структурному програмуванні грають процедури і функції - основні семантичні одиниці програми, що проектується, які містять описання окремих підзадач, що мають самостійне значення.

Програмування здійснюється комбінуванням основних структур управління. Наприклад, комбінування розгалуження і повторення приводить до блок-схеми



Основний результат можна тепер сформулювати наступним чином:

Управління будь-якого алгоритму може бути реалізовано в виді комбінації основних структур управління.

У реальних мовах структурного програмування застосовують і інші управляючі структури, кожна з яких може бути віднесена до одного з трьох основних типів. Наприклад, у мові Pascal розгалуження - умовний оператор, короткий умовний оператор, оператор варіанта; повторення - оператор циклу з параметром, оператор циклу з передумовою, оператор циклу з постумовою. Наявність додаткових структур управління породжує надлишковість у виразних можливостях мови, що дозволяє робити програми більш природними.

Відмітимо, що оператор переходу Goto не включений ні в список основних, ні додаткових операторів управління. Це - не випадковість. Безконтрольне застосування цього оператора приводить до того, що програма втрачає властивості, що вказані вище. Тому структурне програмування часто називають програмуванням без Goto.

2. Основні структури даних.

Продовжимо аналіз прикладів, що приведені на початку параграфа. Звернемо увагу на сукупності даних, з якими працюють ці програми. Дані визначені в розділах типів, змінних і констант:

```
Program Evclid;
      Var a, b, u, v, w: Integer;

Program EquationSol;
  Const Eps = 1e-4;
    Var a, b, u, v, w: Real;

Program Partition ;
  Const n = 16;
     Var f, g, h : array [1..n] of Integer;
          x, b: Integer;
          i, j, k: Integer;
```

Сукупність даних, що визначені у програмі як змінні і константи, називається структурою даних цієї програми. Правильна побудова структури даних програми грає туку ж важливу роль для її ефективності, як і правильна побудова структури управління. Саме взаємодія цих двох аспектів визначає програму.

Теорія структур даних подібна теорії структур управління. Саме, існують основні (стандартні) структури, кожна з яких визначає один із засобів об'єднання даних у структуру. Дані простих типів при цьому грають роль цеглинок, з яких будується весь будинок.

Структури даних у мові Pascal, напевне, найбільш природним чином відображають ідеологію структурного програмування.

Прості дані: цілі числа, дійсні числа, символи, логічні значення, дані - імена.

Засоби структурування даних: масиви, записи, файли, множини, посилання.

Наприклад, визначення типів

```
Word = Array [1..20] of Char;
Man = Record
        Name: Word;
        Age: Integer
            end;
Notebook = array [1..n] of Man;
```

означає масив із записів, перше поле яких - масив із двадцяти символів, а друге - ціле число.

Необхідно розуміти, що зв'язок "дані-управління" полягає не тільки і не стільки у схожості правил їх побудови, скільки в орієнтації структур управління на обробку структур даних. Оператор присвоювання використовує дані простих типів. Логічні дані застосовують для визначення умов. Скалярні типи, що визначаються програмістом, використовують для описання даних-індексів у масивах і селектора варіанта в операторі вибору. Для обробки масивів зручно користуватись оператором циклу з параметром, а для обробки файлів - операторами While і Repeat. Записи з варіантами потрібно оброблювати операторами варіанта. Посилальні (динамічні) структури потрібно оброблювати рекурсивно. Цей список можна продовжити.

3. Методологія програмування "зверху-вниз".

Застосування у програмуванні концепції структур даних і управління потребує специфічної методології процесу розробки програм. Цей підхід називають програмуванням "зверху-вниз". Суть метода - у наступному:

1.Процес розробки програми складається з послідовності кроків, на кожному з яких програміст

- уточнює структуру програми;
- уточнює структуру даних.

2. Уточнення структури управління полягає у визначенні того оператора управління, який треба використовувати для реалізації алгоритму і тих елементів оператора, які приймають участь в управлінні (умов, меж циклів і т.п.)

3. Уточнення структури даних складається у визначенні й описанні даних, що оброблюються вибраним оператором.

4. Визначення структури даних програми починається з описання входу-виходу, тобто з визначення структури початкових і вихідних даних.

5. При роботі користуються принципом мінімальної необхідності: уточнення здійснюють лише з тим ступенем точності, яка необхідна на даному кроці.

6. При розробці декількох фрагментів програми краще уточнити кожний з них на один крок, а не один - повністю (особливо тоді, коли це уточнення пов'язано з принциповими рішеннями).

7. Основними структурними одиницями програми є процедури і функції. Кожну відносно самостійну підзадачу оформлюють як підпрограму (процедуру або функцію).

4.Приклад: Розв'язок системи лінійних рівнянь.

Розглянемо приклад програмування задачі, яка реально виникла у роботі авторів.

Задача: Реалізувати процедуру розв'язання системи лінійних рівнянь, що отримана в процесі застосування метода невизначених коефіцієнтів до задачі обчислення невизначеного інтеграла від раціонального виразу.

Розв'язок: Стислість формулювання задачі обманлива: у ній скриті ті обмеження, які роблять цю постановку точною. Для уточнення задачі необхідно по-перше, указати поле коефіцієнтів системи рівнянь, по-друге - уточнити поняття розв'язка системи у термінах точне - наближене. Розглянемо два варіанта уточнення задачі:

Варіант 1. Поле коефіцієнтів - поле раціональних чисел. Елемент цього поля - або ціле число A, або дріб, що не скорочується, виду A/B, де знаменник B - натуральне число, а чисельник A - ціле число. Треба знайти точний розв'язок системи.

Варіант 2. Поле коефіцієнтів - поле дійсних чисел. Елемент цього поля - число типу Real. Треба знайти наближений розв'язок системи з указаною точністю.

У нашому випадку мають місце наступні обмеження:

- Коефіцієнти рівнянь системи - раціональні числа;
- Необхідно отримати точний розв'язок системи у виді набору раціональних чисел;
- Завжди існує єдиний розв'язок системи;
- Кількість рівнянь системи і кількість невідомих співпадають.

На першому кроці уточнюємо задачу в виді:

```
Program LinearSystem;
   Type Solution = ...;
        LinSys = ...;
    Var S: LinSys;
        X: Solution;
        n: Integer;
    Procedure SysInp(var A: LinSys);
        Begin
```

```
    {введення системи рівнянь A}
  End;
Procedure SysSol(A: LinSys; var Y: Solution);
  Begin
    {розв'язок Y системи рівнянь A}
    End;
Procedure SolOut(Y: Solution);
   Begin
    {виведення розв'язка Y}
    End;
Begin {основна програма}
  SysInp(S);
  SysSol(S, X);
  SolOut(X)
End. {кінець програми LinearSystem}
```

На наступному кроці уточнення виникає проблема вибору метода розв'язання і уточнення структури даних задачі. Відомим точним методом розв'язання систем лінійних рівнянь є метод Гауса послідовного вилучення невідомих. Основні кроки метода - вибір ведучого рівняння і ведучого невідомого і вилучення ведучого невідомого з інших рівнянь систем. Посібники з лінійної алгебри рекомендують представляти дані у виді n*(n+1) матриці коефіцієнтів системи

x_1	x_2	. . .	x_n	Св. член
a_{11}	a_{12}	. . .	a_{1n}	b_1
a_{21}	a_{22}	. . .	a_{2n}	b_2
. . .	. . .	. . .	. . .	. . .
. . .	. . .	. . .	. . .	. . .
a_{n1}	a_{n2}	. . .	a_{nn}	b_n

Оскільки параметр n залежить від задачі, для представлення матриці A у виді двомірного масиву треба резервувати пам'ять під масив деякого найбільш можливого розміру Nmax, що приводить до неоправданих витрат оперативної пам'яті. Радикальне рішення полягає у використанні динамічних структур даних. Систему можна представити в виді списку рівнянь, а кожне рівняння - списком доданків (членів рівняння):

```
LinSys = ^LS_Item; {список рівнянь}
LS_Item = Record
            LS_Mem : LinEqu;
            NextEqu: LinSys
          End;
```

```
LinEqu = ^EquItem; {список членів рівняння}
EquItem = Record
                EquMem : Monom;
                NextMem: LinEqu
          End;
```

Компромісне рішення - динамічне резервування пам'яті під масив A:

```
LinSys = ^Array[1..Nmax, 1..Nmax] of Rational; {Rational - раціональні числа}
```

Виберемо радикальний шлях рішення і уточнимо процедури SysInp, SysSol, SolOut. Наш вибір визначає Solution як список раціональних чисел.

```
Procedure SysInp(var A: LinSys);
    Var i: Integer;
        P: LinSys;
  Begin {введення системи рівнянь A}
    Read(n); S := Nil;
    For i := 1 to n do begin
        New(P); P^.NextEqu := S; S := P;
        {введення i-того рівняння - списку P^.LS_Mem}
    end
  End;

Procedure SysSol(A: LinSys; var Y: Solution);
  Begin                                  {розв'язок Y системи рівнянь A}
    For i ;= 1 to n do begin             {прямий хід метода Гауса}
      - вибір рівняння, що містить ненульовий коефіцієнт при Xi в якості ведучого;
      - перестановка місцями i-того і ведучого;
      For j := 1 to n do
        If  i <> j
          then  - вилучення  Xi з j-того рівняння
    end;
    For i := n downto 1 do               {обернений хід метода Гауса}
      - обчислення Xi;
  End;

Procedure SolOut(Y: Solution);
    Var i: Integer;
  Begin {виведення розв'язка Y}
    For i := 1 to n do           {Виведення i-тої компоненти розв'язка}
    Writeln(' Задача розв'язана ')
  End;
```

Наступне уточнення структури даних складається у визначенні типу Monom. Якщо ми вирішимо продовжити тактику економії пам'яті, можна не зберігати члени рівняння виду $A_{ij}*X_i$ при $A_{ij} = 0$. У цьому випадку члени рівняння представляються парою

<коефіцієнт, номер невідомого >:

```
Monom = Record
          Coef: Rational;
           Unknown: Integer
         End;
```

Для спрощення структури даних змінимо визначення типу EquItem, вилучивши тип Monom:

```
EquItem = Record
            Coef: Rational;
            Unknown: Integer;
            NextMem: LinEqu
          End;
```

Тепер процедуру введення рівняння реалізуємо як локальну в процедурі SysInp. Введення рівняння - це введення лівої і правої частин рівняння, відмежованих друг від друга введенням знака "=". Введення лівої частини - повторення введень членів рівняння, відмежоване друг від друга введенням знака "+".

```
  Procedure SysInp(var A: LinSys);
      Var i: Integer;
          P: LinSys;
 Procedure EquInp(Var E: LinEqu);
      Var Q: LinEqu;
    Sign: Char;
 Procedure MemInp(var R: LinEqu);
{ тіло процедури MemInp }
    Begin
      Writeln('Введення першого члена лівої частини');
      MemInp(Q); E := Q;
      Write('Натисни = або +');
      Sign := ReadKey;
      While Sign = '+' do begin
        Writeln('Введення наступного члена рівняння);
        MemInp(Q^.NextMem); Q := Q^.NextMem;
        Write('Натисни = або +');
        Sign := ReadKey
      end;
```

```
        Writeln('Введення вільного члена рівняння);
        MemInp(Q^.NextMem); Q := Q^.NextMem;
        Q^.NextMem := Nil
        End;{процедури введення рівняння}

    Begin {введення системи рівняння A}
        Read(n); S := Nil;
        For i := 1 to n do begin
          New(P); P^.NextEqu := S; S := P;
          EquInp(P^.LS_Mem}
        end
    End;
```

Розділ операторів процедури SysInp набув закінченого вигляду. Введення члена рівняння оформимо в виді процедури MemInp, означивши тип Rational, значеннями якого є раціональні числа.

Увага! Введення чисельного типу Rational означає необхідність реалізації набору процедур, що забезпечують роботу з раціональними числами. Оскільки тип Rational необхідний для розв'язків багатьох інших задач, його треба оформити як модуль RAT і активізувати в програмі директивою Uses. Принципи проектування модуля ми розглянемо нижче. Зараз нам знадобляться тільки імена процедур введення і виведення раціональних чисел - RatInp і RatPrint.

```
Procedure MemInp(var R: LinEqu);
    Begin
        New(R);
        With R^ do begin
            RatInp(Coef);
             If Sign <> '='
                then Read(UnkNown)
                else UnkNown := n + 1
        end
    End;
```

Наш алгоритм розв'язання системи (процедура SysSol) перетворить список рівнянь у список пар виду <номер невідомого, значення невідомого >. Тому для виведення розв'язка системи немає необхідності формувати список X. Достатньо установити посилання X типу LinSys на S і вивести значення X у потрібному виді. Це значить, що тип Solution можна вилучити, зробивши відповідні зміни у заголовках процедур.

```
Procedure SysSol(A: LinSys; var Y: LinSys);
```

```
Procedure SolOut(Y: LinSys);
```
Уточнимо процедуру виведення розв'язка:
```
Procedure SolOut(Y: LinSys);
        Var i: Integer;
              P: LinSys;
      Begin {виведення розв'язка Y}
        P := Y;
        For i := 1 to n do begin
        {Виведення і-тої компоненти розв'язка}
        With P^, LS_Mem^ do begin
          Writeln('X[',i,'] = ');
          RatPrint(Coef)
        end;
        P := P^.NextEqu
      end;
      Writeln(' Задача розв'язана ')
  End;
```

Ми завершили проектування процедур введення-виведення з точністю до засобів модуля RAT. Продовжимо уточнення основної процедури програми - процедури SysSol.

```
Procedure SysSol(A: LinSys; var Y:LinSys);
       Var P, Q: LinSys;
            TempRef: LinEqu;
             i: Integer;
    Begin {розв'язок Y системи рівнянь A}
      P := A;
      For i := 1 to n do begin {прямий хід метода Гауса}
       {вибір ведучого рівняння}
        Q := P;
        While Q^.LS_Mem^.Unknown <> i do
          Q := Q^.NextEqu;
          {перестановка місцями і-того і ведучого}
          TempRef := P^.LS_Mem;
           P^.LS_Mem := Q^.LS_Mem;
           Q^.LS_Mem := TempRef;
         {вилучення Xi з рівнянь системи}
         {1. нормалізація ведучого рівняння}
         {2. вилучення Xi з верхньої частини системи}
         {3. вилучення Xi з нижньої частини системи}
         P := P^.NextEqu;
```

```
    end;
    Y := A;
End;
```

Для вибору ведучого рівняння використовується послідовний перегляд рівнянь системи за допомогою посилань Q. Єдиний розв'язок системи гарантує існування ведучого рівняння- рівняння з ненульовим коефіцієнтом при X_i.

Перестановка рівнянь здійснюється переброской посилань P^.LS_Mem і Q^.LS_Mem.

Основна частина алгоритму - вилучення невідомого X_i. Вона складається з кроків 1, 2, 3, що виконуються послідовно.

Крок 1 - нормалізація ведучого рівняння, тобто ділення всіх членів рівняння на коефіцієнт при X_i.

Крок 2 - вилучення X_i з верхньої частини системи. j-те рівняння верхньої частини (j < i) має вид $X_j + A_{ji1}*X_{i1} + ... + A_{jik}*X_{ik} = B_j$, причому i1 >= i оскільки воно нормалізовано і невідомі X_k, k < i, вилучені на попередніх кроках основного циклу. Тому вилучення здійснюється при i1 = i. Треба домножити ведуче рівняння на A_{ji1} і обчислити його з j-того рівняння.

Крок 3 - вилучення Xi з нижньої частини системи. j-те рівняння нижньої частини (j > i) має вид $A_{ji1}*X_{i1} + ... + A_{jik}*X_{ik} = B_j$, причому i1 >= i. Вилучення здійснюється точно також, як і на кроку 2.

Кроки 2 і 3 настільки схожі, що природно було би реалізувати їх одною і тою ж процедурою. Для цього необхідно всього лише реалізувати однакове представлення рівнянь з нижньої і верхньої частин системи, тобто не зберігати у списку рівняння верхньої частини системи відповідний ведучий член X_j !

Таким чином, на виході кроку 1 - процедури нормалізації ми повинні отримати "хвіст" $A_{ji1}*X_{i1} + ... + A_{jik}*X_{ik} = B_j$ нормалізованого рівняння. Основна частина алгоритму буде виглядати так:

```
{вилучення Xi з рівнянь системи}
{нормалізація і-того рівняння й отримання "хвоста"}
        EquNorm(P);
        Q := A;
        For j := 1 to n do begin
          If i <> j
    then  If Q^.LS_Mem^.Unknown = i
              {вилучення  Xi з j-того рівняння}
                then EquTrans(Q, P);
                Q := Q^.NextEqu
        end;
```

У нашій версії метода Гауса та його частина, яку називають зворотним ходом, співміщена з прямим ходом метода. Ми приводимо матрицю системи одразу до діагонального вигляду. Зазначимо, що структура даних виходу процедури SysSol змінилась: тепер і-тий член списку Y має вид <n + 1, значення X_i>. Процедура SysSol набула завершеного виду. Залишилось уточнити процедури EquNorm(P) і EquTrans(Q, P), які оброблюють окремі рівняння.

```
{нормалізація рівнянь і виділення "хвоста"}
 Procedure EquNorm(var P: LinSys);
        Var LCoef: Rational;
            TempRef: LinEqu;
      Begin
       TempRef := P^.LS_Mem;
       New(LCoef);
       LCoef^ := TempRef^.Coef^;                    {ведучий коефіцієнт }
       {установка посилання на “хвіст” рівняння}
       P^.LS_Mem := P^.LS_Mem^.NextMem;
       Dispose(TempRef);                            {збирання сміття}
       TempRef := P^.LS_Mem;                        {ініціалізація циклу}
       Repeat
         TempRef^.Coef := DivRat(TempRef^.Coef, LCoef);
         TempRef := TempRef^.NextMem                {наступний член}
       until TempRef = Nil;
       Dispose(LCoef)
   End                                                {процедури нормалізації};
```

Найбільш цікава з точки зору техніки програмування процедура EquTrans елементарного перетворення лінійного рівняння E_1 за допомогою ведучого рівняння E_2.

$E_1 := E_1 - C*E_2$

Якби рівняння були представлені масивами коефіцієнтів, це перетворення реалізувалось б за допомогою одного арифметичного циклу. У нашій же структурі даних необхідно реалізувати перетворення списків. Нехай рівняння мають вид

$E_1 = A_1*X + T_1$, $E_2 = A_2*Y + T_2$, де A_1*X, A_2*Y - перші члени списків (голови), а T_1 і T_2 - хвости списків. Можливі наступні випадки:

Випадок 1. Невідомі X і Y однакові. Тоді:

Якщо $A_1 - C*A_2 <> 0$

то $(A_1*X + T_1) - C*(A_2*X + T_2) = (A_1-C*A_2)*X + (T_1 - C*T_2)$

інакше $(A1*X + T_1) - C*(A_2*X + T_2) = (T_1 - C*T_2)$

Випадок 2. Номер X менше, ніж номер Y. Тоді:

$(A_1*X + T_1) - C*(A_2*X + T_2) = A_1*X + (T_1 - C*E_2)$

Випадок 3. Номер X більше, ніж номер Y. Тоді:

$(A_1*X + T_1) - C*(A_2*X + T_2) = (-C*A_2)*Y + (E_1 - C*T_2)$

У кожному з цих випадків права частина відповідного співвідношення - рівності є сума (голова + хвіст) результуючого рівняння, причому обчислення хвоста легко організувати за допомогою рекурсії і трохи складніше - ітерації. Ознака закінчення обчислень - вільні члени в головах списків і відсутність хвостів. Перед застосуванням описаного алгоритму (процедура EquDiff) обчислюється множник C і з рівняння, що перетворюється вилучається перший член. Всі обчислення супроводжуються “збиранням сміття” - звільненням пам’яті, що займається членами, які вилучаються.

```
Procedure EquTrans(var Q: LinSys; P: LinSys);
    Var LCoef: Rational;
        CurP, CurQ: LinEqu;
Procedure EquDiff(var RefQ, RefP: LinEqu; C: Rational);
    Var NextP, NextQ: LinEqu;
        NextC: Rational;
         II ,JJ: Integer;
        Finish: Boolean;
{Вилучення члена рівняння з нульовим коефіцієнтом}
Procedure MemDel(var RefQ: LinEqu);
     Var TempRef: LinEqu;
  Begin
      TempRef := RefQ^.NextMem;
      RefQ^.NextMem := RefQ^.NextMem^.NextMem;
      RefQ^.Coef := TempRef^.Coef;
      RefQ^.Unknown := TempRef^.UnkNown;
      Dispose(TempRef) {збирання сміття}
      End {процедури вилучення члена рівняння};
 {вставка члена рівняння}
Procedure MemInc(var RefQ, RefP: LinEqu; C: Rational);
    Var C0: Rational;
        TempRef: LinEqu;
  Begin
      New(TempRef);
      TempRef^.Coef := RefQ^.Coef;
      RefQ^.Coef := MinusRat(MultRat(RefP^.Coef, C));
      TempRef^.Unknown := RefQ^.UnkNown;
      RefQ^.Unknown := RefP^.Unknown;
      TempRef^.NextMem := RefQ^.NextMem;
      RefQ^.NextMem := TempRef;  RefQ := TempRef
  End {процедури вставки члена рівняння};

Begin
```

```
        NextQ := RefQ; {покажчики на голову списку}
        NextP := RefP;
        Finish := False; {признак закінчення обчислень}
      Repeat
        II := NextQ^.Unknown; {номера невідомих}
        JJ := NextP^.Unknown;
       {випадок 1 і випадок закінчення обчислень}
      If II = JJ
       then begin {обчислення коефіцієнта рівняння}
         NextC := SubRat(NextQ^.Coef, MultRat(NextP^.Coef, C));
         If II = n + 1
           then begin {випадок закінчення обчислень}
             NextQ^.Coef := NextC;
             Finish := True end
           else {випадок 1}
             if NextC^.Num <> 0
               then begin
                 NextQ^.Coef := NextC;
                 NextQ := NextQ^.NextMem;
                 NextP := NextP^.NextMem end
               else begin
                 MemDel(NextQ);
                  NextP := NextP^.NextMem end
              end
       else if II < JJ
         then {випадок 2}
           NextQ := NextQ^.NextMem
        else begin {випадок 3}
           MemInc(NextQ, NextP, C);
           NextP := NextP^.NextMem end
      until Finish
    End {процедури EquDiff перетворення рівняння};
    Begin
      {підготовка до виклику процедури EquDiff}
       CurQ := Q^.LS_Mem;
       CurP := P^.LS_Mem;
       New(LCoef);
       Lcoef^_:=_CurQ^.Coef^;
       Q^.LS_Mem :=_Q^.LS_Mem^.NextMem;
       Dispose(CurQ);
       CurQ := Q^.LS_Mem;
```

```
    EquDiff(CurQ, CurP, LCoef)
  End {вилучення невідомого};
```

Проектування процедур обробки рівнянь завершено з точністю до процедур арифметичних дій із раціональними числами. Наступний етап - проектування процедур модуля RAT.

5. Проектування модулів. Модуль RAT.

Модуль RAT містить всі засоби, що забезпечують застосування у програмах раціональних чисел. Сюди входять:

- описання типу Rational, констант типу Rational;
- процедури введення-виведення раціональних чисел;
- функції перетворення числових типів і типу Rational;
- функції арифметичних операцій і порівнянь раціональних чисел;
- засоби, що використовуються для реалізації вищевказаних процедур (внутрішні засоби модуля).

Проектування почнемо з визначення поняття раціонального числа. Раціональні числа - це дроби виду Num/Den, де Num - чисельник, а Den - знаменник. Для забезпечення коректності і єдиності представлення раціонального числа у виді пари <Num, Den> (дробу Num/Den) нехай виконуються наступні обмеження:

Den > 0, НОД(Num, Den) = 1, Якщо Num = 0, то Den = 1

Таке представлення ми будемо називати канонічною формою. Дії над дробами природно і зручно реалізовувати у виді функцій. Але функції мови Pascal повертають скалярні значення. Тому представлення дробу у виді запису у цьому випадку непридатне. Для того, щоб обійти цю чисто лінгвістичну перешкоду, уточнимо тип Rational як посилання на запис:

```
Type
  Rational = ^RatValue;
  RatValue = Record
  Num, {чисельник }
  Den: LongInt {знаменник}
End;
```

{Тип LongInt - один із цілочисельних типів в TP-6. Множина значень визначена константою MaxLongInt = 2^{31} - 1.}

Нижче описані деякі (далеко не всі) засоби RAT, необхідні для роботи з раціональними числами. Ключовою функцією модуля є функція CanRat, що приводить дріб до канонічної форми. У свою чергу, CanRat використовує цілочисельну функцію GCD, яка обчислює найбільший спільний дільник (НСД) двох натуральних чисел.

A / B = A div НСД(A, B) / B div НСД(A, B)

```
{----------НСД двох натуральних чисел-------------------------}
Function GCD(u, v: LongInt): LongInt;
```

```
    Begin
      While (u <> 0) and (v <> 0) do
     If u > v
       then u := u mod v else v := v mod u;
       If u = 0 then GCD := v else GCD := u
    End;

{----------канонізація раціонального числа----------}
    Function CanRat(X: Rational): Rational;
        Var u, v, w: LongInt;
     Begin
       u := X^.Num; v := X^.Den;
       If  v = 0
          then CanRat := Nil {дріб із нульовим знаменником}
          else begin
            If v < 0 {знаменник > 0}
              then begin u := -u; v := -v end;
              w := GCD(Abs(u), v);
               If  w <> 1
                then {скорочення чисельника і знаменника}
                  begin u := u div w; v := v div w end;
                  New(R);
                  Z^.Num := u;
                  Z^.Den := v ;
                  CanRat := Z;
        end
    End;
```

Як уже відзначалось, процедури введення-виведення потребують дуже старанного проектування, що враховує багато факторів (зручність використання, захист від несанкціонованих дій і т.д.). Ми опишемо тільки їх тривіальні макети.

```
{ВВЕДЕННЯ-ВИВЕДЕННЯ}
    Procedure RatInp(var X: Rational);                    {Введення}
          Var Ch: Char;
       Begin
        New(X);
        Write('введення чисельника '); Read(X^.Num);
        Ch := ReadKey;
        If Ch = '/'
          then begin
            Write('введення знаменника '); Read(X^.Den);
```

```
                X := CanRat(X)
              end
              else X^.Den := 1
          End;

        Procedure RatPrint(X: Rational);                    {Виведення}
            Begin
             If  X = Nil
                  then Write('Ділення на нуль')
                  else begin
                    Write(X^.Num);
                    If X^.Den <> 1
                       then begin Write('/'); Write(X^.Den) end;
                  end;
              WriteIn
            End;
{АРИФМЕТИЧНІ ОПЕРАЦІЇ}
        Function AddRat(X, Y: Rational): Rational;          {Додавання}
          Begin
            New(Z);
            Z^.Num := X^.Num*Y^.Den + X^.Den*Y^.Num;
            Z^.Den := X^.Den*Y^.Den;
            AddRat := CanRat(Z)
          End;

        Function SubRat(X, Y: Rational): Rational;          {Віднімання}
          Begin
            New(Z);
            Z^.Num := X^.Num*Y^.Den - X^.Den*Y^.Num;
            Z^.Den := X^.Den*Y^.Den;
            SubRat := CanRat(Z)
          End;

        Function MultRat(X, Y: Rational): Rational;    {Множення}
          Begin
            New(Z);
            Z^.Num := X^.Num*Y^.Num;
            Z^.Den := X^.Den*Y^.Den;
            MultRat := CanRat(Z)
          End;
        Function DivRat(X, Y: Rational): Rational;          {Ділення}
```

```
            Begin
              If Y^.Num = 0
                    then DivRat := Nil
                    else begin
                      New(Z);
                      Z^.Num := X^.Num*Y^.Den;
                      Z^.Den := X^.Den*Y^.Num;
                      DivRat := CanRat(Z)
                    end
              End;

        Function MinusRat(X: Rational): Rational; {число з знаком мінус}
          Begin
            New(Z);
            Z^.Num := -X^.Num;
            Z^.Den := X^.Den;
            MinusRat := Z
          End;

{ПОРІВНЯННЯ}
        Function EquRat(X, Y: Rational): Boolean; {Рівність}
          Begin
             EquRat := (X^.Num = Y^.Num) and (X^.Den = Y^.Den)
           End;

Function GrtRat(X, Y: Rational): Boolean; {Порівняння на "більш"}
  Begin
             GrtRat := SubRat(X, Y)^.Num > 0
           End;

 {ПЕРЕТВОРЕННЯ ТИПІВ}
        Function RatReal(X:Rational): Real; {Раціональні в дійсні}
          Begin
             RatReal := X^.Num/X^.Den
           End;
Function IntRat(X: LongInt): Rational; {Ціле в раціональне}
          Begin
            New(Z);
            Z^.Num := X; Z^.Den := 1;
            IntRat := Z
           End;
```

Кожна з функцій, що описані вище, реалізує одну з дій у відповідності з його математичним визначенням. Так, наприклад, додавання спирається на формулу

A/B + C/D = CanRat((A*D + B*C)/(B*D))

Оскільки модуль планується використовувати при програмуванні багатьох прикладних задач, особливу увагу треба приділити його оптимізації. З цією метою необхідно, в першу чергу, виділити і оптимізувати ключеві засоби модуля. У нашому прикладі це функції GCD і CanRat.

Функцію GCD у професійних програмах реалізують звичайно мовами нижнього рівня, використовуючи так званий бінарний алгоритм Евкліда.

Ще один шлях підвищення ефективності - зменшення середньої кількості викликів функції CanRat у функціях модуля. У функції AddRat, наприклад, це можна зробити, розглянувши поряд з загальною формулою додавання її частинні випадки, в яких виклик CanRat не потрібний. Попутно зменшується і кількість арифметичних операцій. На справді, при визначенні додавання виділимо частинні випадки додавання дробу і цілого числа, а також дробів з рівними знаменниками

A/B + C/D = CanRat((A*D + B*C)/(B*D)); (*)
A/B + C/B = CanRat((A + C)/B);
A/B + C = (A + B*C)/B;
A + C/D = (A*D + C)/D

Функція AddRat, що реалізована у відповідності з цим визначенням, у середньому ефективніше, ніж , яка описана у модулі.

Оформлення модуля.

У системі програмування TP-6 поряд із стандартними модулями можна застосовувати і власні модулі. Модуль містить: заголовок модуля, інтерфейсну частину, реалізаційну частину і ініціалізаційну частину.

Заголовок модуля має вид:

Unit < Ім'я модуля >.

Ім'я модуля повинно співпадати з іменем файла, що містить цей модуль.

Інтерфейсна частина має вид:

Interface

<

- Описання констант, міток, типів і змінних, що доступні для використання зовнішніми програмами;
- Заголовки процедур і функцій, що доступні для використання зовнішніми програмами;
- Uses-директиви з іменами модулів, що доступні для використання зовнішніми програмами.

>

Все доступні зовні засоби модуля повинні бути описані в інтерфейсі цього модуля.

Реалізаційна частина має вид:

Implementation

<

- Описання всіх засобів модуля, які скриті від зовнішніх програм;
- Uses-директиви з іменами модулів, що використовуються в цьому модулі і скриті від зовнішніх програм;
- Повні описання всіх процедур і функцій модуля.

>

Ініціалізаційна частина - розділ операторів модуля. Він виконується перед виконанням зовнішньої програми, що використовує модуль. У зовнішню програму модуль включається директивою Uses.

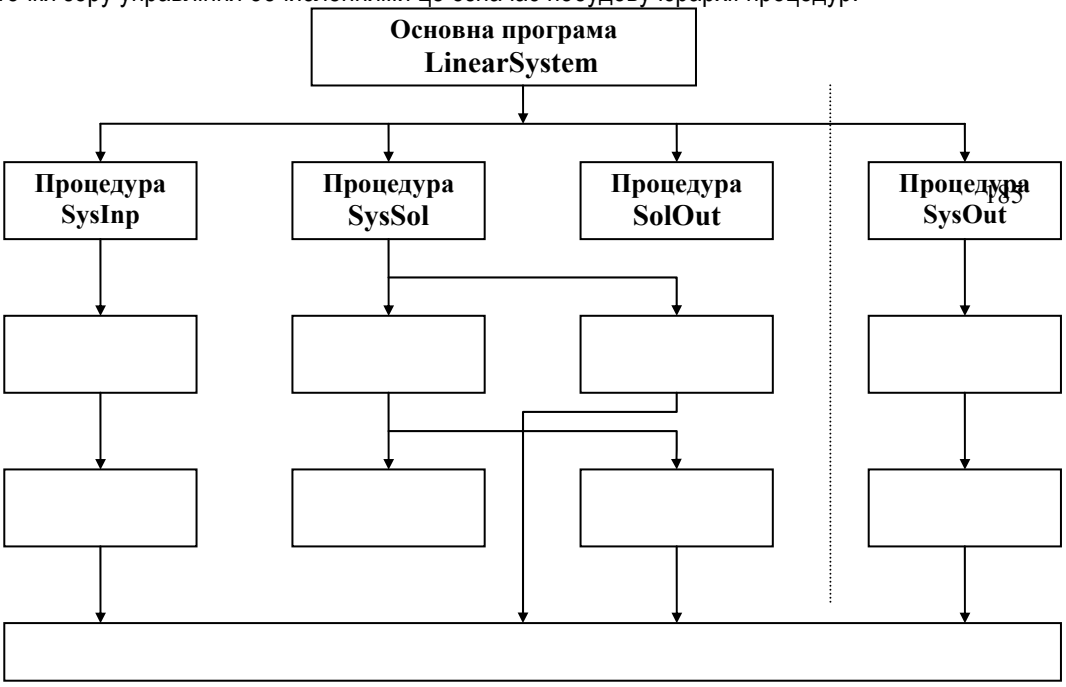
В якості приклада приведемо оформлення модуля RAT.

{заголовок модуля}

```
        Unit RAT; {<RAT.pas>}
        Uses CRT;
{інтерфейсна частина}
        Interface
           Type
             Rational = ^RatValue;
              RatValue = Record
                            Num, {чисельник }
                            Den: LongInt {знаменник}
                         End;
        Procedure RatInp(var X: Rational);
        Procedure RatPrint(X: Rational);
        Function AddRat(X, Y: Rational): Rational;
        Function SubRat(X, Y: Rational): Rational;
        Function MultRat(X, Y: Rational): Rational;
        Function DivRat(X, Y: Rational): Rational;
        Function MinusRat(X: Rational): Rational;
        Function EquRat(X, Y: Rational): Boolean;
        Function GrtRat(X, Y: Rational): Boolean;
        Function RatReal(X:Rational): Real;
        Function IntRat(X: LongInt): Rational;
{реалізаційна частина}
        Implementation
             Var Z: Rational;
 {повні описання всіх процедур і функцій модуля}
 {Ініціалізаційна частина}
 Begin
   ClrScr;
   Writeln(' Модуль Rat v.0.0 ')
 End.
```

6. Висновки.

Підведемо деякі висновки проектування програми LinearSystem. У процесі проектування "зверху-вниз" ми отримали чітко відокремлені за змістом чотири рівня програми: рівень системи, рівень рівняння, рівень члена рівняння, рівень коефіцієнтів. З точки зору управління обчисленнями це означає побудову ієрархії процедур:

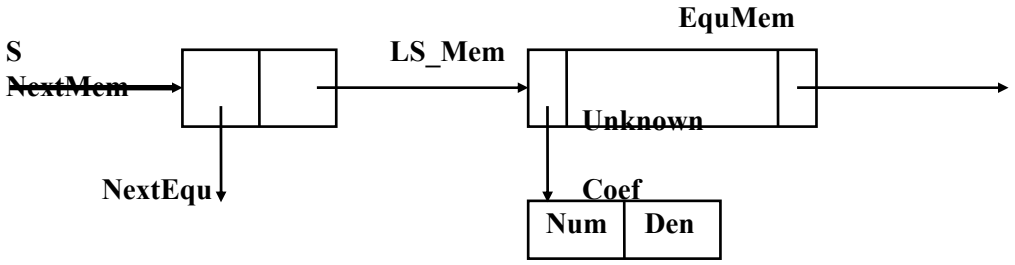


{Процедури SysOut, EquOut, MemOut використовувались для відладки і в тексті не описані.}

З точки зору даних це означає побудову ієрархії типів даних і відповідній ієрархічної структури даних:

Типи: LinSys ⟶ LinEqu {Monom} Rational

Структура даних:



{Тип Monom і поле EquMem було вирішено явним чином не вводити, оскільки на цьому рівні структура даних достатньо проста.}

Побудова програми як чотирьохрівневої ієрархії обумовлено ієрархією математичних абстракцій, що використовується в математичній теорії систем лінійних рівнянь:

Система -> Рівняння -> Член рівняння-> Коефіцієнт

Суворе слідування теорії (підкорення законам відповідній предметній області) необхідно для створення правильної (надійної) програми, а конкретна реалізація цієї ієрархії абстракцій обумовлена інженерною проблемою економії обчислювальних ресурсів.

Проектування модуля RAT представляє з себе окрему задачу - задачу з іншої предметної області. Замінивши модуль RAT, наприклад, модулем COMPLEX, у якому реалізовані дії з комплексними числами, ми отримаємо програму розв'язання системи лінійних рівнянь з комплексними коефіцієнтами. З іншого боку, наявність модуля RAT дозволяє легко програмувати і інші задачі, які потребують дій з раціональними числами.

7. Вправи.

1.Реалізувати всі функції арифметичних дій модуля RAT у відповідності з визначеннями, які аналогічні визначенню (*) функції AddRat.

2.Використовуючи модуль RAT і представлення системи лінійних рівнянь у виді n*(n+1) матриці, яка реалізована як двомірний масив, реалізувати іншу версію задачі розв'язка системи лінійних рівнянь.

3.Реалізувати модуль COMPLEX як розширення типу REAL. Підключити цей модуль замість RAT до програми LinearSystem і модифікувати цю програму для розв'язка систем лінійних рівнянь з комплексними коефіцієнтами.

4.Реалізувати модуль COMPRAT як розширення типу Rational. Використовувати цей модуль разом з RAT для модифікації програми LinearSystem, яка розв'язує системи лінійних рівнянь з комплексними раціональними коефіцієнтами.

АЛГОРИТМИ ТА ПРОГРАМИ

1. Поняття алгоритму.

Точне математичне визначення алгоритму та вивчення цього поняття - предмет спеціальної математичної дисципліни - теорії алгоритмів, яка, в свою чергу, спирається на апарат математичної логіки. Тут ми розглянемо на змістовному (неформальному) рівні лише основні характерні риси цього поняття.

Алгоритм - це правило перетворення інформації, застосування якого до заданої (вхідної) інформації приводить до результату - нової інформації.

Так, наприклад, застосування правила додавання дробів до 1/2 і 2/3 дає результат 7/6.

Основну увагу в теорії алгоритмів приділяється методам завдання (описання, конструювання) алгоритмів.

Алгоритм - це скінченний набір інструкцій по перетворенню інформації (команд), виконання яких приводить до результату. Кожна інструкція алгоритму містить точний опис деякої елементарної дії по перетворенню інформації, а також (в явному або неявному вигляді) вказівку на інструкцію, яку необхідно виконувати наступною.

Так, алгоритм додавання дробів можна задати такою послідовністю команд:

Приклад 1. Алгоритм додавання дробів.

Вхід: A/B, C/D;

8. Обчислити Y = B*D; {Перейти до наступної команди}
9. Обчислити X_1 = A*D; {Перейти до наступної команди}
10. Обчислити X_2 = B*C; {Перейти до наступної команди}
11. Обчислити X = X_1+X_2; {Перейти до наступної команди}
12. Обчислити Z = НСД(X,Y); {Перейти до наступної команди}
13. Обчислити E = X div Z; {Перейти до наступної команди}
14. Обчислити F = Y div Z; {Завершити роботу}.

Вихід: E/F

Наш алгоритм складається з 7-ми інструкцій, кожна з яких містить опис однієї з арифметичних дій над цілими числами: додавання, множення, обчислення НСД і цілочисленне ділення div. Крім того, кожна інструкція (у неявному вигляді) містить вказівку на наступну виконувану інструкцію. Таким чином, алгоритм описує деталізований по кроках процес перетворення інформації. Рівень деталізації опису визначається набором можливих команд. Цей набір називають набором команд Виконавця або Інтерпретатора.

При цьому мається на увазі, що алгоритми виконує Виконавець (Інтерпретатор) - деякий пристрій, що однозначно розрізнює і точно виконує (інтерпретує) кожну команду алгоритму.

Для виконання нашого алгоритму Виконавець повинен, очевидно, вміти оперувати з цілими числами, виділяти чисельник і знаменник дробів, а також складати з пари цілих чисел дріб. Крім того, Виконавець повинен вміти запам'ятовувати результати виконання операцій і переходити до виконання наступної команди.

Уявимо собі, що у нашому розпорядженні знаходиться Виконавець, який інтерпретує команди - операції цілочисельної арифметики - додавання віднімання, множення, обчислення неповної частки (div) та остачі (mod), обчислення НСД і НСК з запам'ятовуванням результатів і вмінням переходити до наступної команди. Тоді для цього Виконавця можна складати найрізноманітніші алгоритми арифметичних обчислень - тобто обчислень, заданих формулами типу

X = НСД ((A + B) div 100, (A * B - 7) mod 10)

використовуючи команди, аналогічні командам алгоритму з приклада 1.

Приклад 2. Алгоритм поділу відрізка навпіл за допомогою циркуля та лінійки.

Вхід: Відрізок AB;
Побудувати коло O_1 з центром A і радіусом AB;
Побудувати коло O_2 з центром B і радіусом AB;
Знайти точки C і D перетину кіл O_1 і O_2;
Побудувати відрізок CD;
Знайти точку E перетину AB і CD.
Вихід: Точка E - середина відрізку AB.

У прикладі 2 Виконавець має набір команд, за допомогою яких можна розв'язувати геометричні задачі на побудову за допомогою циркуля і лінійки.

Виконання алгоритму полягає у послідовному виконанні кожної побудови і переході до виконання наступної команди.

Відзначимо, що нумерувати команди алгоритму не треба, якщо Виконавець завжди переходить до наступної команди.

Приклад 3. Алгоритм розв'язування наведеного квадратного рівняння $x^2 + px + q = 0$;

Вхід: Коефіцієнти p і q рівняння $x^2 + px + q = 0$;
Обчислити $D = p^2 - 4q$;
Якщо D < 0 то (відповідь "Розв'язків немає"; Перейти до 1);
Якщо D = 0 то (обчислити x = -p/2; Перейти до 1);

Обчислити $x_1 = (-p + \sqrt{D}) / 2$;

Обчислити $x_2 = (-p - \sqrt{D}) / 2)$;

1: Завершити роботу.

Вихід відповідь “Розв’язків немає” або корінь х або корені x_1, x_2.

В цьому прикладі використовується команда виду

Якщо <умова>

то (<послідовність команд>)

Виконуючи цю команду, Виконавець перевіряє істинність умови. Якщо умова виконана, Виконавець переходить до виконання першої команди з послідовності команд, яка стоїть після слова то і команди, і виконання алгоритму йде тим чи іншим шляхом, що знаходиться в дужках. Якщо ж умова не виконана, Виконавець переходить до виконання наступної команди. Такі команди називають вибираючими, умовними або розгалуженнями. Вибираючі команди містять у собі інші команди, які виконуються в залежності від результатів перевірки умов.

Іншою характерною командою, яка використовується у прикладі, є команда переходу. Вона має вид Перейти до < N >, причому число N використовується в запису алгоритму як спеціальна відмітка деякої команди. У прикладі використовує команда переходу Перейти до 1, а числом 1 відмічена команда 1: Закінчити роботу.

Виконання команди переходу заключається в тому, що Виконавець переходить до виконання команди, позначеної відміткою N (порушуючи при цьому природну послідовність виконання команд).

2. Характерні властивості алгоритмів.

Поняттю алгоритму притаманні такі властивості:

1. Елементарність. Кожна команда з набору команд Виконавця містить вказівку виконати деяку елементарну (не деталізуєму більш повно) дію, яку Виконавець однозначно розуміє і виконує.

Поняття елементарності тому відносне. Так в алгоритмі прикладу 1 міститься команда обчислення НСД двох чисел. Це означає, що Виконавець вміє знаходити НСД, причому алгоритм обчислення (алгоритм Євкліда або якийсь інший) захований від людини, яка складає алгоритми для цього Виконавця. Якщо набір команд Виконавця не містить команди обчислення НСД, обчислення НСД певно визначено у виді алгоритму.

Приклад 4. Алгоритм Євкліда обчислення найбільшого спільного дільника цілих додатних чисел А і В: НСД (А,В).

Вхід A,B;
Обчислити U = A;
Обчислити V = B;
Поки U <> V виконувати
Якщо U < V
то Обчислити V = V - U
інакше Обчислити U = U - V;
Обчислити D = U.
Вихід: D - найбільший спільний дільник A і B.

У цьому прикладі використана команда повторення. Вона має вид
Поки <Умова> виконувати <Команда>
Виконуючи цю команду, Виконавець перевіряє істинність Умови. Якщо Умова істинна, Виконавець виконує Команду, вказану після слова виконувати і повторює перевірку Умови. Виконання Команди і перевірка Умови повторюються до тих пір, поки Умова істинна. Якщо Умова хибна, Виконавець переходить до виконання команди, наступної за командою повторення. В цьому ж прикладі використаний ще один різновид команди розгалуження - команда виду.
Якщо < Умова > то <Команда 1 > інакше < Команда 2 >
Виконуючи цю команду, Виконавець перевіряє Умову. Якщо умова виконана, виконується Команда 1, в іншому випадку виконується команда 2. Далі Виконавець переходить до наступної команди. Відмітимо, що команда повторення, як і команди розгалуження, містить у собі інші команди.

2.Визначеність. Виконання алгоритму суворо визначено. Це означає, що на кожному кроку Виконавець не тільки точно виконує команду, але й однозначно визначає наступну команду, що буде виконуватись. Тому повторне виконання алгоритму для одних і тих же вхідних даних у точності повторює перше його виконання.
Так, у виконанні алгоритму у прикладі 3 можливі розгалуження, які, однак, однозначно визначені умовами D < 0, D = 0.

3. Масовість. Алгоритми, як правило, описують хід рішення не одної-єдиної задачі, а цілого класу однотипних задач.

Так у прикладі 2 описаний алгоритм додавання будь-яких двох дробів. Одна-єдина задача, яка розв'язується Виконавцем у даний момент, визначена значеннями вихідних даних A, B, C, D. Зміна вхідних даних означає розв'язування іншої задачі з цього ж класу задач.

Аналогічно, алгоритм прикладу 2 будує середину будь-якого відрізка, заданого його кінцями, а у прикладі 3 за допомогою алгоритму розв'язується будь-яке наведене квадратне рівняння.

4. Результативність. Виконання будь-якого алгоритму повинно бути закінчено через скінченне число кроків (тобто виконання скінченного числа команд) з деяким результатом.

Так, у прикладі 2 результат - точка E - середина відрізку AB. Алгоритм прикладу 3 видає результат "Розв'язків немає" навіть у тому випадку, коли рівняння не має дійсних коренів, оскільки його дискримінант менше нуля.

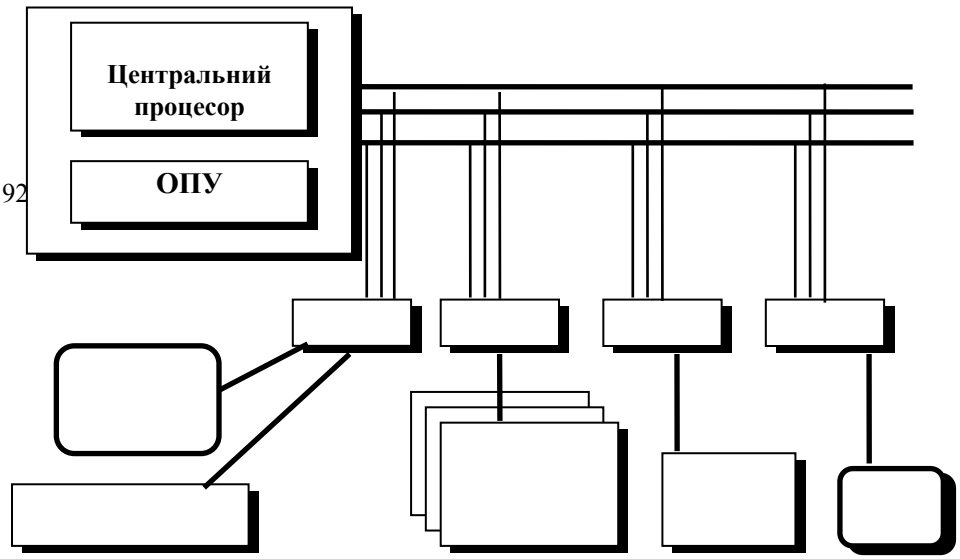
Відмітимо, однак, що кількість кроків алгоритму, який розв'язує деяку задачу, заздалегідь невідомо і може бути дуже великим, тому властивість результативності конкретного алгоритму часто потрібно спеціально доводити. Так, перевірка властивості результативності в прикладі 4 потребує доведення того, що виконання команди повторення закінчиться за скінчену кількість кроків.

Розглянуті приклади показують, що поняття Виконавця є однією з основних абстракцій, які використовують для вивчення алгоритмів. Саме система команд Виконавця і є уточнення набору засобів, за допомогою яких будуються алгоритми. У наших прикладах системи команд Виконавця предметно-орієнтовані. Так, у прикладі 1 Виконавець має справу з цілочисельною арифметикою, а у прикладі 3 - з арифметикою дійсних чисел. Особливо наочно це демонструє Виконавець прикладу 2, команди якого виконують геометричні побудови. Разом з цим очевидно, що існують команди, які повинні входити у систему команд будь-якого Виконавця, який претендує на універсальність у деякій предметній області. Це команди розгалуження, повторення, переходу. Ці команди безпосередньо не вказують на перетворення даних, а лише на керування цими перетвореннями.

Незважаючи на всю різноманітність форм представлення інформації та операцій її перетворення, які використовує людина у своїй діяльності, виявилось можливим створення універсального Виконавця, система команд якого дозволяє промоделювати будь-яку іншу систему команд. Таким Виконавцем є ЕОМ.

3. ЕОМ як універсальний Виконавець.

На рис. 1 представлена блок-схема ЕОМ.



Зовнішні пристрої ЕОМ.

Пристрої введення інформації призначені для введення інформації в ЕОМ. Пристрої введення перетворюють інформацію з форми, призначеної для користувача, у форму, призначену для збереження та обробки в ЕОМ - у двійковий код. Найбільш поширені пристрої введення - клавіатура, сканер, "миша", та інші.

Пристрої виведення інформації призначені для виведення інформації (результатів) у формі, призначеної для користувача - у виді чисел, текстів, малюнків і т.п.

Характерними для персональної ЕОМ пристроями виведення є монітор (дисплей), принтер, плотер.

Зовнішні запам'ятовуючі пристрої (ЗЗП) призначені для тривалого (при вимкненій ЕОМ) збереження інформації. У теперішній час найбільш поширені накопичувачі на гнучких магнітних дисках (НГМД), накопичувачі на твердих магнітних дисках (НМД), накопичувачі на магнітних стрічках (НМС).

Узгодження сигналів, якими обмінюються пристрої ЕОМ у процесі роботи, здійснюють інтерфейсні блоки (контролери). Інтерфейсний блок - це сукупність апаратних і програмних засобів, які підтримують процес обміну даними між пристроями ЕОМ, різними по швидкодії, рівню сигналів, засобу кодування інформації і таке інше.

Центральні пристрої ЕОМ.

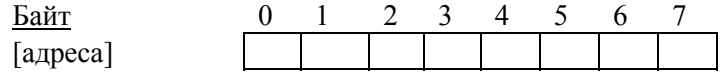
Ядро ЕОМ складають так звані центральні пристрої - центральний процесор і оперативний запам'ятовуючий пристрій (ЦП і ОЗП).

Центральні пристрої призначені для оперативного зберігання та перетворення інформації. Сукупність центральних пристроїв об'єднана в системний блок.

Вся інформація, необхідна для виконання алгоритму - початкові, проміжні і вихідні дані, а також сам алгоритм, зберігається в ОЗП у закодованому виді.

ОЗП представляє собою сукупність спеціальних комірок, кожна з яких призначена для збереження двійкового коду інформації фіксованого обсягу. Кожна комірка пам'яті має

свій номер, який називається адресою. У сучасних ПЕОМ комірка ОЗП зберігає 1 байт інформації - 8 двійкових цифр, які називаються бітами.



Характерною особливістю (ОЗП) є те, що доступ до даних, які там зберігаються, здійснюється за їх адресами. Час доступу до даного не залежить від адреси комірки, в якій воно зберігається. Тому ОЗП називають пам'яттю з прямим (випадковим) доступом. Розміром (об'ємом) ОЗП називають кількість її комірок. Розмір ОЗП виражають у байтах, кілобайтах (Kb) і мегабайтах (Mb). 1 кілобайт = 1024 байта, 1 мегабайт = 1024 кілобайта. І дані, і команди як правило, займають у ОЗП декілька підряд адресованих байтів.

Центральний процесор - пристрій, призначений для виконання алгоритмів, які зберігаються в ОЗП у виді набору команд. Кожний центральний процесор має свою систему команд Виконавця. Система команд реального процесора містить десятки команд, і її вивчення - предмет окремого курсу навчання. Ми розглянемо лише основні принципи побудови машинної мови.

Поняття про машинну мову.

Набір команд процесора містить:

арифметико-логічні команди - команди арифметичних дій над двійковими числами і логічних дій на двійковими векторами;

команди управління - команди переходу, розгалужень, повторень, і деякі інші команди;

команди пересилання даних - команди, за допомогою яких обмінюються даними ОЗП і ЦП;

команди введення-виведення даних - команди, за допомогою яких обмінюються даними ЦП і зовнішні пристрої.

Кожна команда містить код операції, яку вона виконує і інформацію про адреси даних, над якими ця операція виконується. Крім цього, команда (безпосередньо - команди керування і опосередковано - інші команди) містить інформацію про адресу команди, яка буде виконуватися наступною. Таким чином, будь-яка послідовність команд, яка розміщена в ОЗП, фактично представляє собою алгоритм, записаний в системі команд процесора - машинну програму.

Найбільш поширеною зараз є схема ЕОМ з загальною шиною. Загальна шина - це центральна інформаційна магістраль, яка зв'язує зовнішні пристрої з центральним процесором. Вона складається з шини даних, шини адреси і шини управління. Шина даних призначена для обміну даними між ОЗП і зовнішніми пристроями. По шині адреси передаються адреси даних. Ця шина однонаправлена. Шина управління служить каналом обміну управляючими сигналами між зовнішніми пристроями і центральним процесором.

Таким чином, мова процесора - це набір команд, кожна з яких описує деяку елементарну дію по перетворенню інформації, яка представлена у двійковому коді. Універсальне використовування двійкового коду представлення інформації самих різноманітних форм приводить до того, що програма рішення навіть достатньо простої задачі містить сотні машинних команд. Написати таку програму, використовуючи машинні команди, дуже непросто навіть кваліфікованому програмісту. Реальні програми складаються з десятків і сотень тисяч машинних команд. Тому будь-яка технологія проектування програми повинна опиратися на заходи, характерні для людського мислення, оперувати звичними для людини поняттями з тієї предметної області, якій належить задача.

Іншими словами, програміст (проектувальник алгоритмів) повинен мати можливість сформулювати свій алгоритм мовою звичних понять; потім спеціальна програма повинна виразити ці поняття за допомогою машинних засобів, - здійснити переклад (трансляцію) тексту алгоритму на мову машини.

Ця необхідність і привела до появи мов програмування високого рівня як мов запису алгоритмів, які призначені для виконання на ЕОМ.

МОВИ ПРОГРАМУВАННЯ І СИСТЕМИ ПРОГРАМУВАННЯ.

1. Мови програмування високого рівня.

Мови програмування високого рівня грають роль засобу зв'язку між програмістом і машиною, а також між програмістами. Ця обставина накладає на мову багато обов'язків:

1.Мова повинна бути близькою до тих фрагментів природничих мов, які забезпечують конкретну предметну область діяльності людини; (Мова, яка орієнтована на ділові сфери використань, повинна містити поняття, які використовуються у цьому виді діяльності: рахунок, база даних і т.п.).

2.Всі засоби мови повинні бути формалізовані у такому степені, щоб їх можна було реалізувати як машинні програми;

(наприклад, речення "Знайти документ X у базі Y" повинно генерувати програму в машинній мові, яка здійснює потрібний пошук).

3.Мова програмування не тільки підтримує предметно-орієнтовну діяльність , але і стимулює її розвиток (поняття бази даних, обчислювальної мережі привело до революції у діловій діяльності).

4.Мова програмування - дещо більш, чим засіб опису алгоритмів: він несе в собі систему понять, на основі яких людина може обдумувати свої задачі, і нотацію, за допомогою якої він може виразити свої розуміння з приводу рішення задачі.

Вивчаючи нову мову програмування, краще всього до неї відноситися, як до будь-якої іншої іноземної мови: засоби мови приймати як дані від Бога, навіть якщо вони нам здаються незрозумілими, поганими або непотрібними.

2. Коротка історія розвитку мов програмування.

Ідея мови програмування з'явилась так само давно, як і універсальні обчислювальні машини - на межі 40-50 років. Вже на перших кроках їх експлуатації виявилися недоліки використання машинного коду, визначились методи усунення або зменшення цих недоліків: використання бібліотек стандартних програм, імен замість адрес, попереднього розподілу пам'яті і т.п.

Великий вплив на наступні розробки виявила мова Fortran, створена у IBM під керівництвом Дж. Бекуса (1954-57 р.р.). У той же час М.Г.Хоппер (Ramington-Rand Univac) і її група розробили мову обробки комерційної інформації Flow-Matic. М.Г.Хоппер належить термін "компілятор". Таку назву мала її перша програма, яка транслювалась.

Перші виробничі мови програмування з'явились на межі 50-60 років, знаменуючи собою нову епоху в розвитку обчислювальних машин і методів обробки інформації. Ці мови високого рівня були реалізовані на перших ЕОМ 2-го покоління.

Ось деякі дати:

1957 р. Fortran США, IBM, Дж. Бекус: по суті перша широко застосовувана мова, орієнтована на науково-інженерні і чисельні задачі.

1960 р. Cobol США, Об'єднаний комітет виробників та користувачів ЕОМ: мова для комерційних задач.

1960 р. Algol-60. Поліпшений варіант мови Algol-58, Європа, США, міжнародна робоча група: універсальна мова, пращур Pascal -я і багатьох інших мов європейського стилю.

1965 р. BASIC Дж. Кемені, Т.Куртц, США, Дармутський коледж: мова для починаючих.

1969 р. Logo С.Пейперт, США, Массачусетський технологічний інститут: мова для дітей.

1966 р. PL-1 група IBM, США: Багатоцільова мова для систем колективного користування.

1968 р. Algol-68. Європа, міжнародна робоча група: європейська відповідь на PL - 1.

1970 р. Pascal Н. Вірт, Швейцарія, федеральний інститут технології, Цюрих: мова для навчання спеціалістів в області інформатики.

1959 р. Lisp Дж.Маккарті, США, Массачусетський технологічний інститут: мова функціонального програмування.

1972 р. Prolog А.Колмєрое і його колеги з лабораторії Штучного інтелекту, Марсельський університет, Франція: мова логічного програмування, що завоювала широку популярність як мова для задач обробки баз знань.

1972-75 р.р. С і його розвиток С++. Д.Креніган, Д.Річі, Б.Страуструп, AT & T Bell Lab.,: мови системного програмування, які отримали широке розповсюдження завдяки своїй ефективності і підтримці ведучих компаній, що розробляють програмне забезпечення.

1975 р. Modula - 2 Н. Вірт, Розвиток мов Pascal і Modula для системного програмування.

Перші мови програмування несли у собі явно виражені ознаки орієнтації на структуру ЕОМ. Вважалось, що програми, написані ними, призначені для виконання на ЕОМ. (Fortran - програми до цих пір пишуть на спеціальних бланках, орієнтованих на перфорацію. Ще одна яскраво виражена ознака машинної орієнтації - мітки і оператор GOTO.)

В результаті теоретичного осмислення процесів, які відбувалися у програмуванні, був вироблений так званий структурний підхід до написання програм, а для його реалізації розроблені такі мови, як Pascal, Modula - 2. Ідеологи структурного підходу вважають, що ЕОМ призначені для виконання програм, а не програми - для виконання на ЕОМ.

Перенесення акцентів з ЕОМ на програми ще більш яскраво виявилось у появі принципово нових стилів програмування - функціонального програмування (Lisp), логічного програмування (Prolog), алгебраїчного програмування (Reduce, APS).

У цих мовах центральну роль грають не процедури обробки даних, а співвідношення між даними, які повинні виконуватись у процесі виконання програми. Тому

ці мови, на відміну від процедурних (вказівних, імперативних), отримали назву декларативних (описових).

Сучасний етап у розвитку програмування характеризується наступними рисами:

- Розвиток мов програмування для мультіпроцесорних і мультімашинних систем;
- Розвиток декларативних мов програмування, орієнтованих на задачі штучного інтелекту;
- Розвиток об'єктно-орієнтованих мов, в яких ієрархія абстракцій дозволяє нарощувати засоби мов, одночасно змінюючи архітектуру ЕОМ стосовно до класу проблем, який розглядається.

3. Основні етапи проектування програми.

Процес проектування програми подібний процесам проектування складних систем. Ось його основні етапи:

- Постановка задачі і вибір її математичної моделі;
- Розробка алгоритму розв'язування задачі;
- Вибір апаратного обладнання і мови програмування;
- Написання програми;
- Налагодження і редагування;
- Контрольні випробування;
- Експлуатація.

Насправді проектування програми - не лінійний, а циклічний процес, на кожному кроку якого можливе повернення до будь-якого з попередніх кроків:

У теперішній час праця програміста у значній мірі регламентована спеціальними методиками, довідковими посібниками і стандартами. Розробники операційних систем і систем програмування велику увагу приділяють спеціальним пакетам програм, які автоматизують працю програміста. Але, не дивлячись на все, процес розробки програм все ще далекий від досконалості.

4.Технологія трансляції програм.

Взагалі ЕОМ не розрахована на те, щоб розуміти LOGO, Pascal або інші мови програмування високого рівня. Апаратура розпізнає і виконує тільки машинну мову, програма на якій являє собою не більш ніж послідовність двійкових чисел.

Поява мов програмування була пов'язана з осмисленням того факту, що переклад алгоритму, написаного "майже" природною мовою, на машинну мову може бути автоматизований і, отже, покладений на плечі машини. Тут важливо розрізняти мову і її реалізацію. Сама мова - це система запису, яка регламентується набором правил, що визначають його лексику і синтаксис. Реалізація мови - це програма, яка перетворює цей запис у послідовність машинних команд у відповідності з прагматичними і семантичними правилами, визначеними у мові.

Існують два основних засоби реалізації: компілятори і інтерпретатори.

Компілятори транслюють весь текст програми, написаний мовою програмування, у машинний код у ході одного безперервного процесу. При цьому створюється повна програма у машинних кодах, яку потім можна виконувати без участі компілятора.

Інтерпретатор у кожний момент часу розпізнає і виконує по одному реченню (оператору) програми, за ходом справи перетворюючи речення, яке оброблюється у машинну програму. Різниця між компілятором і інтерпретатором подібна різниці між синхронним перекладом усної мови і письмовим перекладом тексту.

У принципі будь-яка мова може бути і компилюємою, і інтерпретируємою, однак у більшості випадків кожна мова має засіб реалізації, якому віддають перевагу. Напевно, така перевага - дещо більше, ніж данина традиції. Вибір визначений самою мовою. Fortran, Pascal, Modula-2 в основному компілюють. Такі мови як Logo, Fort майже завжди інтерпретують. BASIC і Lisp широко використовується в обох формах. Кожний з цих засобів має як свої переваги, так і недоліки:

Основні переваги компіляції:

Швидкість виконання готової програми;

Незалежність програми від системи реалізації;

Основні недоліки компіляції:

Деякі незручності, які перевіряються програмістом при написанні і налагодженні великих програм;

Порівняно великий об'єм пам'яті, який займає компілятор в ОЗП;

5. Поняття про систему програмування.

Розглянемо послідовність змін, що відбуваються з програмою під час виконання (у процесі компіляції).

Текст програми, який написаний мовою програмування, називається вихідним модулем. Достатньо складні програми можуть складатися з кількох модулів, взаємодіючих друг з другом. Вихідний модуль - це вхідний потік для програми - компілятора, яка

- Здійснює лексичний аналіз вхідного потоку [блок лексичного аналізу];
- Здійснює синтаксичний аналіз вхідного потоку [блок синтаксичного аналізу];
- Генерує машинний код, який є перекладом вихідного модуля на мову ЕОМ в умовних адресах [генератор коду];

В результаті цих перетворень на виході отримується об'єктний модуль.

Навіть якщо ми маємо справу з одним вихідним модулем, для успішного виконання програми необхідно "зв'язати" її з деякими іншими програмами (наприклад, з стандартними процедурами введення-виведення, які реалізовані в мові). Ці функції виконує програма - редактор зв'язків. Вихідний потік цієї програми називають завантаженим модулем.

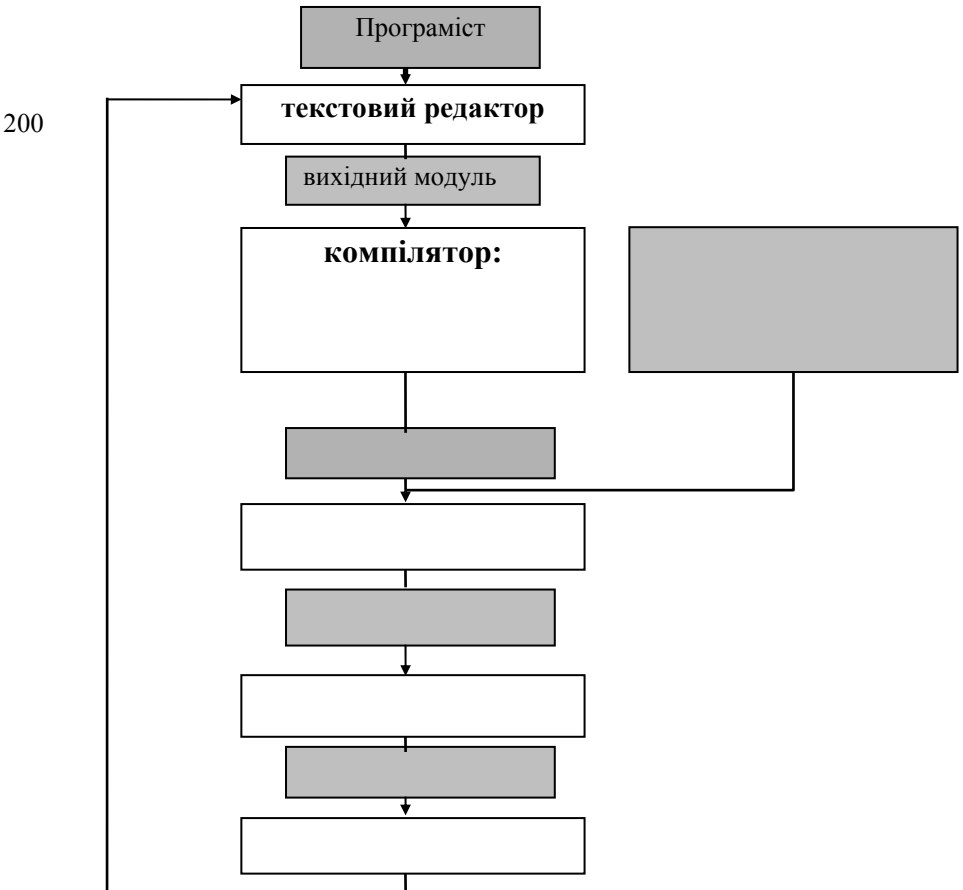
Сучасна технологія застосовування ЕОМ потребує, щоб виконуючу програму можна було розміщувати в довільному місці ОЗП. Тому і завантажений модуль написаний в умовних адресах. Розміщенням завантаженого модуля в пам'ять займається програма - завантажувач. Як правило, програми, які щойно написані, містять безліч помилок. Помилки бувають:

а) синтаксичні і лексичні (виявляються на етапах лексичного і синтаксичного аналізу). Наприклад, помилка y := sos(x) замість y:= cos(x) - лексична, а помилка в операторі if x < 0 then y:= 0; else y:= 1 - синтаксична.

б) семантичні (виявляються на етапі налагодження). Наприклад, помилка в операторі присвоювання - ділення на нуль: x:= y; z:= 1/ (x - y).

в) логічні (виявляються на етапі контрольних випробувань). До логічних ми відносимо такі помилки, в результаті яких програма не виконує того, що ми від неї чекаємо. Для автоматизації процесу пошуку і усунення семантичних і частково логічних помилок використовуються спеціальні програми, які називають налагоджувачами.

Процес перетворення тексту вихідного модуля в модуль, який виконується можна зобразити схематично:



Звичайно в склад системи програмування включають власний текстовий редактор, інші сервісні програми. Для керування роботою будь-яких частин системи програмування використовують керуючу програму, яку називають оболонкою.

Таким чином, для роботи в мові програмування використовуються спеціальні пакети програм, які називають системами програмування (СП). У склад СП входять:

- Оболонка
- Текстовий редактор
- Компілятор
- Редактор зв'язків
- Завантажувач
- Налагоджувач
- Бібліотеки стандартних процедур і функцій
- Сервісні програми

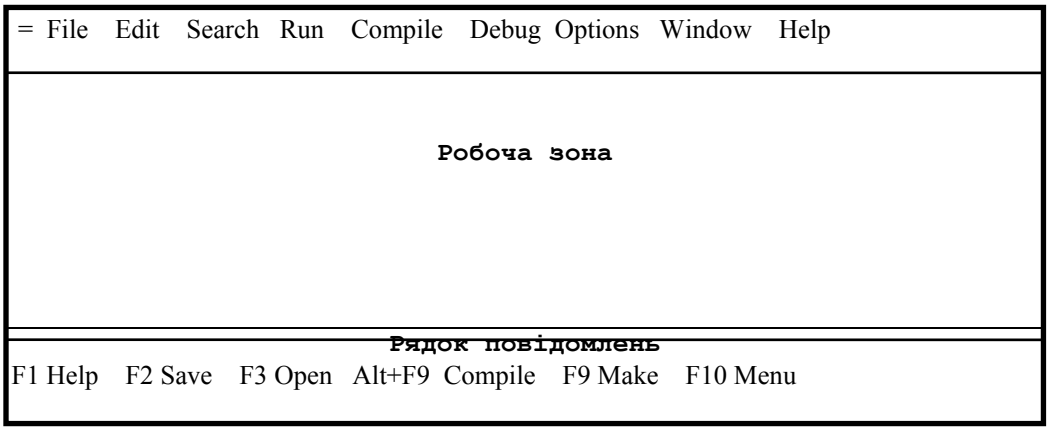
6. Короткий опис системи Turbo Pascal.

Система програмування Turbo-Pascal (TP) була створена фірмою Borland. Перша версія системи, яка з'явилась на ринку в 1985 році, (TP v. 3.0) швидко завоювала популярність завдяки добре реалізованій ідеї інтеграції всіх частин системи в єдиному середовищі програмування. У теперішній час як професіоналами, так і в навчанні активно використовується система TP v.6.0, можливості якої значно розширені. Нещодавно на ринку з'явилась наступна - 7-ма версія системи і треба очікувати, що цей ряд буде продовжуватись.

Інтегрована Інструментальна Оболонка (ІІО). Виклик системи TP виконується директивою Turbo, яка ініціює екран ІІО. Весь процес програмування виконується в середовищі ІІО.

Екран Інтегрованої Інструментальної Оболонки

Зона головного меню



Екран ІІО ТР розділений на 3 зони: - зона головного меню, робоча зона, рядок повідомлень.

Зона головного меню (верхній рядок екрану) містить набір груп команд, які використовуються програмістом при роботі в системі ТР.

=	етикетка системи, команди роботи з екраном;
File	команди роботи з файлами;
Edit	команди редагування файлу, які опрацьовують групи рядків;
Search	команди пошуку і заміни;
Run	команди запуску програми на компіляцію і наступне виконання;
Compile	команди запуску програми на компіляцію;
Debug	команди налагоджувача програм;
Options	команди конфігурування системи;
Window	команди роботи з різними вікнами;
Help	довідкова інформація про систему.

Рядок повідомлень (нижній рядок екрану) демонструє деякі команди, які найбільш часто використовують ІІО і комбінації клавіш для їх швидкого виконання (не звертаючись до головного меню). Наприклад, комбінація Alt - F9 запускає програму на компіляцію. В цей рядок система також вводить службові і довідково-інформаційні повідомлення.

Основне місце на екрані займає **Робоча зона**. В робочій зоні розташовано декілька вікон, в які виводяться тексти програм, вхід/ вихід програми, відладочна інформація і т.п. Будь-яке з вікон може бути активізоване (зроблено доступним для редагування). Кожне вікно можна закрити (за допомогою “мишки”, комбінацією Alt - F3 або відповідною командою головного меню).

На закінчення відмітимо, що робота в системі ТР потребує певних навичок, які вироблюються достатньо швидко в процесі практичної роботи в системі з використанням як довідкових матеріалів самої системи, так і спеціальних учбових посібників з роботи в ТР.

МОВА ПРОГРАМУВАННЯ PASCAL (ВСТУП).

Мова програмування Паскаль розроблена видатним швейцарським ученим і педагогом в області програмування Н.Віртом. Попереднє повідомлення з'явилось в 1968 р. В 1971 році запрацював перший компілятор переглянутої версії, яка набула статус стандарту.

Паскаль призначався для навчальних цілей - для викладання основ програмування студентам - майбутнім спеціалістам в області інформатики. Дуже швидко мова завоювала популярність не тільки в середовищі викладачів і студентів, але й серед професіоналів завдяки своїй компактності, гнучкості, старанній розробці концепції.

Існуючі сьогодні реалізації мови, зберігши його стандарт в якості ядра, володіють дуже потужними додатковими засобами, що сприяє широкому використанню мови.

1. Алфавіт мови.

В мові використовуються:

7.Латинські букви (великі і маленькі), знак підкреслення "_"

8.Цифри 0,...,9

9.Математичні символи +, -, *, /, <, >, =

10.Роздільники: ; " ' . : ^

11.Дужки ( ) [ ] { }

12.Інші символи (що використовуються для друку): букви національних алфавітів, !, ?, \, ...

У різних версіях можуть використовуватися різні набори символів. Зараз широко використовується набір символів коду ASCII (American Standard Code for Information Interchange).

Цей код передбачає розширення для національних алфавітів, символів псевдографіки, які можуть змінюватись від версії до версії.

2. Концепція даних.

Дані є загальне поняття для всього того, з чим оперує ЕОМ. Мови програмування дозволяють нам абстрагуватись від деталей представлення даних на машинному рівні перш за все за рахунок введення поняття типу даних.

У мові Pascal представляються числа і рядки.

Цілі числа записуються в десятковій системі числення: 137, -56, +26.

Дійсні числа використовують також десяткову нотацію, причому ціла частина відокремлюється від дробової не комою, а крапкою. Для позначення порядку числа в якості роздільника використовується буква E. Наприклад, -5.1E14 означає -5.1, помножене на 10 у степені 14 ($-5{,}1*10^{14}$). Порядки можуть бути і від'ємними: 6.74E-8, -56.89E-10.

Послідовності символів в лапках, називаються рядками. Якщо у рядок треба включити лапки, то замість неї записують дві лапки:

' рядок з символів ', ' апостроф ' ' у слові '

В мові дуже старанно в відповідності з концепцією структурного програмування пророблено поняття типу даних. Існують так звані прості (елементарні) типи, які або визначені як стандартні, або визначаються програмістом. Визначений також механізм описання і використання складних типів з більш простих як з базових.

3. Імена та їх застосування.

Іменем в мові називається послідовність (латинських) букв, знака підкреслення "_" і цифр, що починається з букви або знака підкреслення. Хоча імена можуть бути будь-якої довжини, в реалізації кількість значущих символів в імені може бути обмежено. У стандарті мові імена розрізняються по першим восьми символам. Це означає, що імена VeryLongNumber, VeryLongCardinal у стандарті мови позначають (іменують) один і той же об'єкт. Крім цього, мова не розрізняє великих і маленьких букв. Тому імена Sin, SIN, sin не розрізняються.

Імена використовуються для позначення констант, типів, змінних, процедур і функцій. Наприклад:

Pi, Constant - імена констант; x, y1, y2, Counter - імена змінних;

Integral, MaxMin - імена процедур; Man, Color, WeekDay - імена типів;

Деякі імена визначені наперед. Наприклад:

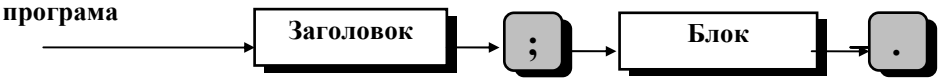
Sin - ім'я для позначення функції синус; Read - ім'я для позначення процедури читання;

Вони називаються стандартними. Всі інші імена вибираються програмістом на його погляд. Однак, для того, щоб програма краще читалась, рекомендується вибирати імена, що несуть інформацію про названий об'єкт.

Нажаль, практично всі реалізації мови допускають використання тільки латинських букв в іменах, тому програми не так зрозумілі неангломовним користувачам, як цього хотілось би.

4. Структура Pascal-програми.

Програма на мові Pascal складається з заголовку і блоку. Блок називають тілом програми. Заголовок програми відділений від тіла крапкою з комою. Крапка, що стоїть після блоку, служить ознакою завершення програми. Таким чином, програма має вид:



Відзначимо, що малюнок, приведений вище, більш точно, наочно і коротко визначає поняття програми, ніж попередній текст. Тому в подальшому ми, наслідуючи традицію, що йде від автора мови, будемо застосовувати саме такий спосіб означень мовних конструкцій, який називають мовою синтаксичних діаграм.

5. Поняття про лексику, прагматику, синтаксис і семантику мови.

Будь-яка мова програмування схожа з природними мовами. Як і природна мова, вона має свій алфавіт, словник, знаки пунктуації (роздільники), за допомогою яких можна утворювати більш складні мовні конструкції, подібні реченням природної мови. Словник мови програмування складається з чисел, слів і деяких інших символів. Елементи цього словника називають лексемами. Приклади лексем:

394, -5678, 12.456, 67.5e8 - числа; Integer, Cos, MaxInt - імена; (,) - дужки.

Мова програмування містить набір правил побудови лексем. Сукупність цих правил називається лексикою мови. Лексичні правила для чисел, строк і імен наведені вище.

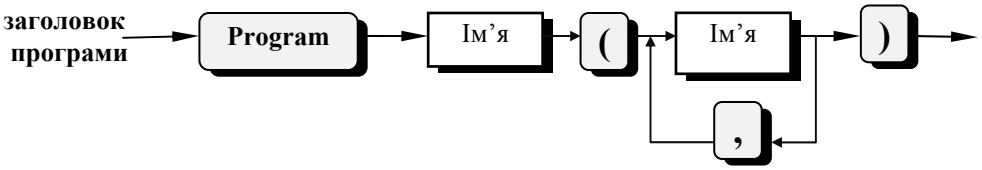
Оскільки текст будь-якої програми є послідовність лексем, основна задача лексичного аналізу - перевірка правильності написання і ідентифікація лексем в цьому тексті. Кожна лексема має свою інтерпретацію (смисл). Так, послідовність цифр, розмежована крапкою, інтерпретується як дійсне число в десятинній нотації, а Cos - як ім'я функції. Сукупність інтерпретацій лексики мови називається його прагматикою.

Правила утворення більш складних конструкцій мови називаються синтаксичними. Сукупність синтаксичних правил утворює синтаксис мови програмування. Одно з синтаксичних правил - правило описання заголовку програми - наведено вище.

Також , як і лексеми, інші конструкції мови інтерпретуються як дії або описання. Наприклад, оператор присвоювання x := x+2 має смисл "скласти значення змінної x з числом 2 і результат інтерпретирувати як (нове) значення цієї ж змінної". Сукупність інтерпретацій синтаксичних правил називається семантикою мови. Можна сказати, що вивчення мови програмування полягає у вивченні його синтаксису і семантики.

6. Синтаксичні діаграми як засіб визначення мови.

Правила побудови синтаксичних діаграм пояснимо на прикладі діаграми заголовку програми:



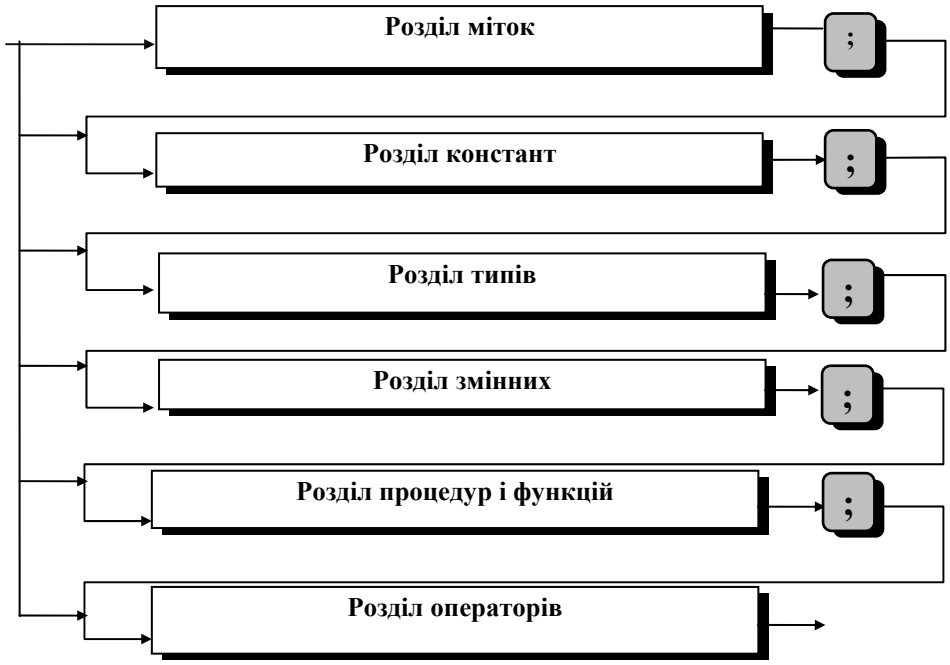
Мовна конструкція, яка визначається діаграмою, указана на початку діаграми (у прикладі це - заголовок програми). Власно діаграма має вид схеми, рух уздовж якої визначає синтаксично правильну мовну конструкцію.

В діаграмі виділені два види об'єктів: термінальні символи або ланцюжки (лексеми) і мовні конструкції, що визначаються іншими (зокрема, цією ж) діаграмами. Вони називаються нетермінальними об'єктами. Термінальні об'єкти обмежуються в овали, а нетермінальні - у прямокутники. Направлення руху уздовж діаграми (обходу) вказують стрілки, які з'єднують об'єкти.

Треба відмітити, що існують і інші метамови (мови описання мов), наприклад - мова нормальних форм Бекуса-Наура.

На завершення наведемо визначення блока мовою синтаксичних діаграм:

Блок



Таким чином, блок програми складається з шести розділів, кожний з яких, за винятком розділу операторів, може бути відсутнім. Розділи блоку відокремлені один від одного крапкою з комою.

ПРОСТІ ТИПИ ДАНИХ. ЛІНІЙНІ ПРОГРАМИ.

Лінійними (нерозгалуженими) називають програми, в яких відсутні обчислення, зв'язані з перевіркою деякої умови і вибором того чи іншого продовження обчислень у залежності від значення цієї умови (розгалуження). Будь-яка складна програма містить розгалуження. Разом з тим будь-яка програма містить лінійні фрагменти.

Для того, щоб складати найпростіші лінійні програми, необхідно вивчити поняття заголовку програми, розділу констант, розділу змінних, розділу операторів.

1. Заголовок програми.

Синтаксична діаграма заголовку програми зображена на сторінці 15.
У стандарті мови використовуються стандартні файли з іменами:
Input - вхідний стандартний файл (ім'я стандартного пристрою введення).
Output - вихідний стандартний файл (ім'я стандартного пристрою виведення).

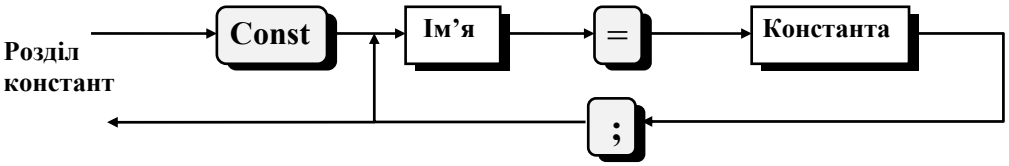
Приклади заголовка :

program LinearUnequation (Input, Output);
program GrafTrans;

Однак практично у всіх реалізаціях мови на ПЕОМ ці імена можна опускати, оскільки вони визначені за замовченням (клавіатура і алфавітно-цифровий екран користувача). Тому далі використовуються заголовки програм без цих параметрів.

2. Константи і їх використання. Розділ констант.

Розділ констант визначається наступною діаграмою:



В розділі констант визначаються імена як синоніми констант. Під константою розуміється або деяке число, або ім'я константи, можливо з знаком, або рядок.

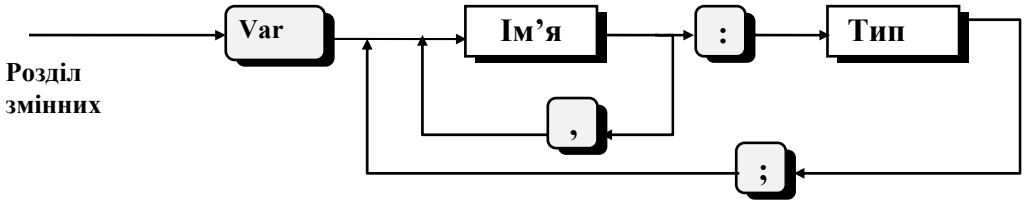
Приклад розділу констант:

```
const Pi = 3.1415926; alfa = 7.1219;
MinInt = -MaxInt;
Line = '_____';
FirstLine = '_____ Список групи _____';
```

Використання імен констант робить програму більш зрозумілою. Крім цього, програміст може згрупувати константи - параметри програми у розділі констант: тут вони легше піддаються контролю і зміні.

3. Змінні програми. Розділ змінних.

Розділ змінних визначений діаграмою:



Змінні використовуються в програмі для позначення даних. На відміну від констант, значення змінних можуть змінюватись у процесі виконання програми.

Будь-яка змінна, яка зустрічається в якому-небудь операторі з розділу операторів повинна бути описана в розділі змінних. Опис змінної зв'язує з новою змінною її ім'я і тип. Інформація, яка знаходиться в розділі змінних, використовується компілятором для:

1.Розподілу пам'яті. Розподіл (резервування) пам'яті для змінних, що описані в розділі змінних, робить компілятор на етапі генерації коду. Для кожної змінної в ОЗП відводиться певне місце. Розмір цієї частини пам'яті визначається типом змінної.

2.Правильної інтерпретації дій над даними. Наприклад, складання цілих чисел інтерпретується не так, як складання дійсних чисел або рядків.

3.Контролю правильності використання змінних. Помилка, яка допущена при написанні змінної в розділі операторів, приведе до повідомлення про синтаксичну помилку, так як ця змінна не описана в розділі змінних.

Приклад розділу змінних:

```
var     Root1, Root2, Discriminant : Real;
        Index, Counter : Integer;
        A,B,C : Real;
        Letter : Char;
        IsSolution : Boolean;
```

4. Стандартні прості типи даних

У мові Паскаль визначені 4 стандартних простих даних:
Integer (цілий);
Real (дійсний);
Char (символьний).
Boolean (логічний);

Для повного опису кожного типу даних, які використовуються в мові програмування, необхідно знати:

- множину допустимих значень для даних цього типу;
- допустимі операції над даними цього типу;
- функції, що визначені на даних цього типу або приймають значення в цьому типі;
- допустимі відношення на даних цього типу.

Тип даних Integer .

Значеннями цілого типу Integer є елементи підмножини (відрізка) цілих чисел, яка залежить від реалізації. Це означає, що існує стандартна константа з ім'ям MaxInt, така що для будь-якого даного X типу Integer

MaxInt < X < MaxInt

Найбільш поширене для 16 розрядних ПЕОМ значення MaxInt = $2^{15} - 1 = 32767$.

Операції:

* - множення; div - цілочисельне ділення; mod - остача від цілочисельного ділення;

+ -додавання; - -віднімання;

Функції:

Abs(x) - | x |;
Sqr(x) - x^2;
Trunc(x) - відкидання дробової частини від дійсного x;
Round(x) - округлення дійсного x;
Succ(x) - x + 1;
Pred(x) - x - 1;

З деякими іншими функціями ми познайомимось пізніше - при визначення інших типів даних.

Відношення:

< - менше <= - менше або дорівнює
> - більше >= - більше або дорівнює
= - дорівнює <> - не дорівнює

Тип даних Real.

Значеннями дійсного типу є елементи підмножини дійсних чисел, яка залежить від реалізації. В TP-6 діапазон типу Real [$2.9*10^{-39}$ ... $1.7*10^{38}$]

Операції:

* -множення; / - ділення;
\+ - додавання; - - віднімання;

Функції:

Abs(x) - модуль x;
Sqr(x) - x у квадраті;
Sqrt(x) - корінь з x.
Sin(x) - sin x;
Cos(x)- cos x;
Arctan(x)- arctg x;
Ln(x) - ln x;
Exp(x) - e^x;
Відношення: такі самі, як і для типу Integer.

Числові типи Integer і Real сумісні. Це означає, що дані типу Integer можуть оброблятися, як дійсні числа і результат буде мати тип Real.

Тип даних Char.

Значеннями символьного типу є елементи скінченої і упорядкованої множини символів. Символи цієї множини визначаються реалізацією. Вони повинні підтримуватись на пристроями введення-виведення. Більшість ПЕОМ підтримує ASCII - код, тому в реалізації мови множина значень типу Char співпадає з деяким розширенням ASCII - символів. ASCII код кожному символу, що визначений в ньому, ставить у відповідність один байт.

Незалежно від реалізації множина символів включає:
A, B, C, ... , Z, _ (знак підкреслення)
1, ... , 9 - (десяткові цифри)
Символ “пробіл”.

Функції:

Ord(x) - порядковий номер x.
Chr(n)- символ з порядковим номером N.
Pred(x)- символ, який передує x.
Succ(x) - символ, який слідує за x.

Відношення.

Як вже повідомлялось, тип даних Char упорядкований. Це означає, що дані типу Char можна порівнювати, як і дані числових типів, за допомогою відношень:

= , <> , > , < , >= , <= .

Порядок на множені букв латинського алфавіту погоджений з алфавітним, а на множині цифр - з числовим.

Логічний тип даних Boolean буде описаний нижче - коли ми будемо вивчати поняття умови.

5. Поняття виразу. Значення виразу. Тип виразу.

Вирази складаються з змінних, констант, функцій, знаків операцій у відповідності з загально-прийнятими математичними правилами. Точне поняття виразу у мові може бути визначено за допомогою синтаксичних діаграм.

Вираз задає порядок обчислення його значення, оснований на загальноприйнятих правилах. Ці правила визначають семантику виразу за допомогою поняття старшинства (пріоритету) операцій. Найбільший пріоритет мають функції і логічна операція not, далі слідують мультиплікативні операції, адитивні операції і відношення. Операції, які мають більший пріоритет, виконуються раніше, ніж операції з меншим пріоритетом.

Таблиця пріоритетів.

Функції, not.

Мультиплікативні операції: * , / ,div , mod , and

Аддитивні операції: : + , - , or

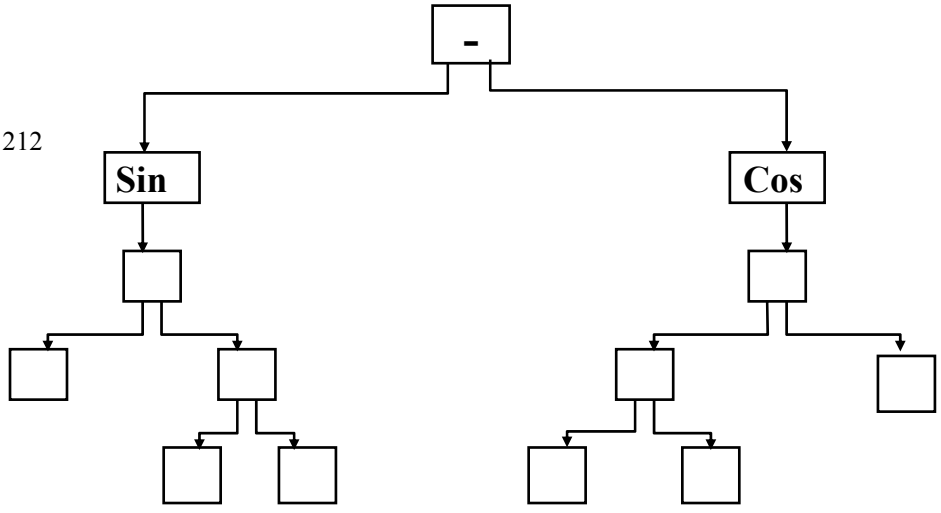
Відношення: = , <> , > , < , >= , <= , in

Операції одного пріоритету обчислюються зліва направо. Це відповідає групуванню дужок у бездужковому виразі уліво.

a + b + c = (a + b) + c, a * b * c = (a * b) * c

Вирази, що стоять у дужках, обчислюються незалежно один від одного.

Важливо розуміти, що в ході обчислення значення виразу кожний проміжний результат - дане деякого типу, точно визначеного знаком операції або функції і типами операндів. Будь-яка невідповідність типу значення операнда приведе до помилки, яка виявляється компілятором при синтаксичному аналізі. Наочне уявлення про структуру виразу дає так зване дерево виразу. Наприклад, вираз sin(x+pi/2) - cos(2*y-pi) може бути представлений у виді дерева:

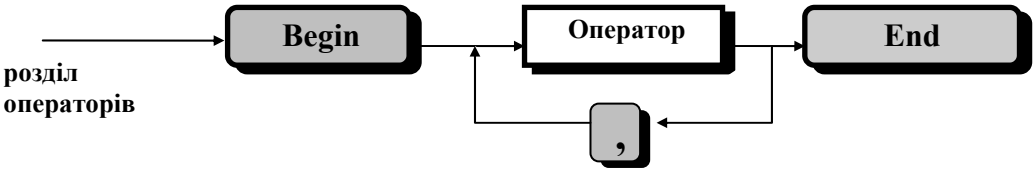


Обчислення значення виразу здійснюється у відповідності з рухом по гілках від листів до кореня - знизу уверх.

6. Розділ операторів. Оператор присвоювання.

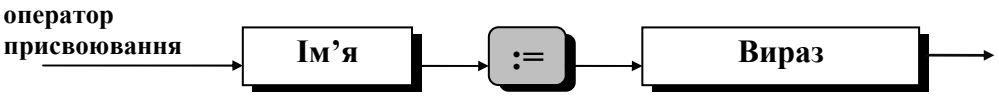
Дії, що роблять над даними, описуються у розділі операторів.

Синтаксична діаграма розділу операторів має вид:



Самим основним, фундаментальним оператором мови є оператор присвоювання. За допомогою оператора присвоювання здійснюється перетворення інформації.

Він має вид: **< ім'я > := < вираз >**



Ім'я ліворуч від символу присвоювання := є ім'я змінної, якій присвоюється значення виразу, який стоїть праворуч. Тому поряд з значенням виразу важливим атрибутом є його тип. Тип виразу у правій частині оператора присвоювання повинен співпадати або бути сумісним з типом змінної з лівої частини. Компілятор на етапі синтаксичного аналізу програми здійснює цю перевірку - так званий контроль типів. Допустиме присвоювання змінних будь-яких типів, за винятком типу File.

Root1 := Pi*(x - y)
Solution := Discriminant >=0
Discriminant := Sqrt(b*b-4*a*c)/2/A
Index := Index + 1
Letter := Succ(Succ(Letter))

7. Оператори введення - виведення.

Для організації введення - виведення даних у мові Pascal використовуються оператори - процедури Write, Read, Writeln, Readln. За допомогою цих операторів організується введення - виведення даних в/з файли Input ,Output.

Текстові файли Input ,Output представляються користувачу як текст, що поділений на рядки і забезпечений ознакою кінця тексту і ознаками кінців рядків. Кожний рядок може містити числа або символьні дані (тобто рядок складається з декількох даних типів Integer, Real, Char). Читання / запис здійснюється через т.н. буфер файла. В момент звернення до файла його буфер встановлений на деяке дане - елемент файла. Буфер файла може бути переміщений або до наступного даного, або до першого даного наступного рядка.

Оператор Read(x) читає дане з Input у змінну x і переміщує буфер до наступного даного.

Оператор Write(x) переміщує буфер у наступну позицію і пише дане в Output з змінної x.

Оператор Readln(x) читає дане з нового рядка з файла Input у змінну x.

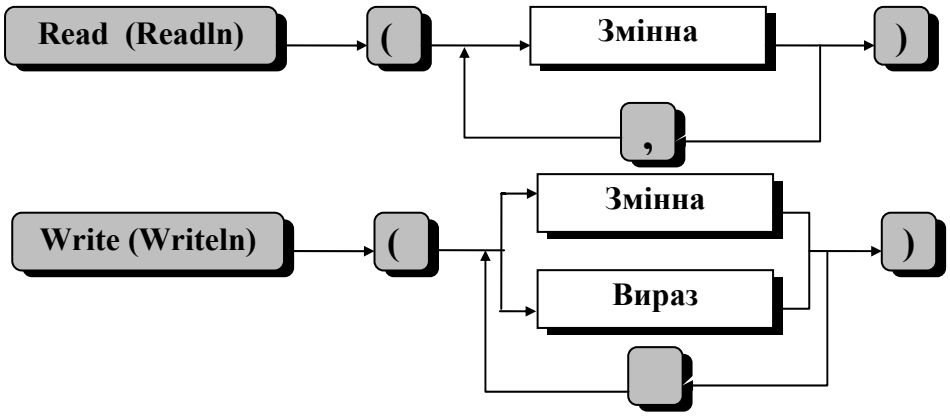
Оператор Writeln(x) пише дане з нового рядка в Output з змінної x.

Оператори введення/виведення можуть використовуватись у більш загальній формі:

Read(<список змінних>), Readln(<список змінних>)

Write(<список виразів або рядків>) Writeln(<список виразів або рядків>)

Ця форма визначається синтаксичними діаграмами:



Тепер ми володіємо знаннями, яких достатньо для написання лінійних (які не розгалужуються) програм.

8. Приклад лінійної програми.

Приклад 1. Програма обчислення площі кола, вписаного у трикутник і площі кола, описаного навколо трикутника.

```
Program Triangle; {програма обчислює площі, вписаного в трикутник і описаного навколо трикутника кіл}
Const  Pi = 3.1415926;
        Line = '_____';
Var a, b, c : Real;
     Sint, Sout : Real;
     S, p : Real;
Begin
 { введення даних}
  Write(' введіть сторони трикутника a b c ');
  Readln(a, b, c);
  {обчислення }
  p := (a + b + c)/2;
  S := Sqrt(p*(p - a)*(p - b)*(p - c));
  Sout := Pi*sqrt(a*b*c/4/S);
  Sint := Pi*sqrt(2*S/p);
  {друк відповіді}
  Writeln; Writeln (Line);
  Writeln('Sвпис = ', Sout); Writeln(Line);
  Writeln('Sопис = ', Sint); Writeln(Line)
End .
```

Оператори Readln і Writeln можуть застосовуватись без параметрів (тобто без тексту у дужках разом з дужками), для переходу до введення/ виведення з наступного рядка або для пропускання рядка.

Для оформлення введення в операторах Write, Writeln можна використовувати рядок. Наприклад: writeln('***', ' площа уписаного кола = ', sint, '***'). Текст програми може містити коментарії. Коментарій - це деякий текст, який стоїть у фігурних дужках. Коментарії

використовують для пояснень, необхідних для кращого розуміння програми. Добре прокоментована програма - признак кваліфікації і сумлінності програміста.

Особливу увагу треба звертати на проектування введення/ виведення. У професіональних програмах це - одна з найбільш важливих проблем. Тут треба виділити наступні аспекти:

- Захист програми від помилок користувача при введенні даних;
- Орієнтація на користувача, який неознайомлений з програмою - введення/ виведення в режимі дружелюбного діалогу.

Обміркування цих аспектів, однак, виходить за рамки цієї книги.

9. Поняття складності виразу. Оптимізація обчислень.

Основний критерій якості програми - її правильність. Строге математичне поняття правильності програми виходить за рамки нашого розгляду. Припустимо, що наші програми виконують саме ті дії, які ми від них чекаємо.

У цьому припущенні критеріями якості програми є, наприклад, її розмір, об'єм пам'яті, що відведений під дані, швидкість виконання, і т.п.

Швидкість виконання програми, що не розгалужується, визначається кількістю обчислень, які необхідні для отримання значень виразів, що містяться в операторах присвоювання і операторах виведення.

Кількість обчислень, необхідних для отримання значення виразу, будемо називати його складністю. Розглянемо приклад:

U := X*Cos(Alfa) + Y*Sin(Alfa) (1)

V := -X*Sin(Alfa) + Y*Cos(Alfa) (2)

Для обчислення значення U необхідно виконати два обчислення значення тригонометричної функції, два множення й одне додавання. Це і складність правої частини оператора (1). Права частина оператора (2) має ту саму складність, що і оператора (1). Зрозуміло, швидкість обчислень значень елементарних функцій, операцій множення і додавання різні і залежать від реалізації мови програмування. Однак реально припускати, що швидкості обчислень всіх елементарних трансцендентних функцій тотожні між собою, швидкості виконання мультиплікативних операцій тотожні, і швидкості виконання адитивних операцій також тотожні.

Хай T_f, T_m і T_a - час обчислення відповідно функцій, множення і додавання, а T(U) - час обчислення значення U. Тоді

$T(U) = 2T_f + 2T_m + T_a$, T(V) = T(U) (3)

Час обчислення значення виразу і є міра його складності. Зрозуміло, що програміст повинен турбуватись про зменшення складності виразів, що входить в програму. Для цього використовують:

а)Тотожні перетворення, що зменшують складність;

б) Попередні обчислення загальних підвиразів;

Приклади:

U := 4*Sin(X)*Cos(X) + 3 <==> U := 2*Sin(2*X) + 3
U := Sin(X)^2 - 3*Sin(X) + 2 <==> U := (Sin(X) - 2)*(Sin(X) - 1) <==>
Y := Sin(X); U := (Y - 2)*(Y - 1)

Між величинами T_f, T_m і T_a існують співвідношення

$$T_f >> T_m > T_a \quad (4)$$

які при програмуванні обчислень не можна ігнорувати. Тому особливу увагу треба приділяти функціональній і мультиплікативній складності за рахунок адитивної.

Наступний приклад по суті визначає оптимальну форму запису полінома для задачі обчислення його значення.

Приклад 2 (Схема Горнера)

Обчислити значення $Y = a_0x^3 + a_1x^2 + a_2x + a_3$;

Оскільки операції піднесення до степеня у мові немає, можна замінити його множеннями:

$Y = a_0*x*x*x + a_1*x*x + a_2*x + a_3$;

У цьому варіанті $T(Y) = 6T_m + 3T_a$;

Якщо виносити x за дужки там, де це можливо, отримаємо:

$Y = ((a_0x + a_1)x + a_2)x + a_3$;

$T(Y) = 3T_m + 3T_a$ {Схема Горнера}.

Можна показати, що схема Горнера обчислення багаточлена є оптимальною.

10. Оптимізація лінійних програм.

Задача зменшення складності програми, що містить декілька виразів, носить більш складний характер. Тут оптимізації підлягають одночасно декілька виразів, які обчислюються послідовно.

Приклад 3. Програма обчислення координат вектора, повернутого на кут Alfa.

```
Program Vector;
Var X, Y : Real;
Alfa : Real;
U, V : Real;
Begin
{ Введення X, Y, Alfa }
U := X*Cos(Alfa) + Y*Sin(Alfa);
V := -X*Sin(Alfa) + Y*Cos(Alfa);
{ Виведення U, V }
End.
```

У цьому варіанті складність програми T(U, V) є $T(U, V) = 4T_f + 4T_m + 2T_a$

Здійснимо попереднє обчислення функції Sin, Cos:

```
{ Описати допоміжні змінні Fsin, Fcos }
Begin
{ Введення X, Y, Alfa }
Fsin := Sin(Alfa); Fcos := Cos(Alfa);
U := X*Fcos + Y*Fsin;
V := -X*FSin + Y*Fcos;
{ ВиведенняU, V }
End.
```

Отримаємо: $T(U, V) = 2T_f + 4T_m + 2T_a$

В результаті перетворення складність зменшилась приблизно вдвічі. Можна ще зменшити мультиплікативну складність, якщо обчислити U і V наступним способом:

```
A := (Fcos + Fsin)*(X + Y);
B := X*Fsin; C := Y*Fcos;
U := A - B - C;
V := C - B;
```

Тоді $T(U, V) = 2T_f + 3T_m + 5T_a$

Рішення питання про те, який з двох варіантів перетворень програми краще по швидкодії, залежить від реалізації множення у комп'ютері.

Приклад показує, що прийоми оптимізації оператора присвоювання ще в більшій степені застосовувані для оптимізації лінійних програм, що складається з декількох операторів. Менш очевидним є прийом оптимізації, що заключається в використанні співвідношень між виразами - правими частинами операторів. Розглянемо його на наступному прикладі:

Приклад 4. Відомо, що рівняння $x^2 - px + q$ має два рішення. Знайти їх.

```
Варіант 1:
Discriminant := Sqrt(p*p - 4*q);
Root1 := ( p + Discriminant )/2;
Root2 := ( p - Discriminant )/2;
Варіант 2:
Discriminant := Sqrt(p*p - 4*q);
Root1 := (p + Discriminant)/2;
Root2 := p - Root1;
```

Обчислення у другому варіанті містять на одну ділення менше. Тут оптимізація досягнута завдяки наявністю співвідношення між Root1 і Root2: Root1+ Root2 = p.

Відмітимо на закінчення, що розглянуті оптимізаційні прийоми мають смисл застосовувати тоді, коли обчислення повторюються в програмі достатньо часто і час виконання програми - критичний параметр.

11. Задачі і вправи.

Скласти програму розв'язку задачі:

1. Знайти гіпотенузу, площа і гострі кути прямокутного трикутника, заданого катетами.

2. Змішано V_1 літрів води температурою t_1 з V_2 літрами води температури t_2. Знайти об'єм V і температуру t утвореної суміши.

3. Знайти радіус кола з центром в (X_0,Y_0), дотичною до якого є пряма y = kx + b.

4.Обчислити центр ваги системи з трьох матеріальних точок на площині з масами M_1, M_2, M_3 і координатами (X_1,Y_1), (X_2,Y_2), (X_3,Y_3).

5.Розв'язати систему лінійних рівнянь методом Крамера.

$$\begin{cases} a_{11}x_1 + a_{12}x_2 = b_1 \\ a_{21}x_1 + a_{22}x_2 = b_2 \end{cases}$$

{Вважати, що її визначник не дорівнює нулю }

6.Обчислити координати точки A(X,Y) при повороті системи координат на кут Alfa і паралельному переносі на вектор a = (u, v).

7.Знайти корінь степені n і n-ту степінь позитивного дійсного числа a.

8.Обчислити цілі коефіцієнти A, B, C квадратного рівняння по його раціональним кореням $x_1 = n_1 / m_1$, $x_2 = n_2 / m_2$.

9. Обчислити внутрішні кути трикутника, заданого довжинами сторін.

10.Перерахувати координати точки з полярної системи в декартову систему координат.

11.Перерахувати координати точки з декартової системи в полярну систему координат.

12. Розрахувати координати матеріальної точки, пущеної з початковою швидкістю V_0 під кутом Alfa до горизонту в напрямі вектора a = (X_0,Y_0) в момент часу t.

13. Обчислити суму, добуток і частку двох комплексних чисел $z_1 = a+bi$, $z_2 = c+di$.

14. Багаточлени F(x) = ax + b і G(x) = cx + d задані своїми коефіцієнтами. Знайти коефіцієнти багаточлена H(x) = F(x)*G(x).

15.Багаточлени F(x) = ax + b і G(x) = cx + d задані своїми коефіцієнтами. Знайти коефіцієнти багаточленів $H_1(x) = F(G(x))$ і $H_2(x) = G(F(x))$.

Задачі

5.Знайти розв'язок вправи 13, яке використовує 3 множення.

6.Знайти розв'язок вправи 14, яке використовує 3 множення.

7.Знайти розв'язок приклада 3, який використовує лише одну операцію обчислення тригонометричної функції.

8.Використовуючи розв'язок задачі 2, знайти схему множення двох квадратних трьохчленів, який використовує 6 множень.

ПРОГРАМИ, ЩО РОЗГАЛУЖУЮТЬСЯ.

1. Поняття умови. Тип даних Boolean (логічний).

Умови використовуються в програмах для організації розгалужень і дій, що повторюються. Умовою в мові є логічний вираз - вираз типу Boolean. Булевські значення - це логічні істинні значення: : True (істина) і False (хибність).

Цей тип даних, як і інші прості типи даних, упорядкований. На ньому визначені функції Ord, Succ, Pred.

Таким чином, мають місце наступні співвідношення:

False < True ,

Ord (False)=0, Ord (True)=1,

Succ (False)=True, Pred (True)=False.

На множені < True, False > визначені логічні операції.

Операції:

And - логічна кон'юнкція (і)

Or - логічна диз'юнкція (або)

Not - логічне заперечення (ні)

Ці операції визначаються наступними таблицями істинності:

And	False	True
False	False	False
True	False	True

Or	False	True
False	False	True
True	True	True

x	Not x
False	True
True	False

Відношення, що були визначені раніше для простих стандартних типів є операціями, результат яких має логічний тип. Іншими словами, булевське значення дає будь-яка з операцій відношень : =, < > , <= , < , > , >= , in.

Для типу Boolean визначені стандартні функції, які приймають значення цього типу (логічні значення):

Odd(X)

{ Odd(X) = True, якщо X - ціле непарне число
Odd(X) = False, якщо X - ціле парне число}

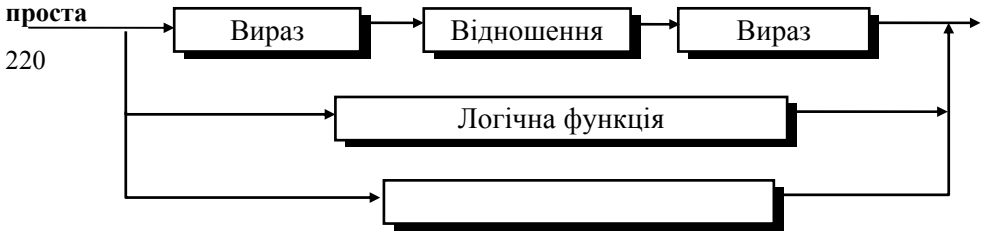
Eoln(F) { кінець рядка в текстовому файлі}

Eof(F) { кінець файла}

Функції Eoln(F) і Eof(F)

Умови можна класифікувати як прості і складні.

Прості умови визначені діаграмою:



умова

Складні умови конструюються з простих за допомогою логічних операцій.

Приведемо приклади простих і складних виразів типу Boolean (умов).

Прості вирази типу Boolean (умови):

Sin(2*x) > S, (X + Y) mod Prime = 0,

b*b > 4*a*c ,

Number div Modulo = 2, Odd(A*P + B),

Flag ;

Складні вирази типу Boolean (умови) :

а) (a + i > в) or (x [Index] = c)

{Тут a, в, c - змінні типу Real, x - масив дійсних чисел , Index - змінна типу Integer }

б) Odd (n) And (n < 10e4)

в) Eof(f) Or (f^.data = 0) {f - файл дійсних чисел}

г) Not(beta) And (gamma) {beta і gamma - змінні типу Boolean}

д) (A > B) = (C > D)

Логічні вирази перетворюються за законами логіки висловлювань. Наприклад,

Not((A > 0) And (B <> 0)) <==> Not(A > 0) Or Not(B <> 0) <==> (A <= 0) Or (B = 0)

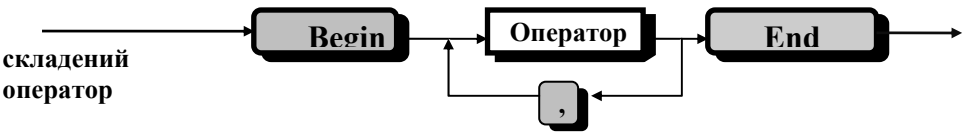
Перетворення логічних виразів часто приводять до зменшення їх складності і, тим самим, оптимізації програми за часом.

2. Складений оператор.

Декілька операторів, що виконуються послідовно, можна об'єднувати в один складений оператор.

Складений оператор передбачає виконання операторів, які в нього входять - компонент у порядку їх написання. Службові слова Begin і End грають роль дужок операторів - вони виділяють тіло складного оператора.

Складений оператор визначається діаграмою:



Зверніть увагу на те, що розділ програми представлений як один складений оператор.

Приклади складених операторів:

а)

```
Begin
 Write(‘ Введіть координати вектора: ‘);
 Readln(a, b, c);
 Length := sqrt(a*a + b*b+ c*c);
 Write(‘ довжина (a,b,c) дорівнює ‘, Length)
end
```

б)

```
Begin
 u := u*x/n;
 s := s+u
End
```

в) Begin writeln (‘ рівняння коренів не має ’) End

3. Оператори вибору: умовний оператор.

Оператори вибору призначені для виділення операторів, які їх представляють - компонент одного - єдиного, який і виконується. Таким чином, оператори вибору реалізують управляючу структуру “розгалуження”. У якості оператори вибору в мові визначені умовний оператор і оператор варіанта.

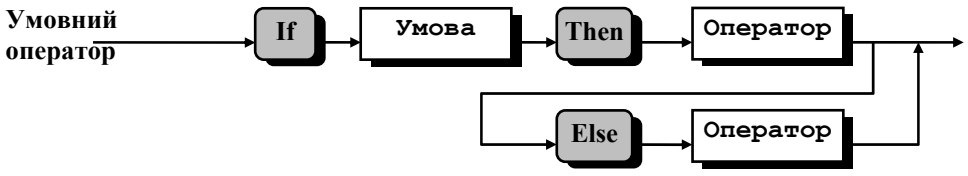
Існують дві форми умовного оператора:

If < умова > then < оператор >

If < умова > then < оператор > else < оператор >

Вони відповідають базовим управляючим структурам короткого і повного розгалуження. Умова - це вираз типу Boolean.

Синтаксична діаграма умовного оператора має вид:



Приклади умовних операторів:

а) If a >= b then Max := a else Max := b

б) If IntFun(i) mod 3 = 0 then write(i)

в) If (a11*a22 = a12*a21) And
((a11*b2 <> a12*b1) Or
(b1*a22 <> b2*a21))
then Write(‘ система розв’язків не має ’)
else Write(‘ система має розв’язки ’)

г) If x <= 0
then begin
u :=x*x - 2*x + 3;
v :=1/2*x + 1
end
else begin
u :=1/3*x+2;
v :=x*x+3*x-2
end

Зверніть увагу на те, що в тілі умовного оператора може використовуватись і інший умовний оператор. Це створює можливість реалізовувати багатозначне розгалуження. Наприклад:

```
If Discriminant < 0
  then If LeadCoef < 0
    then Write(‘ Розв’язків немає ’)
    else Write(‘ Розв’язки - вся числова вісь ’)
  else If LeadCoef < 0
    then Write(‘ Розв’язки - між коренями рівняння ’)
    else Write(‘ Розв’язки - поза коренями рівняння ’)
```

Відмітьте однак, що наступна конструкція мови, що складається з вкладених розгалужень синтаксично двозначна (допускає два різних варіанта синтаксичного аналізу).

If <умова 1> then If <умова 2> then <оператор 1> else <оператор 2>

1 варіант:

```
If < умова 1>
 then begin
        I f < умова 2> then <оператор 1> else <оператор 2>
       end
```

2 варіант:

```
If< умова 1>
 then begin
  If < умова 2> then <оператор 1>
 end
else <оператор 2>
```

Для усунення двозначності в мові обрано 1-ий варіант інтерпретації у відповідності з правилом: роздільнику else відповідає найближчий попередній роздільник then.

4. Програми, що розгалужуються. Приклад.

Програмами, що розгалужуються, називаються програми, в яких використовуються оператори розгалуження.

Приклад 1.

```
Program SquareEquation;
 Const Line = '----------------------------';
 Var a, b, c, Root1, Root2: real;
 Solution: Integer;
 Discriminant: Real;
 Begin
  {Введення даних}
  Write(' Введіть коефіцієнти рівняння (через пробіл):');
  Readln(a, b, c) ; Writeln(Line);
            {Обчислення}
 Discriminant := Sqr(b) - 4*a*c;
 If Discriminant < 0
  then Solution := 0 { Немає коренів }
  else If Discriminant = 0 { Один корінь }
    then begin
      Solution := 1;
      Root1 := -b/a;
      Writeln ('x1= ' ,Root1)
    end
   else { Два кореня } begin
     Solution := 2;
     Root1 := (-b + Sqrt(Discriminant))/(2*a);
     Root2 := -b/a - Root1;
     Writeln('x1= ', Root1, ' x2= ', Root2)
   end ;
     Writeln(Line);
     Writeln(' Кількість розв'язків дорівнює: ', Solution)
End.
```

5. Оптимізація програм, що розгалужуються за часом.

Складність програми, що розгалужується, визначається складністю умови і складністю обчислень гілок. На відміну від програми, що не розгалужується, час виконання

тут залежить від гілки, якою слідує процес виконання. Тому для таких програм має смисл поняття складності програми в гіршому випадку і складності програми в середньому.

Хай $T_п$ - складність програми за часом в гіршому випадку, $T_у$ - складність за часом умови і $T_{в1}$, $T_{в2}$ - складності гілок за часом програми. Тоді має місце співвідношення:

$T_п = T_у + Max(T_{в1}, T_{в2})$ (1)

Складність за часом у середньому визначається формулою

$T_п = T_у + P_у T_{в1} + (1 - P_у) T_{в2}$ (2)

де $P_у$ - ймовірність виконання умови.

Оскільки умовою є логічний вираз, загальні прийоми оптимізації виразів застосовуються і для логічних виразів. Час Тл виконання логічних операцій And, Or, Not, =, <> значно менше часу виконання аддитивних операцій, а час виконання операцій <, >, <=, >= дорівнює часу виконання аддитивних операцій.

$T_f >> T_m > T_a > T_л$ (3)

Розглянемо приклад: потрібно з'ясувати, являється чи дорівнює нулю одне з двох чисел A, B .

1 варіант умови: A * B = 0

2 варіант умови: (A = 0) Or (B = 0)

У першому варіанті використані множення і порівняння, у другому - 3 логічних операції. Складність 2-го варіанта менша.

У програмах з багатозначним розгалуженням, коли в управлінні використовується декілька умов або обчислень логічних виразів, існує можливість оптимізації управління обчисленнями. Наприклад, наступні обчислення еквівалентні:

```
If x < 0 then                    <==>   If (x < 0) And (Y < 0)
If y < 0 then z := 1                      then z := 1
If x < 0
then Flag := False               <==>   Flag := (x >= 0)
else Flag := True
```

У наступному прикладі умови розгалужень спрощені за рахунок використання співвідношень, що виконуються після обчислення попередньої умови:

Приклад 2. Програма обчислення значення кусково-визначеної функції:

$$y = \begin{cases} 2x - 1 & при\ x < -1 \\ x^2 + 1 & при\ -1 <= x < 1 \\ 2x + 1 & при\ x >= 1 \end{cases}$$

1 варіант (який часто зустрічається у починаючих)

```
If x < -1 then y := 2*x - 1;
If (-1 <= x)And(x < 1) then y := Sqr(x) + 1;
If x >= 1 then y := 2*x + 1;
```

2 варіант (оптимальний)

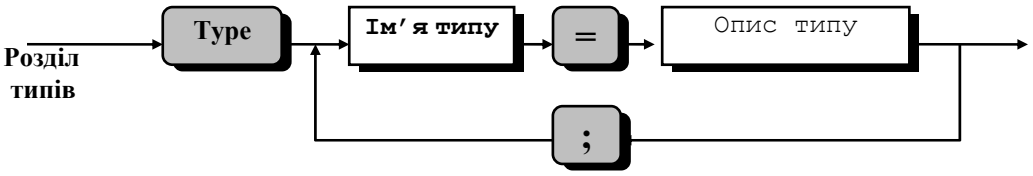
```
If x < -1 then y := 2*x - 1
  else If x < 1 then y := Sqr(x) + 1
  else y := 2*x + 1;
```

Для отримання 2-го варіанта відмітимо, що умови < -1, (-1 <= x)And(x < 1), x >= 1 взаємно протилежні і у сукупності тотожно істинні. Тому (-1 <= x) і x >= 1 можна замінити на else.

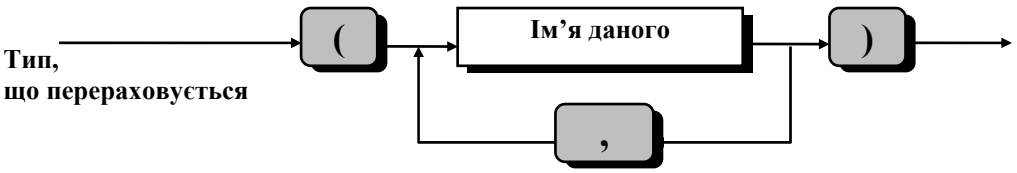
6. Тип, що перераховується. Розділ типів.

Поряд з стандартними типами даних у мові Pascal широко використовуються типи, що визначаються програмістом. Один з таких типів - це тип, що перераховується. Визначення цього типу задає упорядковану множину значень шляхом перерахування імен, що позначають ці значення.

Типи даних, що визначає програміст, описуються у спеціальному розділі - розділі типів. Розділ типів визначений синтаксичною діаграмою:



Тип даних, що перераховується, визначається наступною діаграмою:



Приклади визначень типів, що перераховуються:

а) Type Weekday = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);

Colour = (Red, Orange, Yellow, Green, Blue, Black);

Operation = (Plus, Minus, Times, Divide)

Відмітимо, що стандартний тип Boolean, якщо б його треба було описати, виглядав би як : type Boolean = (False, True);

Для аргументів типу, що перераховується, визначені такі стандартні функції:

Succ(x) - значення, наступне за x.

Pred(x) - значення, попереднє x.

Ord(x) - порядковий номер x.

До значення типу, що перераховується, можуть бути застосовані відношення:
=, <>, <, <=, >=, > .

Упорядкованість значень визначається порядком перераховування констант в описанні типу. Наприклад:

Red < Orange < Yellow < Green < Blue < Black ;

Описання типу змінної може бути дано і в розділі змінних. Наприклад, описання:
Type Figure = (Triangle, Circle, Rhombus, Square);

Var f: Figure; еквівалентне описанню Var f: (Triangle, Circle, Rhombus, Square); однак у другому випадку описання типу становиться анонімним: тип описаний, але не має імені. Використання цього типу обмежене. Тому 1-ий варіант більш відповідає стилю мови.

7.Оператори вибору: оператор варіанта

Оператор варіанта складається з виразу, який називається селектором, і списку операторів, кожний з яких відмічений константою того ж типу, що й селектор. Селектор повинен бути скалярного типу, але не дійсного.

Оператор варіанта обчислює значення селектора і вибирає для виконання оператор, одна з міток якого дорівнює цьому значенню. По закінченню виконання вибраного оператора управління передається на виконання наступного за оператором варіанта оператора.

Якщо значення селектора не співпадає ні з однією з міток, то вибирається оператор, помічений ключовим словом else. Цей оператор повинен бути останнім у списку варіантів. Якщо значення селектора не співпадає ні з однією з міток і else відсутнє, то оператор варіанта ігнорується.

Оператор варіанта має вид:

Case < вираз {селектор}> of <список міток варіанта > : < оператор >;

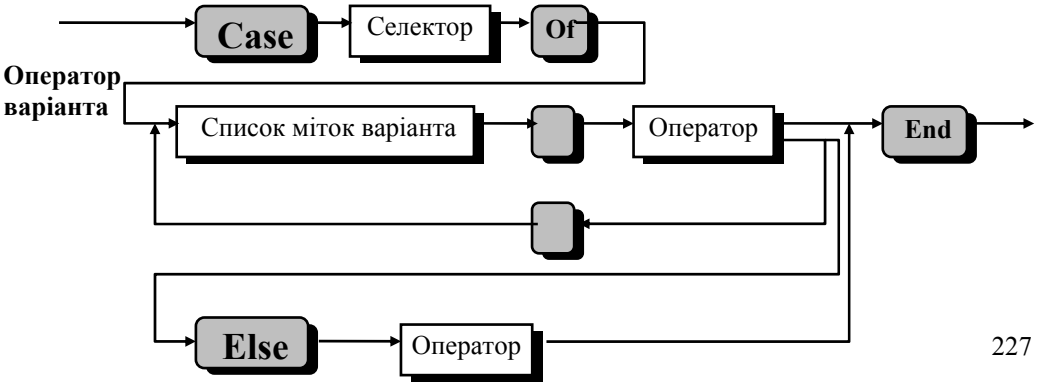
. . . . . .

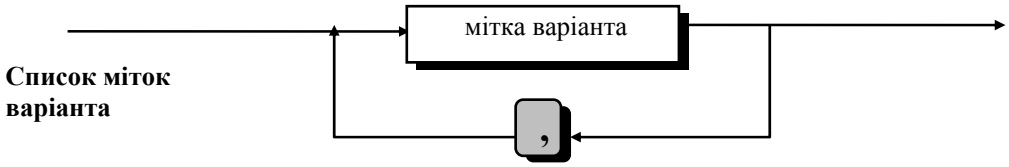
< список міток варіанта > : < оператор >

[else < оператор >]

end

Мовою синтаксичних діаграм це виглядає так:





Приклади операторів варіанта:

а) Select : = Index mod 4;
```
                    case Select of
                      0 : x := y*y + 1;
                      1 : x := y*y - 2*y;
                      2,3 : x := 0
                    end;
```
В цьому прикладі Select приймає значення 0, 1, 2, 3. Це досягнуто обчисленням Select := Index mod 4.

Таким чином, замість імені Select можна використовувати вираз Index mod 4:

а)
```
case Index mod 4 of
  0 : x := y*y + 1;
  1 : x := y*y - 2*y;
  2,3 : x := 0
end;
```
б)
```
case ch of
  ‘a’,’b’,’c’ : ch := succ(ch);
  ‘y’,’z’ : ch := pred(ch);
  ‘f’,’g’ : {порожній варіант};
else ch := pred(pred(ch)
end;
```
Програма в наступному прикладі обчислює знак однієї з тригонометричних функцій у залежності від квадранта декартової площини.

Пример 3.
```
program Sign_of_Function;
Type Fun = (Unknown, FSin, FCos, Ftg, Fctg);
 var FunNumber, Quoter: Integer;
```

```
TrigFun : Fun;
Begin
  Write(‘ Введіть номер тригонометричної функції ‘);
  Readln(FunNumber);
{ Обчислення імені функції }
Case FunNumber of
  1: TrigFun := FSin;
  2: TrigFun := FCos;
  3: TrigFun := FTg ;
  4: TrigFun := FCtg
else begin
  TrigFun := Unknown;
  Writeln(‘ Невідома функція ’)
end
end;
Write(‘ Введіть номер квадранта ‘); Readln(Quoter);
{ Обчислення знака функції }
case TrigFun of
FSin: case Quoter of
  1, 2: Writeln (‘ знак синуса +’);
  3, 4: Writeln (‘ знак синуса -‘)
  end;
  FCos: case Quoter of
1, 3: Writeln (‘ знак косинуса +’);
2, 4: Writeln (‘ знак косинуса  -‘)
end;
FTg, FCtg: case Quoter of
          1, 4: Writeln (‘ знак тангенса і котангенса +’);
  2, 3: Writeln (‘ знак тангенса і котангенса -‘) end;
Unknown: Writeln(‘ Функція не визначена ’)
end
end.
```

8. Вправи.

І. Сформулюйте умови для оператора розгалуження:

11.Білий кінь розташований на полі (x, n). Чорний пішак розташований на полі (y, m). Чи знаходиться пішак під боєм коня?

12.Білий слон розташований на полі (x, n).Чорний пішак розташований на полі (y, m). Інших фігур на полі немає. Чи знаходиться пішак під боєм слона?

13.Біла шашка розташована на полі (x, n). Чорна шашка розташована на полі (y, m). Чи знаходиться чорна шашка під боєм білої?

14.Біла дамка розташована на полі (x, n).Чорна шашка розташована на полі (y, m). Чи знаходиться чорна шашка під боєм білої дамки? (Інших фігур на дошці немає)

15.Вектор a = (x, y), вектор b = (u, v). Чи являються вектори паралельними або перпендикулярними?

16.Вектор a = (x, y), вектор b = (u, v). Чи можна повернути вектор a проти годинної стрілки на деякий кут, менший 180^0 так, щоб вектори стали співнаправленими?

17.Чи пересікаються коло O_1 з центром (x, y) і радіусом R_1 з колом O_2 з центром (u, v) і радіусом R_2?

18.Чи є трикутник з вершинами $A(x_1, y_1)$, $B(x_2, y_2)$, $C(x_3, y_3)$ рівнобедреним?

19.Чи можна з відрізків a, b, c скласти трикутник і чи можна цей трикутник помістити у коло радіуса R?

20.Пряма задана рівнянням Y = kX + b. Чи лежить точка A(u, v) над цією прямою?

II. Напишіть програму розв'язування задачі:

1.Розв'яжіть квадратну нерівність $ax^2 + bx + c < 0$.

2.Розв'яжіть систему лінійних рівнянь:

$$\begin{cases} a_{11}x + a_{12}y = b_1 \\ a_{21}x + a_{22}y = b_2 \end{cases}$$

3.Обчислити внутрішні кути трикутника, з вершинами $A(x_1, y_1)$, $B(x_2, y_2)$, $C(x_3, y_3)$

4.Знайти найкоротшу сторону трикутника з вершинами $A(x_1, y_1)$, $B(x_2, y_2)$, $C(x_3, y_3)$

5.Знайти максимум і мінімум з трьох цілих чисел.

6.Обчислити значення функції:

1, якщо x > 0

Sign(x) = 0, якщо x = 0

-1, якщо x < 0 .

7.Обчислити найбільше і найменше значення функції $Y=ax^2 +bx+c$ на відрізку [p ; q].

8.Знайти область визначення функції: $y=1/(x^2 + px + q)$.

9.Розв'язати біквадратне рівняння $ax^4 + bx^2 + c = 0$

10.Знайти квадрат найменшої площини з сторонами, паралельними осям координат, який містить три точки площини $A(x_1,y_1)$, $B(x_2,y_2)$, $C(x_3,y_3)$.

11.Здійснити переклад не більш чим трьохзначного цілого додатного числа у відповідний йому складений числівник українською, російською або англійською мовою.

12.За номером місяця і номером дня знайти день тижня, що припадає на цю дату.

ОПЕРАТОРИ ПОВТОРЕННЯ З ПАРАМЕТРОМ І МАСИВИ.

1. Оператор циклу з параметром.

Цикли - основний засіб у програмуванні, що дозволяє коротко записувати алгоритм, який здійснює велику кількість дій.

Для реалізації циклічних алгоритмів у мові Паскаль використовуються оператори повторення:

- оператор циклу з параметром;
- оператор циклу з передумовою;
- оператор циклу з постумовою.

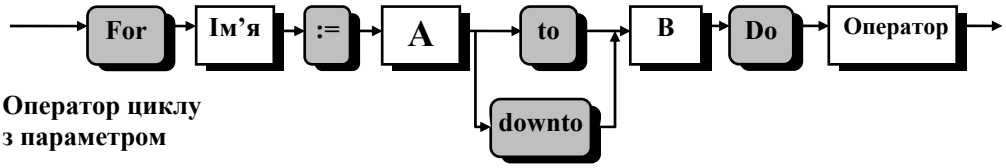
У цьому параграфі вивчається оператор циклу з параметром. Такий оператор передбачає повторне виконання деякого оператора з одночасною зміною значення, що присвоюється керуючою змінною (параметру цього циклу). Він має вид:

For < параметр > := <початкове значення > to <кінцеве значення > do <оператор >

або

For < параметр >:=< початкове значення > downto <кінцеве значення > do <оператор>

Синтаксична діаграма оператора циклу з параметром:



Оператор циклу з параметром

Тут

Ім'я - це ім'я змінної - параметра циклу;

A - початкове значення параметра циклу;

B - кінцеве значення параметра циклу;

Оператор - тіло циклу.

Параметр циклу, початкове і кінцеве значення повинні бути одного й того ж скалярного типу (крім дійсного). Початкове і кінцеве значення обчислюються лише один раз - при вході в цикл, і, отже, повинні бути визначені до входу в цикл і не можуть бути змінені в тілі циклу.

Якщо початкове і кінцеве значення розділяє службове слово to, то після виконання оператора (тіло циклу) параметр циклу v приймає значення Succ(v), якщо ж дільником початкового і кінцевого значень служить слово downto, то параметр циклу v після виконання тіла циклу приймає значення Pred(v). Зокрема, якщо параметр v має тип Integer, то одночасно з виконанням тіла циклу здійснюється або присвоювання v := v + 1 (to), або v := v - 1 (downto). Додатково (примусово) змінювати значення параметра в тілі циклу не рекомендується, оскільки контроль за правильністю виконання такого циклу дуже затруднений. Якщо в циклі з to початкове значення більше, ніж кінцеве, то цикл не виконується взагалі. Аналогічно, якщо в циклі з downto початкове значення менше, ніж кінцеве, цикл також не виконується.

Стандарт мови локалізує дію параметра тільки в самому циклі, тому при нормальному виході з циклу, тобто коли параметр циклу вже прийняв всі можливі значення між початковим і кінцевим значеннями, значення параметра вважається невизначеним (В реалізації мови це правило часто не виконується).

Приклад 1.

```
Program NFactorial; var Factorial, Argument: Integer; i : Integer;
  Begin
    Write(‘ введіть аргумент факторіала ‘);
     Readln(Argument) ;
     Factorial := 1;
     For i := 2 to Argument do Factorial := i*Factorial;
     Writeln(Argument,’! = ‘, Factorial)
  End.
```

У цьому прикладі

i - параметр циклу;

2 - початкове значення параметра циклу;

Argument - кінцеве значення параметра циклу;

Factorial := i*Factorial - тіло циклу.

Відмітимо, що на вході Argument > 7 значення Factorial вийде за межі типу Integer і результат буде неправильним.

В наступному прикладі обчислюються суми $\sum_{n=0}^{\infty} x^n/n^2$, $\sum_{n=0}^{\infty} x^n/n^3$

Приклад 2.

```
Program SeriesSumma;
  Var Summa1, Summa2, x : Real;
  u1, u2 : Real;
  v : Real;
  Index, n : Integer;
Begin
```

```
 Write(‘ введіть x і n ‘); Readln(x, n);   {Ініціалізація змінних, що використовуються в
    циклі}
 Summa1 := 1; Summa2 := 1;                { суми рядів }
  v := 1; { x^n }
  For Index := 1 to n do begin
    v := v*x;
    u1 := v/Sqr(Index);  u2 := u1/Index;
    Summa1 := Summa1 + u1;   Summa2 := Summa2 + u2;
   end;
  Writeln( ‘ S1 = ‘, Summa1,’ ‘,’ S2 = ‘, Summa2)
End.
```

Зверніть увагу на те, що всі оператори, які необхідно виконувати в циклі, об’єднані у складений оператор.

Часто крок зміни змінної, яка управляє циклом, відрізняється від 1, -1. Наступний приклад демонструє використання циклу з параметром у таких обчисленнях.

Приклад 3. Табулювання функції дійсної змінної.

```
Program Tabulation;
   const Pi=3.14159262;
   var MinBound, MaxBound, Step: Real;
   x, y : Real;   Coef : Real;
   i, n : Integer;
 Begin
  Write(‘Введіть межі табулювання ‘); Readln(MinBound, MaxBound);
  Write(‘Введіть крок табулювання ‘); Readln(Step);
  n := Round((MinBound - MaxBound)/Step);
  x := MinBound;   Coef := 1/Sqrt(Pi);
  for i := 0 to n do  begin
    y := Coef * exp(-Sqr(x)/2);
    writeln(‘ x = ‘,x,’ y = ‘,y);
    x := x + Step
  end;
End.
```

Програма табулює функцію $y=1/\pi\ e^{-x2/2}$ в інтервалі значень [MinBound, MaxBound] з кроком Step. Зверніть увагу на те, що перед входом у цикл обчислюється N - верхня межа параметра циклу, а в тілі циклу обчислюється наступне значення x.

Приклад 4.

```
program Alfabet;
  Var Letter : char;
```

```
Begin
 For Letter := 'z' downto 'a' do Write(Letter, ',')
End.
```

2. Циклічні програми. Складність циклічної програми. Оптимізація циклічних програм.

Циклічними називають програми, що містять оператори циклів. Циклічна програма може містити декілька операторів циклу, що виконуються послідовно або входять в інші оператори . Найпростіші циклічні програми містять один оператор циклу і оператори, що керують введенням-виведенням. До найпростіших відносяться циклічні програми розглянутих прикладів.

Складність $T_ц$ найпростішої циклічної програми, що містить арифметичний цикл, визначається формулою $T_ц = NT_b$ де N - кількість повторень циклу, T_b - складність тіла циклу. Якщо арифметичний цикл завершується нормально, то N = |Ord(MaxVal) - Ord(MinVal)|, де MaxVal, MinVal - нижня і верхня межі параметра. Таким чином, оптимізація програми за часом полягає у зменшенні складності тіла циклу і (якщо це можливо) зменшенні кількості повторень циклу.

Для оптимізації тіла циклу використовують наступні прийоми:

Попереднє (поза циклу) обчислення підвиразів, значення яких не змінюється в циклі. Наприклад, у програмі Tabulation до входу в цикл обчислено значення Coef=1/Sqrt(Pi);

Використання співвідношень, що зв'язують змінні, які змінюються в циклі, для обчислень однієї з них через інші. У програмі SeriesSumma змінні u1 і u2 - члени ряду - визначені як функції від x, i:

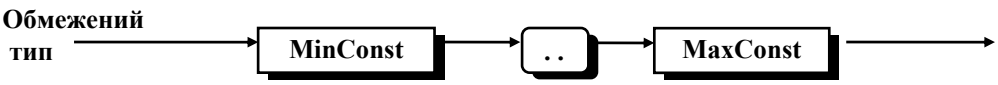
$$u1(x, i) = x^i / i^2; \quad u2(x, i) = x^i / i^3;$$

Тому має місце рівність u2*i = u1, яка використовується для обчислення u2.

3. Обмежені типи.

Обмежений тип у мові Паскаль можна визначити, накладаючи обмеження на вже визначений скалярний тип - його називають базовим скалярним типом. Обмеження визначаються діапазоном - мінімальним і максимальним значеннями констант базового скалярного типу. Обмеження стандартного типу Real не допускається.

Синтаксична діаграма обмеження має вид:



Наприклад:

а) Type Day = 1..30; - обмеження на тип integer;

б) Digit = ‘0’..’9’; - обмеження на тип char;

в) Type Weekday = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);

Workday = Monday .. Friday; - обмеження на скалярний тип Weekday.

Базовий скалярний тип визначає допустимість всіх операцій і відношень над значеннями обмеженого типу. Обмежені типи дають можливість описувати алгоритм у більш наочній формі. Крім цього, у процесі трансляції і виконання програми з’являється можливість економити пам’ять і здійснювати контроль присвоювань.

Приклад 5.

```
Program Time_Table;
  Type Weekday = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);
       Workday = Monday .. Friday;
Var i : Workday;
 Begin
   For i := Monday to Friday do
    Case i of
     Monday: Writeln(‘фізика‘, ’інформатика‘, ’історія’);
    Tuesday, Friday: Writeln(‘мат. аналіз‘,’педагогіка‘, ‘англійська’);
    Wednesday, Thursday: Writeln(‘фізика ‘,’алгебра‘,’інформатика’)
   end
End.
```

4. Складні (складені) типи.

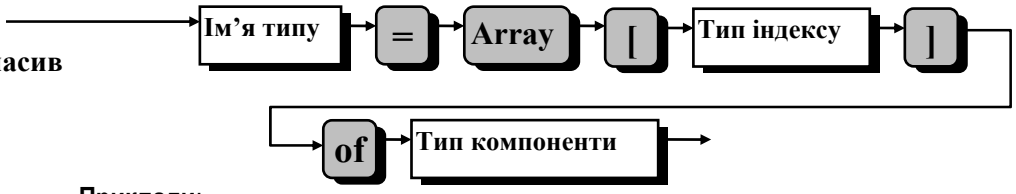
У мові Pascal реалізований механізм визначення складних (складених) типів даних. Новий тип даних визначається як структурована сукупність даних-компонент стандартних або раніше визначених типів. Оскільки типи компонент можуть також складеними, можна будувати складні ієрархії типів. Методи структурування даних у мові дозволяють будувати масиви, записи, множини і файли. Ці методи називають статичними, оскільки їх опис здійснюється попередньо. Більш складні структури можна створити динамічно - у процесі виконання програми - за допомогою засилок. При вивченні складних типів основна увага приділяється способам конструювання даного і способам доступу до компонентів даного.

5. Регулярний тип. Масиви.

Значеннями регулярного типу являються масиви. Масив - це найбільш поширена структура даних. У багатьох мовах програмування, що були одними з перших мов високого рівня, (Fortran, Algol-60, Basic) цей єдиний явно визначений складний тип.

Масив - це послідовність однотипних даних, що об'єднана загальним іменем, елементи (компоненти) якої відрізняються (ідентифіцируються) індексами. Індекс елемента вказує місце (номер) елемента в масиві. Кількість елементів масиву фіксовано і визначено в його описі. Доступ до елемента масиву здійснюється обчисленням значення його індексу. Тому масиви - це структури даних з прямим (випадковим) доступом. Всі компоненти масиву є однаково доступними. При визначенні регулярного типу задається і тип компонент, і тип індексів. Саме визначення має вид:

<ім'я типу > = Array [< тип індексу >] of <тип компоненти >;



Приклади:

а) Type LinearTable = Array [0..100] of Integer;

б) type Word = Array [1..20] of Letter;

Letter = 'a'..'z';

Order = Array [Letter] of Integer;

в) type Matrix = array [1..N] of array [1..M] of Real;

г) Tenzor = array [-10..10] of array [0..20] of array [0..3] of Real;

У прикладі в) M і N - константи цілого типу. Зверніть увагу на те, що значення типу Matrix - M*N матриці - визначаються як масиви, компонентами яких, в свою чергу, є масиви з дійсних чисел.

Регулярний тип, значеннями якого є багатомірні масиви (наприклад, в) і г)), можна визначати в скороченому виді:

Type <ім'я> = Array [<Тип> {,<Тип>}] of <тип компоненти>;

Наприклад:

а) Type matrica = array [1..N,1..M] of real;

б) Type Index1 = -10..10;
Index2 = 0..20;
Index3 = 0..3;

Tenzor = Array [Index1, Index2, Index3] of Real;

в) Type Labyrinth = array [1..100,1..100] of Boolean;

Типи Real і Integer не можуть бути типами індексів!

Компонента змінної регулярного типу - компонента масиву явно позначається іменем змінної, за яким у квадратних дужках слідують індекси; індекси являються виразами типу індексу. Наприклад, Table[1, i+j], T[2*i+1, (j*j) mod i], S[Monday, Friday]. Зверніть увагу на те, що на відміну від індексних виразів, межі індексів у змінних - масивах повинні бути константами. Значення індексних виразів повинні бути значеннями типу індексу, тобто знаходитись в межах, що визначені межами індексів.

Розглянемо приклади:

Приклад 6. Програма обчислює скалярний добуток вектора V і вектора V`, отриманого з V перестановкою координат у зворотному порядку.

```
Program ScalarMult;
 Const n = 10;
  Type Vector =  array[1..n] of Real;
   Var V : Vector;   Summa : Real;   i : Integer;
 Begin
  For i := 1 to n do begin { блок читання вихідного вектора }
   Write('Введіть координату вектора : ');  Readln(V[i]);
  end;
  Summa := 0; { блок обчислень}
  For i := 1 to n do
   Summa := Summa + V[i]*V[n-i+1];
   write(' Результат : ', Summa)
 End.
```

Блок обчислень можна оптимізувати за часом. Зазначимо, що Summa обчислюється за формулою: : Summa = V[1]*V[n] + V[2]*V[n-1] +...+ V[2]*V[n-1] + V[1]*V[n]. Отже, її доданки, рівновіддалені від кінців, рівні. Тому кількість повторень циклу можна зменшити вдвоє. При цьому необхідно враховувати парність числа n:

```
{Program ScalarMult1;}
For i := 1 to n div 2 do   Summa := Summa + V[i]*V[n-i+1];
 If Odd(n)
  then Summa := 2*Summa
  else Summa := 2*(Summa + V[n div 2 + 1];
```

Приклад 7. Програма розкладу матриці в суму симетричної і кососиметричної матриць.

Позначимо через Mat, Sym, Assym відповідно вхідну, симетричну і кососиметричну матриці. Тоді

Sym[i, j] = (Mat[i, j] + Mat[j, i])/2

Assym[i, j] = (Mat[i, j] - Mat[j, i])/2

Блоки введення - виведення даних пропонуємо читачеві оформити самостійно. Використовуючи уроки попередньої програми, врахуємо властивості симетрії матриць Sym, Assym:

Sym[i, j] = Sym[j, i], Assym[i, j] = - Assym[j, i];

```
          Program MatrixSymmetry;
  Const n = 10;
   Type Msize = 1 .. n ;
        Matrix = Array [Msize, Msize] of Real;
    Var Mat, Sym, Assym : Matrix;
        i, j : Msize;
Begin
{блок читання вихідної матриці}
For i := 1 to n do
 For j := 1 to i do
{ перебор усіх елементів під головною діагоналлю }
 begin
  Sym[i, j] := (Mat[i, j] + Mat[j, i])/2;
  Assym[i, j] := (Mat[i, j] - Mat[j, i])/2;
  Sym[j, i] := Sym[i,j];                  {симетрія}
  Assym[j, i] := -Assym[i, j]             {симетрія}
 end;
{блок друкування отриманих матриць}
End.
```

Ще одне джерело зайвих дій - пересилання даних в масиви, що супроводжуються обчисленнями індексних виразів. Ми двічі звертаємось до Mat[i,j], Mat[j,i], Sym[i,j], Assym[i,j]. Введенням декількох простих допоміжних змінних усунемо цей недолік:

```
{Program MatrixSymmetry1;}
{Var X, Y, U, V : Real;}
For i := 1 to n do
 For j := 1 to i do
{перебор усіх елементів під головною діагоналлю}
 begin { тіло внутрішнього циклу }
  X := Mat[i, j], Y := Mat[j, i];
  U := (X + Y)/2; V := U - X ;    {співвідношення U + V = X}
  Sym[i, j] := U; Sym[j, i] := U;    {симетрія}
  Assym[i, j] := V; Assym[j, i] := -V {симетрія}
 end;
```

На цьому прикладі продемонстровано прийом оптимізації обчислень, що використовують змінні з індексами - усунення зайвих до них звернень. Змінні з індексами подібні функціям: для обчислення їх значень необхідно обчислювати значення індексів!

6. Пошук елемента в масиві.

Задача пошуку елемента в послідовності - одна з важливих задач програмування як з теоретичної, так і практичної точок зору. Ось її формулювання:

Нехай A = {a_1, a_2, ...} - послідовність однотипних елементів і b - деякий елемент, який володіє властивістю P. Знайти місце елемента b в послідовності A. Оскільки представлення послідовності в пам'яті може бути здійснено в виді масиву, задачі можуть бути уточнені як задачі пошуку елемента в масиві A:

1.Знайти максимальний елемент масиву;

2.Знайти даний елемент масиву;

3.Знайти k-тий за величиною елемент масиву;

Найбільш прості і часто оптимальні алгоритми основані на послідовному перегляді масиву A з перевіркою властивості P на кожному елементі.

Приклад 8. Пошук мінімального елемента в масиві.

```
Program MinItem;
  Const n = 23;
  Var A : Array [1..n] of Real;
        Min, Item : Real;   Index, i : Integer;
  Begin
   {Блок введення масиву}
   Index := 1; Min := A[1];
   For i := 1 to n do begin
     Item := A[i];
     If Min > Item
      then begin Index := i; Min := Item end
   end;
   {Виведення значень Index, Min}
  End.
```

Змінна Item введена для того, щоб уникнути повторних обчислень індексного виразу в A[i]. Ми пропонуємо читачу розглянути поведінку цього алгоритму і його модифікації у випадку, коли мінімальних елементів у масиві декілька і треба знайти перший, останній, а також всі мінімальні елементи.

7. Ефективність алгоритму за часом.

Раніш, ніж розглянути задачу пошуку даного елемента, відзначимо важливу відмінну особливість алгоритмів обробки масивів: їх складність визначається характерними параметрами - розмірами вхідних масивів. У розглянутих вище прикладах розмір входу - константа n.

Програма	Складність	Оцінка складності
ScalarMult	$T(n) = T_b*n$	$T(n) = O(n)$
ScalarMult1	$T(n) = T_b*n/2$	$T(n) = O(n)$
MatrixSymmetry	$T(n) = T_b*n*(n + 1)/2$	$T(n) = O(n^2)$
MatrixSymmetry1	$T(n) = T_b*n*(n + 1)/2$	$T(n) = O(n^2)$
MinElement	$T(n) = T_b*n$	$T(n) = O(n)$

Тут T (b) - складність тіла (внутрішнього) циклу. В реальних програмах величина T_b залежить від багатьох факторів, зв'язаних з апаратурою, операційною системою і системою програмування. Тому якість алгоритму, реалізованого у виді програми і не залежного від зовнішніх обставин можна оцінити функцією параметра задачі. При цьому основна властивість функції оцінки складності - швидкість її росту. Для алгоритму ScalarMult ця функція зростає лінійно. Цей факт відображений формулою $T(n) = O(n)$. Складність алгоритму MatrixSymmetry оцінюється квадратичною функцією. Тому $T(n) = O(n^2)$. (Точне визначення запису O(n) дано в математичному аналізі.)

Абстрагування від деталей реалізації програми дає можливість оцінювати алгоритми розв'язування задач і їх реалізацію у виді програми. Мірою ефективності алгоритму є оцінка його складності.

Розглянемо задачу пошуку даного елементу в масиві. Очевидний алгоритм її розв'язування, як і в попередній задачі - послідовний перегляд масиву і порівняння кожного елемента масиву з даним. Відміна полягає в тому, що коли елемент знайдений, перегляд можна припинити. Це означає, що виконання циклу переривається. У мові є засіб переривання - оператор переходу.

8. Мітки. Оператор переходу. Застосування оператора переходу для дострокового виходу з циклу.

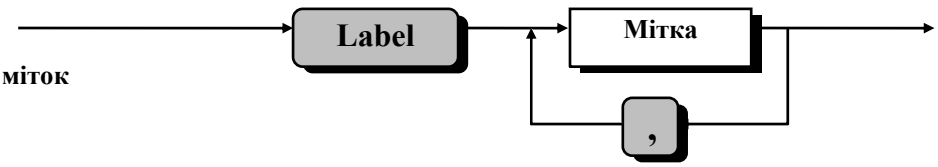
Оператор переходу вказує, що подальша робота (виконання програми) повинна продовжуватись з іншої точки програми, а саме, з оператора, відміченого міткою.

Оператор має вид:

Goto < мітка >

Мітка представляє собою ціле число без знака, що складається не більш ніж з 4 цифр. У розділі операторів кожна мітка може зустрічатися тільки перед одним оператором. Кожна мітка, що зустрічається в розділі операторів, повинна бути описана в розділі міток.

Розділ міток визначений наступною діаграмою:



Розділ міток

Після мітки, що відмічає оператор, треба ставити двокрапку. Наприклад:
1995 : x := x +I; 1 : read(Y);

Увага! Дія оператора переходу усередину складного оператора ззовні не визначено.

Оператор переходу слід використати у незвичайних, виняткових ситуаціях, коли доводиться порушувати природну структуру алгоритму. Треба пам'ятати, що будь-який алгоритм може бути реалізований без застосування Goto без втрати ефективності. Цей факт має принциповий характер. Саме тому структурний стиль програмування іноді називають "Програмування без Goto".

У якості єдиного приклада програми з Goto розглянемо задачу пошуку елемента в одномірному масиві.

Приклад 9.

```
Program Search_in_Array;
  Label 1;
  Const n = 100;
    Var A : Array[1..n] of Real;
        b : Real;
        Flag : Boolean;
           i : Integer;
  Begin
   {Блок читання масиву A і елемента b}
   Flag := true;
   For i := 1 to n do
     If A[i] = b then begin
       Flag := false; goto1
   end; { переривання циклу }
   1:If Flag
     then Writeln(' Елемент ',b,' у  масиві відсутній ')
     else Writeln(' елемент ',b,' стоїть на ',i,'-тому  місці ');
  End.
```

Вишуканий (елегантний) розв'язок цієї задачі без застосування Goto буде розглянуто нижче.

9. Постановка задачі сортування.

Під сортуванням послідовності розуміють процес перестановки елементів послідовності у визначеному порядку. Мета такої впорядкованості - полегшення подальшої обробки даних (зокрема, задачі пошуку). Тому задача сортування - одна з найбільш важливих внутрішніх задач програмування.

Цікаво, що задача сортування є ідеальним прикладом великої кількості різноманітних алгоритмів, розв'язування одної і тої ж задачі. Розглядаючи різні методи сортування, ми побачимо, як зміна алгоритму приводить до нових, більш ефективних у порівнянні з простими, розв'язувань задачі сортування.

Крім цього, послідовності можна представити (реалізувати) в пам'яті різними структурами даних. Як і слід очікувати, ефективність алгоритмів стає дуже залежною від реалізації послідовності.

Нехай дана послідовність елементів a_1, a_2, ... , a_n. Елементи цієї послідовності - дані довільного типу, на якому визначено відношення порядку "<<" (менше) таке, що будь-які два різні елементи можна порівняти.

Сортування означає перестановку елементів послідовності a_{k1}, a_{k2}, ... , a_{kn} таку, що $a_{k1} << a_{k2} << ... << a_{kn}$.

Приклад: послідовність документів, кожний з яких містить інформацію про людину, включаючи його вік. Потрібно розмістити документи цієї послідовності у порядку за віком, починаючи з старшого.

Сортування масивів.

Якщо послідовність a_1, a_2, ... , a_n реалізована як масив a[1..n], вся вона розміщена в адресованій пам'яті. Тому наряду з вимогами ефективності за часом основна вимога - економне використання пам'яті. Це означає, що алгоритм не повинен використовувати додаткові масиви і пересилки з масиву a в ці масиви.

Постановка задачі сортування в загальному виді передбачає, що існують тільки два типи дій з даними сортованого типу : порівняння двох елементів (x << y) і пересилання елемента (x := y). Тому зручна міра складності алгоритму сортування масиву a[1..n] за часом - кількість порівнянь C(n) і кількість пересилань M(n).

Прості алгоритми сортування.

Прості алгоритми сортування можна класифікувати як сортування обміном, сортування вибором і сортування включеннями.

Сортування обмінами.

Основна дія сортування обмінами - порівняння двох елементів і, якщо результат порівняння від'ємний, перестановка їх місцями:

Якщо при i < j a[i] > a[j] то переставити a[i] і a[j].

В найбільш простому варіанті порівнюються елементи a[j] і a[j+1], що стоять поряд:

Приклад 10.

```
Program BublSort;
  Const n = 100;
    Var a : array[1..n] of Real;
        i, j : Integer;
       TempMem : Real;
 Begin
  { Блок введення масиву }
  For i := n - 1 downto 1 do
  For j := 1 to i do
  If a[j] > a[j+1]
    then begin
      TempMem := a[j+1]; a[j+1] := a[j]; a[j] := TempMem end;
  { Блок виведення масиву }
 End.
```

Аналіз алгоритму сортування обмінами.

Аналіз алгоритму полягає в обгрунтуванні його властивостей. Важливішими властивостями алгоритму є: коректність (правильність), оцінка складності за часом і пам'яттю, а також і деякі інші властивості.

Для обгрунтування алгоритму сортування необхідно довести, що алгоритм завжди (незалежно від входу) завершує свою роботу і його результат - зростаюче впорядкований масив. Оцінка складності (ефективності) алгоритму за часом полягає у находженні C(n), M(n) у гіршому випадку, у середньому.

Оскільки алгоритм сортує масив "на місці", його складність по пам'яті - константа.

До інших властивостей алгоритму можна віднести властивість стійкості. (Алгоритм сортування називається стійким, якщо він не переставляє місцями тотожні елементи.) Здійснимо аналіз алгоритму сортування обмінами.

Завершуваність.

Алгоритм BublSort завжди завершує свою роботу, оскільки він використовує тільки цикли з параметром, і в тілі циклів параметри примусово не змінюються.

Коректність.

Внутрішній цикл алгоритму (з параметром j) здійснює послідовний перегляд перших і елементів масиву. На кожному кроці перегляду порівнюються і, якщо це необхідно, переставляються два сусідніх елемента. Таким чином, найбільший серед перших і +1 елементів " спливає " на і +1-е місце. Тому після виконання оператора If має місце співвідношення:

$$a[j+1] = Max(a[1], ..., a[i], a[j+1]) \quad (1)$$

а після завершення циклу (i = j) має місце співвідношення

$$a[i+1] = Max(a[1], ..., a[i], a[i+1]) \quad (2)$$

Зовнішній цикл (з параметром i) керує довжиною частини масиву, що переглядається: вона зменшується на 1. Тому після завершення внутрішнього циклу має місце співвідношення:

$$a[i+1] \le a[i+2] \le \ ... \le a[n] \quad (3)$$

Після завершення зовнішнього циклу отримаємо:

$$a[1] \le \ a[2], a[2] \le \ a[3] \le \ ... \le \ a[n] \quad (4)$$

тобто масив відсортований.

Відзначимо, що більш докладне обгрунтування складається перш за все в доказі співвідношень (1), (3). Це можна зробити методом математичної індукції.

Ефективність за часом.

Зовнішній цикл виконався n-1 разів. Внутрішній цикл виконується i разів (i = n-1, n-2, ..., 1). Кожне виконання тіла внутрішнього циклу складається одно порівняння і, можливо, з одної перестановки. Тому $C(n) = 1+2+ ... +n-1 = n*(n - 1)/2$, $M(n) \le n*(n - 1)/2$. У гіршому випадку (коли елементи вихідного масиву розміщені в порядку спадання)

$C(n) = n*(n - 1)/2 = O(n^2)$, $M(n) = n*(n - 1)/2 = O(n^2)$

Стійкість.

Для доказу стійкості достатньо відмітити, що переставляються тільки сусідні нерівні елементи.

Сортування вибором.

Приклад 11.

Основна дія сортування вибором - пошук найменшого елемента в частині масиву, що переглядається і перестановка з першим елементом частини, що переглядається:

```
For i := 1 to n - 1 do begin
 k := Індекс( Min(a[i], ..., a[n]));
 Переставити a[i], a[k]
end;
Program SelectSort;
  Const n = 10;
    Var a : array[1..n] of Real;
       i, j, MinIndex : Integer;
       TempMem : Real;
  Begin
   {Блок введення масиву}
   For i := 1 to n - 1 do  begin
```

```
  { пошук мінімального елемента }
   MinIndex := i;
   for j := i + 1 to n do
   If a[j] < a[MinIndex] then MinIndex := j;
   {перестановка елементів}
   TempMem := a[MinIndex]; a[MinIndex] := a[i]; a[i] := TempMem
  end;
 {Блок виведення масиву}
End.
```

Аналіз алгоритму сортування вибором.

Внутрішній цикл здійснює пошук мінімального елемента. Після виконання оператора If має місце співвідношення Min = Min(a[i], a[i+1], ... , a[j]), а після завершення циклу Min = Min(a[i], a[i+1], ... , a[n]). Після перестановки маємо a[i] = Min(a[i], a[i+1], ... , a[n]).

Зовнішній цикл керує довжиною частини масиву, що переглядається. Після виконання тіла зовнішнього циклу початок масиву вже відсортований: $a[1] \le a[2] \le .... \le a[i]$

Після завершення зовнішнього циклу отримаємо:

$a[1] \le a[2] \le ... \le a[n-1]$, a[n-1] = Min(a[n-1], a[n]), тобто масив відсортований.

Зовнішній цикл виконався n-1 разів. Внутрішній цикл виконується i-1 разів (i = n-1, n-2, ...,1). Кожне виконання тіла внутрішнього циклу складається в одному порівнянні. Тому

$C(n) = 1 + 2 + ... + n - 1 = n(n - 1)/2$,

Перестановка елементів здійснюється у зовнішньому циклі. Тому

$M(n) = 3(n - 1)$

Також, як і алгоритм сортування обмінами, алгоритм, що аналізується, стійкий. Дійсно, блок пошуку мінімального елемента в хвості масиву шукає мінімальний елемент з найменшим індексом, тому перестановки будуть стійкими. Можна зробити висновок, що просте сортування вибором ефективніше сортування простими обмінами за критерієм M(n). Якщо послідовність, що сортується, складається з даних великого розміру, цей критерій може мати вирішальне значення.

10.Задачі і вправи.

І.Скласти програму табулювання функції z = f(x, y) у прямокутнику [a, b]*[c, d] з кроком табуляції h і точністю ε

	Функція z = f(x,y)	[a, b]	[c, d]	h	ε
1	$z = \ln(1 + x^2 + y^2)$	[-2, 3]	[-1, 3]	0.1	10^{-5}
2	$z = \sin^2(2x+y)$	[$-\pi$, $\pi/2$]	[-2π, π]	0.2	10^{-6}
3	$z = \exp(x^2+y^2)$	[-2, 2]	[-2, 2]	0.1	10^{-5}

4	$z = 1/(tg^2(x+y)+1)$	$[-\pi/2, \pi/2]$	$[-\pi, \pi]$	0.2	10^{-4}
5	$z = \ln(x+sqrt(x^2+y^2))$	[-2, 3]	[-2, 3]	0.1	10^{-5}

II. Скласти програму розв'язування задачі і оцінити її складність:

1.Дано масив A[1..n]. Знайти суму всіх елементів A, що стоять між A[1] і A[n].

2.Дано масив A[1..n]. Підрахувати середнє арифметичне всіх від'ємних і всіх додатних його елементів.

3.Послідовність з n точок площини задана масивами X[1..n] і Y[1..n] координат. Знайти точку, найменш віддалену від початку координат.

4.Дано масив A[1..n]. Знайти всі його елементи, менші, ніж всі попередні.

5.Дано масив A[1..n]. Знайти в цьому масиві найбільшу за кількістю елементів зростаючу підпослідовність елементів, що йдуть підряд.

6.Дано масив A[1..n], що складається з непарного (n = 2k+1) числа попарно нерівних елементів. Знайти середній по величині елемент у масиві.

7.Дано масиви A[1..n] і B[1..m]. Знайти всі елементи масиву A, що не входять у B.

8.Дано масиви A[1..n] і B[1..m]. Знайти всі елементи масиву A, що входять у B.

9.Дано масив A[1..n]. Величину S_{ij} визначимо наступним чином:

$S_{ii} = A[i]$,

$S_{ij} = A[i] + A[i+1] + ... + A[j]$ при $i < j$

$S_{ij} = S_{ji}$ при $i > j$

Знайти Max S_{ij} при $1 \le i, j \le n$

III. Скласти програму розв'язування задачі і оцінити її складність:

7.Помножити матрицю A розміром m x n на вектор b.

8.Перемножити матриці A і A'. (A' - A транспонована)

9.Перемножити матриці A і B.

10.Дано масив A[1..n, 1..m]. Знайти сідлову точку масиву або встановити її відсутність. (Елемент двомірного масиву називається сідловою точкою, якщо він максимальний у своєму стовпці і мінімальний у своєму рядку.)

11.Дано масив A[1..n, 1..m]. Знайти стовпець, сума квадратів елементів якого мінімальна.

12.Дано масив A[1..n, 1..m]. Знайти всі елементи A менші, ніж усі сусідні. (Сусідніми називаються елементи, індекси яких відрізняються на 1).

ІТЕРАЦІЙНІ ЦИКЛИ.

1. Оператори повторення While і Repeat.

У попередньому параграфі ми вивчили оператор повторення з параметром (For).

Це оператор використовується лише у випадку, коли заздалегідь відома кількість повторень тіла циклу. У більш загальному випадку, коли кількість повторень заздалегідь невідома, а задана деяка умова закінчення (або продовження) циклу, у мові Pascal використовують інші оператори повторення: оператор циклу з передумовою While і оператор циклу з постумовою Repeat.

Оператор циклу з передумовою визначений діаграмою:

Оператор циклу з передумовою

Оператор (тіло циклу) виконується до тих пір, поки умова істинна. Якщо при першій перевірці умова виявилась хибною, оператор не виконується ні разу.

Приклад 1. Знайти найменший натуральний розв'язок нерівності $x^3 + ax^2 + bx + c > 0$ з цілими коефіцієнтами.

Очевидний алгоритм пошуку розв'язку складається у послідовному обчисленні значень $Y = x^3 + ax^2 + bx + c$ для x = 1, 2, 3, ... до тих пір, поки $Y \le 0$.

```
Program UneqvSolution;
        Var a, b, c : Integer;
                X : Integer;  Y : Real;
      Begin
        Write(‘ введіть коефіцієнти a, b, c : ‘); Readln(a, b, c);
         X := 1; Y := a + b + c + 1; { Ініціалізація циклу }
        While Y <= 0 do begin
         X := Succ(X); { Наступне значення X }
         Y := ((X + a)*X + b)*X + c { Наступне значення Y }
      end;
      Writeln(‘X = ‘, X, ‘ Y = ‘, Y )
  End.
```

Для обгрунтування цього методу вимагається єдине: впевнитись у тому, що цикл коли-небудь завершиться, тобто що розв'язок нерівності існує! Відмітимо, що

$\lim (x^3 + ax^2 + bx + c) = +\infty$

$x \to +\infty$

Тому при деякому x_0 $x_0^3 + ax_0^2 + bx_0 + c > 0$ Покладемо $x = |a| + |b| + |c|$. Тоді $x^3 = (|a| + |b| + |c|)x^2 > |a|x^2 + |b|x + c$ Отже, $x^3 + ax^2 + bx + c > x^3 - (|a|x^2 + |b|x + c) > 0$ Тому кількість повторень циклу не більше ніж величини $|a| + |b| + |c|$, що дає можливість не тільки обгрунтовувати завершення, але й оцінити складність програми у гіршому випадку.

Оператор циклу з постумовою визначений діаграмою:

Тіло циклу Repeat виконується до тих пір, поки умова приймає значення False. Дії, що містяться в тілі циклу, будуть виконані у крайньому випадку один раз. Таким чином, умова є умовою закінчення циклу.

Приклад 2. Знайти номер найменшого числа Фібоначчі, що ділиться на 10. Послідовність Фібоначчі { F(n) } визначається рекуррентно:

F(1) = F(2) = 1, F(n+2) = F(n+1) + F(n)

Як і у попередньому прикладі, обчислюємо F_n для n = 3, 4, ... до тих пір, поки не знайдемо елемент послідовності, що ділиться на 10. Проблема обгрунтування методу залишається тією ж: чи існує потрібний елемент?

```
Program Fibbonachy;
  Var Fib1, Fib2 : Integer;
        Index : Integer;  Buf : Integer;
    Begin
Fib1 := 1; Fib2 := 1; Index := 2; { Ініціалізація циклу }
Repeat
  Buf := Fib2;
  Fib2 := Fib2 + Fib1;                  { Наступне число Фібоначчі }
  Fib1 := Buf;                          { Попереднє число Фібоначчі }
  Index := Succ(Index)                  { Номер числа Фібоначчі }
 until Fib2 mod 10 = 0;
 Writeln('Номер = ', Index, ' Число = ', Fib2)
End.
```

Цикли While і Repeat називають ще ітераційними циклами, оскільки за їх допомогою легко реалізувати різного роду ітераційні обчислення (обчислення, в яких кожний наступний результат є уточненням попереднього). Умова закінчення циклу - досягнення відхилення результату Y_n от Y_{n-1} деякої допустимої похибки ε:

$$|Y_n - Y_{n-1}| < \varepsilon$$

Приклад 3: Обчислити з заданою точністю значення кореня рівняння x = cos(x) на відрізку [0 ; 1] методом ділення відрізка навпіл .

Метод наближеного розв'язування рівняння діленням відрізка навпіл - один з найбільш простих і поширених чисельних методів. Суть його полягає у послідовному уточненні меж відрізка, на якому існує корінь. На кожному кроці методу цей відрізок ділиться навпіл, а потім вирішується питання про те, в якій половині (лівій чи правій) знаходиться корінь. Зменшення відрізка вдвоє теоретично гарантує завершення циклу пошуку з заданою точністю.

```
Program TransEquation;
  Const a = 0; b = 1;
        Epsilon = 1e-5;
   Var u, v, Root : Real;
        Fu, Fr : Real;
  Begin
   u := a; v := b; { ініціалізація циклу}
   Fu := Cos(u) - u;
   Repeat
     Root := (u + v)/2 ; { середня точка відрізка }
     Fr := Cos(Root) - Root;
     if Fu*Fr > 0 { вибір правої або лівої половини }
      then begin u := Root; Fu := Fr end
      else v := Root
   until Abs(v - u) < Epsilon;
  writeln (‘ корінь= ‘, Root, ‘ з точністю ‘,Epsilon)
 End.
```

Для того, щоб знайти кількість N повторень у циклі, відмітимо, що на k-тому кроку циклу виконана нерівність $| v - u | = (b - a)/2^k$. Тому при найменшому k такому, що $(b - a)/2^k < \varepsilon$ цикл завершиться. Неважко обчислити N:

$$N = [\log_2(b - a)/ \varepsilon] \qquad (1)$$

Як показують розглянуті приклади, одна з основних проблем аналізу програм, що містять ітераційні цикли, складається в обгрунтуванні завершення циклу (і програми). На відміну від алгоритмів з арифметичними циклами, ця проблема не є тривіальною!

Оскільки кількість повторень ітераційного циклу не визначена, задача оцінки складності програми часто також не проста.

2. Алгоритми пошуку і сортування. Лінійний пошук у масиві.

Приклад 4. Розглянемо красивий розв'язок задачі пошуку даного елемента в масиві, що використовує цикл While. Насправді, замість використання циклу For з достроковим виходом при допомозі Goto можна застосувати цикл While:

```
Program WhileSearch;
  Const n = 100;
    Var A : Array[1..n] of Real;
        b : Real; i : Integer;
  Begin
  {Блок читання масиву A і елемента b}
   i := 1;
   While (i <= n) and (A[i] <> b) do i := Succ(i);
     If i = n + 1
       then Writeln('Елемент ',b,'у масиві відсутній')
       else Writeln('елемент ',b, розміщений на',i,'-тому місці');
     End.
```

Недоліком такої реалізації є по-перше, використання складної умови в циклі, по-друге - можливо некоректне обчислення підвиразів A[i] <> b при i = n +1. Не дивлячись на те, що умова завідомо хибна (i <= n = False), вираз A[n+1] <> b не визначено!

Доповнимо масив A ще одним елементом: A[n+1] = b, і всі недоліки легко ліквідуються!

```
Program LinearSearch;
  Const n = 101; { Розмір масиву збільшений на 1 }
    Var A : Array[1..n] of Real;
        b : Real;  i : Integer;
  Begin
   { Блок читання масиву A і елемента b }
   i := 1;
   A[n] := b; { Доповнимо масив "бар'єрним" елементом }
   While A[i] <> b do i := Succ(i);
    If i = n
      then Writeln('Елемент ',b,'в масиві відсутній')
      else Writeln('елемент ',b,'розміщений на',i,'-тому місці');
   End.
```

Поліпшений алгоритм сортування обмінами.

Приклад 5. У програмі BublSort цикл For використовується в якості зовнішнього. Це приводить до того, що він виконується рівно n - 1 разів, навіть якщо масив вже упорядкований після декількох перших проходів. Від зайвих проходів можна позбавитися, застосувавши цикл Repeat і перевіряючи у внутрішньому циклі масив на упорядкованість:

```
Program BublSort1;
        Const n = 100;
  Var a : array[1..n] of Real;
      i, j : Integer;
     TempMem : Real; isOrd : Boolean;
 Begin
  {Блок введення масиву}
  i := n - 1;
  Repeat
   isOrd := True; { ознака упорядкованості}
   For j := 1 to i do
    If a[j] > a[j+1]
     then begin
      TempMem := a[j+1]; a[j+1] := a[j]; a[j] := TempMem;
       isOrd := False { виявлено“засортування” }
      end;
    i := Pred(i)
   until isOrd;
  {Блок виведення масиву}
 End.
```

Цей алгоритм можна ще покращити, якщо кожний наступний прохід починати не з початку (j = 1), а з того місця, де на попередньому проході відбувся перший обмін:

```
{LowIndex : Integer;}
Repeat
isOrd := True; { ознака упорядкованості}
LowIndex := 1; { відрізок A[1..LowIndex] упорядкований}
For j := LowIndex to i do
 If a[j] > a[j+1]
 then begin
  TempMem := a[j+1]; a[j+1] := a[j]; a[j] := TempMem;
   If isOrd then begin LowIndex := j; isOrd := False
  end
 end;
 i := Pred(i)
until isOrd;
```

Відзначимо однак, що наш алгоритм працює не симетрично: якщо найбільший елемент “спливає” на своє місце за один прохід, то найменший елемент “потоне” на 1-е місце, знаходячись початково на n-тому місці, за n-1 прохід. При цьому здійсниться максимально можливе число порівнянь. Усунути асиметрію можна, чергуючи проходи

вперед і назад. Реалізацію цієї ідеї, як і подальші поліпшення алгоритму сортування простими обмінами, ми надаємо читачеві в якості вправи.

Всі поліпшення методу, що розглядається відносяться до ефективності в середньому. Жодне з них не зменшує оцінки складності C (n) у гіршому випадку. (Доведіть це самостійно!)

Бінарний пошук в упорядкованому масиві.

Приклад 6. Задача пошуку суттєво спрощується, якщо елементи масиву упорядковані. Стандартний метод пошуку в упорядкованому масиві - це ділення відрізка навпіл, причому відрізком є відрізок індексів 1..n. Насправді, нехай m (k < m < l) - деякий індекс. Тоді якщо A[m] > b, далі елемент треба шукати на відрізку k..m-1, а якщо A[m] < b - на відрізку m+1..l.

Для того, щоб збалансувати кількість обчислень в тому і іншому випадках, індекс m треба вибирати так, щоб довжини відрізків k..m, m..l були (приблизно) рівними. Описану стратегію пошуку називають бінарним пошуком.

```
Program BinarySearch;
  Const n = 100;
   Var A : Array[1..n] of Real;
       b : Real; Buffer : Real;
       k, l, m : Integer;

 Begin
   { Блок читання упорядкованого масиву A і елемента b }
   k := 1; l := n;
   Repeat
   m := (k + l) div 2; Buffer := A[m];
   If Buffer > b
    then l := Pred(m)
    else k := Succ(m)
   until (Buffer = b) or (l < k);
   If A[m] = b
    then Writeln('Індекс = ', m)
    else Writeln( 'Елемент не знайдений')
  End.
```

Неважко отримати оцінку числа C (n) порівнянь методу, застосувавши формулу (1) прикладу 3. Нехай M - кількість повторень циклу. Тоді

$$M \leq [\log_2 (n)]$$

Оскільки на кожному кроку здійснюється 2 порівняння, у гіршому випадку

$$C(n) = 2[\log_2 n] = O(\log_2 n) \qquad (2)$$

3. Алгоритми сортування масивів (продовження). Сортування вставками.

Ще один простий алгоритм сортування - сортування вставками базується на наступній ідеї: припустимо, що перші k елементи масиву A[1..n] вже упорядковані:

$A[1] \le A[2] \le ... \le A[k], A[k+1], ..., A[n]$

Знайдемо місце елемента A[k+1] в початковому відрізку A[1],...,A[k] і вставимо елемент на своє місце, отримавши упорядковану послідовність довжини k+1. Оскільки початковий відрізок масиву упорядкований, пошук треба реалізувати як бінарний. Вставці елемента на своє місце повинна передувати процедура зсуву “хвоста” початкового відрізка для звільнення місця.

```
Program InsSort;
  Const n = 100;
    Var A : array[1..n] of Integer;
        k : Integer;
        l, r, i, j : Integer;
        b : Integer;
  Begin
  {Блок читання масиву A}
   For k := 1 to n-1 do begin
    b := A[k+1];
    If b < A[k]
     then begin
      l := 1; r := k; {Бінарний пошук}
     Repeat
       j := (l + r) div 2;
       If b < A[j]  then r := j  else  l := j + 1;
      until (l = j);{ Зсув “хвоста” масиву на 1 позицію праворуч}
      For i := k downto j do A[i+1] := A[i];
     A[j] := b ; { Пересилання елемента на своє місце}
    end
   end;
  {Блок виведення масиву A}
  End.
```

Оцінимо ефективність алгоритму. Пошук міста елемента A[k+1] потребує, як показано вище, $O(\log_2 k)$ порівнянь. Тому у гіршому випадку кількість порівнянь C (n) є

$C(n) = O(\log_2 2) + ... + O(\log_2(n-1)) + O(\log_2 n) = O(n \log_2 n)$

Зсув “хвоста” на кожному кроку зовнішнього циклу у гіршому випадку потребує k перестановок. Тому у гіршому випадку

$M(n) = 1 + ... + (n-2) + (n-1) = n*(n-1)/2 = O(n^2)$

Таким чином, алгоритм сортування вставками значно ефективніший, ніж всі розглянуті раніше алгоритми за числом порівнянь. Однак число перестановок у гіршому випадку також буде значним, як і у самому неефективному алгоритмі - сортуванню простими обмінами.

4. Задачі і вправи.

Вправа 1. Визначити число членів нескінченного ряду чисел, необхідне для обчислення його суми з точністю ε.

1. $\sum_{n=0}^{\infty} [(-1)^n /(2n+1)] = \pi/4$
2. $\sum_{n=0}^{\infty} [(-1)^n/(2n)!] = \cos 1$
3. $\sum_{n=0}^{\infty} [(-1)^n/(2n+1)!] = \sin 1$

Вправа 2. Знайти суму членів функціонального ряду з заданою похибкою ε.

1. $\sin x = \sum_{n=0}^{\infty} (-1)^n(x^{2n+1})/(2n+1)!$
2. $\ln x = \sum_{n=1}^{\infty} (-1)^n * x^n / n;\ x>1/2$
3. $\operatorname{arctg} x = \sum_{n=0}^{\infty} (-1)^n\, x^{2n+1}/(2n+1);\ |x|<1$

Вправа 3. Знайти всі трьохзначні члени послідовності F_n, визначеної рекурентними співвідношеннями:

1. $F_0 = 1,\ F_1 = 1,\ \ F_{n+2} = F_{n+1} + F_n$ при $n > 0$
2. $F_0 = 5,\ F_1 = 1,\ F_2 = 1,\ \ F_{n+3} = F_{n+2} + F_{n+1} - F_n$ при $n > 0$

Вправа 4.

4.Дано раціональне число p/q. Побудувати його розклад у ланцюговий дріб.

5.Побудувати розклад натурального числа N у добуток степенів його простих дільників.

6.Перевести ціле число N у 2-ічну систему числення. Узагальнити алгоритм на випадок p-ічної системи числення.

4.Відомо, що якщо d = НСД(a,b), то існують такі числа u, v, що d = au + bv. По даним a, b знайти d, u, v.

5.Число N називається досконалим, якщо воно співпадає з сумою своїх власних дільників. (Наприклад, 6=1+2+3). Знайти всі досконалі числа, які не перевищують M.

6.Знайти номер найменшого числа Фібоначі, що закінчується 2-ма нулями. Знайти само це число.

Вправа 5.

4.Дано масив A[1..n] дійсних чисел. Знайти у цьому масиві найменше додатне число.

5.Дано масив A[1..n] цілих чисел. Знайти в цьому масиві найбільшу кількість нулів, що йдуть підряд.

6.Дано масив A[1..n] цілих чисел. Відрізок індексів k .. m називається відрізком росту масиву, якщо A[k] < A[k+1] < ... < A[m]. Знайти відрізок росту максимальної довжини.

Задача 1. Масив A[1..n] складається з даних типу Color = (white, black). Скласти програму сортування масиву A, яка перевіряє значення (колір) кожного елемента тільки один раз.

Задача 2. Масив A[1..n] складається з даних типу Color = (white, red, black). Скласти програму сортування масиву A, яка перевіряє значення (колір) кожного елемента тільки один раз.

Задача 3. Масив A[1..n] складається з дійсних чисел, причому для будь-якого $2 \le i \le n-1$ виконана нерівність $A[i] \le (A[i-1] + A[i+1])/2$ Скласти програму пошуку найменшого і найбільшого елементів масиву, складність якої $C(n) = O(\log_2(n))$

Задача 4.Задані упорядковані масиви A[1..n] і B[1..m] цілих чисел і ціле число C. Скласти програму пошуку таких індексів i і j, що C = A[i] + B[j], складність якої $C(n) = O(\log_2(n\ m))$

Задача 5. Множина точок площини $\{A_i = (X_i, Y_i)\}$ задана масивами координат X[1..n] і Y[1..n]. Випуклою лінійною оболонкою цієї множини називається така його підмножина B, що багатокутник з вершинами в точках з B є випуклими і містить всю множину A. Скласти програму пошуку випуклої лінійної оболонки множини A. Оцінити її складність.

Задача 6.

а) Реалізувати арифметичний цикл за допомогою циклу While;

б) Реалізувати арифметичний цикл за допомогою циклу Repeat;
в) Реалізувати цикл While за допомогою циклу Repeat;
г) Реалізувати цикл Repeat за допомогою циклу While;
д) Реалізувати цикл While за допомогою операторів If і Goto;
є) Реалізувати цикл Repeat за допомогою операторів If і Goto;

ПРОЦЕДУРИ І ФУНКЦІЇ

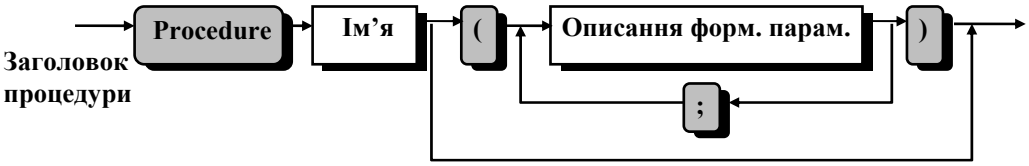
1. Опис процедур.

Технологія програмування мовою програмування типу мови Pascal пропонує проектувати програми методом послідовних уточнень.

На кожному етапі - кроку уточнення програміст розбиває задачу на деяке число підзадач, визначаючи тим самим деяку кількість окремих підпрограм. Концепція процедур (підпрограм) дозволяє виділити підзадачу як явну підпрограму.

У мові Паскаль процедури визначаються в розділі процедур і функцій за допомогою описань процедур. Звернення до процедури здійснюється оператором процедури.

Описання процедури таке ж, як і описання програми, але замість заголовка програми фігурує заголовок процедури. Заголовок має вид:



<u>Приклади заголовка процедури:</u>

procedure Picture;
procedure Power(X: real; n: Integer; var u, v: Real);
procedure Integral(a, b, epsilon: Real; var S: Real);

2.Формальні параметри. Локальні і глобальні об'єкти.

Як бачимо з прикладів, у розділі формальних параметрів перелічуються імена формальних параметрів, а потім вказується їхній тип. Таким чином, кожне описання формальних параметрів з точки зору синтаксису має такий же вигляд, як і описання змінних у розділі змінних. Перед деякими описаннями ставиться службове слово Var. Такі параметри називаються параметрами-змінними. Якщо перед описанням службове слово Var не стоїть, це - параметри-значення. Різницю між цими типами параметрів описано нижче.

Мітки, імена констант, типів, змінних, процедур і функцій, що описуються в тілі процедури, а також всі імена, що вводяться в розділи формальних параметрів є локальними в описанні процедури, яка називається областю дії для цих об'єктів. За межами цієї області вони не відомі.

Резервування пам'яті під локальні об'єкти здійснюється таким чином, що вона "захоплюється" процедурою при її виклику оператором і звільняється при виході з процедури. Такий механізм розподілу пам'яті називається динамічним. Динамічний розподіл дозволяє економити пам'ять, що адресується.

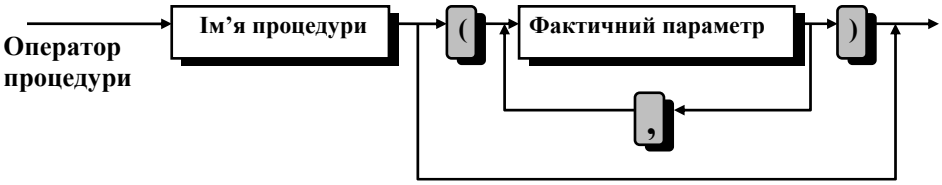
Об'єкти, що описані в основній програмі, доступні для використовування в цій процедурі. Вони називаються глобальними. Відмітимо, що локальний і глобальний об'єкти цілковито різні, але можуть мати одне й те ж ім'я. Якщо, наприклад, змінна X описана як в основній програмі, так і в процедурі, то в процедурі вона використовується як локальна. Якщо ж в процедурі змінна X не описана, то в процедурі вона використовується як глобальна.

3. Оператор процедури. Фактичні параметри.

Оператор процедури має вид:

< ім'я > або < ім'я > (< список фактичних параметрів >)

Синтаксична діаграма оператора процедури:



<u>Приклади операторів процедури:</u>

Picture

Power((a + b)/2, 3, degree, root)

Integral (0, P/2, 1E-6, SUMMA)

Зверніть увагу на відповідність між заголовком процедури і оператором процедури. Між списками формальних і фактичних параметрів встановлена взаємно-однозначна відповідність, що визначена їх місцями в списках. Ця відповідність ілюструється наступним прикладом:

Приклад 1. Розглянемо заголовок процедури і оператор цієї процедури:

```
Procedure Integral ( a, b, eps: real; var s: real );
   Integral ( -Pi/2, Pi/2, 1E-6, summa );
```

Відповідність:

Формальний параметр		Фактичний параметр
Значення	a	Вираз -Pi/2
Значення	b	Вираз Pi/2
Значення	eps	Дане 1E-6
Змінна	s	Змінна Summa

Як було вказано вище, параметри бувають 2-х видів: параметри-значення і параметри-змінні. Якщо перед описанням параметрів ніякого службового слова немає, мова йде про параметри-значення. Перед описанням параметрів-змінних ставиться службове слово var. При зверненні до процедури (в процесі виконання оператора процедури) формальним параметрам-значенням присвоюються значення відповідних фактичних параметрів, а замість імен формальних параметрів-змінних підставляються відповідні фактичні параметри - імена змінних, а потім виконується підпрограма, що описана процедурою.

Якщо x1, x2, ..., xn - фактичні параметри-змінні, відповідні формальним параметрам-змінним v1, ... , vn, то x1, x2, ..., xn повинні бути різними. Фактичними параметрами-значеннями можуть бути вирази або дані відповідних типів.

Розглянемо приклад:

Приклад 2. Програма обчислює координати точки (x_0, y_0) при послідовних поворотах і паралельних переносах системи координат.

```
Program Coordinates;
  Const Pi = 3.141592;
    Var Alfa, Beta : Real;
        x0, y0, x1, y1, x2, y2 : Real;
        x, y : Real;
 Procedure Rotate(x, y, Fi: Real; var u, v: Real );
       var cosFi, sinFi : Real; { локальні змінні }
      begin
        Fi := Fi*Pi/180 ;
        cosFi := Cos(Fi); sinFi := Sin(Fi);
```

```
        { параметри x, y захищені від глобальних змінних x, y }
         u := x * cosFi - y * sinFi ;
         v := x * sinFi + y * cosFi
        end ;
  Procedure Move(x, y, a, b : Real; var u, v: Real);
     begin
       u := x + a ; v := y + b
      end;
   begin
     Read (x0, y0);  Read (Alfa);  Rotate(x0, f0, alfa, x, y);
     Read (x1, y1);     Move(x, y, x1, y1, x, y);
     Read (Beta);       Rotate(x, y, Beta, x, y);
     Read ( x2, y2 );  Move(x, y, x2, y2, x, y);
     Writeln ('==============================');
     Writeln ('абсциса точки : ', x);
     Writeln ('ордината точки : ', y);
   end.
```

Параметри-значення використовуються для передачі даних в процедуру. Це значить, що для параметра-значення на час виконання процедури резервується пам'ять, розмір якої визначений типом параметра і яка заповнюється при виклику процедури. Таким чином, використання параметрів-значень при передачі даних великого об'єму може привести до неоправданих витрат часу процесора і пам'яті, що адресується.

Нехай, наприклад, змінна A типу Sequence - масив з 1000 дійсних чисел і Procedure MaxMin(X : Sequence; var Max, Min : Real) - пошук максимального і мінімального елементів масиву X. Тоді при зверненні до процедури MaxMin за допомогою оператора MaxMin(A, Sup, Inf) компілятор виділить пам'ять (6 - 10 байт на кожний елемент масиву - всього 6000 - 10000 байт) і здійснить 1000 циклів пересилання чисел з A в X. Якщо ж параметр X визначити як параметр-змінну: Procedure MaxMin(var X : Sequence; var Max, Min : Real) ні пам'яті, ні пересилань не знадобиться.

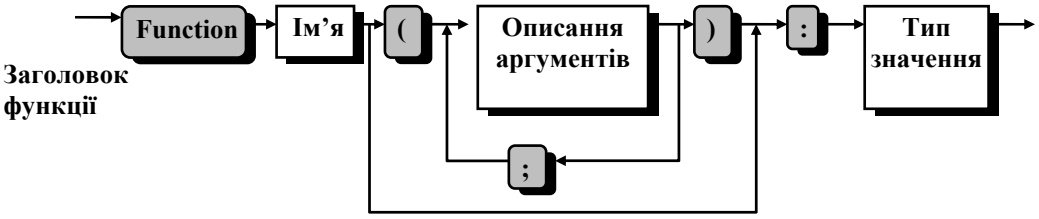
4. Функції.

Поряд із стандартними функціями, в мові можна визначити і інші необхідні програмі функції. Функція - це підпрограма, що визначає одне - єдине скалярне або засилкове значення, що використовується при обчисленні виразу. Описання функції має, по суті, такий самий вид, як і описання процедури. Різниця тільки у заголовку, який має вид:

Function < ім'я > : < тип результату > ;

або **Function < ім'я > (<список описань формальних параметрів >): < тип результату >;**

Синтаксична діаграма заголовка функції:



Зверніть увагу на описання результату, який визначає тип значення функції.

Таким чином, для функції визначені всі ті поняття, які були сформульовані для процедур.

Ім'я, що задане в заголовку функції, іменує цю функцію. В середині описання функції - в розділі операторів - повинно бути присвоювання, що виконується, в лівій частині якого стоїть ім'я функції, а в правій - вираз що має тип значення функції.

Приклади опису функцій.

Приклад 3. Функція GCD (алгоритм Евкліда) обчислює найбільший спільний дільник двох натуральних чисел x і y.

```
Function GCD (x, y : Integer) : Integer ;
  Begin
    While x <> y do
      If x < y
       then y := y - x
       else x := x - y ;
       GCD := x
   End;
```

Приклад 4. Функція IntPow підносить дійсне число x до степеня N. ($Y = x^N$)

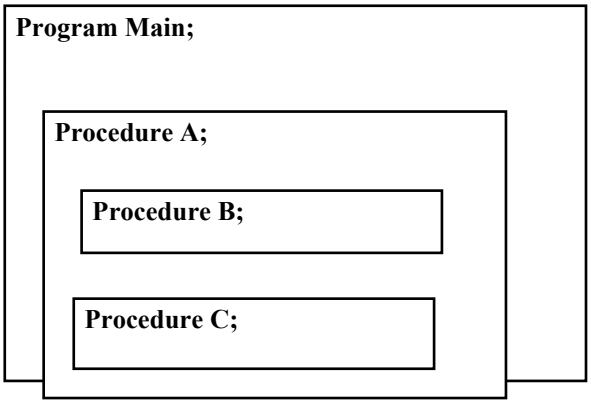
```
Function IntPow(x: Real; N: Integer) : Real;
    Var i: Integer;
  Begin
    IntPow := 1;
    For i:=1 to Abs(N) do IntPow := IntPow * x;
    If N < 0 then IntPow := 1/IntPow
   End;
```

Приклад 5. Програма обчислює найбільший спільний дільник послідовності натуральних чисел, яка представлена масивом.

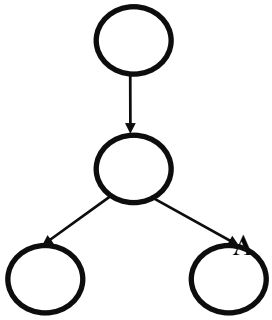
```
Program GCD_of_Array;
```

```
   Const n = 100 ;
    Var i, D : Integer;
        A : Array[1..n] of Integer;
Function GCD (x, y : Integer) : Integer ;
  Begin
   While x <> y do
    If x < y
     then y := y - x
     else x := x - y;
     GCD := x
   End;
Begin { основна програма }
{Процедура читання масиву натуральних чисел}
  D := GCD (A[1], A[2]);
  For i := 3 to n do D := GCD(D, A[i]);
  writeln ( ‘ НОД послідовності = ‘ , D )
 End.
```

Кожна процедура або функція може, в свою чергу, містити розділ процедур і функцій, в якому визначені одна або декілька процедур і функцій. В цьому випадку кажуть про вкладення процедур. Кількість рівнів вкладень може бути довільним. Структура вкладення ілюструється малюнком:



C



B

Поняття локальних і глобальних об’єктів поширюються і на вкладені процедури. Наприклад, змінна, описана в процедурі А локальна по відношенню до основної програми і глобальна для процедур В і С, вкладених в А.

В деяких випадках необхідно з процедури здійснити виклик іншої процедури, описаної в тому ж розділі процедур і функцій. Наприклад, процедура C може містити оператор виклику процедури B. В цьому випадку компілятор правильно обробить текст програми, оскільки процедура B описана до процедури C. Якщо ж з процедури B необхідно звернутись до C, для правильної обробки виклику C необхідно використовувати механізм так званого попереднього описання C. Описання, що опереджає процедури (функції) - це її заголовок, услід за яким через ";" стоїть службове слово Forward. У тексті програми описання, що опереджає, повинно передувати процедурі, в якій процедура, що попередньо описана, викликається.

Якщо процедура або функція описані попередньо, описанню її тіла передує скорочений заголовок, що складається тільки з відповідного службового слова і імені - без списку описань параметрів.

```
Procedure A (x : TypeX; Var y : TypeY); Forward;
Procedure B (z : TypeZ) ;
Begin
... A( p, q); ...
End;
Procedure A;
Begin
 ...
End;
```

5. Рекурсивно-визначені процедури і функції.

Описання процедури A, в розділі операторів якої використовується оператор цієї процедури, називається рекурсивним. Таким чином, рекурсивне описання має вид

```
Procedure A(u, v : ParType);
 ...
Begin
...; A(x, y); ...
End;
```

Аналогічно, описання функції F, в розділі операторів якої використовується виклик функції F, називається рекурсивним. Рекурсивне описання функції має вид

```
Function F(u, v : ArgType) : FunType;
 ...
Begin
...; z := g(F(x, y)); ...
 End;
```

Використання рекурсивного описання процедури (функції) приводить до рекурсивного виконання цієї процедури (обчисленню цієї функції). Задачі, що формулюються природнім чином як рекурсивні, часто приводять до рекурсивних розв'язків.

Приклад 6. Факторіал.

Розглянемо рекурсивне визначення функції n!=1*2*...*n (n-факторіал). Нехай F(n) = n! Тоді

1.F(0) = 1

2.F(n) = n*F(n - 1) при n > 0

Засобами мови це визначення можна сформулювати як обчислення:

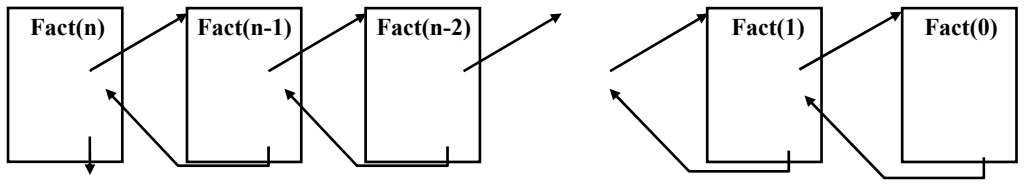
```
If n = 0
 then F := 1
 else F := F(n - 1) * n
```

Оформивши це обчислення як функцію і змінивши ім'я, отримаємо:

```
Function Fact(n: Integer): Integer;
  Begin
   If n = 0
    then Fact := 1
    else Fact := Fact(n - 1) * n
   End;
```

Обчислення функції Fact можна представити як ланцюжок викликів нових копій цієї функції з передачею нових значень аргументу і повернень значень функції в попередню копію.

n-1 n-2 n-3 1 0



n! (n-1)! (n-2)! 1 1

Ланцюжок викликів обривається при передачі нуля в нову копію функції. Рух у прямому напрямку (розгортання рекурсії) супроводжується тільки обчисленням умови і викликом. Значення функції обчислюється при згортанні ланцюжка викликів. Складність обчислення $T_{fact}(n)$ функції Fact можна оцінити, виписавши рекурентне співвідношення:

$T_{fact}(n) = T_{fact}(n-1) + T_m + T_l + T_c$

Для того, щоб обчислити Tfact(n), треба здійснити одну перевірку, одне множення і один виклик $T_{fact}(n-1)$. Виклик T_{fact} потребує затрат часу T_c на "адміністративні" обчислення: передачу параметра, запам'ятовування адреса повернень і т.п. Поклавши $C = T_m+T_l+T_c$, отримаємо $T_{fac}t(n) = T_{fact}(n-1) + C$. Неважко тепер показати, що $T_{fact}(n) = Cn$.

Приклад 7. Перестановки. Згенерувати всі перестановки елементів скінченої послідовності, що складається з букв.

Спробуємо звести задачу до декількох підзадач, більш простих, ніж вихідна.

Нехай $S = [s_1, s_2, ..., s_n]$ - набір символів.

Через Permut(S) позначимо множину всіх перестановок S, а через Permut(S,i) - множину всіх перестановок, в яких на останньому місці стоїть елемент s_i. Тоді

$Permut(S) = Permut(S,n) \cup Permut(S, n-1) \cup ... \cup Permut(S,1)$

Елемент множини Permut(S, i) має вид $[s_{j2},..., s_{jn},s_i]$ де j2,..., jn - всі можливі перестановки індексів, не рівних I. Тому Permut(S, i) = (Permut(S \ s_i), s_i) і $Permut(S)=(Permut(S \setminus s_1),s_1)+...+ (Permut(S \setminus s_n), s_n)$.

Отримане співвідношення виражає множину Permut(S) через множини перестановок наборів з (n-1) символу. Доповнивши це співвідношення визначенням Permut(S) на одноелементній множині, отримаємо:

1. Permut({s}) = {s}
2. $Permut(S) = (Permut(S\setminus s_1), s_1) + ... + (Permut(S\setminus s_n), s_n)$

Уточнимо алгоритм, опираючись на представлення набору S в виді масиву S[1..n] of char.

По перше, визначимо параметри процедури Permut:

k - кількість елементів в наборі символів;

S - набір символів, що переставляються.

Алгоритм починає роботу на вхідному наборі і генерує всі його перестановки, що залишають на місці елемент s[k]. Якщо множина перестановок, в яких на останньому місці стоїть s[j], уже породжена, міняємо місцями s[j-1] і s[k], виводимо на друк отриманий набір і застосовуємо алгоритм до цього набору. Параметр k керує рекурсивними обчисленнями: ланцюжок викликів процедури Permut обривається при k = 1.

```
Procedure Permut(k : Integer; S : Sequence);
Var j : integer;
   Begin
     if k <> 1 then Permut(k - 1, S);
     For j := k - 1 downto 1 do begin
     Buf := S[j]; S[j] := S[k]; S[k] := Buf;
     WriteSequence(S); Permut(k - 1, S)
     end
   End;
   Begin { Розділ операторів програми}
   {Генерація вихідного набору S}
```

```
    WriteSequence(S); {Виведення першого набору на друк}
     Permut(n, S)
   End.
```

Оцінимо складність алгоритму за часом у термінах C(n): Кожний виклик процедури Permut(k) містить k викликів процедури Permut(k-1) і 3(k-1) пересилання. Кожний виклик Permut(k-1) супроводжується передачею масиву S як параметра-значення, що за часом еквівалентне n пересиланням. Тому мають місце співвідношення

$$C(k) = kC(k-1) + nk + 3(k-1),\ C(1) = 0,\ \text{звідки}\ C(n) = (n+3)n!$$

Оцінимо тепер розмір пам'яті, необхідної для алгоритму. Оскільки S - параметр-значення, при кожному виклику Permut резервується n комірок (байтів) для S, а при виході з цієї процедури пам'ять звільнюється. Рекурсивне застосування Permut призводить до того, що ланцюжок вкладених викликів має максимальну довжину (n - 1):

Permut(n) --> Permut(n-1) --> ... --> Permut(2) --> Permut(1)

Тому дані цього ланцюжка потребують n^2 - n комірок пам'яті, тобто алгоритм потребує пам'ять розміром $O(n^2)$. Кількість перестановок - елементів множини Permut(S) дорівнює n!. Тому наш алгоритм "витрачає" C(n)/n! = O(n) дій для породження кожної перестановки. Зрозуміло, такий алгоритм неможна називати ефективним. (Інтуїція нам підказує, що ефективний алгоритм повинен генерувати кожну перестановку за фіксовану, незалежну від n кількість дій.) Джерело неефективності очевидне: використання S як параметра-значення. Це дозволило нам зберегти незмінним масив S при поверненні з рекурсивного виклику для того, щоб правильно переставити елементи, готуючи наступний виклик.

Розглянемо інший варіант цього алгоритму. Використаємо S як глобальну змінну. Тоді при виклику Permut S буде змінюватись. Отже, при виході з рекурсії масив S треба поновлювати, використовуючи обернену перестановку!

```
Program All_permutations;
  Const n = 4;
   Type Sequence = array[1..n] of char;
    Var S : Sequence;
        Buf : char;
          i : Integer;
Procedure WriteSequence;
     begin
      For i := 1 to n do Write(S[i]); Writeln
      end;
Procedure Permut(k : Integer);
      Var j : integer;
Procedure Swap(i, j : Integer);
     begin  Buf := S[j]; S[j] := S[i]; S[i] := Buf  end;
    Begin
      if k > 1 then Permut(k - 1);
```

```
      For j := k - 1 downto 1 do begin
       Swap(j, k); {Пряма перестановка}
       WriteSequence; Permut(k - 1);
       Swap(k, j) {Обернена перестановка}
      end
    End;
  Begin
    {Генерація вихідного набору S}
    WriteSequence;  Permut(n)
   End.
```

Тепер оцінка C(n) виходить з співвідношень

$$C(k) = kC(k-1) + 3(k-1),\ C(1) = 0 \text{ , т.е. } C(n) = O(n!)$$

Цікаво, що цей варіант не потребує і пам'яті розміру $O(n^2)$ для збереження масивів. Необхідна тільки пам'ять розміру O(n) для збереження значення параметра k.

6.Приклади рекурсивних описів процедур і функцій.

Декілька прикладів рекурсивних процедур, розглянутих у цьому пункті, допоможуть кращому засвоєнню техніки застосування рекурсії. Ми також побачимо переваги і недоліки рекурсивних описань.

Приклад 8. Повернені послідовності.

Розглянемо послідовність V(n), що задана рекурентно:

1.$V(0) = V_0$, $V(1) = V_1$, ..., $V(k) = V_k$;

2.V(n+k) = F(V(n+k-1), ... V(n+1), V(n), n)

Таке визначення задає послідовність V(n) фіксуванням перших її декількох членів і функції F, що обчислює кожний наступний член, виходячи з попередніх. Рекурентне визначення "дослівно переводиться" в функцію, що обчислюється рекурсивно:

```
Function V(n: Integer) : Integer;
  Begin
   Case n of
     0:  V := V0;
      . . . . .
     k:  V := Vk
    else V := F(V(n+k-1), ... V(n+1), V(n), n)
   end
End;
```

Один з прикладів повернених послідовностей - послідовність Фібоначчі, програму обчислення якої ми вже розглядали. Ось її рекурсивна версія:

```
Function RF(n: Integer) : Integer;
  Begin
   Case n of
```

```
  0,1: RF := 1
  else RF := RF(n-1) + RF(n-2)
 end
End;
```

Нажаль, ця красива і прозора версія вкрай неефективна: її складність визначається з співвідношення $T_{rf}(n) = T_{rf}(n-1) + T_{rf}(n-2) + 1$.

Функція $T_{rf}(n)$ дорівнює функції RF(n). Неважко показати, що $T_{rf}(n) = RF(n) > 2^{n-2}$ при n>5.

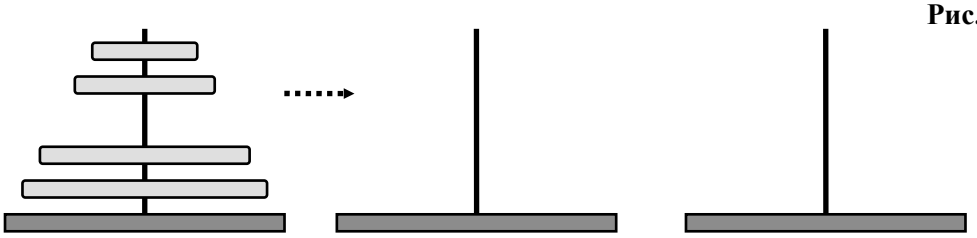
Таким чином, складність обчислення RF(n) експоненціальна. Складність же ітеративного алгоритму обчислення Fib лінійна. У загальному випадку ситуація точно така ж: можна побудувати ітеративний алгоритм обчислення поверненої послідовності лінійної складності, рекурсивна ж версія при k>=1 має експоненціальну складність.

Приклад 9. Ханойські башти.

Класичний приклад застосування рекурсії для описання ефективного алгоритму - задача про ханойські башти. На перший з трьох стержнів, закріплених вертикально на підставці, насаджено декілька кілець різних діаметрів так, що кільце меншого діаметра лежить на кільці великого діаметра. Два інших стержня порожні (рис. А).

У задачі треба переставити всі кільця з 1-го стержня на 2-ий за декілька кроків. Кожний крок - це перестановка верхнього кільця одного стержня на верхнє кільце іншого стержня з дотриманням правила: діаметр кільця, що переставляється повинен бути менше, ніж діаметр кільця, на яке здійснюється перестановка.

Рис. А



1 стержень **2 стержень** **3 стержень**

Нехай N - кількість кілець на стержні, I - номер кільця, з якого здійснюється перестановка і J - номер кільця, на який кільця треба переставити. Змінні N,I,J - параметри процедури HanojTower, керуючої перестановками. Крок розв'язку - процедура Step з параметрами I, J.

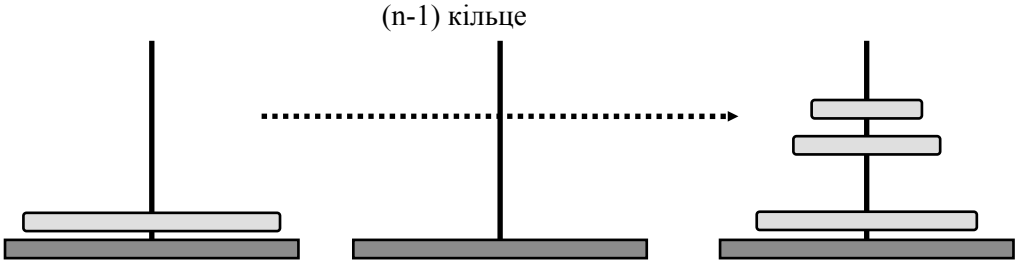
HanojTower (N,I,J) - процедура, що переставляє N кілець з I-того стержня на J-тий.

Step (I, J) - процедура , що переставляє 1-не кільце з I-того стержня на J-тий.

Відмітимо, що якщо I і J - номера 2-стержней, то 6-I-J - номер третього стержня.

Припустимо, що ми перемістили N-1 кільце з I-того стержня на 6-I-J стержень. (рис. B)

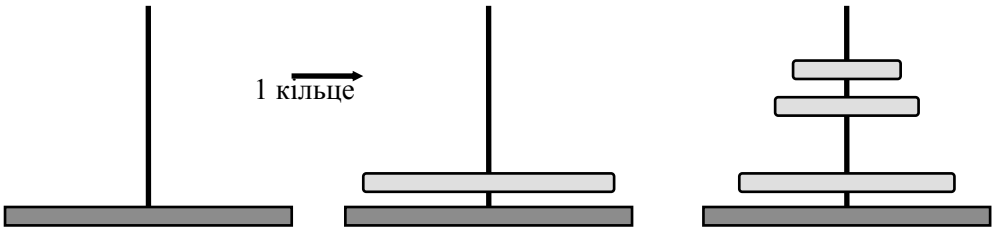
Рис. B



I стержень **J стержень** **6-I-J стержень**

Тоді можна перемістити кільце з стержня I на J. (рис. C)

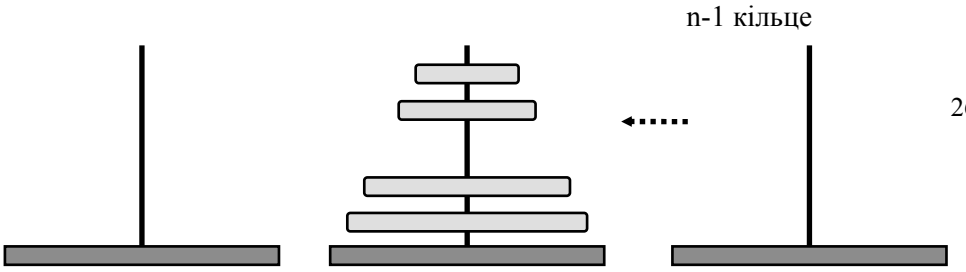
Рис. C



I стержень **J стержень** **6-I-J стержень**

Відмітимо тепер, що на стержні J лежить кільце з найбільшим діаметром, тобто цей стержень можна використовувати без порушення обмежень, пов'язаних з величинами діаметрів. Тому можна тепер переставити всю піраміду з N-1 кільця з стержня 6-I-J на J, (рис. D) і задача розв'язана!

Рис. D



I стержень **J стержень** **6-I-J стержень**

Відмітивши, що при N = 1 задача розв'язується за допомогою процедури Step (I, J), опишемо процедуру HanojTower (N,I,J):

```
Procedure HanojTower(N, I, J: Integer);
  Begin
    If N = 1
     then Step(I, J)
     else begin
      HanojTower(N-1, I, 6-I-J);
      Step(I, J);
      HanojTower(N-1, 6-I-J, J)
     end
    End;
```

Процедуру Step(I, J) можна реалізувати, використовуючи представлення даних у масиві Rings[1..N, 1..3] і графічну візуалізацію переміщення кілець.

Визначимо складність алгоритму за часом C(n) у термінах кількості кроків (викликів Step), вписавши рекурентне співвідношення: $C(n) = 2C(n-1) + 1$

Легко тепер показати, що $C(n) = 2^n - 1$. Доведено, що ця кількість кроків є мінімально можливою, тому наш алгоритм оптимальний.

Приклад 10. Лінійні діафантові рівняння. Перерахувати всі невід'ємні цілі розв'язки лінійного рівняння $a_1x_1 + a_2x_2 + ... + a_nx_n = b$ з цілими додатними коефіцієнтами.

Як і в попередніх прикладах, опишемо алгоритм рекурсивно, здійснивши зведення вихідної задачі до задачі меншого розміру.

Перепишемо вихідне рівняння в виді: $a_1 x_1 + a_2 x_2 + ... + a_{n-1} x_{n-1} = b - a_n x_n$

Організуємо перебор всіляких значень x_n, при яких права частина $b - a_n x_n > 0$. $x_n = 0, 1, ... y$, де $y = b \text{ div } a_n$. Тоді перші n-1 компонент розв'язка $(x_1, ..., x_{n-1}, x_n)$ вихідного рівняння - розв'язок рівняння $a_1 x_1 + a_2 x_2 + ... + a_{n-1} x_{n-1} = b - a_n x_n$.

Таким чином ми звели розв'язок вихідного рівняння до розв'язка y+1 рівняння з n-1 невідомим, і, отже, можемо реалізувати алгоритм рекурсивно. Визначимо умови виходу з рекурсії:

при b = 0 існує єдиний розв'язок - (0, 0, ..., 0)

при n = 1 якщо b mod a_1 = 0 то x_1 = b div a_1 інакше розв'язків немає.

Таким чином, виходити з рекурсивних обчислень треба в двох (крайніх) випадках. Ми встановили і параметри процедури: n і b.

```
Procedure Solution(n, b : integer);
    Var i, j, y, z : Integer;
  Begin
   If  b = 0
    then begin
     For j := 1 to n do X[j] := 0; WriteSolution end
     else If (n = 1) and (b mod a[1] = 0)
       then begin X[1]:= b div a[1]; WriteSolution end
      else If n > 1
        then begin
          z := a[n]; y := b div z;
          For i := 0 to y do begin
          X[n] := i; Solution(n - 1, b - z*i)
        end
    end
 End;
 Program AllSolutions;
    Const n = 4;
     Type Vector = array[1..n] of Integer;
      Var a, X : Vector; b : Integer;  i : Integer;
   {Procedure WriteSolution друкує  розв'язок X[1..n] }
   {Procedure Solution}
   Begin
    {Введення масиву коефіцієнтів a[1..n] і св. члена b}
    Solution(n, b)
   End;
```

Приклад 11. Підведення числа до натурального степеня. Піднести дійсне число a у натуральний степінь n.

Раніше ця задача була розв'язана методом послідовного домноження результату на a. Однак її можна розв'язати ефективніше, якщо застосувати метод половинного ділення степеня n. Саме: $a^n = (a^{n/2})^2$. Оскільки число n не обов'язково парне, формулу треба уточнити:

a^n = $(a^{n \text{ div } 2})^2$ * $a^{n \text{ mod } 2}$. Доповнивши визначення a^n визначенням a^1 = a і замінивши домноження на $a^{n \bmod 2}$ розбором випадків парний-непарний, отримаємо:

```
Function CardPower(a : Real; n : Integer) : Real;
var b : Real;
```

```
Begin
 If n = 1
  then CardPower := a
  else begin
    b := Sqr(CardPower(a, n div 2));
    If n mod 2 = 0
     then CardPower := b
     else CardPower := a*b
  end
End;
```

Доведіть, що функція CardPower використовує не більш $O(\log_2 n)$ множень і піднесень у квадрат.

Зрозуміло, функцію CardPower можна реалізувати без циклів і рекурсій, за допомогою формули: при $a > 0$ $a^n = e^{n * \ln a}$, але цей метод не придатний наприклад, якщо в натуральний степінь підноситься багаточлен.

Приклад 12. Перетин зростаючих послідовностей.

Нехай $A = [a_1, a_2,...,a_n]$ і $B = [b_1, b_2, ... , b_m]$ - дві зростаючі числові послідовності. Перетином цих послідовностей називається зростаюча послідовність $C = [c_1, c_2,..., c_k]$, що складається з тих і тільки тих чисел, які належать обом послідовностям. Треба знайти $C = A \cap B$.

Зведемо задачу до кількох підзадач, більш простих, ніж вихідна. Поділимо для цього вихідні послідовності на частини

$A = [a_1] \cup A_2$; $B = [b_1] \cup B_2$, де $A_2 = [a_2,...,a_n]$, $B_2 = [b_2,...,b_n]$

Тоді $A \cap B = (a_1 \cup A_2) \cap (b_1 + B_2) = a_1 \cap b_1 \cup a_1 \cap B_2 \cup A_2 \cap b_1 \cup A_2 \cap B_2$

(Дужки в позначеннях одноелементних множин опущені)

Так як послідовності A і B зростають, маємо:

Якщо $a_1 = b_1$ то $A \cap B = a1 \cap b1 \cup A2 \cap B2 = a1 \cup A2 \cap B2$

Якщо $a1 < b1$ то $A \cap B = b1 \cap A2 \cup A2 \cap B2 = A2 \cap B$

Якщо $a1 > b1$ то $A \cap B = a1 \cap B2 \cup A2 \cap B2 = A \cap B2$

Ці відношення зводять вихідну задачу до задач менших розмірів, тому їх можна використовувати для рекурсивного описання обчислень.

Процедура Intersect пошуку перетину залежать від параметрів і та j - номерів початкових елементів підпослідовностей A[i..m] і B[j..n], представлених масивами A[1..m] і B[1..n]. Знайдений елемент перетину роздруковується.

```
Procedure Intersect(i, j : Integer);
  Begin
   If (i <= m) and (j <= n)
    then If a[i] = b[j]
     then begin Write(a[i]); Intersect(i+1, j+1) end
```

```
  else If a[i] < b[j]
    then Intersect(i+1, j)
    else Intersect(i, j+1)
End;
```

7. Переваги і недоліки рекурсивних алгоритмів.

Розглянуті приклади показують, що застосування рекурсивних означень само по собі не приводить до хороших розв'язків задач. Так, задачі прикладів 6, 8, 12 допускають більш ефективні і достатньо прості ітеративні розв'язки. У прикладі 12 це продемонстровано особливо наочно. У прикладах 6 і 8 рекурсія моделює простий цикл, який витрачає час на обчислення, що зв'язані з керуванням рекурсивними викликами і пам'ять на динамічне збереження даних. Те ж саме можна сказати і про приклад 11. Ми вже розглядали ітеративну версію метода ділення навпіл в інших прикладах.

Безсумнівним достоїнством алгоритмів з прикладів 8, 11, 12 є їх простота і обгрунтованість. При програмуванні ми по-суті одночасно обгрунтовували правильність алгоритму. У прикладах 7, 9,10 розглядались комбінаторні задачі, розв'язки яких не зводяться до простого застосування циклу, а потребують перебору варіантів, природнім чином, що керується рекурсивно. Ітеративні версії розв'язків цих задач складніші по управлінню. Можна сказати, що основна обчислювальна складність комбінаторної задачі полягає не в реалізації алгоритму її розв'язку, а в самому методі розв'язку.

Пізніше ми побачимо, як рекурсія в поєднанні з іншими методами використовується для побудови ефективних програм.

8.Задачі і вправи.

<u>Вправа І.</u>

9.Опишіть процедури додавання і множення n*n матриць, що складаються з дійсних чисел. За допомогою цих процедур складіть програму обчислення матриці B =AX + XA + E за даними матрицями A і X.

10.Опишіть процедуру пошуку найбільшого і найменшого елементів у масиві. За допомогою цієї процедури складіть програму, що визначає, чи співпадають найбільший і найменший елементи двох масивів A [1..n] і B [1..n].

11.Опишіть процедури пошуку середнього за величиною елемента у масиві A [1..2n+1] і середнє арифметичне елементів цього масиву. За допомогою цих процедур складіть програму, яка порівнює середнє за величиною і середнє арифметичне у масиві A [1..99].

12.Опишіть процедуру пошуку найбільшого і найменшого елементів у масиві A[1..n]. За допомогою цієї процедури складіть програму, що визначає координати вершин найменшого прямокутника зі сторонами, паралельними осям координат, який містить в собі множину точок $T_1(X_1,Y_1)$, ... , $T_n(X_n,Y_n)$.

13.Опишіть процедуру перевірки розкладення натурального числа в суму двох квадратів. Складіть програму, яка вибирає з даного масиву ті і тільки ті числа, які розкладаються в суму двох квадратів.

14.Опишіть функцію перевірки простоти числа. Опишіть функцію перевірки числа на розкладність в суму виду $1+2^a$. Складіть програму, яка вибирає з даного масиву чисел ті і тільки ті його елементи, які задовольняють обом властивостям.

15.Опишіть процедуру перетворення цілого числа з десяткової системи в р-ічну систему числення. Опишіть процедуру перетворення правильного десяткового дробу в р-ічний з заданою кількістю розрядів. Складіть програму перетворення довільного дійсного числа з 10-вої в р-ічну систему числення.

16.Опишіть процедуру пошуку найменшого елемента в рядку матриці. Опишіть функцію перевірки на максимальність даного елемента в стовпці матриці. Складіть програму пошуку сідлової точки матриці.

Вправи II.

1.Опишіть функцію f(x) - найбільший власний дільник числа x. Складіть програму пошуку всіх власних дільників числа N.

2.Опишіть функцію y = arcsin(x). Складіть програму розв'язку рівняння sin(ax+b) = c.

3.Опишіть функцію f(x) = max (t*sin(t)).

$$t \in [0,x]$$

Складіть Програму табулювання функції f(x) на відрізку [a,b].

4.Опишіть функцію f(x) - число, яке отримується з натурального числа x перестановкою цифр у зворотному порядку. Складіть програму, яка роздруковує симетричні числа з масиву натуральних чисел.

5.Опишіть функції L(x,y) і Φ(x,y), які визначені формулами $L(x,y) = |x + iy|$, $\Phi(x, y) = \arg(x + iy)$. Складіть програму переведення послідовності комплексних чисел $z_1, ..., z_n$, $z_j = x_j + iy_j$, в тригонометричну форму.

Вправи III.

1.Реалізуйте алгебру раціональних чисел як бібліотеку процедур і функцій, що містить арифметичні дії і відношення порядку на множині раціональних чисел.

2.Реалізуйте алгебру комплексних чисел як бібліотеку процедур і функцій, що містить арифметичні дії над комплексними числами в алгебраїчній формі.

3.Реалізуйте алгебру - скінчене поле GF(p) лишків за простим модулем p як бібліотеку процедур і функцій, що містить арифметичні дії над лишками. GF(p) = (0, 1, ..., p-1)

4.Реалізуйте алгебру 3 x 3 матриць над полем дійсних чисел як бібліотеку процедур і функцій, що містить додавання, віднімання, множення матриць, обертання невироджених матриць, ділення матриці на невироджену матрицю, множення матриці на число, обчислення визначника матриці.

Задачі.

1.Перечислити всі вибірки по K елементів у масиві A з N елементів. Масив містить перші N натуральних чисел: A[i] = i. Знайти їх кількість і оцінити складність алгоритму.

2.Перечислити всі підмножини множини (масиву A) з задачі 1. Знайти їх кількість і оцінити складність алгоритму.

3.Дано масив A[1..n] цілих додатних чисел і ціле число b. Перечислити всі набори індексів j1, j2, ..., jk такі, що A[j1] + A[j2] + ... + A[jk] = b. Оцінити складність алгоритму.

4.Дано масив A[1..n] цілих чисел і ціле число b. Перечислити всі набори індексів j1, j2, ... , jk такі, що A[j1] + A[j2] + ... + A[jk] = b. Оцінити складність алгоритму.

5.Відома в теорії алгоритмів функція Акермана A(x, y) визначена на множні натуральних чисел рекурсивно:

A(0, y) = y + 1,

A(x, 0) = A(x - 1, 1),

A(x, y) = A(x - 1, A(x, y - 1));

а)Опишіть функції A(1, y), A(2, y), A(3, y), A(4, y) явним чином;

б)Реалізуйте рекурсивну програму обчислення A(x, y);

в) Реалізуйте програму обчислення A(x, y) без рекурсії.

ШВИДКІ АЛГОРИТМИ СОРТУВАННЯ І ПОШУКУ.

У теперішньому параграфі ми встановимо нижню грань обчислювальної складності задачі сортування масиву і завершимо вивчення алгоритмів сортування і пошуку в масивах, розглянув так звані швидкі сортування.

1. Нижня оцінка часу задачі сортування масиву за числом порівнянь.

Теорема. Будь-який алгоритм сортування масиву, що використовує тільки порівняння і перестановки, у гіршому випадку потребує мінімум $O(n \log_2 n)$ порівнянь.

Доведення. Нехай $a_1, a_2, ..., a_n$ - підмножина елементів, вибраних з абстрактної множини U, на якій визначено відношення лінійного порядку. (Це означає, що будь-які два різних елемента a і b з U порівняльні: a < b або b < a.) Ця підмножина породжує n! перестановок, і кожна перестановка може бути вихідною послідовністю (входом) для застосування алгоритму сортування. Іншими словами, задача сортування розміру n має n! різних входів. Позначимо цю множину входів через Per(A). Наприклад, при n = 3 підмножина {a, b, c} породжує 3! = 6 різних входів: [a b c], [a c b], [b a c], [b c a], [c a b], [c b a].

Оскільки множина U абстрактна, алгоритм сортування здійснює перестановки, що сортують вхід B, виходячи тільки з перевірки умов, що містять порівняння двох елементів A, прямо або опосередковано вибраних з масиву B по їх індексах. Нехай S - деяка підмножина з Per(A).

Умова, що містить порівняння a[i] ? a[j], на одній частині множини S дасть результат - True, для іншої - False. Нехай T(S), F(S) - ці підмножини. Тоді |S| = |T(S)| + |F(S)|. Отже, кількість елементів принаймні в одній з підмножин T(S), F(S) не менше |S|/2. Нехай C(1,B), C(2,B), ..., C(k, B) - послідовність значень умов, що перевіряються алгоритмом послідовно на вході B довжини k і ConSeq(k, B) - множина входів, які генерують ту ж послідовність умов. Тоді

$\cup$(ConSeq(k, B), B $\in$Per(A)) = Per(A)

і, отже, існує такий вхід B, для якого

$|ConSeq(k, B)| \geq |Per(A)|/2^k$ (*)

Нехай B і B' - два (різних) входи і P - деяка перестановка. Легко показати, що з $B \neq B'$ слідує $P(B) \neq P(B')$. Тому послідовності перестановок, що сортирують різні входи, будуть різними. Але кожна послідовність перестановок однозначно визначена відповідною послідовністю умов. Отже, кожному входу B відповідає своя послідовність значень умов C(1,B), C(2,B), ..., C(m, B),що перевіряється в процесі виконання програми сортування на цьому вході. Таким чином,

ConSeq(m, B) = {B}. (**)

З (*), (**) випливає існування такого входу B, для якого виконана нерівність $|Per(A)|/2^m \leq 1$, тобто $n!/2^m \leq 1$ або $m \geq \log(n!)$. За формулою Стірлінга

$\log(n!) \approx \log_2((n/e)^n) = n(\log_2 n - 1) = O(n \log_2 n)$.

Ми показали, що для будь-якого алгоритму сортування за допомогою порівнянь і перестановок існує такий вхід, на якому $m >= O(n \log_2 n)$

Таким чином, функція складності в гіршому випадку алгоритму сортування C(n) обмежена знизу величиною $O(n \log_2 n)$. Кінець доведення.

Спам'ятаймо, що алгоритм сортування вставками має таку ж саму за порядком величини функцію C(n). Таким чином, ця оцінка може бути досягнена.

Питання про оцінку функції кількості перестановок M(n) залишилося відкритим. Можна припустити, що кожній перестановці передує деяке порівняння. Тоді загальна кількість перестановок, що здійснюється алгоритмом, не перевищує C(n) і, отже, ефективні алгоритми сортування повинні мати складність $O(n \log_2 n)$ як за порівняннями, так і по перестановках. Нижче ми розглянемо деякі такі алгоритми.

2. Швидкі алгоритми сортування: Сортування деревом.

Алгоритм сортування деревом TreeSort по суті є поліпшенням алгоритму сортування вибором. Процедура вибору найменшого елементу вдосконалена як процедура побудови т.н. дерева, що сортує. Дерево, що сортує - це структура даних, у якій представлений процес пошуку найменшого елемента методом попарного порівняння елементів, що стоять поряд. Алгоритм сортує масив у два етапи.

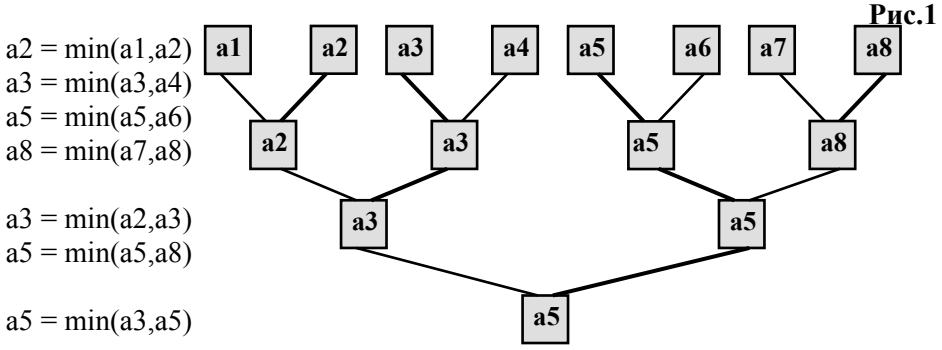
I етап: побудова дерева, що сортує;

II етап: просівання елементів по дереву, що сортується.

Розглянемо приклад:

Нехай масив A складається з 8 елементів (мал. 1, 1-ий рядок). Другий рядок складається з мінімумів елементів першого рядка, що стоять поряд. Кожний наступний рядок складається з мінімумів елементів попереднього, що стоять поряд.

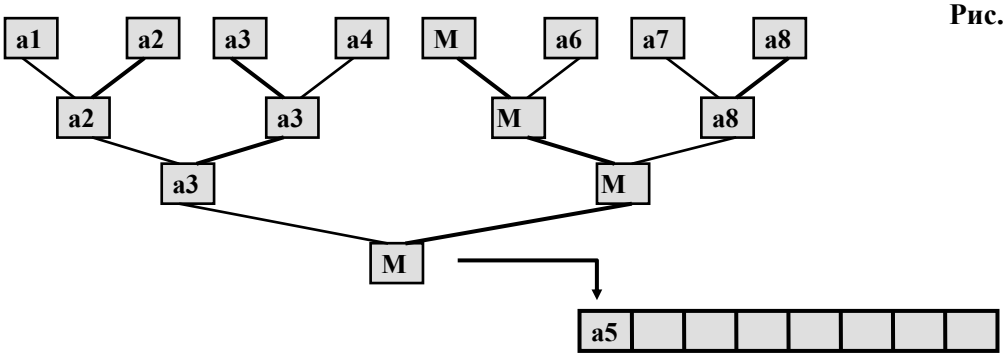
Рис.1



Ця структура даних називається сортуючим деревом. У корені дерева, що сортує, знаходиться найменший елемент. Крім цього, в дереві побудовані шляхи елементів масиву від листя до відповідного за величиною елемента вузла - розгалуження. (На рис. 1 шлях мінімального елемента a5 - від листа a5 до кореня відмічений товстою лінією.)

Коли дерево побудовано, починається етап просівання елементів масиву по дереву. Мінімальний елемент пересилається у вихідний масив B і всі входження цього елемента в дереві замінюються на спеціальний символ M.

Рис. 2



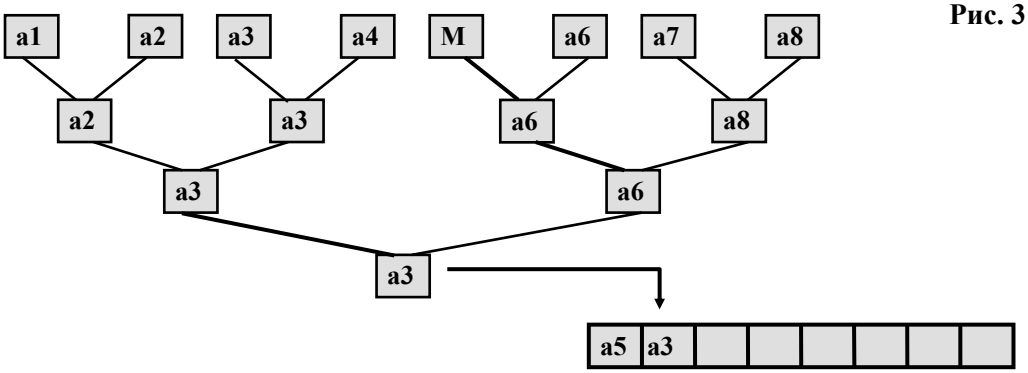
B

1 2 ...

8

Потім здійснюється просівання елемента уздовж шляху, відміченого символом M, починаючи з листка, сусіднього з M (На рис. 2 зверху-вниз) і до кореня. Крок просівання - це вибір найменшого з двох елементів, що зустрілись на шляху до кореня дерева і його пересилання у вузол, відмічений M. Просівання 2-го елемента показано на мал. 3 (Символ M більше, ніж будь-який елемент масиву.)

Рис. 3



a6 = min(M, a6)
a6 = min(a6, a8)
a3 = min(a3, a6) B
b2 := a3 1 2 ...
8

Просівання елементів відбувається до тих пір, поки весь вихідний масив не буде заповнений символами M, тобто n разів:

For I := 1 to n do begin
Відмітити шлях від кореня до листка символом M;
Просіяти елемент уздовж відміченого шляху;
B[I] := корінь дерева
end;

Обгрунтування правильності алгоритму очевидне, оскільки кожне чергове просівання викидає у масив B найменший з елементів масиву A, що залишився.

Дерево, що сортує можна реалізувати, використовуючи або двомірний масив, або одномірний масив ST[1..N], де N = 2n-1 (див. наступний розділ). Оцінимо складність алгоритму в термінах M(n), C(n). Перш за все відмітимо, що алгоритм TreeSort працює однаково на всіх входах, так що його складність у гіршому випадку співпадає зі складністю в середньому.

Припустимо, що n - степінь 2 ($n = 2^l$). Тоді дерево, сортує має l + 1 рівень (глибину l). Побудова рівня I потребує $n / 2^I$ рівнянь і пересилань. Таким чином, I-ий етап має складність

$C_1(n) = n/2 + n/4 + ... + 2 + 1 = n - 1$, $M_1(n) = C_1(n) = n - 1$

Для того, щоб оцінити складність II-го етапу $C_2(n)$ і $M_2(n)$ відмітимо, що кожний шлях просівання має довжину l, тому кількість порівнянь пересилань при просіванні одного елемента пропорційна l. Таким чином, $M_2(n) = O(l\ n)$, $C_2(n) = O(l\ n)$.

Оскільки $l = \log_2 n$, $M_2(n)=O(n \log_2 n))$, $C_2(n)=O(n \log_2 n)$, але $C(n) = C_1(n) + C_2(n)$, $M(n) = M_1(n) + M_2(n)$. Так як $C_1(n) < C_2(n)$, $M_1(n) < M_2(n)$, остаточно отримаємо оцінки складності алгоритму TreeSort за часом:

$M(n) = O(n \log_2 n)$, $C(n) = O(n \log_2 n)$,

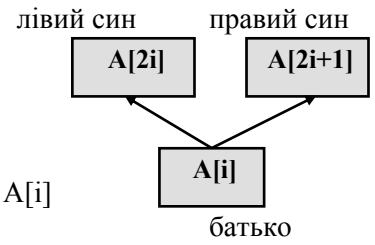
У загальному випадку, коли n не є степенем 2, дерево, що сортує, будується дещо по іншому. “Зайвий” елемент (елемент, для якого немає пари) переноситься на наступний рівень. Легко бачити, що при цьому глибина дерева, що сортує, дорівнює $[\log_2 n] + 1$. Удосконалення алгоритму II етапу очевидне. Оцінки при цьому міняють лише мультиплікативні множники. Алгоритм TreeSort має суттєвий недолік: для нього потрібна додаткова пам’ять розміром 2n - 1.

Пірамідальне сортування.

Алгоритм пірамідального сортування HeapSort також використовує представлення масиву у виді дерева. Цей алгоритм не потребує додаткових масивів; він сортує "на місці". Розглянемо спочатку метод представлення масиву у виді дерева:

Нехай A [1 .. n] - деякий масив. Співставимо йому дерево, використовуючи наступні правила:

1.A[1] - корінь дерева ;
2.Якщо A[i] - вузол дерева і $2i \le n$,
то A[2*i] - вузол - "лівий син" вузла A[i]
3.Якщо A[i] - вузол дерева і $2i + 1 \le n$,
то A[2*i+1] - вузол - "правий син" вузла A[i]

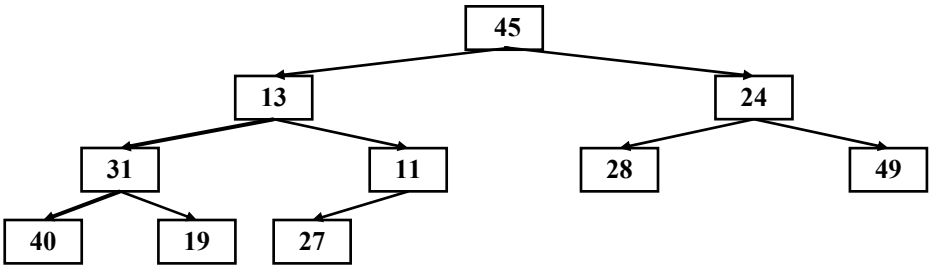


Правила 1 - 3 визначають у масиві структуру дерева, причому глибина дерева не перевищує $[\log_2 n] + 1$. Вони ж задають спосіб руху по дереву від кореня до листя. Рух уверх задається правилом 4:

4.Якщо A[i] - вузол дерева і i > 1
то A[i mod 2] - вузол - "батько" вузла A[i];

Приклад1. Нехай A = [45 13 24 31 11 28 49 40 19 27] - масив. Відповідне йому дерево має вид:

Рис. 5



Зверніть увагу на те, що всі рівні дерева, за винятком останнього, повністю заповнені, останній рівень заповнений зліва і індексація елементів масиву здійснюється зверху-вниз і зліва-направо. Дерево впорядкованого масиву задовольняє наступним властивостям:

$$A[i] \le A[2*i],\ A[i] \le A[2*i+1],\ A[2*i] \le A[2*i+1].$$

Як це не дивно, алгоритм HeapSort спочатку будує дерево, яке задовольняє прямо протилежним співвідношенням

$$A[i] \ge A[2*i],\ A[i] \ge A[2*i+1] \qquad (6)$$

а потім міняє місцями A[1] (найбільший елемент) і A[n].

Як і TreeSort, алгоритм HeapSort працює в два етапи:

I.Побудова дерева, що сортує;

I I.Просівання елементів по дереву, що сортує.

Дерево, що представляє масив, називається сортуючим, якщо виконуються умови (6). Якщо для деякого і ця умова не виконується, будемо говорити, що має місце (сімейний) конфлікт у трикутнику і (мал.4).

І на І-ому, і на ІІ-ому етапах елементарна дія алгоритму полягає у розв'язку сімейного конфлікту: якщо найбільший з синів більше, ніж батько, то переставляються батько і його син (процедура ConSwap).

У результаті перестановки може виникнути новий конфлікт в тому трикутнику, куди переставлений батько. Таким чином можна говорити про конфлікт (роду) у піддереві з коренем у вершині і. Конфлікт роду розв'язується послідовним розв'язком сімейних конфліктів - проходом по дереву зверху-вниз. (На мал.5 шлях розв'язку конфлікту роду у вершині 2 відмічений.) Конфлікт роду розв'язаний, якщо прохід завершився (i > n div 2) або у результаті перестановки не виникнув новий сімейний конфлікт. (процедура Conflict)

```
Procedure ConSwap(i, j : Integer);
    Var b : Real;
    Begin
     If a[i] < a[j] then begin
       b := a[i]; a[i] := a[j]; a[j] := b
     end
    End;
Procedure Conflict(i, k : Integer);
      Var j : Integer;
    Begin
      j := 2*i;
      If j = k
       then ConSwap(i, j)
       else if j < k then begin
          if a[j+1] > a[j] then j := j + 1;
           ConSwap(i, j); Conflict(j, k)
        end
      End;
```

І етап - побудови дерева, що сортує - оформимо в виді рекурсивної процедури, використовуючи визначення:

Якщо ліве і праве піддерева T(2i) і T(2i+1) дерева T(i) , що сортують, то для перебудови T(i) необхідно розрішити конфлікт роду в цьому дереві.

Procedure SortTree(i : Integer);

```
begin
 If i <= n div 2 then begin
   SortTree(2*i);
   SortTree(2*i+1);
    Conflict(i, n)
  end
end;
```

На II-ому етапі - етапі просівання - для k от n до 2 повторюються наступні дії:

1.Переставити A[1] і A[k];

2.Побудувати дерево, що сортує початкового відрізка масиву A[1..k-1], усунувши конфлікт роду в корені A[1]. Відмітимо, що 2-а дія потребує введення в процедуру Conflict параметра k.

```
Program HeapSort;
  Const n = 100;
    Var A : Array[1..n] of real;
        k : Integer;
   {процедури ConSwap, Conflict, SortTree, введення, виведення}
   Begin
    { введення }
    SortTree(1);
    For k := n downto 2 do begin
     ConSwap(k, 1); Conflict(1, k - 1)
    end;
   { виведення }
   End.
```

Оцінимо складність алгоритму у термінах C(n), M(n). Оскільки кожна перестановка пов'язана з порівняннями в процедурі ConSwap, $M(n) \le C(n)$. Далі, в процедурі Conflict порівняння передує виклику ConSwap, тому загальне число порівнянь не перевищує подвоєного числа викликів ConSwap. Отже, C(n) можна отримати, оцінивши кількість звернень до ConSwap.

Складність I-го етапу $C_1(n)$ задовольняє співвідношенню $C_1(n) \le 2C_1(n / 2) + C_0(n)$ де $C_0(n)$ - складність процедури Conflict(1, n). Для $C_0(n)$ маємо: $C_0(n) \le C_0(n/2) + 1$, оскільки виклик Conflict(i, n) містить один (рекурсивний) виклик Conflict(2i, n) і ланцюжок викликів обривається при i > n/2. Тому $C_0(n) \le c_0 \log_2 n$. Таким чином,

$C_1(n) \le 2\, C_1(n / 2) + c_0 \log_2 n$.

Нехай k - деяке число. Тоді

$C_1(n) <= 2^k C_1(n/2^k) + c_0 (2^{k-1} \log_2(n/2^{k-1}) + ... + \log_2(n/2) \log_2 n)$

Оскільки $C_1(1) = 0$, маємо $C_1(n) <= c_0 (\log_2 n + 2 \log_2(n/2) + ... (n/2) \log_2(2))$. Замінимо $\log(n/2^i)$ на $\log_2 n$, посиливши нерівність. Отримаємо $C_1(n) <= (c_0 \log_2 n)(1 + 2 + 4 + ... n/2)$

Сума в дужках не перевищує n. Тому

$$C_1(n) <= c_0 n \log_2 n \quad (7)$$

Арифметичний цикл II-го етапу виконується n - 2 разів і кожне повторення містить ConSwap(k, 1) і Conflict(1, k - 1). Тому його складність $C_2(n)$ можна оцінити нерівністю $C_2(n) <= (1+C_0(n-1)) + (1+C_0(n-2)) + ... + (1+C_0(2))$ Замінимо $C_0(i)$ на $C_0(n-1)$, посиливши нерівність. Отримаємо

$$C_2(n) <= n + n\, C_0(n-1) = c_1 n \log_2 n \quad (8)$$

Остаточно $C(n) = C_1(n) + C_2(n) = (c_0+c_1)\, n \log_2 n$, або

$$C(n) = O(n \log_2 n) \quad (9)$$

Ми показали, що пірамідальне сортування (з точністю до мультиплікативної константи) оптимальне: його складність співпадає (з тією ж точністю) з нижньою оцінкою задачі.

Швидке сортування Хоара.

Удосконаливши метод сортування, оснований на обмінах, К.Хоар запропонував алгоритм QuickSort сортування масивів, що дає на практиці відмінні результати і дуже просто програмується. Автор назвав свій алгоритм швидким сортуванням.

Ідея К.Хоара полягає у наступному:

Виберемо деякий елемент x масиву A випадковим способом;

Розглядаємо масив у прямому напрямку (i= 1, 2, ...), шукаючи у ньому елемент A[i] не менший, ніж x;

3.Розглядаємо масив у зворотному напрямку(j = n, n-1,..), шукаючи у ньому елемент A[j] не більший, ніж x;

4.Меняємо місцями A[i] і A[j];

Пункти 2-4 повторюємо до тих пір, поки i < j;

В результаті такого зустрічного проходу початок масиву A[1..i] і кінець масиву A[j..n] опиняються розділеними “бар’єром” x: $A[k] \le x$ при $k < i$, $A[k] \ge x$ при $k > j$, причому на розділ ми затратимо не більш n/2 перестановок. Тепер залишилось зробити ті ж дії з початком і кінцем масиву, тобто застосувати їх рекурсивно!

Таким чином, процедура Hoare, що описана нами, залежить від параметрів k і m - початкового і кінцевого індексів відрізку масиву, що оброблюється.

```
Procedure Swap(i, j : Integer);
     Var b : Real;
   Begin
     b := a[i]; a[i] := a[j]; a[j] := b
   End;
Procedure Hoare(L, R : Integer);
     Var left, right : Integer;
         x : Integer;
   Begin
    If L < R then begin
```

```
      x := A[(L + R) div 2];                    {вибір бар'єра x}
      left := L; right := R ;
      Repeat                                          {зустрічний прохід}
       While A[left] < x do left := Succ(left);   {перегляд уперед}
       While A[right] > x do right := Pred(right); {перегляд назад}
        If left <= right then begin
         Swap(left, right);                       {перестановка}
         left := Succ(left);  right := Pred(right);
        end
       until left > right;
       Hoare(L, right);                                {сортуємо початок}
       Hoare(left, R)                                  {сортуємо кінець}
     end
    End;
Program QuickSort;
  Const n = 100;
    Var A : array[1..n] of Integer;
{ процедури Swap, Hoare, введення і виведення }
Begin
  Inp;  Hoare(1, n);  Out
 End.
```

Найбільш тонке місце алгоритму Хоара - правильне опрацювання моменту закінчення руху покажчиків left, right. Помітьте, що в нашій версії переставляються місцями елементи, що дорівнюють x. Якщо в циклах While замінити умови (A[left] < x) і (A[right] > x) на (A[left] <= x) і (A[right] >=x), при x = Max(A) індекси left і right пробіжать весь масив і побіжать далі! Ускладнення ж умов продовження перегляду погіршить ефективність програми.

Аналіз складності алгоритму в середньому, що використовує гіпотезу про рівну ймовірність всіх входів, показує, що

$C(n) = O(n \log_2 n)$, $M(n) = O(n \log_2 n)$.

У гіршому випадку, коли в якості бар'єрного вибирається, наприклад, максимальний елемент підмасиву, складність алгоритму квадратична.

3. Пошук k-того в масиві. Пошук медіани масиву.

З задачею сортування тісно пов'язана задача находження k-того за величиною елемента у масиві. З її частковим випадком - пошук 1-го (мінімального) і n-того елемента (максимального) - ми вже знайомі. Аналогічно можна розв'язати задачу пошуку 2-го і (n - 1)-го за величиною елементів, однак кількість порівнянь зростає вдвічі. Перш, ніж розглянути задачу у загальному виді, приведемо необхідне визначення.

К-тим за величиною елементом послідовності називається такий елемент b цієї послідовності, для якого в послідовності існує не більш, ніж k-1 елемент, менший, ніж b і не менше k елементів, менше або рівних b.

Наприклад, у послідовності {2, 1, 3, 5, 4, 2, 2, 3 } четвертим за величиною буде 2.

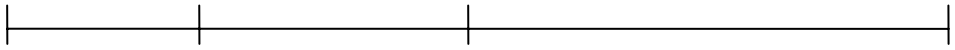
Іншими словами, якщо відсортувати масив, то k-тий елемент опиниться на k-тому місці (b = A[k]). Можна очкувати, що найбільш складним буде алгоритм пошуку елемента при k = n div 2 - тобто елемента, рівновіддаленого від кінців масиву. Цей елемент назвемо медіаною масиву.

Очевидний розв'язок задачі пошуку k-того елемента - повне або часткове (до k) сортування масиву A. Наприклад, алгоритм TreeSort можна перервати після просівання k елементів. Тоді $C(n) = O(n + k \log_2 n)$

Однак ефективний на практиці розв'язок можна отримати, якщо застосувати ідею К.Хоара розділення масиву бар'єром. Насправді, аналіз метода розділення елементів по бар'єру в процедурі Hoare показує, що після її закінчення мають місце співвідношення

Right < Left, при i < Left A[i] <= x, при i > Right A[i] >= x

Таким чином, відрізок First .. Last розбиває на 3 частини:



First	Right	Left	Last
A[i] <= x	A[i] = x	A[i] >= x	

Подальший пошук, отже, можна організувати так:

Якщо k <= right
 то шукати k-тий елемент в A[First .. right]
 інакше якщо k <= left
  то k-тий елемент дорівнює x
  інакше шукати k - тий елемент в A[left + 1 .. Last]

```
Procedure HoareSearch(First, Last : Integer);
      Var l, r : Integer; x : Integer;
     Begin
      If First = Last
        then Write(A[k]) { залишився 1 елемент }
        else If First < Last then begin
         x := A[k]; {вибір бар'єра x}
          l := First; r := Last;
          Repeat {зустрічний прохід}
           While A[l] < x do l := Succ(l);
           While A[r] > x do r := Pred( r );
            If l <= r then begin
             Swap(l, r);
```

```
        l := Succ(l); r := Pred( r );
       end
      until l > r;
  If k <= r
  then HoareSearch(First, r) {1-а частина масиву}
  else If k <= l
   then Write(x) {елемент знайдений}
   else HoareSearch(l+1, Last) {3-ая частина масиву}
  end
End;
```

Автор методу пропонує у якості бар'єрного елемента вибирати A[k], хоча це не дає ніяких переваг перед вибором середнього або випадкового елемента.

Як і алгоритм швидкого сортування, цей алгоритм у гіршому випадку неефективний. Якщо при кожному розділу 1-а частина масиву буде містити 1 елемент, алгоритм витратить O(kn) кроків. Однак в середньому його складність лінійна (незалежно від k). Зокрема, навіть при пошуку медіани A[n div 2]

$$C_{cp}(n) = O(n),\ M_{cp}(n) = O(n)$$

В завершення цього розділу відзначимо, що існує лінійний за складністю у гіршому випадку алгоритм пошуку k-того елемента, однак його теоретичні переваги переростають у практичні при дуже великих значеннях n.

4. Метод "розділяй і володій".

Один з найбільш відомих методів проектування ефективних алгоритмів - метод "розділяй і володій" полягає у зведенні задачі, що розв'язується, до рішення однієї або кількох більш простих задач.

Якщо спрощення задачі полягає в зменшенні її параметрів, таке зведення дає рекурсивний опис алгоритму. Зокрема, зменшення параметра може означати зменшення кількості даних, з якими працює алгоритм.

Характерна ознака методу - зведення до декількох задач рівних за складністю підзадач. Ефективність отриманого алгоритму залежить від, по-перше, від кількості дій, затрачених на зведення задачі до підзадач, по-друге - від балансу складностей підзадач.

Класичний приклад застосування методу - бінарний пошук у впорядкованому масиві. Масив розбивається на дві рівні частини. Використовуючи потім два порівняння, алгоритм або знаходить елемент, або визначає ту половину, в якій його треба шукати - тобто зводить задачу розміром n до задачі розміром n/2. Цей же прийом використовується в алгоритмах Хоара сортування і пошуку, в алгоритмі піднесення числа в натуральну степінь ($8, приклад 11), у деяких інших раніше сформульованих задачах. Розглянемо відомий розв'язок задачі пошуку мінімального і максимального елемента в масиві.

Нехай A[1..n] - числовий масив. Треба знайти Max(A) і Min(A). Припустимо, що $n = 2^k$. Таке припущення дає можливість завжди ділити масиви, отримані навпіл.

Опишемо основну процедуру MaxMin. Нехай S - деяка множина і |S| = m. Розіб'ємо S на рівні за числом елементів частини: $S = S_1 \cup S_2$, $S_1 \cap S_2 = \varnothing$, $|S_1| = m/2$, $|S_2| = m/2$. Знайдемо Max1, Min1 як результат MaxMin(S_1) і Max2, Min2 як результат MaxMin(S_2).

Потім Max := Max(Max1, Max2); Min := Min(Min1, Min2);

```
procedure MaxMin(L, R : Integer Var Max, Min : Real);
     Var Max1, Min1, Max2, Min2 : Real;
    Begin { L, R - межі індексів масиву A }
     If R - L = 1
      then If A[L] > A[R] { вибір Max, Min за одне порівняння}
       then begin
         Max := A[l]; Min := A[R] end
       else begin
         Max := A[R]; Min := A[L] end
       else begin
        C := (R + L - 1) div 2;
        MaxMin(L, C, Max1, Min1);
        MaxMin(C+1, R, Max2, Min2);
        If Max1 > Max2 then Max := Max1 else Max := Max2;
        If Min1 < Min2 then Min := Min1 else Min := Min2
       end
    End;
```

Нехай C(n) - оцінка складності процедури MaxMin в термінах порівнянь. Тоді, очевидно

C(2) = 1,

C(n) = 2 C(n/2) + 2 при n > 2

Звідси C(n) = 2 (n/4 + n/8 + ... + 1) + n/2 = 2(n/2 - 1) + n/2;

C(n) = 3/2 n - 2

Для порівняння відзначимо, що алгоритм пошуку Max і Min методом послідовного перегляду масиву потребує 2n -2 порівнянь. Таким чином, застосування метода "розділяй і володій" зменшило часову складність задачі в фіксоване число раз. Зрозуміло, істинна причина поліпшення - не в застосуванні рекурсії, а в тому, що максимуми і мінімуми двох елементів масиву, що стоять поруч, відшукуються за одне порівняння! Метод дозволив просто і природно описати обчислення.

Довжина ланцюжка рекурсивних викликів в алгоритмі дорівнює $\log_2 n$ -1, і в кожній копії використовується 4 локальних змінних. Тому алгоритм використовує допоміжну пам'ять об'єму O($\log_2 n$), що розподіляється динамічно. Це - плата за економію часу.

5. Метод цифрового сортування.

Іноді при розв'язані задачі типу задачі сортування можна використовувати особливості типу даних, що перетворюються, для отримання ефективного алгоритму. Розглянемо одну з таких задач - задачу про обернення підстановки.

Підстановкою множини 1..n назвемо двомірний масив A[1..2, 1..n] виду

1	2	.....	n-1	n
j1	j2	.....	j n-1	j n

в якому 2-ий рядок містить всі елементи відрізка 1. .n. Підстановка B називається оберненою до підстановки A, якщо B отримується з A сортуванням стовпців A у порядку зростання елементів 2-ого рядка з наступною перестановкою рядків. Треба побудувати алгоритм обчислення оберненої підстановки. З визначення випливає, що стовпець [i, j_i] підстановки A треба перетворити в стовпець [j_i , i] і поставити на j_i-те місце в підстановці B.

```
{Type NumSet = 1..n;
Substitution = array[ 1..2, NumSet] of NumSet; }
Procedure Reverse ( Var A, B : Substitution);
   Begin
    For i := 1 to n do begin
      B[A[2, i], 2] := i; B[A[2, i], 1] := A[2, i]
    end
   End;
```

Складність процедури Reverse лінійна, оскільки тіло арифметичного циклу складається з двох операторів присвоювання. Між тим стовпці підстановки відсортовані.

Аналогічна ідея використовується в так званому сортуванні вичерпуванням (або цифровому сортуванні), що застосовується для сортування послідовностей слів (багаторозрядних чисел). Для кожної букви алфавіту алгоритм створює структуру даних "черпак", в яку пересилаються слова, що починаються на цю букву. В результаті перегляду послідовність виявляється відсортованою по першій букві. Далі алгоритм застосовують рекурсивно до кожного "черпака", сортуючи його по наступним буквах.

6. Здачі і вправи.

1.Дано масив A[1..900] of 1..999. Знайти трьохзначне число, що не знаходиться у цьому масиві. Розглянути два варіанта задачі:

а) Допоміжні масиви в алгоритмі не використовуються;

б) Використовування допоміжних масивів дозволено.

2.Реалізувати алгоритм TreeSort, застосувавши метод вкладення дерева в масив. Для цього використати допоміжний масив B.

3.Реалізувати ітеративну версію алгоритму HeapSort:

а)Замінити рекурсію в процедурі SortTree арифметичним циклом For...downto, що оброблює дерево за рівнями, починаючи з нижнього;

б)Замінити рекурсію в процедурі Conflict ітераційним циклом, керуючим змінною i. { i := 2i або i := 2i + 1 }.

4.Реалізувати ітеративну версію алгоритму процедури HoareSeach, замінивши рекурсію ітераційним циклом.

5-6. Застосувати для розв'язку задач 1-2 параграфа 8 ідею Хоара.

7.Реалізувати алгоритм "тернарного" пошуку елемента в упорядкованому масиві, поділяючи ділянку пошуку на 3 приблизно рівні частини. Оцінити складність алгоритму у термінах C (n). Порівняти ефективності бінарного і тернарного пошуку.

8.Реалізувати алгоритм пошуку максимального і мінімального елементів у масиві (процедура MaxMin) для довільного n.

9.Нехай A[1..n] - масив цілих чисел, упорядкований за зростанням (A[i] < A[i+1]). Реалізувати алгоритм пошуку такого номера i, що A[i]= i методом "розділяй і володій". Оцінити складність алгоритму.

10-12. Застосувати для розв'язку задач 3-5 параграфа 8 метод "розділяй і володій".

13.Реалізувати алгоритм процедури Reverse "на місці", формуючи обернену підстановку в масиві A.

ГРАФІКА В СИСТЕМІ ПРОГРАМУВАННЯ ТР-6.

Характерна особливість систем програмування, реалізованих на персональних комп'ютерах - використання розширень мови засобами обробки графічної інформації. В системі програмування Turbo Pascal ці засоби зосереджені в модулі GRAPH.

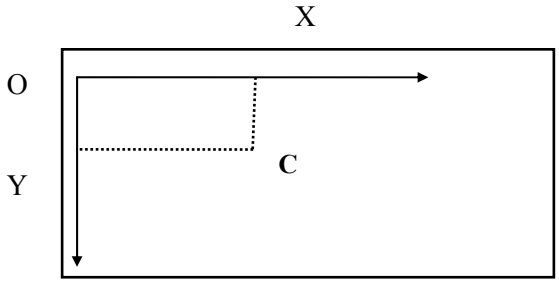
Модуль (UNIT) - це сукупність (бібліотека) описань констант, типів даних, змінних, процедур і функцій. Кожний модуль представляє собою по суті самостійну програму на Паскалі: він може містити декілька операторів, що виконуються перед викликом програми, і здійснюють всю необхідну ініціалізацію. Засоби, що описані в модулі, можна використовувати в програмі. Модуль можна компілювати окремо. Для того, щоб включити модулі в програму, необхідно описати їх у спеціальному розділі описань Uses. Синтаксичне описання цього розділу має вид:

Uses <список модулів >; {необов'язковий параметр}

Всі описання усередині модуля як правило пов'язані. Наприклад, стандартний зовнішній модуль CRT містить засоби, що підтримують взаємодію програми з екраном монітора в алфавітно-цифровому режимі (текстовий екран). Модуль GRAPH призначений для взаємодії програми з екраном у графічному режимі (графічний екран).

1. Графічний екран. Види графічних адаптерів.

Елементарною одиницею інформації, що виводиться в графічному режимі на екран, є точка, покрашена в деякий колір. Ця точка називається пікселом. Позиція піксела на екрані визначається в декартовій системі координат (мал. 1). Відмітимо незвичний напрямок осі OY: вона направлена з лівого верхнього кута екрана униз.



Таким чином, кожний піксел описується трьома параметрами:
(X, Y, C), де C - колір піксела.
X - ціле число з інтервалу [0 ... GetMaxX];
Y - ціле число з інтервалу [0 ... GetMaxY];
C - ціле число з інтервалу [0 ... GetMaxColors]

де GetMaxX, GetMaxY, GetMaxColors - константи, що описані в модулі GRAPH.

Діапазони допустимих значень X, Y, C визначаються технічними характеристиками апаратури комп'ютера, операційною системою і наладкою програми на один з можливих режимів використовування модуля GRAPH - ініціалізацією графіки. Детальні відомості про ініціалізацію графіки знаходяться в документації системи програмування.

Система програмування Turbo-Pascal в MS-DOS підтримує, зокрема, наступні графічні режими:

Відеоадаптер	GetMaxX	GetMaxY	GetMaxColor	Драйвер
CGA	320	200	4 з 16	CGA.BGI
MCGA	320	200	16 з 256	CGA.BGI
EGA	640	200	16 з 256	EGAVGA.BGI
EGA	640	350	16 з 256	EGAVGA.BGI
VGA	640	480	16 з 256	EGAVGA.BGI

Настройка програми на один з графічних режимів виконується автоматично оператором InitGraph, якщо відповідний драйвер установлений на комп'ютері.

InitGraph(< гр. адаптер >, < режим гр.др >, < путь >);

Перехід у текстовий режим здійснюється оператором CloseGraph.

Таким чином, програма, що використовує графіку, має вид:

```
Program MyGraph;
    Uses Graph;
     Var grDriver, grMode, errCode: Integer;
     .............
   Begin
     grDriver := Detect;
     InitGraph(grDriver, grMode, '< шлях до драйвера >'); { приклад шляху: D:\PasSys6\BGI }
     .............
   End.
```

Константа Detect визначена в модулі GRAPH. Її значення дорівнює 0. Detect вказує системі на необхідність підключення драйвера графіки, відповідного типу адаптера дисплея.

Графічний модуль GRAPH містить декілька десятків (більш 50) процедур і функцій, кожна з яких описана в документації системи і доступна програмісту з розділу HELP головного меню.

Тому ми приведемо тільки деякі відомості про GRAPH.

Константи:

процедури Bar3D;

процедури PutImage;
відсікання, кольорів, стиля ліній і ширини;
стиля тексту;
графічні драйвери і графічні режими драйверів;
шаблонів заповнення;
.....

Типи:

ArcCoordsType
FillPatternType
FillSettingsType
LineSettingsType
PaletteType
PointType
TextSettingsTtype
.....

Процедури і функції:

Графічні примітиви:

PutPixel, GetPixel, GetX, GetY,
Line, LineTo, Move, Bar, DrawPoly, Rectangle, ...
Arc, Circle, Ellipse, ...

Обробка кольорів:

GetColor; GetPixel, GetBkColor, ...
SetColor, SetBkColor, SetPalette,...

Установка стиля малювання: SetLineStyle, SetFillStile, ...

Обробка тексту: OutText, OutTextXY, TextWidth, SetTextStyle, ...

Обробка графічного екрана: SetActivePage, PutImage, GetImage, SetViewPort,...

Процедура PutPixel(X, Y, C), наприклад, фарбує точку (X,Y) екрана в колір C. Одночасно графічний курсор поміщається в цю точку.

Процедура Line(X1,Y1, X2,Y2) малює відрізок AB кольором, що встановлений попередньо A = (X1, Y1), B = (X2, Y2).

2. Задача побудови графіка функції.

Як приклад застосування графічних засобів розглянемо задачу побудови графіка функції y = f(x). Побудова графіка функції потребує як мінімум:

Побудову вікна з оформленою системою координат;
Побудову кривої y = f(x) у вікні;
Оформлення необхідних надписів.

а) Оформлення системи координат.

Вікно системи координат визначається параметрами:

(Un, Vn) - координати лівого верхнього кута вікна;

(Uk, Vk) - координати правого нижнього кута вікна;

(Uo, Vo) - координати початку координат;

Mx, My - масштаби системи координат зо x і y.

Всі параметри повинні бути визначені в комп'ютерній системі координат (в координатах пікселів). При цьому(Un, Vn), (Uk, Vk) - константи (нерухоме вікно), а (Uo, Vo), M обчислюються, виходячи з даних, що вводяться користувачем. На практиці користувач частіше всього вказує межі змін змінних x, y - (Xn, Xk), (Yn, Yk). Значення M, Uo, Vo необхідно обчислювати перед побудовою системи координат.

Для простоти ми будемо вважати, що Xn<0<Xk, Yn<0<Yk. Тоді початок системи координат попадає у вікно графіка. У протилежному випадку треба програмувати декілька спеціальних варіантів розміщення осей координат.

Mx = [(Uk - Un)/(Xk - Xn)],

My = [(Vk - Vn)/(Yk - Yn)],

Uo = Uk - [Mx*Xk], Vo = Vn + [My*Yk];

Використовуючи ці формули, напишемо потрібну процедуру:

```
Procedure WinGraf(Xn, Xk, Yn, Yk: Real;       Var Mx, My, Uo, Vo : Integer);
   Begin
    { обчислення параметрів }
    Mx := Round((Uk - Un)/(Xk - Xn));    My := Round((Vk - Vn)/(Yk - Yn));
    Uo := Uk - Round(Mx*Xk);             Vo := Vn + Round(My*Yk);
    { малювання системи координат }
     SetColor(LightGray);
     Bar(Un, Vn, Uk, Vk); { вікно графіка }
     SetColor(Green); { малюємо зеленим кольором}
     SetLineStyle(0, 0, ThickWidth);    {Малюємо товсті суцільні осі}
     Line(Un, Vo, Uk, Vo); { ось OX }
     Line(Uo, Vn, Uo, Vk); { ось OY }
     SetColor(Red);
     Circle(Uo, Vo, 2); {початок координат - невелике коло}
     SetLineStyle(0, 0, 1); {Малюємо тонкі суцільні відмітки1 }
     Line(Uo+Mx, Vo-4, Uo+Mx, Vo+4); { відмітка 1 на OX }
     Line(Uo-4, Vo-My, Uo+4, Vo-My); { відмітка 1 на OY }
     SetTextStyle(0, 0, 2);
     OutTextXY(Uo + Mx - 7, Vo + 16, '1'); { малюємо 1 }
     OutTextXY(Uo - 24, Vo - My - 7 , '1'); { малюємо1 }
      { Подальші прикрашення вікна графіка }
    End;
```

б) Побудова кривої y = f(x) у вікні;

При поточечній побудові кривої на кожному кроці необхідно обчислювати наступні значення (x, y), обчислювати істинні координати (u, v) піксела - образа точки (x, y) на екрані і виводить піксел на екран.

На побудову графіка витрачається більша частина часу. Тому доцільно апроксимувати дугу кривої y = f(x) хордою, вибираючи вузли апроксимації з деяким кроком DrawStep (наприклад, DrawStep = 10 (пікселів)).

Таким чином, процедура малювання графіка має вид:

```
Procedure FuncDraw;
    Const DrawStep = 10;
      Var i: Integer;
          x, y: Real;
          u, v: Integer;
          ArgStep: Real;
          StepNum: Integer;
      Begin
       SetColor(Blue); {колір кривої}
       SetLineStyle(0, 0, NormWidth); {товщина нормальна}
       ArgStep := DrawStep / Mx; {крок зміни x}
       StepNum := Round((Uk - Un) / DrawStep); {кількість вузлів}
       { обчислення початкової точки графіка }
       x := Xn ; u := Un;
       y := sqrt(Abs(x))*sin(2*x);  v := Round(Vo - My*y);
        MoveTo(u, v);
        { цикл малювання апроксимуючою ламаною }
        For i := 1 to StepNum do begin
          x := x + ArgStep; u := u + DrawStep;
          y := sqrt(Abs(x))*sin(2*x); v := Round(Vo - My*y);
          LineTo(u,v);
        end
      End;
```

Робота з текстами і введення-виведення в графічному екрані.

Введення-виведення інформації в режимі графіки суттєво відрізняються від відповідних процедур в алфавітно-цифровому режимі. Виведення тексту на екран здійснюють процедури OutText, OutTextXY.

OutText(TxtStr: String) виводить строку на пристрій виведення (графічний екран) в точку - позицію графічного курсора.

OutTextXY(X, Y: Integer; TxtStr: String) виводить строку на пристрій виведення (графічний екран) в точку (X, Y).

Крім цих процедур, в модулі GRAPH описані процедури і функції, підтримуючі різні стилі виведення. Так, наприклад:

Функції TextHeight(TxtStr:String), TextWidth(TxtStr:String) повертають відповідно висоту і ширину рядка в пікселах.

Процедура SetTextStyle(Font, Direction:Word; CharSize:Word) установлює стиль виведення тексту в графічному режимі, тобто визначає шрифт, стиль тексту і коефіцієнт збільшення символу. Як приклад опишемо процедуру WinText виведення надписів-коментарів у вікно графіка функції.

```
Procedure WinText;
   Begin
    SetColor(LightMagenta);
    SetTextStyle(0, 1, 1);
    OutTextXY(15,15, 'Function Graphics Window');
    SetColor(Cyan);
    SetTextStyle(0, 0, 1);
    OutTextXY(460,15, 'Production of SL.XXI');
    SetTextStyle(0, 0, 2);
    OutTextXY(28, 390, 'Y = Sqrt(Sin(|x|)');
   End;
```

Введення інформації в графічному режимі засобами модуля GRAPH не підтримується. Тому для організації введення необхідно скористуватися або стандартними процедурами, або процедурами модуля CRT (бібліотека алфавітно-цифрового режиму). На практиці це означає посимвольне введення інформації, формування рядка введених символів і перетворення типу рядка до типа даного, необхідного програмі.

Для введення символу використовується функція ReadKey - читання символу з клавіатури. Для відображення введеного символу на екрані (відлуння) використовується одна з графічних процедур виведення. Суперпозиція цих дій має вид Symbol := ReadKey; OutTextXY(x0, y0, Symbol). Для більш повної імітації процедури Read точка введення інформації на екрані звичайно відмічається зображенням курсора, який здвигається на одну позицію вправо при кожному введенні символу.

```
Procedure ReadGrStr(x, y: Integer; var Str: String);
   Const CursW = 8; CursH = 6; {ширина і висота курсора}
    Var Symbol: Char; x1, y1: Integer;
   Begin
    x1 := x + CursW; y1 := y + CursH;
    SetColor(Green); {колір курсора }
    Rectangle(x, y, x1, y1);
    Str := ''; Symbol := ReadKey; {ініціалізація }
    While Ord(Symbol) <> 13 do begin
    SetColor(LightGray); {Колір вікна графіка}
    Bar(x, y, x1, y1); {Віддалити курсор}
```

```
        OutTextXY(x, y, Symbol); {Вивести символ}
         x := x + CursW; x1 := x + CursW;
         Str:= Str + Symbol; {Формування вихідного рядка}
         SetColor(Green); {Вивести курсор}
         Rectangle(x, y, x1, y1);
         Symbol := ReadKey {Рахувати символ}
        end
      End;
```

Відмітимо, що ніяких редагуючих дій процедура ReadGrStr не підтримує. Для реальних програм її треба удосконалити.

Числа вводяться як рядки, перетворюються потім системною процедурою Val. Val(Str: String; var v {компонент Text}; var Code: Integer);

Змінна Code містить номер помилки формату. За її допомогою можна контролювати правильність числового формату введеного рядка.

3. Рекурсивні описи в графіку.

Використовуючи графічні можливості комп'ютера і рекурсивні описи малюнків, можна будувати на екрані зображення красивих мозаїк і кривих.

Розглянемо метод отримання одного з таких малюнків - зображення квадрата Кантора. (Квадрат Кантора відомий в математиці як приклад ніде не щільної множини додатної площі (міри)).

Квадрат Кантора - це фігура, що отримана шляхом вирізання деяких ділянок з початкового суцільного квадрату. Для того, щоб побудувати квадрат Кантора, необхідно:

4.Розбити суцільний (пофарбований в один колір) квадрат на 9 рівних квадратів вертикальними і горизонтальними лініями (3 на 3);

5.Вирізати (пофарбувати в інший колір) середній з 9-ти маленьких квадратів.

6.Застосувати (рекурсивно) дії 1, 2, 3, до 8-ми маленьких суцільних квадратів, що залишилися.

```
Program KantorSet;
   Uses Graph, Crt;
  Const Xmin = 0; Ymin = 0; { розміри квадрата }
        Xmax = 486; Ymax = 486;
        N = 6; { глибина рекурсії }
   Var grDriver, GrMode: Integer; ch: Char;
 Procedure Picture(k: Integer; x0, y0, x3, y3: Integer);
     Var x, y, dx, dy: Integer; i, j: Integer;
   Begin
    If k <> N
     then begin
       dx := (x3-x0) div 3; dy := (y3-y0) div 3;
```

```
          x := x0;
          For i := 0 to 2 do begin
          y := y0;
          For j := 0 to 2 do begin
            If (i = 1) and (j = 1)
              then Bar(x, y, x+dx, y+dy) {вирізаємо квадрат}
              else Picture(k+1, x, y, x+dx, y+dy);
            y := y + dy  end;
          x := x + dx  end;
         k := k + 1
       end
End;
    Begin
      grDriver := Detect;
      InitGraph(grDriver, grMode, 'E:\lvov\pascal\bgi');
       SetBkColor(Blue);
       Picture(1, Xmin, Ymin, Xmax, Ymax);
       ch := ReadKey
     End.
```

Аналогічно можна побудувати і інші симпатичні мозаїки.

4. Робота з сторінками і фрагментами.

Графічне зображення, що виводиться на екран, зберігається в спеціальній області оперативної пам'яті комп'ютера, яка називається відео пам'яттю. Розмір пам'яті, що відводиться під зображення одного графічного екрана, залежить від графічного режиму. Ця частина пам'яті називається сторінкою відеопам'яті (відеосторінкою). Так, у режимі EGA (640 * 350, 16 кольорів) сторінка відеопам'яті займає приблизно 110 Kb. У цьому режимі, якщо відеопам'ять має 256 Kb, визначені дві сторінки - з номерами 0 і 1.

Модуль Graph містить засоби, що підтримують роботу з декількома відеосторінками. Наявність декількох сторінок дозволяє оперувати заздалегідь заготовленими малюнками і їх фрагментами. З цією метою введені поняття видимої сторінки і активної сторінки. Видима сторінка - це сторінка, яка виводиться на екран. На активній сторінці формуються зображення, що виконуються операторами графіки.

Найбільш використовувані засоби, підтримуючі роботу з сторінками і їх фрагментами, коротко описані нижче:

SetVisualPage(Page:Word)- візуалізує сторінку з номером Page;

SetActivePage(Page: Word) - активізує сторінку з номером Page;

GetMem(P, Size) - резервує пам'ять розміру Size і встановлює на неї покажчик P; {Поняття покажчика розглядається в розділі 12}

ImageSize(x1, y1, x2, y2: Integer):Word - повертає розмір відеопам'яті прямокутної області екрана;

GetImage(x1, y1, x2, y2:Integer; var BitMap) - зберігає образ вказаної області в буфері;

PutImage(x, y: Integer; var BitMap; BitBit: Word) - поміщає (двійковий) образ із буфера на екран. Параметр BitBit визначає спосіб накладення образа на активну сторінку;

```
Program ImageExample;
    Uses Graph, Crt;
     Var grDriver, GrMode: Integer;
         Ch :Char;
          P :Pointer; {покажчик на образ у пам'яті}
          Size :Word; {розмір образа}
       Begin
        grDriver := Detect;
         InitGraph(grDriver, grMode, 'E:\lvov\pascal\bgi');
         SetGraphMode(1); { Розмір екрана = 640 * 350 }
         {Приклад для SetVisualPage, SetActivePage}
         SetBkColor(Blue);
         SetActivePage(0); {Малюємо на сторінці 0. На екрані сторінка 0}
         Rectangle(40, 40, 140, 140);  OutTextXY(50, 90, 'First Page');
         Ch := ReadKey;  SetActivePage(1); { Малюємо на сторінці 1. На екрані сторінка
          0}
         SetColor(Magenta);
         Circle(90, 90, 50);
         OutTextXY(45, 90, 'Second Page');
         SetVisualPage(1); {Виводимо на екран стор. 1}
         Ch := ReadKey;    SetVisualPage(0); {Виводимо на екран стор. 0}
         Ch := ReadKey;    SetVisualPage(1); {Виводимо на екран стор. 1}
         Ch := ReadKey;
          {Приклад для GetImage, ImageSize, і PutImage}
          Size := ImageSize(40, 40, 140, 140); {Установили розмір малюнка}
          GetMem(P, Size); {Резервуємо пам'ять, на яку установлений покажчик P }
          SetActivePage(0);
          GetImage(40, 40, 140, 140, P^); {Пересилаємо малюнок в область пам'яті -
значення P}
          Ch := ReadKey;
          SetActivePage(1); {на сторінці 1}
          PutImage(400, 200, P^, NormalPut); {Виводимо на стор.1 малюнок}
          Ch := ReadKey;
          ClearDevice;
          CloseGraph;
```

end.

5.Задачі і вправи.

11.Описати процедуру оформлення системи координат, у якій точка початку координат (Uo,Vo) може виходити за межі вікна графіка. Осі координат у цьому випадку повинні співпадати з відповідними межами вікна графіка.

12.Удосконалити процедуру FuncDraw виведення графіка функції y = f(x) обробкою випадків, коли крива виходить за межі вікна системи координат.

13.Змінити процедуру FuncDraw для виведення на екран графіка функції, заданої параметрично: x = fi(t), y = psi(t), A <= t <= B.

14.Написати програму побудови графіка функції в полярній системі координат.

15.Реалізувати процедури лінійних геометричних перетворень (перенос, поворот, осьова і центральна симетрії, гомотетія).

16.Використовуючи процедури впр.5, написати програму, що демонструє геометричні перетворення плоских геометричних фігур (зокрема - трикутника).

17.Удосконалити процедуру ReadGrStr, перетворивши її в редактор рядка з діями Insert, Delete, BackSpace за допомогою відповідних клавіш.

18.Змінити процедуру ReadGrStr, перетворивши її в редактор дійсного числа.

19.Змінити процедуру ReadGrStr, перетворивши її в редактор цілого числа.

20.Написати програму зображення мозаїки, яка визначена правилами:

а. У центрі квадрата розміром A*A малюється коло радіусом A/2;

б. Квадрат розбивається на 4 рівних (під)квадрата горизонтальною і вертикальною лініями.

в. Дії 1-3 застосовуються рекурсивно до кожного з 4-х квадратів.

Задачі обчислювальної геометрії.

16. Дано набір з n точок площини, заданих координатами $A_1(X_1, Y_1)$, $A_2(X_2, Y_2)$, ..., $A_n(X_n, Y_n)$. Знайти таку точку S(X, Y) на площині, сума відстаней від якої до даних точок набору найменша. Проілюструвати розв'язок на малюнку.

17. Дано набір з n точок площини, заданих координатами: $A_1(X_1, Y_1)$, $A_2(X_2, Y_2)$, ..., $A_n(X_n, Y_n)$. Вибрати з цього набору точки, що знаходяться в вершинах випуклого багатокутника найменшої площини і який містить всі точки набору. Проілюструвати розв'язок на малюнку.

18. Дано набір з n точок площини, заданих координатами: $A_1(X_1, Y_1)$, $A_2(X_2, Y_2)$, ..., $A_n(X_n, Y_n)$. Побудувати коло найменшого діаметра, який містить всі точки набору. Проілюструвати розв'язок на малюнку.

19. Дано набір з n точок площини, заданих координатами: $A_1(X_1, Y_1)$, $A_2(X_2, Y_2)$, ..., $A_n(X_n, Y_n)$. Побудувати прямокутник найменшої площини, який містить всі точки набору. Проілюструвати розв'язок на малюнку.

20. Випуклий n-кутник заданий набором вершин $A_1(X_1, Y_1)$, $A_2(X_2, Y_2)$, ..., $A_n(X_n, Y_n)$.

Розбиття цього багатокутника на трикутники діагоналями, що не перетинаються називається тріангуляція. Вартістю тріангуляції називається сума довжин діагоналей розбиття. Знайти тріангуляцію багатокутника найменшої вартості. Проілюструвати розв'язок на малюнку.

СКЛАДНІ ТИПИ ДАНИХ: ЗАПИСИ І ФАЙЛИ.

1. Складні типи даних у мові Pascal.

Ми вже отримали уявлення про деякі типи даних, які використовуються при програмуванні на Паскалі. Прості типи даних або заздалегідь визначені як стандартні, або визначаються в програмі.

Стандартні типи	Типи, що визначаються програмістом
Integer;	Тип, що перераховується;
Real;	Скалярний тип.
Char:	
Boolean.	

З поняттям простого типу пов'язані:

- ім'я типу;
- множина допустимих значень типу;
- набір операцій, що визначені на типі;
- набір відношень, що визначені на типі;
- набір функцій, що визначені або набувають значення на типі.

З даних простих типів можна конструювати дані складних типів. Єдиним (поки) прикладом складних типів є регулярні типи (значення - масиви). З поняттям складного типу пов'язані:

- ім'я типу;
- спосіб об'єднання даних у дане складного типу;
- спосіб доступу до даних, що утворюють складне дане типу;

Так, визначення

```
Type Vector = array [1..N] of Integer;
Var X : Vector;
```

задає N цілих чисел, що об'єднані в єдине ціле - масив з іменем X. Доступ до компоненти масиву здійснюється за його іменем X і значенням індексного виразу: X[i].

Крім регулярних типів, у мові визначені ще деякі складні типи. Кожне таке визначення задає свій, характерний механізм об'єднання даних у єдине ціле і механізм доступу до даних - компонент складного даного.

Це - комбінований тип (записи), файловий тип (файли), множинний тип (множини) і посилальний тип (посилання).

2. Записи.

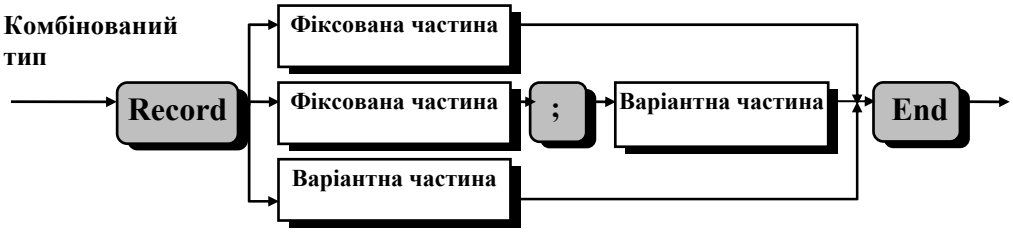
Значеннями так званого комбінованого типу даних є записи. Комбінований тип задає образ структури об'єкта - даного цього типу, кожна частина якого (поле) може мати зовсім різні характеристики.

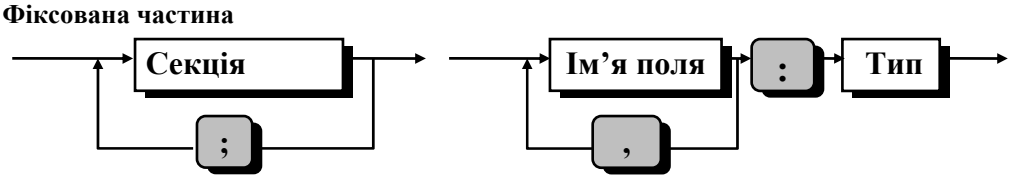
Таким чином, запис - це набір різнотипних даних, що об'єднані спільним іменем. Більш формально, запис містить визначене число компонент, що називаються полями.

У визначенні типу запису задається ім'я і тип кожного поля запису:

<комбінований тип >::= Record < список описань полів > End
<список полів>::= <фіксов. част.> | <фіксов. част.>;<варіант. част.> | <варіант. част.>
<фіксована частина >::= <секція запису > {,<секція запису >}
< секція запису >::= <ім'я поля >{,<ім'я поля >}:< тип > < пусто >

Синтаксис записів, що містять варіантну частину - записів з варіантами - ми визначимо нижче. Відповідні синтаксичні діаграми записів з варіантами:





Приклади.

Приклад 1

Type Complex = Record

```
  Re, Im : Real
 end;
Var z1, z2 : Complex;
```

Приклад 2

```
Type Name = array [1..15] of Char;
          Student = Record
                      F1,F2,F3 : Name;
                      Day : 1..31;
                      Month : 1..12;
                      Year : integer;
                      StudDoc : integer
                    end;
Var Group : array [1..25] of student;
                S : Student;
```

При позначенні компоненти запису в програмі слідом за іменем запису ставиться точка, а потім ім'я відповідного поля. Таким чином здійснюється доступ до цієї компоненти. Наприклад:

```
1)  z1.Re := 2;  z1.Im := 3;
    M := sqrt(sqr(z1.Re) + sqr(z1.Im));
2)  S.F1 := Group[i].F1;
    S.Year := Group[i + 1].Year;
    writeln( Group[i].StudDoc);
```

Запис може входити у склад даних більш складної структури. Можна говорити, наприклад, про масиви і файли, що складаються з записів. Запис може бути полем іншого запису.

Приклад 3

```
Type Name = array[1..20] of Char;
     FullName = Record
       Name1, Name2, Name3 : Name
     end;
     Date = Record
      Day : 1..31;
      Month : 1..12;
      Year : integer
     end;
    Student = Record
     StudName: FullName;
      BirthDay: Date;
```

```
  StudDoc: integer
end;
Var StudGroup : Array [1..30] of Stugent;
    A, B : Student;
```

Наприклад, доступ до поля day змінної A можливий по імені A.BirthDay.Day, а до першої букви поля Name2 імені студента з номером 13 змінної StudGroup - по імені StudGroup[13].StudName.Name2[1]

3.Записи з варіантами.

Синтаксис комбінованого типу містить і варіантну частину, що припускає можливість визначення типу, який містить визначення декількох варіантів структури. Наприклад, запис у комп'ютерному каталозі бібліотеки може мати наступну структуру:

Фіксована частина

Прізвище Ім'я по батькові	{автора}
Назва	{книги}
Видавництво	{його атрибути}
Шифр	{бібліотеки}
Стан	(**видана, у фонді, в архіві**)

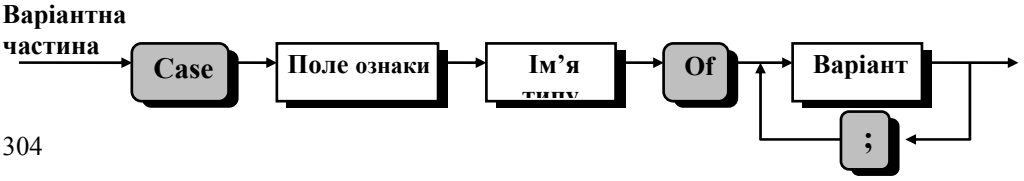
Варіантна частина

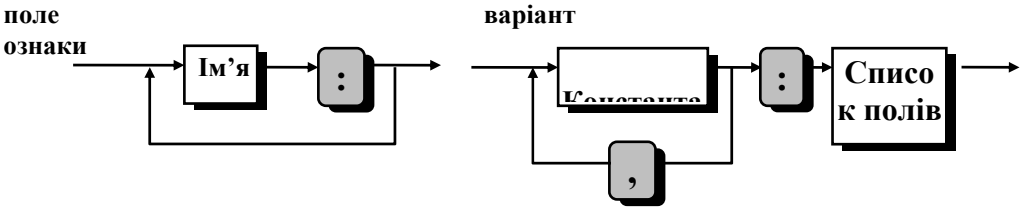
Значення признака	**видана**	**у фонді**	**в архіві**
Поля Варіантної части	Прізвище ім'я, по батькові N% {чит.квитка} Дата {видачі}	адрес {схову}	ім'я {архіву} адрес {схову} дата {архівації}

Синтаксис варіантної частини:

<варіантна частина >::= case <поле ознаки > <ім'я типа > of < варіант >{;<варіант >}
< варіант >::=<список міток варіанта >:(<список полів >) _ < пусто >
<список міток варіанта >::=<мітка варіанта >{;<мітка варіанта >}
<мітка варіанта >::=< константа >
<поле ознаки >::=<ім'я >:< пусто >

Відповідні синтаксичні діаграми:





Описання типу запису в розглянутому прикладі може мати вид:

Приклад 4

```
Book = record
  Author : FullName; {фіксована частина}
BookName: String;
BookCode: Code;
Station : (Readed, inFile, inArchive);
case Station of {поле ознаки}
        Readed: Reader : FullName; {варіантна частина}
        ReadCode : Integer;
        ReadDate : Date;
        inFile: FilAdress : Adress;
        inArc : ArcName : Srting;
        ArcAdress: Adress
    end
end;
```

У нашому прикладі на варіанти вказує поле Station. У залежності від значення цього поля запис має ту чи іншу структуру. Це часта ситуація. Звичайно на варіант запису вказує одне з полів фіксованої частини цього запису. Тому синтаксисом допускається скорочення: опис компоненти, що визначає варіант, (яка називається полем ознаки - дискримінантом), включається в заголовок варіанта. У нашому прикладі 4 це виглядить так:

```
Type BookStation = (Readed, inFile, inArc);
Book = record
        Author : FullName;
        BookName : String;
        BookCode : Code;
        case Station : BookStation of
```

```
            Readed : Reader : FullName;
            ReadCode : Integer;
            ReadDate : Date;
            inFile : FilAdress: Adress;
            inArc : ArcName : String;
            ArcAdress: Adress
  end
end;
```

Всі імена полів повинні бути різними, навіть якщо вони зустрічаються в різних варіантах. (Наприклад, Author, Reader - імена людей, а FilAdress і ArcAdress - адреси, що вказують на місцезнаходження книги на полках сховища). У випадку, коли один з варіантів не містить варіантної частини, він повинен бути оформлений наступним чином:

EmptyVariant :( ) {EmptyVariant - мітка порожнього варіанта}

4.Оператор приєднання.

Якщо A - змінна типу Student із приклада 1, її значення можна змінити групою операторів:

```
A.F1 := ' Іванов '; A.F2 := ' Ілля '; A.F3 := 'Інокентійович ';
A.Day := 14; A.Month := 9; A.Year := 1976;
A.StudDoc := 123;
```

Приведені вище позначення можна скоротити за допомогою оператора приєднання. Заголовок цього оператора відкриває область дії “внутрішніх” імен полів запису, які можуть бути використані як імена змінних. Оператор приєднання має вид:

With <змінна-запис > {,<змінна-запис >} do < оператор >

Приклад 5

```
with A do begin
   F1 := ' Іванов ';  F2 := ' Ілля '; F3 := ' Інокентійович ';
   Day := 14; Month := 9; Year := 1976;
   StudDoc := 123;
 end { оператора with }
```

Більш того, в групі операторів обробки змінної StudGroup[i] (приклад 3) запис може бути сформований наступним чином:

```
 With StudGroup[3], StudName do begin
  Name1 := ' Інокентіївна '; Name2 := ' Інна '; Name3 := ' Іванівна ';
  BirthDay.Day := 14; BirthDay.Month := 9; BirthDay.Year := 1976;
  StudDoc := 123
```

```
 end;
```

Таким чином, оператор виду

With r1,...,rn do S

еквівалентний оператору

With r1 do with r2 ... with rn do S.

Зауваження: В операторі With R do S вираз R не повинен містити змінні, що змінюються в операторі S. Наприклад, оператор With S[j] do j := j + 1 недопустимий!

Як приклад розглянемо програму, яка оброблює масив записів, кожен з яких містить відомості про людину, необхідні для зв'язка з ним по пошті. Обробка запису полягає у пошуку всіх осіб, які народилися в даний день.

Запис має вид:

Повне ім'я (Прізвище Ім'я по батькові)
День народження (День, Місяць, Рік)
Адреса (Індекс, Країна, Місто, Вулиця, № будинку, № квартири)

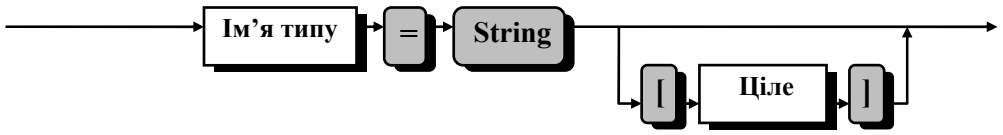
```
Program Adress;
  Const n = 100;
  Type
{ описання типів Name, FullName, Date із приклада 3 }
   Mail = Record Index : LongInt; { довге (4 байта) ціле число }
             Country, Sity, Street : Name;
             House, Flat : Integer
           end;
Person = Record
              Who : FullName;
              When : Date;
              Where : Mail
            end;
  Var
    Today : Date;
    Department : array [1..N] of Person;
    i : Integer;

    Function theSame(var X : Date, var Y : Person): Boolean;
      Begin
        With Y, When do
         theSame := (X.Day = Day) and (X.Month = Month)
       End;
  {Procedure ReadDepartment - введення масиву записів}
```

```
Procedure ReadDate(var Z : Date);
   Begin
     With Z do Read(Day, Month)
    End;
 Procedure WriteMail(var X : Person);
   Begin
     With X, Who do Writeln(Name1, ' ', Name2, ' ', Name3);
     With X, Where do begin
       Writeln(Street, ' ',House, ' ', Flat);
       Writeln(Index, ' ', Sity, ' ', Country);
     end
   End;
 Begin
   ReadDepartment; { введення масиву записів}
   ReadDate(Today);
   For i := 1 to N do
     if theSame(Today, Department[i])
       then WriteMail(Department[i])
   End;
```

5. Рядки і засоби їх обробки.

Значенням рядкового типу даних є рядки. Стандарт мови передбачає використання рядків тільки як констант, що використовуються в операторах виведення Write, Writeln. У розширенні мови Turbo-Pascal рядковий тип даних визначений більш повно.

Визначення рядкового типу слідує діаграмі:



Тут ціле належить діапазону 1..255 і означає максимальну кількість символів у рядку цього типу. Якщо описання типу String використовується без вказівки максимальної кількості символів, це (за замовченням) означає, що під цей тип резервується 255 символів.

Наприклад:

```
Type Name = String[20]; { рядки з 20-ти символів }
  Post = String; { рядки з 255-ти символів }
```

Процедури і функції типу String.

Над рядками визначена операція конкатенації “+”, результат якої - рядок, в якому операнди з’єднанні в порядку їх слідування у виразі. Наприклад:

'Turbo' + 'Pascal' = 'TurboPascal'; 'Turbo_' + 'Pascal ' + 'System' = 'Turbo_Pascal System';

Тому результатом виконання серії операторів

```
X := ' Приклад'; У := ' додавання '; Z := ' рядків'  ;
Writeln(X + Y + Z); Writeln(Y + ' ' + Z + ' ' + X)
```

будуть виведені на екран рядки

Прикладдодаваннярядків
додавання рядків Приклад

Тип String допускає і пустий рядок - рядок, який не містить символів: EmptyStr := '' {лапки, що йдуть підряд}. Вона грає роль нуля (нейтрального елемента) операції конкатенації: EmptyStr + X = X + EmptyStr = X

Над рядками визначені також відношення (операції логічного типу)

" = ", " <> ", " < ", " > ", " <= ", " >= ".

Таким чином, кожний із рядкових типів упорядкований, причому лексикографічно. Це означає, що

а) порядок на рядках погоджений з порядком, заданим на символьному типі (Char);

б) порівняння двох рядків здійснюється посимвольно, починаючи з перших символів;

в) якщо рядок А є початком рядка В, то А < В;

г) пустий рядок - найменший елемент типу.

Наприклад:

а) 'c' < 'k', так як Ord(‘c’) < Ord(‘k’);

б) 'abc' < 'abk', так як перші два символи рядків співпадають, а порівняння третіх дає Ord(‘c’) < Ord(‘k’);

в) 'abc' < 'abkd', так як перші два символи рядків співпадають, а порівняння третіх дає Ord(‘c’) < Ord(k);

г) 'ab' < 'abcd', так як рядок 'ab'- початок рядка 'abcd'.

На рядковому типі даних визначені:

Функції:

а) Length(X: String): Byte; - довжина рядка X; { Length(EmptyStr) = 0 }

б) Pos(Y:String; X:String):Byte; - позиція першого символу першого зліва входження підрядка Y у рядок X. Якщо X не містить Y, Pos(Y, X) = 0.

в) Copy(X:String; Index, Count: Integer):String - підрядок рядка X, що починається з позиції Index і містить Count символів.

г) Concat(X1, X2, .., Xk: String): String; - конкатенація рядків X1, X2, .., Xk. Інша форма запису суми X1+X2+ .. +Xk.

Процедури:

д) Delete(var X: String; Index, Count: Integer); З рядка X видаляється Count символів, починаючи з позиції Index. Результат поміщується в змінну X.

е) Insert(Y:string; var X: String; Index: Integer); В строку X вставляється рядок Y, причому вставка здійснюється починаючи з позиції Index.

Стандартні процедури введення-виведення Паскаля розширені для введення-виведення рядків. Відмітимо, однак, що для введення декількох рядкових даних треба користуватись оператором Readln. Оператор Read у цих випадках може вести себе непередбачено.

Приклад 2. Дано масив A[1..n] of string[20]. Скласти програму заміни всіх перших входжень підрядка L в A[i] на підрядок R. Рядки L і R вводяться з клавіатури в виді рівності L = R. Результат замін відобразити у масив, елементи якого - рівності виду:

A[i]=результат заміни L на R в A[i].

```
Program RewriteArray;
  Const n = 100; Single = 20; Double = 41;
  Type
          Sitem = string[Single];    Ditem = string[Double];
       SWordArray = array[1..n] of Sitem;   DWordArray = array[1..n] of Ditem;
  Var
          A: SWordArray;    B: DWordArray;
          L, R: Sitem;   X : Sitem;
          i, Index : Integer;
 Procedure InpWord(var U, V : Sitem);
   Var X : Ditem;
       j : Integer;
 Begin
  Writeln('_________ Введення рівності L = R ___________');
  Read(X); j := Pos('=', X);
  U := Copy(X, 1, j - 1);
  V := Copy(X, j + 1, Length(X))
  End;


Procedure InpArray;
 begin
  Writeln('====== Введення масиву слів ======');
  For i:=1 to n do Readln(A[i])
 end;
Procedure OutArray;
 begin
  Writeln('====== Виведення масиву слів ====');
  For i:=1 to n do Writeln(B[i])
end;
Begin
 InpArray;                    {введення масиву слів з клавіатури}
 InpWord(L, R);     {введення і аналіз рівності L = R}
```

```
 For i := 1 to n do begin
  X := A[i]; Index := Pos(L, X);
  If Index <> 0
    then begin
      Delete(X, Index, Length(L));
      Insert(R, X, Index)
    end;
   B[i] := A[i] + '=' + X
 end;
 OutArray;                    {виведення масиву слів до друку}
End.
```

6.Файли. Управління файлами.

Програма, яка написана мовою Pascal, повинна якимось чином обмінюватись даними з зовнішніми пристроями (отримувати дані з клавіатури, магнітного диска, виводити дані на екран, принтер і т.д.) Для роботи з зовнішніми пристроями використовуються файли. Файли - це значення файлового типу даних - ще одного стандартного складного типу в мові.

(Послідовний) файл - це послідовність однотипних компонент, яка має ознаку кінця і оброблюється послідовно - від початку до кінця.

Порядок компонент визначається самою послідовністю, подібно до того, як порядок слідування чергового кадру кінофільму визначається його розташуванням на кіноплівці. У будь-який момент часу доступний тільки один елемент файла (кадр кінофільму). Інші компоненти доступні тільки шляхом послідовного просування по файлу.

У результаті виконання програми відбувається перетворення одного текстового файла (який називається Input) в інший текстовий файл (який називається Output). Обидва ці файли є стандартними і використовуються для введення /виведення даних.

Над файлами можна виконувати два явних виду дій:

3.Перегляд (читання) файла.

4.Створення (запис) файла - виконується шляхом приєднання нових компонент у кінець початково порожнього файла.

Файли, з якими працює програма, повинні бути описані в програмі. Частина файлів (що уявляють собою фізичні пристрої) має в операційній системі стандартні імена. Наприклад, для читання даних з клавіатури і виведення результатів на екран монітора ми користуємось стандартними файлами Input і Output. Файл - принтер має ім'я Prn:

Імена нестандартних файлів, що використовуються в програмі, необхідно описувати у розділі змінних. Описання файлового типу відповідає діаграмі:

Файл, компоненти якого є символами, називається текстовим. Він має стандартний тип Text:

```
Type Text = File of Char;
```

Приклади:

```
   Type
ExtClass = File of Person;CardList = File of Integer;
    Var
F : Text;
Data : File of real;
List1, List2 : CardList;
Class10A, Class10B : ExtClass;
```

Для роботи з нестандартними файлами ім'я файла повинно бути зв'язане з реально існуючим об'єктом - зовнішнім пристроєм. Саме, якщо нам необхідно обробити дані з файла, що знаходиться на магнітному диску і який має (зовнішнє) ім'я D:\ExtName.dat, ми повинні повідомити системі, що працюючи з файлом IntName (читаючи з нього дані або записуючи у нього дані), ми працюємо з файлом ExtName.dat, що знаходиться на диску D:.

Для ототожнення внутрішнього імені файла з зовнішнім іменем використовується процедура **Assign**(< внутрішнє ім'я >, < 'зовнішнє ім'я ' >).

Після того, як ім'я файла описано і визначено, можна приступити до роботи з файлом. При використанні нестандартних файлів треба пам'ятати, що перед роботою необхідно відкрити їх, тобто зробити доступними з програми. Для цього треба застосувати одну з двох наступних процедур:

Процедура **Rewrite**(<ім'я файла >) - відкриває файл для запису. Якщо файл раніше існував, всі дані, що зберігались у ньому, знищуються. Файл готовий до запису першої компоненти.

Процедура **Reset**(<ім'я файла >) - відкриває файл для читання. Файл готовий для читання з нього першої компоненти.

По закінченню роботи з файлом (на запис) він повинен бути закритий. Для цього використовується процедура **Close**(<ім'я файла >). Ця процедура виконує всі необхідні машинні маніпуляції, що забезпечують збереження даних у файлі.

Для обміну даними з файлами використовують процедури Read і Write.

Процедура **Read** (<ім'я файла >,<список введення >), читає дані з файла (по замовченню ім'я файла - Input). Список введення - це список змінних.

Процедура **Write** (<ім'я файла >,<список виведення >), записує дані у файл (по замовченню ім'я файла - Output). Список виведення - це список виразів.

Якщо F - файл типу Text, то у списку введення/виведення допустимі змінні/вирази типу Integer, Real, Char, String[N]. В інших випадках типи всіх компонент списку повинні співпадати з типом компоненти файла.

При роботі з файлами застосовують стандартні логічні функції:

Eof(F) (end of file) - стандартна функція - признак кінця файла. Якщо файл F вичерпаний, то Eof(F) = True, в протилежному випадку Eof(F) = False.

Eoln(F) (End of line) - стандартна функція - признак кінця рядка текстового файла. Якщо рядок текстового файла F вичерпаний, то Eoln(F) = True, в протилежному випадку Eoln(F) = False. Функція Eoln визначена тільки для файлів типа Text. Звичайно в програмах використовують або текстові файли, або файли, компонентами яких є структуровані дані (наприклад, записи).

Треба пам'ятати, що дані файлових типів неможна використовувати в якості компонент інших структур даних. Наприклад, неможна визначити масив, компонентами якого є файли, запис, полем якої є файл.

Приклад 3. Програма формування файла як вибірки з компонент іншого файла. Нехай F - файл записів виду (поле ключа, поле даних). Треба сформувати файл G, який містить ті компоненти F, ключі яких задовольняють умові: значення ключа - ціле число з інтервалу]Max, Min [.

```
Program FileForm;
   Type Component = Record
          Key: Integer; { поле ключа }
          Data: String[20] { поле даних}
         End;
          OurFile = File of Component;
               Var F, G : OurFile;  X: Component;
          Max, Min : Integer;
  Function Correct(var U: Component): Boolean;
   begin
    Correct := (U.Key < Max) and (U.Key > Min)
   end;

  Begin
   Read(Max, Min);
   Assign(F, 'D:InpFile.dat');                {визначення значення F }
   Assign(G, 'D:OutFile.dat');        {визначення значення G }
   Reset(F);                                  {файл F відкритий для читання}
   Rewrite(G);                        {файл G відкритий для запису}
   While not(Eof(F)) do begin         {цикл читання F - запису G}
    Read(F, X);
    If Correct(X) then Write(G, X)
   end;
   Close(G)                                   {файл G закритий}
  End.
```

7.Основні задачі обробки файлів.

Специфіка файлового типу, яка пов'язана з послідовним доступом до компонент і розташуванням файлів на зовнішніх носіях, накладає жорсткі обмеження на засоби розв'язку задач обробки файлів. Розглянемо деякі такі задачі. Для визначеності будемо вважати, що всі файли мають тип OurFile із приклада 3 і впорядковані за значенням ключового поля Key (ключу).

Задача 1. Доповнення елемента до файла. Дано файл F і елемент X типу Component. Розширити F, включивши в нього компоненту X з збереженням упорядкованості.

Ось, напевно, єдиний можливий розв'язок:

- Переписувати покомпонентно F у новий файл G до тих пір, поки F^.Key < X.Key ;
- Записати X у G;
- Переписати "хвіст" файла F і G;
- Перейменувати G у F .

Задача 2. Вилучення елемента з файла. Дано файл F і число K - значення ключа елементів, що вилучаються. Скоротити F, вилучивши із нього всі компоненти з ключем K.

Розв'язок аналогічний розв'язку задачі 1:

- Переписувати покомпонентно F у новий файл G до тих пір, поки F^.Key < X.Key;
- Поки F^.Key= K читати F;
- Переписати "хвіст" файла F у G;
- Перейменувати G у F.

Відмітимо, що:

$\Rightarrow$ Розв'язки цих задач потребують послідовного пошуку міста елемента X як компоненти файла. Ефективний розв'язок задачі пошуку (наприклад, бінарний пошук) неможливий.

$\Rightarrow$ У якості вихідного використовується новий файл, оскільки читання/запис в один і той же файл неможливі!

Наступні задачі присвячені обробці двох файлів.

Задача 3. Злиття (об'єднання) файлів. Дано файли F і G. Треба сформувати файл H, який містить всі компоненти як F, так і G.

Алгоритм полягає у послідовному і почерговому перегляді файлів F і G і запису чергової компоненти в H. Почерговість визначається порівнянням значень ключів компонент F і G. Оформимо алгоритм у виді процедури:

```
Procedure FileMerge(var F, G, H: OurFile);
    Var X, Y : Component;
        Flag : Boolean;
```

```
Procedure Step(var U:OurFile; var A, B:Component);
    begin
     Write(H, A);
       If Eof(U)
         then begin Write(H, B); Flag := False end
         else Read(U, A)
    end;

Procedure AppendTail(var U: Ourfile);
     Var A: Component;
     Begin
       While not(Eof(U)) do begin
       Read(U, A); Write(H, A)
      end
     end;

  Begin
    Reset(F); Reset(G); Rewrite(H);
    Flag := True;
    Read(F, X); Read(G, Y);
    While Flag do
     If X.Key < Y.Key
       then Step(F, X, Y)
       else Step(G, Y, X);
    AppendTail(F);
    AppendTail(G);
    Close(H)
  End;
```

Задача 4. Перетин файлів. Дано файли F і G. Треба сформувати файл H, який містить всі компоненти, що входять як у F, так і в G.

Задача 5. Віднімання файлів. Дано файли F і G. Треба сформувати файл H, який містить всі компоненти, що входять у F, але не входять у G.
Розв'язок цих задач аналогічний розв'язку задачі злиття файлів.

8. Сортування файлів.

Упорядкованість компонент файла за одним або кількома ключовими полями - одна з основних умов ефективної реалізації задач обробки файлів. Так, задача роздруку

файла у визначеному порядку слідування компонент, якщо файл не впорядкований відповідним чином, розв'язується за допомогою багатократних переглядів (прогонів) файла. Кількість прогонів при цьому пропорційна кількості компонент.

Відсутність прямого доступу до компонент приводить до того, що розглянуті вище алгоритми сортувань масиву неможливо ефективно адаптувати для сортування файла. На відміну від масивів, основні критерії ефективності алгоритму сортування файла - кількість прогонів файлів і кількість проміжних файлів.

Так, наприклад, алгоритм сортування простими обмінами потребує N прогонів файла, що сортується (N - кількість компонент файла). Алгоритм швидкого сортування взагалі не має сенсу розглядати, оскільки при його реалізації необхідно було би читати файл від кінця до початку!

Розглянемо тому новий для нас алгоритм - алгоритм сортування злиттям, який найбільш ефективний при сортуванні файлів і відноситься до швидких алгоритмів при сортуванні масивів, хоча і потребує додаткової пам'яті.

Алгоритм сортування злиттям.

Нехай послідовність, що сортується, $F = \{ f_1, ..., f_N \}$ представлена в виді двох уже відсортованих половин - F_1 і F_2. Тоді для сортування F достатньо злити (тобто застосувати аналог алгоритму злиття FileMerge) послідовності F_1 і F_2 у вихідну послідовність G. Оскільки упорядковані половинки F_1 і F_2 можна отримати за допомогою тих же дій, ми можемо описати алгоритм сортування злиттям рекурсивно. При цьому, очевидно, необхідно використовувати оптимальну кількість проміжних файлів. Оскільки злиття треба починати з однокомпонентних файлів, точна реалізація описаного алгоритму потребує 2N додаткових файлів, що, звичайно, неприйнятно. Тому ми повинні використовувати один файл для збереження декількох послідовностей, що зливаються. Уточнимо цю ідею:

Нехай файл F, що сортується, містить k-елементні упорядковані підпослідовності $A_1, A_2, ..., A_{2l}$, $k = 2l$.

1.Розділ F полягає у переписі послідовностей Aj з парними номерами в файл F_1, а з непарними номерами - в файл F_2.

Нехай файл F_1 містить k-елементні упорядковані підпослідовності $B_1, B_3, ..., B_{2l-1}$, а F_2 - k-елементні упорядковані підпослідовності $B_2, B_4, ..., B_{2l}$.

2.Злиття F_1 і F_2 полягає в злиттях пар $<B_i, B_{i+1}>$ у підпослідовності $A_{(i+1) \text{ div } 2}$ у файл F. Після обробки F буде містити l 2k-елементних підпослідовностей A_j.

Якщо N - степінь 2-х ($N = 2^m$), то після m-кратного повторення розділення-злиття, починаючи з $A_1=\{f_1\}, ..., A_N=\{f_N\}$, файл F буде містити відсортовану послідовність. У загальному випадку, коли $N \neq 2^m$, керування злиттями-розділеннями трохи ускладнюється:

- при розділі кількість підпослідовностей може опинитися непарною;
- при злитті остання за рахунком підпослідовність одного з файлів може мати меншу, ніж усі інші, кількість елементів.

```
Program MergeSort;
```

```
    Type Component = Record
            Key: Integer; { поле ключа }
            Data: String[20] { поле даних}
          End;
        OurFile = File of Component;
     Var F, F1, F2 : OurFile;
       k, L, t, SizeF: Integer;
{Procedure FileOut(var F :OurFile);}
{Procedure FileInp(var F: Ourfile);}
      Procedure CopyBlock(Var U, V: OurFile; k: Integer);
           Var i: Integer;
               X: Component;
         Begin
           For i := 1 to k do begin
            Read(U, X); Write(V, X)
           end
         End;
{ розділення блоками по k компонент F ==> F1, F2 }
      Procedure Partition(k: Integer);
           Var i: Integer;
         Begin
           Reset(F); Rewrite(F1); Rewrite(F2);
           For i:= 1 to L do
            If Odd(i)
              then CopyBlock(F, F1, k)
              else CopyBlock(F, F2, k);
            If Odd(L)
              then CopyBlock(F, F2, t)
              else CopyBlock(F, F1, t);
              Close(F1); Close(F2)
         End;
{ злиття блоками по k компонент F1, F2 ==> F }
      Procedure Merge(k: Integer);
          Var i: Integer;
      Procedure MergeBlock(t: Integer);
           Var count1, count2: Integer;
               X1, X2: Component;
              Flag: Boolean;
       Procedure AppendTail(var U: Ourfile; var p: Integer; s: Integer);
           Var A: Component;
         Begin
```

```
            While p <= s do begin
            Read(U, A); Write(F, A); p := p + 1
          end;
        End { AppendTail };
Procedure Step(var U: OurFile; Var A, B: Component; var p, q: Integer; s: Integer);
          begin
            Write(F, A); p := p + 1;
            If p <= s
              then Read(U, A)
               else begin  Write(F, B); q := q + 1; Flag := False end
         end { Step };
      Begin { MergeBlock }
        count1 := 1; count2 := 1; Flag := True;
        Read(F1, X1); Read(F2, X2);
        While Flag do
          If X1.Key < X2.Key
            then Step(F1, X1, X2, count1, count2, k)
            else Step(F2, X2, X1, count2, count1, t);
        AppendTail(F1, count1, k); AppendTail(F2, count2, t);
      end { MergeBlock };
      Begin { Merge }
        Rewrite(F); Reset(F1); Reset(F2);
        For i := 1 to (L div 2) do MergeBlock(k);
          If t<> 0
            then if Odd(L)
              then MergeBlock(t)
              else CopyBlock(F1, F, t)
            else if Odd(L)
              then CopyBlock(F1, F, k)
    End { Merge };
    Procedure FirstPartition;
          Var X : Component;
       Begin
        Reset(F); Rewrite(F1); Rewrite(F2);
        SizeF := 0;
        While not(Eof(F)) do begin
          Read(F, X); SizeF := Succ(SizeF);
          If Odd(SizeF)
             then Write(F1, X)
            else Write(F2, X)
         end;
```

```
    Close(F1); Close(F2)
  End { FirstPartition };
 Begin                          {Визначення зовнішніх імен файлів F, F1, F2}
    FileInp(F);
  FirstPartition;      {перше розділення підрахунком SizeF}
  L := SizeF; t := 0;
  Merge(1);                     {перше злиття1-блоків}
  k := 2;
  While k < SizeF do begin
    L := SizeF div k;           {кількість повних k-блоків у F}
    t := SizeF mod k;           { розмір неповного k-блока }
      Partition(k);
    Merge(k);
    k := 2*k
  end;
 FileOut(F);
End.
```

Оцінимо складність алгоритму в термінах C(n), M(n), L(n), де L(n) - число прогонів файла F і n = SizeF.

1.Оцінка L(n).

L(n)= число розділень + число злить. Кожне розділення - виклик процедури Partition, а злиття - виклик Merge. Тому L(n) - подвоєне число повторень тіла циклу While. Звідси, оскільки змінна k, що керує циклом, кожний раз збільшується вдвічі, на L-тому кроку $k = 2^L$, і, отже, L - найбільше число таке, що $2^L < n$, тобто $L = [\log_2 n]$.

$$L(n) = 2[\log_2 n]$$

2.Оцінка C(n).

Порівняння компонент за ключем відбуваються при злитті. Після кожного порівняння виконується процедура Step, яка записує одну компоненту в F. Тому при кожному злитті кількість порівнянь не перевищує n. Звідси $C(n) \le L(n)*n/2$, тобто

$$C(n) \le n [\log_2 n]$$

3.Оцінка M(n).

Процедури Partition і Merge пересилають компоненти файлів. Незалежно від значень ключів, виклик кожної з них або читає F, або записує F. Тому M(n) = nL(n), тобто

$$M(n) = 2n[\log_2 n]$$

Отримані оцінки дозволяють класифікувати алгоритм як ефективний алгоритм сортування послідовних файлів. Його також можна адаптувати до сортування масивів.

Алгоритм сортування злиттям може бути поліпшений кількома способами. Розглянемо лише деякі з них:

а. Помітимо (зауважимо), що процедура Partition носить допоміжний характер. Не аналізуючи ключів, вона просто формує файли F_1 і F_2 для злиття. Тому її можна вилучити, якщо процедуру Merge примусити формувати не F, а одразу F_1 і F_2. При цьому, звичайно, кількість файлів у програмі збільшується. Саме:

1.F розділимо на (F_1, F_2).
2.Визначимо допоміжні файли G_1, G_2
3.Основний цикл алгоритму:
Злиття (F_1, F_2) ===> (G_1, G_2);
Злиття (G_1, G_2) ===> (F_1, F_2);
(Непарні пари блоків зливаємо на 1-ий файл, парні - на 2-ий).
Часова складність алгоритму поліпшується майже вдвічі.

б. Зауважимо, що на початковій стадії роботи алгоритму розміри блоків малі. Їх можна сортувати в оперативній пам'яті, використовуючи представлення в виді масиву і швидкі алгоритми сортування масивів. Таким чином, треба змінити процедуру FirstPartition, визначив її як процедуру внутрішнього сортування k_0-блоків при деякому (максимально можливому) значенні k_0. Цикл While основної програми тепер можна починати з $k = k_0$.

в. В реальних файлах часто зустрічаються вже упорядковані ділянки компонент. Тому файл можна початково розглядати як послідовність упорядкованих ділянок, і зливати не блоки фіксованого розміру, а упорядковані ділянки. Такі сортування називають природними.

9.Задача корегування файла.

Задача корегування файла є одною з основних задач обробки файлів. Розглянемо її формулювання:

Вихідні дані задачі - файл F, що корегується, файл корегувань G. Результат - відкорегований файл H. Будемо вважати, що файл F має тип OurFile. Тоді файл H має той же тип. Файл корегувань G складається з записів з варіантами:

```
Type
    CorComponent = record
                       Key: Integer;
                       job: (Include, Change, Exclude); {включити, змінити,
                  вилучити}
                       Case job of
                         Include, Change: Data: String[20];  Exclude: ()
                        end
                       End;
     CorFile = File of CorComponent;
Var G : CorFile;
```

Це значить, що файл корегувань містить компоненти, які треба включити в основний файл, компоненти, які треба вилучити з файла і компоненти, зміст яких треба змінити (з врахуванням змісту як основного файла, так і файла корегувань).

Файл F вважаємо впорядкованим (за ключем), а файл G, взагалі кажучи, ні. Результатом повинен бути упорядкований відкорегований файл H.

Розв'язок задачі звичайно містить два етапи:

а) Сортування файла корегувань G;

б) Узагальнене злиття файлів F, G у H.

На практиці файл G може бути невеликим. У цьому випадку застосовують внутрішнє сортування. Узагальнений алгоритм злиття робить всі три варіанта обробки файла F за один прогін.

На завершення відзначимо, що сучасні ЕОМ рідко активно використовують зовнішні носії з послідовним доступом у розв'язках задач керування базами даних, тому структури даних, у яких зберігається інформація, як правило, більш складні, ніж послідовні файли.

10. Задачі і вправи.

Файли.

7.Розробіть програму Добавлення елемента до файла.

8.Розробіть програму Вилучення елемента з файла.

9.Розробіть програму Перетин файлів.

10.Розробіть програму Віднімання файлів.

11.Розробіть програму Корегування файлів.

12.Розробіть програму корегування упорядкованих файлів, у яких проводиться зміна значень ключових полів. Файл корегувань містить пари

< старий ключ, новий ключ >.

7.Реалізуйте поліпшення а) алгоритму сортування злиттям.

8.Реалізуйте поліпшення в) алгоритму сортування - сортування природним злиттям.

Записи.

11.Опишіть тип запису - клітинки розкладу занять на факультеті для своєї спеціальності і курсу. Сформуйте файл двохтижневого розкладу для своєї підгрупи. Розробіть програму, яка визначає кількість лекційних, практичних і лабораторних занять у двохтижневому циклі для своєї підгрупи за вказаною дисципліною.

12.Опишіть тип запису - відомості про студента групи, необхідні декану факультету. Сформуйте файл студентів своєї підгрупи. Розробіть програму, яка визначає стан успішності в підгрупі.

13.Опишіть тип запису - відомості про батьків учнів класу, необхідні класному керівнику. Сформуйте файл, що складається не менш, ніж з восьми учнів “Вашого” класу. Розробіть програму, яка за прізвищем і іменем учня друкує відомості про його батьків.

14.Опишіть тип запису - відомості про успішність учня, необхідні для вчителя-предметника зі свого предмета. Сформуйте файл, що складається не менш, ніж з восьми учнів “Вашого” класу. Розробіть програму, яка визначає самого слабшого і самого сильного учня класу.

15.Опишіть тип запису - рядок залікової книжки (екзаменаційна частина). Сформуйте файл іспитів, зданих Вами. Розробіть програму, яка визначає середній бал, складає список екзаменаторів і по номеру семестру роздруковує результати Вашої сесії.

16.Опишіть тип запису - рядок залікової книжки (залікова частина). Сформуйте файл заліків, зданих Вами. Розробіть програму, яка визначає дні, коли Ви здавали по два і більш заліків.

17.Опишіть тип запису - відомості про вік, зріст і вагу учня. Сформуйте файл, що складається не менш, ніж з восьми учнів “Вашого класу”. Розробіть програму, яка визначає всіх учнів, що народилися в даний проміжок часу, вказаний датами початку і кінця і визначає середній зріст і середню вагу цієї групи учнів.

18.Опишіть тип запису - відомості про книгу (наприклад, з інформатики). Сформуйте файл книг, необхідних учителю інформатики. Складіть програму, яка підбирає книги для курсу, номер якого вводиться, друкує імена їх авторів і рік видання.

19.Опишіть тип запису - відомості про товар у магазині. Сформуйте файл товарів, що є в магазині. Розробіть програму корегування масиву товарів і визначення виручки магазину на даний момент часу.

20.Опишіть тип запису - рядок у телефонній книзі. Сформуйте файл записів - вашу телефонну записну книжку. Розробіть програму пошуку номера телефона за прізвищем і пошуку адреси за номером телефона.

Файли рядків (слів).

10. Знайти в файлі F всі оборотні слова і скласти з них файл G.
11. Знайти в файлі F входження слова p, замінити їх на слово q, отримавши новий файл G.
12. Знайти в файлі F всі слова, що представляють числа (у десятковому запису) і отримати числовий файл G, що містить знайдені числа.
13. Знайти в файлі F всі однобуквенні слова і зайві пробіли, і, вилучивши їх, отримати новий файл G.

14. Знайти в файлі F всі слова, що зустрічаються більш одного разу, і скласти файл G, вилучивши з F знайдені слова.
15. У файлі F знайти всі слова, що містять здвоєні букви і скласти з них файл G.
16. Знайти в файлі F всі слова, що містять підслово p і скласти з них файл G.
17. Дано файл F. Відсортувати його в алфавітному порядку.
18. Дано слово p і файл F. Знайти в F всі слова, які можна скласти з букв слова p.

МНОЖИНИ.

1.Множинний тип.

Ще одним складним стандартним типом даних, визначеним у мові Pascal, є множинний тип. Значенням множинного типу даних є множина, що складається з однотипних елементів. Тип елемента множини називається базовим типом. Базовим типом може бути скалярний або обмежений тип. Таким чином, множина значень множинного типу - це множина всіх підмножин базового типу, враховуючи і порожню множину. Якщо базовий тип містить N елементів, відповідний множинний тип буде містити 2^N елементів.

Характерна відміна множинного типу - визначення на ньому найбільш поширених теоретико-множинних операцій і відношень. Це робить множинний тип схожим на прості типи даних. Множинні типи описуються в розділі типів наступним чином:

Type < ім'я типу > = Set of < базовий тип >

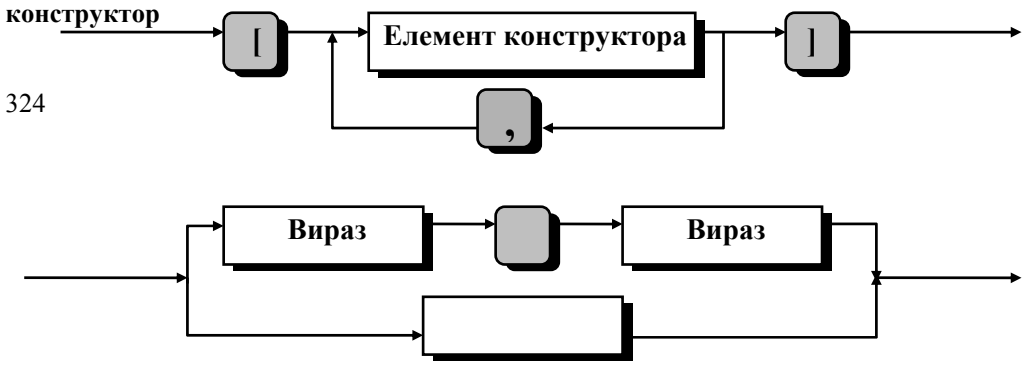
Наприклад,

а) Type Beta = Set of 100..200;
б) Type Glas = Set of char ; {Vowel}
в) Type Color = (red, orange, yellow, green, light_blue, blue, violet);
Paint = Set of Color;
г) Type TwoDigNum = Set of 10..99;

Var A, B: Beta;
llet, flet: Glas;
last, first: Paint;
Sinit: TwoDigNum;

2.Конструктор множини.

Множини будуються з своїх елементів за допомогою конструктора множин. Конструктор представляє собою перелік через кому елементів множини або відрізків базового типу, взятий у дужки [,]. Порожня множина позначається через [ ].



Елемент конструктора

Наприклад:

[ ] - порожня множина
[2, 5 ..7] - множина {2, 5, 6, 7}
['A'..'Z', 'O'..'9'] - множина, що складається з всіх великих латинських букв і цифр
[i + j .. i + 2*j] - множина, що складається з всіх цілих чисел між i + j і i + 2j
Відмітимо, що якщо у виразі [v1..v2] v1 > v2, множина [v1 .. v2] - порожня.

3. Операції і відношення.

До операндів - однотипних множин A і B можна застосувати такі дії:

A + B - об'єднання $A \cup B$

A * B - перетин $A \cap B$

A - B - різниця $A \setminus B$

Між A і B визначені також відношення порядку і рівності

A = B, A <> B, A < B, A <= B, A > B, A >= B;

Відношення порядку інтерпретуються як теоретико-множинні включення.

Якщо A - множина і x - елемент базового типу, то визначено відношення належності **x in A** - x належить A ($x \in A$).

Кожне з відношень, описаних вище, по суті є операцією, результат якої має тип Boolean. Таким чином, якщо Init - змінна типу Boolean, можливе присвоювання Init := A < B. Можливі такі порівняння (A = B) = (C = D).

Наявність операцій над множинами дозволяє застосовувати в програмах оператори присвоювання, в лівій частині яких стоїть змінна типу множини, а в правій - вираз того ж типу. Наприклад:

A := A * [1 .. 10] + B ; B := (A + B)*['A' .. 'Z'] ;

4. Застосування множин у програмуванні.

При реалізації мови розміри множин завжди обмежені константою, що залежить від реалізації. Звичайно ця константа кратна довжині машинного слова. Це відбувається тому, що множини реалізовані в виді логічних (двійкових) векторів наступним чином: кожній координаті двійкового вектора однозначно відповідає один з елементів базового типу.

Якщо елемент a належить множині A, що представляється, то значення координати вектора, відповідне a, дорівнює 1. У протилежному випадку значення відповідної координати дорівнює 0.

Наприклад, якщо множина A описана як Set of 0..15, то його представляє 16-ти мірний двійковий вектор, координати якого перенумеровані від 0 до 15, і i-тій координаті відповідає елемент i базового типу.

```
Базовий тип :          0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
Двійковий вектор :     0  0  1  1  0  1  0  1  0  0   0   1   0   1   0   0
Представлена множина : A = [2, 3, 5, 7, 11, 13]
```

Такий спосіб реалізації дозволяє швидко виконувати операції над множинами і перевірки теоретико-множинних відношень. Тому, наприклад, замість

```
For X := 'A' to 'Z' do
        If (X ='A') or (X ='E') or  (X ='I') or (X ='O') or (X='U')
          then Statement1
          else Statement2
```

краще написати

```
        For X := 'A' to 'Z' do
          If X in ['A','E','I','O','U']
          then Statement1
          else Statement2
```

Остання форма запису не тільки краще читається, але й значно швидше обчислюється.

У системі Turbo-Pascal максимальна кількість елементів у множині дорівнює 256. Таким чином, у якості базового типу можна вибирати, наприклад, Char або відрізок 0..255. У завершення розділу наведемо приклад програми, що використовує множинні типи даних.

Приклад. Побудувати множину всіх простих чисел з відрізка 2..n ($n \leq 255$).

Метод, за допомогою якого ми це зробимо, відомий як “Решето Ератосфена”. Суть цього метода у наступному: Нехай Prime - множина простих чисел, що будується, і Grating - множина, що називається решетом. Алгоритм починає роботу з Prime = [ ]; Grating = [2..n].

Крок основного циклу:

а. Найменший елемент Grating помістити у Prime;

б. Вилучити з Grating всі числа, кратні цьому елементу;

Алгоритм закінчує роботу при Grating = [ ]

```
Program EratosfenGrating;
   Const n = 255;
     Var Grating, Prime: set of 2 .. n ;
         i, Min : integer ;
```

```
Begin
  Grating := [2 .. n] ; Prime := [] ; Min := 2;    {ініціалізація}
  While Grating <> [] do begin        {основний цикл}
    While not(Min in Grating) do      {пошук найменшого елемента в решеті}
      Min := Min + 1;
      Prime := Prime + [Min] ;        {поповнення множин простих чисел}
      For i := 1 to n div Min do      {вилучення кратних із решета}
        Grating := Grating - [i*Min];
  end;
  Writeln('Primes: ');                          {виведення множин простих чисел}
   For i := 1 to n do
     If i in Prime then write(i, ', ')
End.
```

Відмітимо, що доступ до елемента множини у мові не передбачений. У цьому - ще одна якісна відміна множинного типу від інших типів даних. Тому наприклад, для введення множини Prime доводиться перебирати всі елементи базового типу і кожний із них перевіряти на належність Prime.

5. Задачі і вправи.

5.Записати за допомогою конструктора множину X, яка складена з латинських букв a, b, c, d, i, j, k, x, y, z.

6.Записати за допомогою конструктора множину з трьох основних кольорів множинного типу Paint.

7.Записати за допомогою конструктора множину цілих розв'язків квадратної нерівності $x^{\wedge 2} + p*x + q < 0$ у припущенні, що корні відповідного квадратного рівняння лежать в інтервалі [0; 255].

8.Записати за допомогою конструктора множину простих чисел-близнюків з інтервалу 1..30.

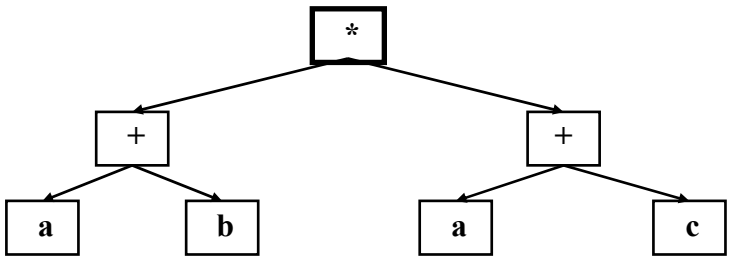
ДИНАМІЧНІ СТРУКТУРИ ДАНИХ.

1. Динамічні інформаційні структури.

У попередніх параграфах були визначені фундаментальні структури даних, що використовуються у процедурному програмуванні: масиви, записи, файли і множини. Фундаментальність цих структур означає, що вони, по-перше, частіше всього використовуються в практиці програмування, і, по-друге, визначають методи структурування даних - тобто методи утворення складних структур із більш простих. Наприклад, можна визначити масив із записів, запис, що складається з множин, файл із масивів, компоненти яких - записи, і т.д. Для кожної з таких структур даних характерна та обставина, що розмір пам'яті, що відводиться для неї, визначається компілятором під час компіляції розділів типів і змінних і залишається незмінним під час виконання програми. Тому такі змінні-структури називаються статичними. Наряду з статичним розподілом пам'яті ми вже використовували в програмах і динамічний розподіл пам'яті - під локальні змінні процедур і функцій. Особливо випукло динамізм тут проявляється при використанні рекурсії.

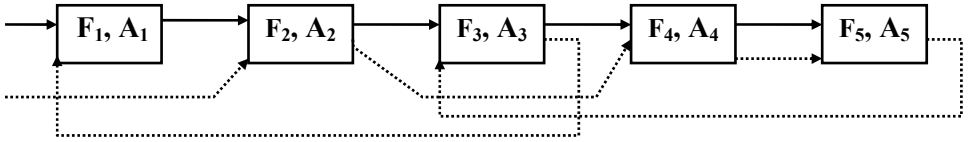
Однак багато задач для своєї ефективної реалізації потребують явних методів динамічного використання пам'яті, тобто описання таких структур даних, розмір і конфігурація яких змінюються під час виконання програм. Такі структури даних називаються динамічними.

Приклад 1. Уявимо собі, що наша програма повинна деяким чином обробляти послідовність символів, яка представляє математичну формулу (арифметичний вираз). Якщо обробка пов'язана з обчисленням значення цієї формули, то представлення формули в виді рядка символів неприродно. Зручніше представити, наприклад, формулу f = (a + b)*(a - c) у виді наступної структури:



Тепер обчислення значення f можна організувати "знизу-уверх", підставляючи результати операції замість знаків операцій. Легко бачити, що такий метод обчислення є універсальним.

Приклад 2. Нам треба обробити набір відомостей про людей (прізвище - F, вік - A), причому обробка включає процедури включення людини у список, вилучення із списку, виведення списку як у алфавітному порядку по прізвищам, так у порядку зменшення віку. Дані для цієї задачі зручно уявити у виді структури:



в якій F_i - прізвища, A_i - віки людей, суцільні стрілки вказують на людей що йдуть в алфавітному порядку., а пунктирні - на людей, що йдуть по росту.

Тоді включення - вилучення елементів можна робити переорієнтацією стрілок, а порядок виведення легко отримати, слідуючи по відповідним стрілкам. Нижче ми розглянемо і інші приклади задач, у програмуванні яких зручно використовувати динамічні структури.

Розглянуті приклади показують, що динамічні структури даних представляють із себе сукупність елементів, кожний з яких містить як деяку значущу інформацію, так і інформацію про зв'язки з іншими елементами структури. Інформацію про зв'язки називають посиланнями або покажчиками.

Динамічні структури даних, що реалізуються засобами мови Паскаль, представляються у виді сукупності записів, кожна з яких містить інформаційні поля і поля посилань (покажчиків) на інші записи структури.

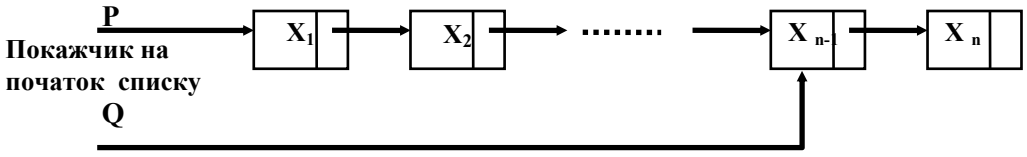
Елемент динамічної структури даних

Посилання на деякий елемент - це по суті адреса першого (молодшого байта) фрагмента оперативної пам'яті, відведеної під цей елемент.

Для реалізації ефективних алгоритмів розв'язків задач вирішальну роль грають способи об'єднання елементів у структури даних. Для кожного такого способу характерна як топологія структури, так і методи її обробки. Нижче ми розглянемо деякі з таких динамічних структур, які по суті є стандартними.

У програмуванні часто використовують наступні динамічні структури: списки, стеки, черги, дерева, графи і т.д. Точні математичні визначення цих структур, як ми побачимо нижче, використовують рекурсивні описання. Для попередніх пояснень краще всього використовувати графічні зображення.

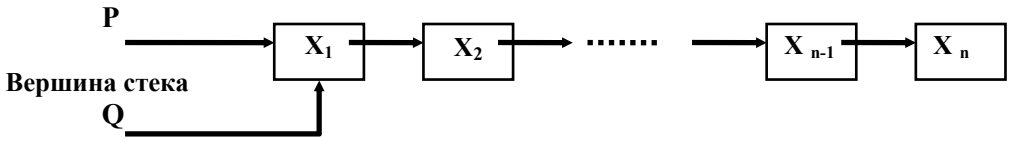
Список



Покажчик на елемент, що оброблюється

Список - це такий спосіб представлення послідовності однотипних елементів, при якому вставка і вилучення елемента допускаються у довільному її місці, а доступ до елемента, що оброблюється, здійснюється послідовним переглядом від початку списку до його кінця.

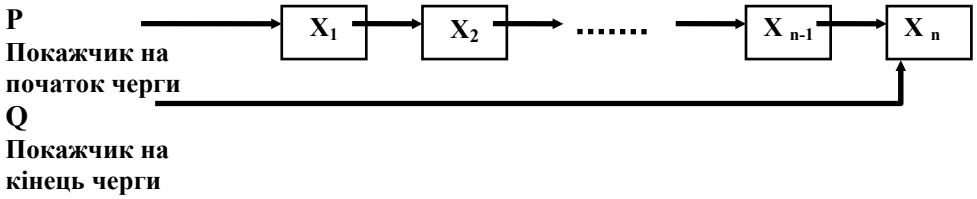
Стек



Покажчик на елемент, що оброблюється

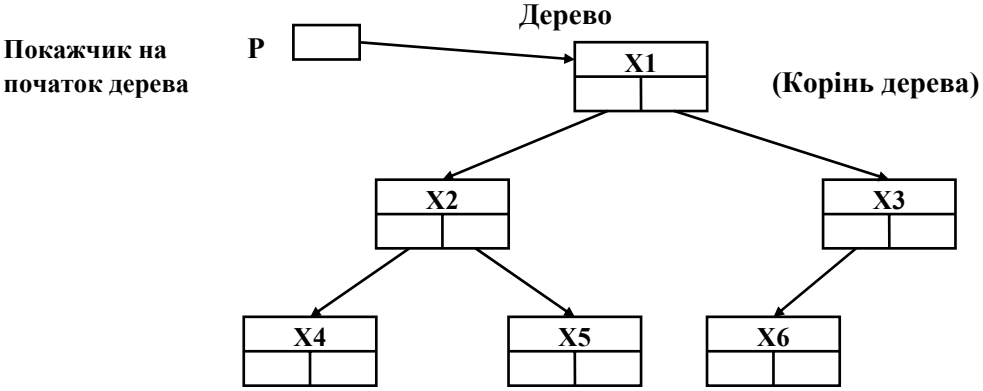
Стек - це такий спосіб представлення послідовності однотипних елементів, при якому вставка і вилучення елемента допускаються тільки на її початку - вершині стека. Доступ до інших елементів стека не підтримується методами обробки стека.

Черга

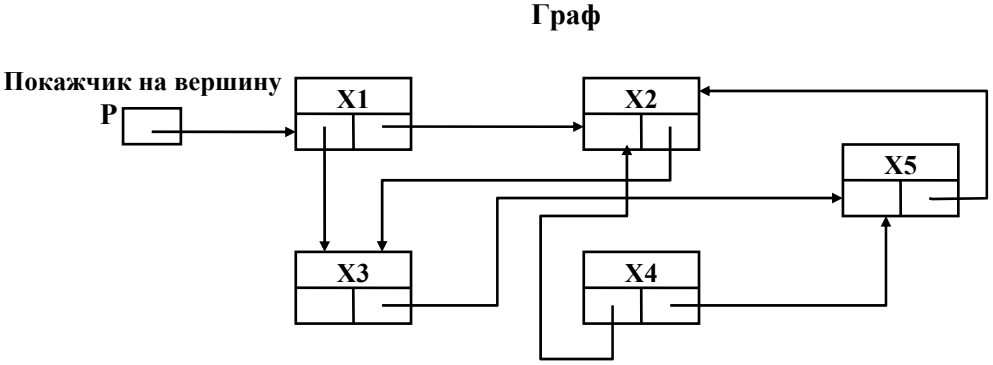


Черга - це такий спосіб представлення послідовності однотипних елементів, при якому вставка елемента допускається тільки в її хвості, а вилучення - тільки на початку. Доступ до інших елементів черги не підтримується методами обробки черги.

Списки, черги і стеки - це т.д. лінійні динамічні структури. Вони представляють з себе так звані послідовності записів, у кожній з яких, за винятком останньої, покажчик виставлений на наступну. Відрізняються вони друг від друга способом обробки: зміна списку здійснюється вставкою або вилученням елемента у довільному місці. У стеку елементи добавляють або вилучають в одному й тому ж місці - вершині стека. У черзі запису добавляються у хвіст, а вилучаються з голови.



Дерева - це структури даних, що розгалужуються. Кожний запис у дереві, за винятком одного - кореневого - має одного попередника. Кожний елемент дерева, за винятком так званого листя, вказує на декілька “спадкоємців”. Точне визначення дерева як динамічної структури даних буде дане нижче.



Граф утворюють декілька записів, покажчики яких виставлені довільним чином. У програмуванні використовуються і інші типи динамічних інформаційних структур (двохзв'язні списки, кільця, і т.д.).

Ще раз відмітимо, що використання посилань частіше всього - чисто технічний прийом. Пам'ять для розміщення змінної може виділятись автоматично при виклику процедури і звільнятись при виході з неї. Однак явне використання посилань дозволяє використовувати при програмуванні більш різноманітні структури. Тому у сучасних мовах програмування (зокрема, в Паскалі) введені засоби, що маніпулюють як даними, так і посиланнями на них.

2. Посилальний тип даних. Посилання.

Значеннями посилального типу даних є посилання (покажчики) на інші дані. Посилальні типи описуються у розділі типів наступним чином:

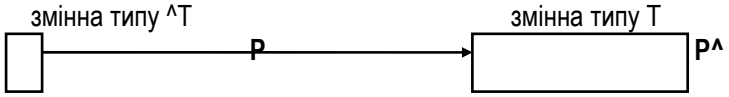
Type <ім'я посилального типу > = ^ <ім'я базового типу >

Ім'я змінної може бути зв'язано з описанням її типу безпосередньо в розділі змінних, або з іменем типу, вже описаного в розділі типів. Стиль мови віддає перевагу явному використанню імен типів.

Приклади описань:

```
1. Type Vector = Array[1..100] of Integer;
                 Place = ^ Vector;
       Var a, b: Place;
або
       Var a, b: ^ Array[1..100] of Integer;
   2. Type Item = Record
 Name: String[20];
                          Age: Integer
end;
Point = ^ Item;
   Var x, y: Point;
```

Якщо p - покажчик на змінну типу T, то p^ - позначення самої цієї змінної. Ця система позначень може бути продемонстрована так:



Ще раз укажемо на те, що при описанні посилальної змінної p пам'ять резервується тільки для неї. Значенням p по суті є початкова адреса розміщення p^ у пам'яті.

Для резервування пам'яті під дане типу T**^** у мові використовується процедура New.

New (< змінна посилального типу >)

Виконання процедури New(p) полягає у заповненні комірки пам'яті, відведеної під змінну p початковою адресою ділянки пам'яті, в якій буде розміщено значення p^. Розмір цієї ділянки визначається типом змінної p.

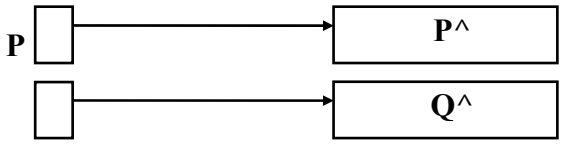
Для визволення пам'яті, відведеної раніш під T^, використовується процедура Dispose.

Dispose(<змінна посилального типу>)

Приклад 3.

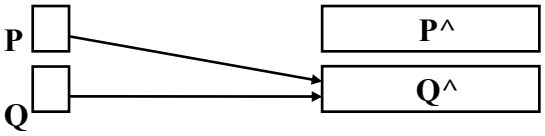
```
Program ExamNewDis;
    Type Vector = Array[1..100] of Integer;
           Place = ^Vector;
             Item = Record
                      Name: String[20];
                        Age: Integer
                     end;
            Point = ^Item;
     Var a, b: Place;
         x, y: Point;
             i: Integer;
Begin
  New(x); New(y);
  Read(x^.Name);  Read(x^.Age);
  y^ := x^;
 Dispose(x);
 New(a);
 For I:= 1 to 100 do  a^[i] := Sqr(i)+1;
 b := a;
 Dispose(a);
End.
```

Тут при зверненні до процедури New(x) буде відведено 22 байта під x^, при виконанні New(y) - 22 байта під y^, а при виконанні New(a) - 200 байт під a^. (Ми вважаємо, що під ціле число відводиться 2 байта). Оператор y^ := x^ пересилає дані з запису x^ у запис y^. Пам'ять, яку займає x^, звільняється оператором Dispose(x). Для переводу покажчика з одного даного на інше використовується оператор присвоювання. Нехай, наприклад, p і q - змінні одного посилального типу і має місце ситуація



Q

Тоді після виконання оператора p := q схема зміниться:



Зверніть увагу на те, що дані, на які до присвоювання посилалась p, стають недоступними! У прикладі, що розглядається, посилання b встановлено оператором b := a на масив a^. Тому оператор Dispose(a), звільняючи пам'ять, позбавить захисту не тільки масив a^, але й масив b^! Тому наступний оператор New може розмістити дані на місці масиву b.

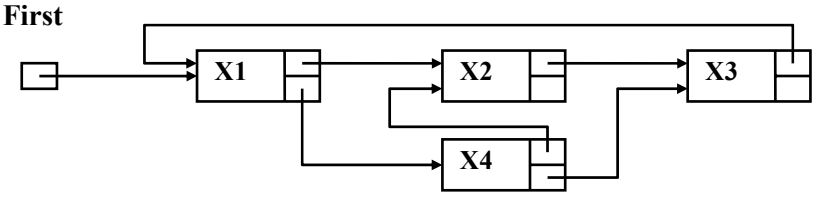
На прикладах з інформаційними динамічними структурами ми бачили, що деякі поля посилальних типів можуть бути не заповнені посиланнями на інші записи, причому програміст повинен явним чином це вказувати. Для цього використовується стандартне ім'я Nil. (Після виконання операторів p := Nil; q := Nil покажчики p і q вказують на одне й те ж дане - у нікуди!)

3. Програмування інформаційних динамічних структур.

Найпростіші характерні прийоми обробки динамічних структур даних розглянемо на наступному прикладі:

Приклад 4.

а) Побудувати динамічну структуру даних, що зображена на малюнку:



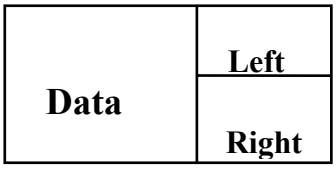
(Дані xi - цілі числа.)

б) Прочитати дані в наступній послідовності x1 x4 x3 x1, x1 x2 x3.

Розв'язок.

Тип елемента структури визначмо наступним чином:

```
Type Point = ^ Item;
             Item = record
                        Data: Integer;
                        Right, Left: Point
                    End;
```



Для побудови структури використовуємо дві змінні типу Point - p і First.

```
Program Structure;
  Type Point = ^ Item;
       Item = Record
                Data: Integer;
                Right, Left: Point
                End;
   Var p, First: Point;
 Begin
  New(p); First := p; Read(p^.Data);          {побудований перший елемент структури}
  New(p^.Left); Read(p^.Left^.Data);          {побудований другий елемент структури}
  New(p^.Right); Read(p^.Right^.Data);        {побудований четвертий елемент структури}
  p := p^.Left;
  New(p^.Left); p := p^.Left; Read(p^.Data);
  p^.Right := Nil; p^.left := First;          {побудований третій елемент структури}
  p := First^.Right;                                {p установлений на 4-ий елемент }
  p^.Left := First^.Left;
  p^.Right := First^.Left^.Left;              {покажчики виставлені. Побудова закінчена}
                                                    {початок блока читання}
  Writeln('x1,x4,x3,x1:');
  p := First;
  Write(p^.Data,''); p := p^.Right;
  Write(p^.Data,''); p := p^.Right;
  Write(p^.Data,''); p := p^.Left;
  Write(p^.Data,'');
  p := First;
  Writeln; Writeln('x1,x2,x3:');
  Write(p^.Data,''); p := p^.Left;
  Write(p^.Data,''); p := p^.Left;
  Write(p^.Data,'');
  Writeln('Кінець роботи')
 end.
```

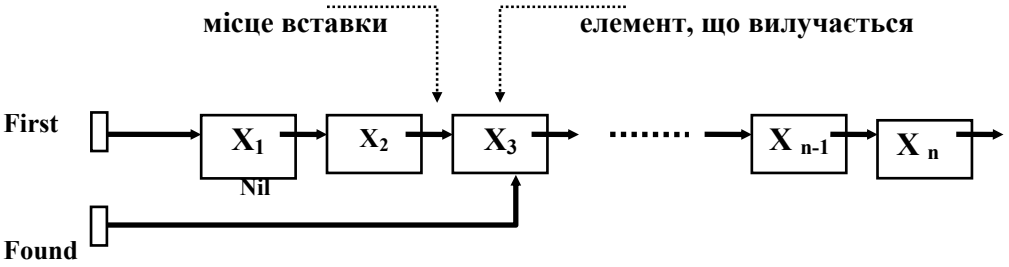
Посилання p використовувалось для обходів структури, а First - як покажчик на початковий елемент структури.

4. Списки.

У цьому розділі розглядаються процедури, що реалізують стандартні засоби роботи зі списками: пошук елемента, вилучення елемента, вставка елемента. Аналогічними засобами користуються і при обробці інших інформаційних структур. Елемент списку описується наступним чином:

```
Type Point = ^ Item;
      Item = Record
               Data: Integer; Next: Point
             End;
```

Відмітимо характерну деталь: описання елемента динамічної структури рекурсивне! Таким чином, описання динамічних структур неможливе без явного описання типів елементів цих структур.



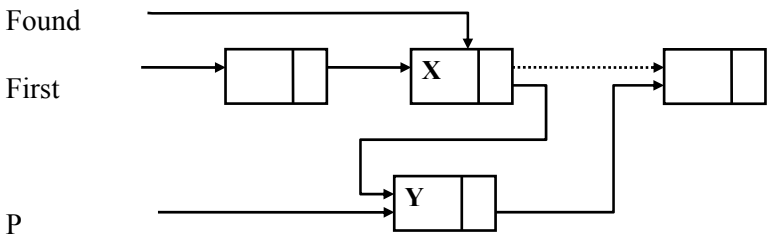
а) Пошук. Процедура Search здійснює пошук елемента списку з числом x у якості значення поля Data і повертає посилання Found на цей елемент. Якщо такий елемент у списку відсутній, Found = Nil.

```
Procedure Search(var Found, First: Point; x: Integer);
   Begin
     Found := First;
     While (Found <> Nil) and (Found^.Data <> x)
       do Found := Found^.Next
   End;
```

б) Вставка. Процедура InsList добавляє елемент у список на місце, що передує Found^ (див. малюнок). Посилання Found встановлюється на вставлений елемент.

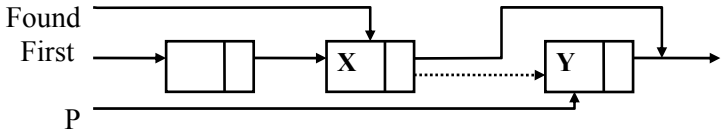
```
Procedure InsList(var Found: Point, x:Integer);
      Var p: Point;
    Begin
```

```
    New(p);
    p^.Data := Found^.data;
    Found^.Data := x;
   p^.Next := Found^.Next;
   Found^.Next := p
End;
```



в) Вилучення. Процедура DelList вилучає з списку елемент Found^. Посилання Found встановлюється на елемент, що йде за вилученим.

```
Procedure DelList(Found: Point);
      Var p: Point;
          y: Integer;
   Begin
     y := Found^.Next^.Data;
     p:= Found^.Next;
    Found^.Next := Found^.Next^.Next;
    Found^.Data := y;
    Dispose(p) {збирання сміття}
 End;
```



Відмітимо один явний недолік процедур InsList і DelList: вони непридатні для обробки останнього елемента списку! InsList для правильної роботи потребує наявності елемента Found^.Next^, а DelList - елемента Found^.Next^.Next. Тому, якщо треба вставити в список останній елемент, покажчик Found повинен бути виставлений на Nil, але тоді Found^.Next не визначений! Аналогічна ситуація має місце і при вилученні останнього елемента.

У нашій постановці задачі цей недолік виправити непросто. Суть ускладнень у тому, що до елементів списку, покажчики яких треба перекинути, немає прямого доступу: посилання Found виявляється встановленим на елемент, наступний за потрібним. Укажемо два виходи з цієї ситуації:

а) Останній елемент списку можна вважати ознакою кінця, поле Data якого не містить значущої інформації і покажчик Found на нього за умовою не може бути встановлений.

б) Можна змінити умови вставки-вилучення: посилання Found за умовою встановлюється на елемент, що передує місцю вставки або елементу, що вилучається. Тоді окремо (поза процедур) треба розглядати випадок, коли оброблюється 1-ий елемент. Самі ж процедури в цьому варіанті спрощуються.

```
Procedure Ins(var Found: Point, x:Integer);
    Var p: Point;
  Begin
     New(p);
     p^.Data := x;
     p^.Next := Found^.Next;
     Found^.Next := p;
  End;
Procedure Del(Found: Point);
    Var p: Point;
  Begin
    p:= Found^.Next;
    Found^.Next = Found^.Next^.Next;
    Dispose(p) {збирання сміття}
  End;
```

Часто вставці/вилученню передує пошук місця зміни списку. Для правильної роботи процедур Ins і Del процедуру Search необхідно модифікувати. Переглядати список треба "на крок уперед". Розглянемо варіант процедури пошуку місця елемента з значенням поля Data = x у списку First із упорядкованими за зростаючими значеннями полів Data.

```
Procedure ForwardSearch(var Found, First: Point;  var isFirst: Boolean; x: Integer);
   Begin
     Found := First;
      If First^.Data >= x
         then isFirst := True
         else begin
           isFirst := False:
           While (Found^.Next <> Nil) and  (Found^.Next^.Data < x)
             do Found := Found^.Next;
         end
   End;
```

Якщо isFirst = True, то місце - перше, інакше на місце вказує Found^.Next.

Задачі на списки.

Елемент списку описується наступним чином:

```
Type Point = ^ Item;
     Item = Record
              Key: Integer;
              Data: Real;
              Next: Point
            End;
```

5. Реалізувати процедуру злиття двох списків, елементи яких розташовані за зростанням значень полів Key. При рівних значеннях полів Key значення полів Data додаються. Оцінити складність алгоритму за часом у термінах довжин списків.
6. Реалізувати процедуру зчеплення (конкатенації) двох списків. Оцінити складність алгоритму за часом у термінах довжин списків.
7. Реалізувати процедуру сортування списку злиттям. Оцінити складність алгоритму за часом і пам'яттю (у гіршому випадку).
8. Реалізувати процедуру побудови списку, значущі елементи якого зберігаються у файлі F
   а) F: File of Record Key: Integer; Data: Real End; із збереженням порядку розташування елементів у файлі. Оцінити складність алгоритму за часом.
   б) F: File of Record Key: Integer; Data: Real End; Список повинен бути впорядкований за значеннями полів Key. Оцінити складність алгоритму за часом.
5. Реалізувати процедури Push і Pop відповідно добавлення і вилучення елемента для стека.
6. Реалізувати процедури Push і Pop відповідно добавлення і вилучення елемента для черги.

5. Дерева.

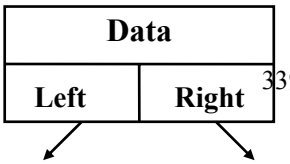
Динамічні структури даних зручно визначати за допомогою рекурсивних визначень. Список, наприклад, визначається наступним чином:

<список > ::= Nil | <елемент списку > ⟶ < список >

Тут символ "::=" означає "є за визначенням", "|" - "або", "⟶ " - покажчик (посилання).

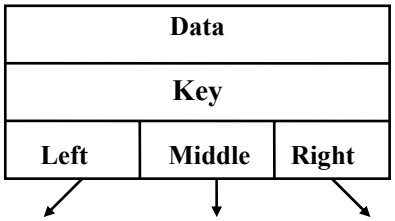
Аналогічно можна визначити і структури, що розгалужуються - так звані дерева. Елемент такої структури (вузол дерева) визначається як запис, що містить декілька полів посилань. Наприклад:

```
Type Point = ^ Item;
```

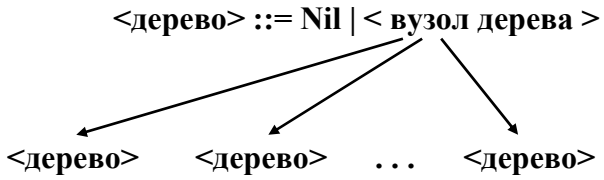


```
Item = Record
         Data: Integer;
         Right, Left: Point
       End;
```

```
Type
    Link = ^ TreeVert;
    TreeVert = Record
                 Data: Real;
                 Key : Integer;
                 Left,  Middle,  Right : Link
               End;
```



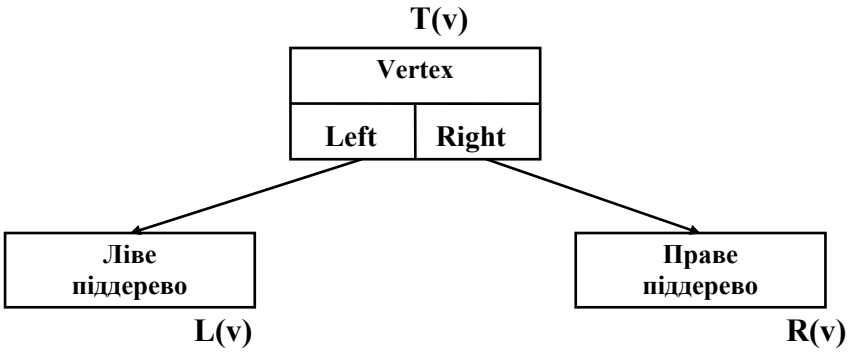
Кількість посилальних полів вузла дерева називається степенем розгалуження (арністю) вузла. Рекурсивне визначення дерева тоді має вид:



Таким чином, дерево може бути або виродженим (Nil), або складено з вузла дерева, всі покажчики якого виставлені на дерева. У цьому випадку вузол називають коренем дерева, а дерева, на які виставлені покажчики - піддеревами. Якщо піддерево складається з одного вузла, всі покажчики якого встановлені на Nil, його називають листом. Інші вузли дерева називають проміжними. Сукупність посилальних полів може бути оформлена як запис або як декілька полів запису (як у наших прикладах). Часто сукупність посилальних полів визначається в виді масиву посилань, або організується у виді списку посилань. У цих випадках говорять про впорядковані дерева, так як піддерева одного кореня виявляються впорядкованими або індексами масиву, або за порядком доступу. Якщо всі вузли дерева мають один і той же степінь розгалуження, можна говорити про степінь розгалуження дерева. Дерева, степінь розгалуження яких дорівнює двом, називають бінарними. Бінарні дерева - одна з найбільш поширених структур, що розгалужуються, які застосовуються у програмуванні.

Бінарні дерева.

Бінарне дерево має вид



Як і для списків, основними процедурами обробки дерев є пошук, вставка і вилучення вузла дерева. Ці процедури ми розглянемо на прикладі, в якому сортування послідовності реалізована за допомогою побудови і наступної обробки бінарного дерева.

Приклад 5: Нехай A = a_1, a_2,..., ,a_n - послідовність цілих чисел. Для того, щоб відсортувати послідовність A, що складається з елементів a_i

а) побудуємо бінарне дерево, що задовольняє наступній властивості: для будь-якого вузла дерева v будь-який елемент L(v) ≤ v ≤ будь-який елемент R(v)

б) побудуємо послідовність із вузлів дерева, в якій елементи розташовані у відповідності з цим же принципом: послідовність {L(v)}, v, послідовність {R(v)}.

Легко бачити, що вихідна послідовність буде впорядкованою.

Розв'язок.

Послідовність A ми реалізуємо у виді списку. Нам знадобляться процедури:

- побудови списку з чисел, що вводяться з клавіатури;
- побудови дерева, що задовольняє властивості п. а);
- побудови списку, що задовольняє властивості п. б);
- виведення списку.

Типи елементів динамічних структур даних задачі:

```
Type Point = ^ Item;
        Item = Record
                Key: Integer; Next: Point
                End;
Link = ^ Vertex;
```

```
Vertex = Record
                Key: Integer;
        Left, Right: Link;
                End;
```

Програма розв'язку задачі:

```
 Program ListSort;
{описання типів даних}
    Var A, B: Point;
        LastB:Point; {кінець списку B}
          Tree: Link;
        {описання процедур}
        {процедура введення списку}
        {процедура побудови дерева з списку}
        {процедура побудови списку з дерева}
        {процедура виведення списку}

        Begin {розділ операторів}
        InpList(A);                              {процедура введення списку}
        Tree := Nil;
        TreeBild(Tree, A);              {процедура побудови  дерева Tree з списку A}
        B := Nil;
        ListBild(Tree, B);          {процедура побудови списку B з дерева Tree}
        OutList(B);                     {процедура виведення списку}
      End.                              {кінець програми}

  Procedure InpList(var P: Point);    {процедура введення списку}
      Var Ch: Char;
          Q: Point;
   Begin
     P := Nil;                        {порожній список}
     Writeln('Для введення числа нажміть y');
     ch := Readkey;
     While ch = 'y' do begin
     New(Q);                          {формування нового елемента списку}
     Writeln('Input Item:');
     Read(Q^.Key);
     Q^.Next := P;                    {включення нового елемента в список}
     P := Q;                          {покажчик списку - на початок списку}
     Writeln('Продовжити введення? Y/N ');
```

```
      ch:= Readkey
    end
  End;

 Procedure TreeBild(var T: Link; P: Point);     {процедура побудови дерева з списку}
       Var x: Integer;
   Procedure Find_Ins(var Q: Link; x: Integer);
     Procedure Ins(var S: Link);
        Begin                                          {процедури вставки елемента}
          New(S);
          S^.Key := x;
          S^.Left := Nil; S^.Right := Nil
        End;                                           {процедури вставки елемента}

Begin                                                  {процедур пошуку і вставки елемента}
         x := P^.Key;
         If Q = Nil
           then Ins(Q)
           else if x < Q^.Key
             then Find_Ins(Q^.Left, x)
             else Find_Ins(Q^.Right, x)
       End;                                            {процедури пошуку і вставки елемента}

      Begin                                            {процедури побудови дерева з списку}
        If P <> Nil
         then begin
          Find_Ins(T, P^.Key);
          TreeBild(T, P^.Next)
        end
      End;                                             {процедури побудови дерева з списку}
```

Тонким місцем організації управління в процедурі побудови дерева є передача посилань на вершини структур, що оброблюються в виді параметрів-змінних. Це дозволяє породжувати вузли дерева без використання додаткових покажчиків.

```
 {процедура побудови списку з дерева}
    Procedure ListBild(var T: Link; var F: Point);
          Var Temp: Point;
        Begin
          If T <> Nil
            then begin
```

```
            ListBild(T^.Left, F);
            New(Temp);                    {породження нового елемента}
            Temp^.Key := T^.Key;
            Temp^.Next := Nil;
             If F = Nil
               then begin
                 F := Temp; LastB := Temp
               end
          else begin
            LastB^.Next := Temp; LastB := Temp
           end;
           {змінна LastB - покажчик на останній елемент списку, що будується}
          ListBild(T^.Right, LastB^.Next)
        end
    End;                                          {процедури побудови списку з дерева}

      Procedure OutList(var P: Point);    {процедура виведення списку}
    Var x: Integer;
  Begin
    If P <> Nil
     then begin
      Write(P^.Key, ' ');
      OutList(P^.Next)
     end
  End;                                            {процедури виведення списку}


      {процедура виведення дерева - використовувалась при налагодженні}
Procedure OutTree(var T: Link);
  Begin
    If T <> Nil
     then begin
       OutTree(T^.Left);                {ліве піддерево}
       Write('V = ', T^.Key);           {корінь дерева}
       OutTree(T^.Right);                       {праве піддерево}
     end
  End;                                            {процедури виведення дерева}
```

У процедурах ListBild і OutTree використаний загальний принцип обробки дерева - так званий обхід дерева зліва - направо:
L(v) - обробка лівого піддерева; v - обробка кореня; R(v) - обробка правого піддерева.

Обробка L(v) і R(v) реалізована як рекурсивний виклик процедури обробки дерева. Крім обходу зліва направо, в задачах використовуються також і інші порядки переліку вузлів. Для бінарних дерев це:

L(v) - v - R(v) - зліва направо;
R(v) - v - L(v) - справа наліво;
v - L(v) - R(v) - зверху вниз;
L(v) - R(v) - v - знизу вверх.

Якщо в процедурі ListBild застосувати обхід справа наліво, елементи списку B будуть упорядкованими за зменшенням. У процедурі Find_Ins по суті застосований обхід зверху вниз у модифікації: v - L(v) або R(v) (у залежності від результату порівняння x < Q^.Key).

Якщо в виді дерева представлено арифметичний вираз (приклад 1), процедуру обчислення його значення треба програмувати обходом знизу-вверх: обчислити значення лівого операнда, правого операнда, результат бінарної операції. Неважко побачити, що якщо при побудові дерева Tree будь-яке ліве піддерево буде містити стільки же вузлів, скільки і відповідне праве, для його побудови треба $O(n*\log_2 n)$ порівнянь. Однак у гіршому випадку (якщо, наприклад, вихідний список упорядкований), для вставки кожного наступного вузла алгоритм буде слідувати по одній і тій же гілці (наприклад, лівій). Тому алгоритм тратить $O(n^2)$ порівнянь, тобто не є ефективним. Однак його можна модифікувати таким чином, щоб процедура TreeBild будувала т.н. збалансоване дерево. Тоді алгоритм сортування стане ефективним.

Задачі на деревах.

1.Клас арифметичних виразів, який містить однобуквені змінні, операції +, -, *, / і круглі дужки, назвемо класом найпростіших виразів. Реалізувати:

процедуру побудови дерева найпростішого арифметичного виразу, який заданий у виді рядка.

процедуру обчислення значення найпростішого арифметичного виразу, який заданий деревом при значеннях змінних, що вводяться з клавіатури.

процедуру перетворення дерева в рядок.

2. У параграфі 9, алгоритм пірамідального сортування, викладений метод представлення масиву в виді бінарного дерева. Реалізувати процедуру побудови цього дерева по масиву V[1..n].

3. Книга складається з 3-розділів. Кожний розділ містить 3 параграфа, а кожний параграф - 3 пункти. Всі вказані структурні частини книги мають назви. Реалізувати :

процедуру побудови змісту книги в виді тернарного дерева;
діалогову процедуру пошуку назви кожної частини книги за її номером;
виведення змісту на екран за допомогою різних обходів;
пошук розділу по його назві.

4. Реалізувати процедури порівняння двох однотипних бінарних дерев T_1 і T_2 на рівність ($T_1 = T_2$) і вкладення ($T_1 \le T_2$), використовуючи наступні рекурсивні визначення:

Через root(T) позначимо корінь дерева T. Тоді

$T_1 = T_2$, якщо $root(T_1) = root(T_2)$ і $L(root(T_1)) = L(root(T_2))$, $R(root(T_1)) = R(root(T_2))$

$T_1 \le T_2$, якщо $root(T_1)=root(T_2)$ або T_1=Nil і $L(root(T_1))\le L(root(T_2))$, $R(root(T_1))\le R(root(T_2))$

5. Реалізувати процедуру пошуку в дереві T_1 піддерева, який дорівнює T_2. Використати процедуру порівняння з попередньої задачі.

6. Реалізувати процедури, які ілюструють графічно:

побудову бінарного дерева;

пошук вузла з даним ключем різними обходами.

СТРУКТУРНЕ ПРОГРАМУВАННЯ.

Структурне програмування.

З виникненням мов високого рівня (кінець 50-х років) з'явилась можливість проектування великих програмних систем. Програмування з мистецтва комбінування машинних команд, таємницями якого володіли вибрані, перетворилося в індустрію виробництва програмного обладнання ЕОМ. До початку 70-их років витрати на виробництво програм перевищили витрати на виробництво апаратури. Тому проблема розробки ефективних і надійних програмних систем стала центральною задачею інформатики. Використання мов високого рівня зняло лише частину проблем (таких, наприклад, як проблема розподілу обчислювальних ресурсів), породивши одночасно нові проблеми (наприклад, у зв'язку з неефективністю машинного коду, що генерується транслятором у порівнянні з кодом "ручної роботи", виникла задача оптимізації). Дослідження цих проблем привели, зокрема, до формування науково-обгрунтованого стилю програмування - структурного програмування.

Структурне програмування - це технологія проектування програм, яка базується на суворому визначенні засобів мови і методів їх використання. До засобів мови відносяться стандартні типи даних і оператори управління обчисленнями.

1. Основні структури управління.

Розглянемо алгоритми розв'язку трьох задач, що відносяться до різних предметних областей:

а) Алгоритм Евкліда:

```
Program Evclid;
    Var a, b, u, v, w: Integer;
  Begin
    Read(a,b);
    u := a; v := b;
    While u <> v do begin
      w := u - v;
      If w > 0 then u := w else v := -w;
    end;
   Write(u)
 end.
```

б) Наближений розв'язок рівняння методом ділення навпіл:

```
Program EquationSol;
   Const Eps = 1e-4;
     Var a, b, u, v, w: Real;
  Begin
```

```
        Read(a, b);
        u := a; v := b;
        While u - v >= Eps do begin
          w := (u + v)/2;
          If f(u)*f(w) > 0  then u := w  else v := w;
         end;
Write(u)
End.
```

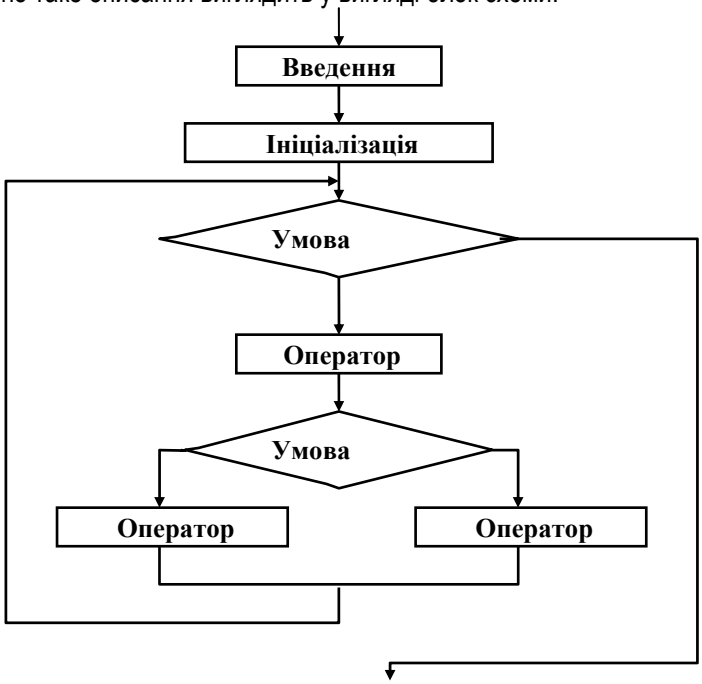
в) Розподіл масиву на два масиви за бар'єрним елементом 0:

```
Program Partition ;
        Const n = 16;
           Var f, g, h : array [1..n] of Integer;
                 x, b: Integer;
                 i, j, k: Integer;
       Begin
         { введення масиву f }
         i := 1; j := 1; k := 1;
         While f[i] <> 0 do begin
           x := f[i];
          If x > 0
            then begin
              g[j] := x; j := j + 1; i := i + 1 end
           else begin h[k] := x; k := k + 1; i := i + 1 end;
        end;
        Write(j, k)
       End.
```

Програма Evclid працює з цілими числами. Предметна область програми EquationSol - область дійсних чисел. Третя програма - Partition - оброблює масиви, причому тип компонент масиву не має суттєвого значення. Не зважаючи на різницю в предметних областях, між цими програмами існує схожість, що грає велику роль для розуміння суті програмування. Насправді, розділ операторів кожної з цих програм може бути описаний так:

```
Begin
  < процедура введення >;
  < послідовність простих операторів >;
  while <умова> do begin
   <оператор>;
   if <умова> then < оператор > else < оператор > ;
  end;
 < процедура виведення >
End.
```

Ще більш наочно таке описання виглядить у вигляді блок схеми:

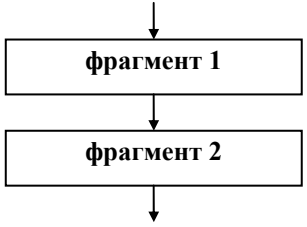


Те загальне, що об'єднує розглянуті програми, називають структурою управління. Структура управління програми грає роль її несучої конструкції, скелета. Зрозуміло, що правильне проектування програми неможливе без правильної побудови її структури управління.

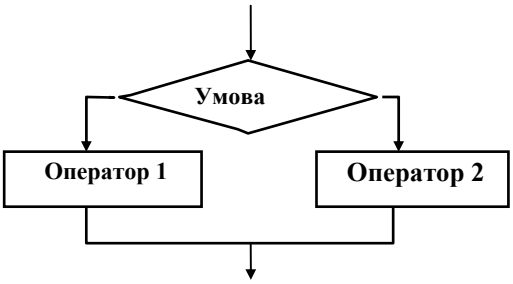
Основним досягненням у теорії програмування 60-х років є усвідомлення і теоретичне осмислення того факту, що існують декілька основних структур управління, оперування якими приводить до створення як завгодно складних за управлінням програм, причому ефективність програм при цьому не погіршується, а такі властивості, як читабельність, надійність суттєво поліпшуються. Процес програмування набуває наукових рис системного проектування.

До основних структур управління відносяться:

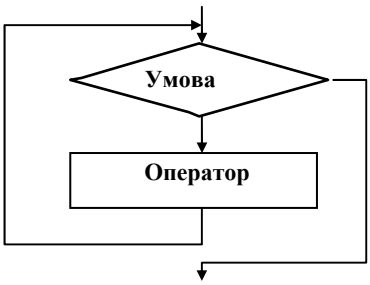
- **послідовне виконання**



•розгалуження

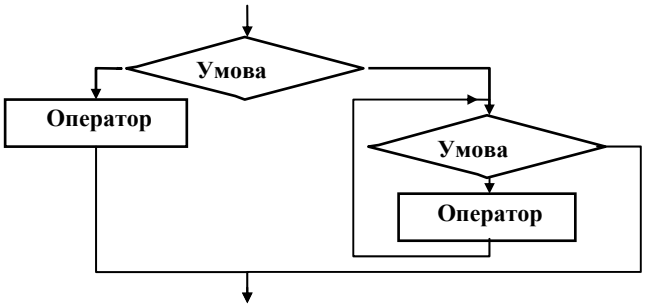


•повторення



Особливу роль у структурному програмуванні грають процедури і функції - основні семантичні одиниці програми, що проектується, які містять описання окремих підзадач, що мають самостійне значення.

Програмування здійснюється комбінуванням основних структур управління. Наприклад, комбінування розгалуження і повторення приводить до блок-схеми



Основний результат можна тепер сформулювати наступним чином:

Управління будь-якого алгоритму може бути реалізовано в виді комбінації основних структур управління.

У реальних мовах структурного програмування застосовують і інші управляючі структури, кожна з яких може бути віднесена до одного з трьох основних типів. Наприклад, у мові Pascal розгалуження - умовний оператор, короткий умовний оператор, оператор варіанта; повторення - оператор циклу з параметром, оператор циклу з передумовою, оператор циклу з постумовою. Наявність додаткових структур управління породжує надлишковість у виразних можливостях мови, що дозволяє робити програми більш природними.

Відмітимо, що оператор переходу Goto не включений ні в список основних, ні додаткових операторів управління. Це - не випадковість. Безконтрольне застосування цього оператора приводить до того, що програма втрачає властивості, що вказані вище. Тому структурне програмування часто називають програмуванням без Goto.

2. Основні структури даних.

Продовжимо аналіз прикладів, що приведені на початку параграфа. Звернемо увагу на сукупності даних, з якими працюють ці програми. Дані визначені в розділах типів, змінних і констант:

```
Program Evclid;
      Var a, b, u, v, w: Integer;

Program EquationSol;
  Const Eps = 1e-4;
    Var a, b, u, v, w: Real;

Program Partition ;
  Const n = 16;
     Var f, g, h : array [1..n] of Integer;
          x, b: Integer;
          i, j, k: Integer;
```

Сукупність даних, що визначені у програмі як змінні і константи, називається структурою даних цієї програми. Правильна побудова структури даних програми грає туку ж важливу роль для її ефективності, як і правильна побудова структури управління. Саме взаємодія цих двох аспектів визначає програму.

Теорія структур даних подібна теорії структур управління. Саме, існують основні (стандартні) структури, кожна з яких визначає один із засобів об'єднання даних у структуру. Дані простих типів при цьому грають роль цеглинок, з яких будується весь будинок.

Структури даних у мові Pascal, напевне, найбільш природним чином відображають ідеологію структурного програмування.

Прості дані: цілі числа, дійсні числа, символи, логічні значення, дані - імена.

Засоби структурування даних: масиви, записи, файли, множини, посилання.

Наприклад, визначення типів

```
Word = Array [1..20] of Char;
Man = Record
        Name: Word;
        Age: Integer
            end;
Notebook = array [1..n] of Man;
```

означає масив із записів, перше поле яких - масив із двадцяти символів, а друге - ціле число.

Необхідно розуміти, що зв'язок "дані-управління" полягає не тільки і не стільки у схожості правил їх побудови, скільки в орієнтації структур управління на обробку структур даних. Оператор присвоювання використовує дані простих типів. Логічні дані застосовують для визначення умов. Скалярні типи, що визначаються програмістом, використовують для описання даних-індексів у масивах і селектора варіанта в операторі вибору. Для обробки масивів зручно користуватись оператором циклу з параметром, а для обробки файлів - операторами While і Repeat. Записи з варіантами потрібно оброблювати операторами варіанта. Посилальні (динамічні) структури потрібно оброблювати рекурсивно. Цей список можна продовжити.

3. Методологія програмування "зверху-вниз".

Застосування у програмуванні концепції структур даних і управління потребує специфічної методології процесу розробки програм. Цей підхід називають програмуванням "зверху-вниз". Суть метода - у наступному:

8.Процес розробки програми складається з послідовності кроків, на кожному з яких програміст

- уточнює структуру програми;
- уточнює структуру даних.

9. Уточнення структури управління полягає у визначенні того оператора управління, який треба використовувати для реалізації алгоритму і тих елементів оператора, які приймають участь в управлінні (умов, меж циклів і т.п.)

10. Уточнення структури даних складається у визначенні й описанні даних, що оброблюються вибраним оператором.

11. Визначення структури даних програми починається з описання входу-виходу, тобто з визначення структури початкових і вихідних даних.

12. При роботі користуються принципом мінімальної необхідності: уточнення здійснюють лише з тим ступенем точності, яка необхідна на даному кроці.

13. При розробці декількох фрагментів програми краще уточнити кожний з них на один крок, а не один - повністю (особливо тоді, коли це уточнення пов'язано з принциповими рішеннями).

14. Основними структурними одиницями програми є процедури і функції. Кожну відносно самостійну підзадачу оформлюють як підпрограму (процедуру або функцію).

4.Приклад: Розв'язок системи лінійних рівнянь.

Розглянемо приклад програмування задачі, яка реально виникла у роботі авторів.

Задача: Реалізувати процедуру розв'язання системи лінійних рівнянь, що отримана в процесі застосування метода невизначених коефіцієнтів до задачі обчислення невизначеного інтеграла від раціонального виразу.

Розв'язок: Стислість формулювання задачі обманлива: у ній скриті ті обмеження, які роблять цю постановку точною. Для уточнення задачі необхідно по-перше, указати поле коефіцієнтів системи рівнянь, по-друге - уточнити поняття розв'язка системи у термінах точне - наближене. Розглянемо два варіанта уточнення задачі:

Варіант 1. Поле коефіцієнтів - поле раціональних чисел. Елемент цього поля - або ціле число A, або дріб, що не скорочується, виду A/B, де знаменник B - натуральне число, а чисельник A - ціле число. Треба знайти точний розв'язок системи.

Варіант 2. Поле коефіцієнтів - поле дійсних чисел. Елемент цього поля - число типу Real. Треба знайти наближений розв'язок системи з указаною точністю.

У нашому випадку мають місце наступні обмеження:

- Коефіцієнти рівнянь системи - раціональні числа;
- Необхідно отримати точний розв'язок системи у виді набору раціональних чисел;
- Завжди існує єдиний розв'язок системи;
- Кількість рівнянь системи і кількість невідомих співпадають.

На першому кроці уточнюємо задачу в виді:

```
Program LinearSystem;
   Type Solution = ...;
        LinSys = ...;
    Var S: LinSys;
        X: Solution;
        n: Integer;
    Procedure SysInp(var A: LinSys);
        Begin
```

```
    {введення системи рівнянь A}
  End;
Procedure SysSol(A: LinSys; var Y: Solution);
  Begin
    {розв'язок Y системи рівнянь A}
    End;
Procedure SolOut(Y: Solution);
   Begin
    {виведення розв'язка Y}
    End;
Begin {основна програма}
  SysInp(S);
  SysSol(S, X);
  SolOut(X)
End. {кінець програми LinearSystem}
```

На наступному кроці уточнення виникає проблема вибору метода розв'язання і уточнення структури даних задачі. Відомим точним методом розв'язання систем лінійних рівнянь є метод Гауса послідовного вилучення невідомих. Основні кроки метода - вибір ведучого рівняння і ведучого невідомого і вилучення ведучого невідомого з інших рівнянь систем. Посібники з лінійної алгебри рекомендують представляти дані у виді n*(n+1) матриці коефіцієнтів системи

x_1	x_2	. . .	x_n	Св. член
a_{11}	a_{12}	. . .	a_{1n}	b_1
a_{21}	a_{22}	. . .	a_{2n}	b_2
. . .	. . .	. . .	. . .	. . .
. . .	. . .	. . .	. . .	. . .
a_{n1}	a_{n2}	. . .	a_{nn}	b_n

Оскільки параметр n залежить від задачі, для представлення матриці A у виді двомірного масиву треба резервувати пам'ять під масив деякого найбільш можливого розміру Nmax, що приводить до неоправданих витрат оперативної пам'яті. Радикальне рішення полягає у використанні динамічних структур даних. Систему можна представити в виді списку рівнянь, а кожне рівняння - списком доданків (членів рівняння):

```
LinSys = ^LS_Item; {список рівнянь}
LS_Item = Record
            LS_Mem : LinEqu;
            NextEqu: LinSys
          End;
```

```
LinEqu = ^EquItem; {список членів рівняння}
EquItem = Record
                EquMem : Monom;
                NextMem: LinEqu
          End;
```

Компромісне рішення - динамічне резервування пам'яті під масив A:

```
LinSys = ^Array[1..Nmax, 1..Nmax] of Rational; {Rational - раціональні числа}
```

Виберемо радикальний шлях рішення і уточнимо процедури SysInp, SysSol, SolOut. Наш вибір визначає Solution як список раціональних чисел.

```
Procedure SysInp(var A: LinSys);
    Var i: Integer;
        P: LinSys;
  Begin {введення системи рівнянь A}
    Read(n); S := Nil;
    For i := 1 to n do begin
        New(P); P^.NextEqu := S; S := P;
        {введення i-того рівняння - списку P^.LS_Mem}
    end
  End;

Procedure SysSol(A: LinSys; var Y: Solution);
  Begin                                  {розв'язок Y системи рівнянь A}
    For i ;= 1 to n do begin             {прямий хід метода Гауса}
      - вибір рівняння, що містить ненульовий коефіцієнт при Xi в якості ведучого;
      - перестановка місцями i-того і ведучого;
      For j := 1 to n do
        If  i <> j
          then  - вилучення  Xi з j-того рівняння
    end;
    For i := n downto 1 do               {обернений хід метода Гауса}
      - обчислення Xi;
  End;

Procedure SolOut(Y: Solution);
    Var i: Integer;
  Begin {виведення розв'язка Y}
    For i := 1 to n do             {Виведення i-тої компоненти розв'язка}
    Writeln(' Задача розв'язана ')
  End;
```

Наступне уточнення структури даних складається у визначенні типу Monom. Якщо ми вирішимо продовжити тактику економії пам'яті, можна не зберігати члени рівняння виду $A_{ij}*X_i$ при $A_{ij} = 0$. У цьому випадку члени рівняння представляються парою

<коефіцієнт, номер невідомого >:

```
Monom = Record
          Coef: Rational;
           Unknown: Integer
         End;
```

Для спрощення структури даних змінимо визначення типу EquItem, вилучивши тип Monom:

```
EquItem = Record
            Coef: Rational;
            Unknown: Integer;
            NextMem: LinEqu
          End;
```

Тепер процедуру введення рівняння реалізуємо як локальну в процедурі SysInp. Введення рівняння - це введення лівої і правої частин рівняння, відмежованих друг від друга введенням знака "=". Введення лівої частини - повторення введень членів рівняння, відмежоване друг від друга введенням знака "+".

```
  Procedure SysInp(var A: LinSys);
      Var i: Integer;
          P: LinSys;
 Procedure EquInp(Var E: LinEqu);
      Var Q: LinEqu;
    Sign: Char;
 Procedure MemInp(var R: LinEqu);
{ тіло процедури MemInp }
    Begin
      Writeln('Введення першого члена лівої частини');
      MemInp(Q); E := Q;
      Write('Натисни = або +');
      Sign := ReadKey;
      While Sign = '+' do begin
        Writeln('Введення наступного члена рівняння);
        MemInp(Q^.NextMem); Q := Q^.NextMem;
        Write('Натисни = або +');
        Sign := ReadKey
      end;
```

```
        Writeln('Введення вільного члена рівняння);
        MemInp(Q^.NextMem); Q := Q^.NextMem;
        Q^.NextMem := Nil
        End;{процедури введення рівняння}

    Begin {введення системи рівняння A}
        Read(n); S := Nil;
        For i := 1 to n do begin
          New(P); P^.NextEqu := S; S := P;
          EquInp(P^.LS_Mem}
        end
    End;
```

Розділ операторів процедури SysInp набув закінченого вигляду. Введення члена рівняння оформимо в виді процедури MemInp, означивши тип Rational, значеннями якого є раціональні числа.

Увага! Введення чисельного типу Rational означає необхідність реалізації набору процедур, що забезпечують роботу з раціональними числами. Оскільки тип Rational необхідний для розв'язків багатьох інших задач, його треба оформити як модуль RAT і активізувати в програмі директивою Uses. Принципи проектування модуля ми розглянемо нижче. Зараз нам знадобляться тільки імена процедур введення і виведення раціональних чисел - RatInp і RatPrint.

```
Procedure MemInp(var R: LinEqu);
    Begin
        New(R);
        With R^ do begin
            RatInp(Coef);
             If Sign <> '='
                then Read(UnkNown)
                else UnkNown := n + 1
        end
    End;
```

Наш алгоритм розв'язання системи (процедура SysSol) перетворить список рівнянь у список пар виду <номер невідомого, значення невідомого >. Тому для виведення розв'язка системи немає необхідності формувати список X. Достатньо установити посилання X типу LinSys на S і вивести значення X у потрібному виді. Це значить, що тип Solution можна вилучити, зробивши відповідні зміни у заголовках процедур.

```
Procedure SysSol(A: LinSys; var Y: LinSys);
```

```
Procedure SolOut(Y: LinSys);
```
Уточнимо процедуру виведення розв'язка:
```
Procedure SolOut(Y: LinSys);
        Var i: Integer;
              P: LinSys;
      Begin {виведення розв'язка Y}
        P := Y;
        For i := 1 to n do begin
        {Виведення і-тої компоненти розв'язка}
        With P^, LS_Mem^ do begin
          Writeln('X[',i,'] = ');
          RatPrint(Coef)
        end;
        P := P^.NextEqu
      end;
      Writeln(' Задача розв'язана ')
  End;
```

Ми завершили проектування процедур введення-виведення з точністю до засобів модуля RAT. Продовжимо уточнення основної процедури програми - процедури SysSol.

```
Procedure SysSol(A: LinSys; var Y:LinSys);
       Var P, Q: LinSys;
            TempRef: LinEqu;
             i: Integer;
    Begin {розв'язок Y системи рівнянь A}
      P := A;
      For i := 1 to n do begin {прямий хід метода Гауса}
       {вибір ведучого рівняння}
        Q := P;
        While Q^.LS_Mem^.Unknown <> i do
          Q := Q^.NextEqu;
          {перестановка місцями і-того і ведучого}
          TempRef := P^.LS_Mem;
           P^.LS_Mem := Q^.LS_Mem;
           Q^.LS_Mem := TempRef;
         {вилучення Xi з рівнянь системи}
         {1. нормалізація ведучого рівняння}
         {2. вилучення Xi з верхньої частини системи}
         {3. вилучення Xi з нижньої частини системи}
         P := P^.NextEqu;
```

```
    end;
    Y := A;
End;
```

Для вибору ведучого рівняння використовується послідовний перегляд рівнянь системи за допомогою посилань Q. Єдиний розв'язок системи гарантує існування ведучого рівняння- рівняння з ненульовим коефіцієнтом при X_i.

Перестановка рівнянь здійснюється переброской посилань P^.LS_Mem і Q^.LS_Mem.

Основна частина алгоритму - вилучення невідомого X_i. Вона складається з кроків 1, 2, 3, що виконуються послідовно.

Крок 1 - нормалізація ведучого рівняння, тобто ділення всіх членів рівняння на коефіцієнт при X_i.

Крок 2 - вилучення X_i з верхньої частини системи. j-те рівняння верхньої частини ($j < i$) має вид $X_j + A_{ji1}*X_{i1} + ... + A_{jik}*X_{ik} = B_j$, причому $i1 >= i$ оскільки воно нормалізовано і невідомі X_k, $k < i$, вилучені на попередніх кроках основного циклу. Тому вилучення здійснюється при $i1 = i$. Треба домножити ведуче рівняння на A_{ji1} і обчислити його з j-того рівняння.

Крок 3 - вилучення Xi з нижньої частини системи. j-те рівняння нижньої частини ($j > i$) має вид $A_{ji1}*X_{i1} + ... + A_{jik}*X_{ik} = B_j$, причому $i1 >= i$. Вилучення здійснюється точно також, як і на кроку 2.

Кроки 2 і 3 настільки схожі, що природно було би реалізувати їх одною і тою ж процедурою. Для цього необхідно всього лише реалізувати однакове представлення рівнянь з нижньої і верхньої частин системи, тобто не зберігати у списку рівняння верхньої частини системи відповідний ведучий член X_j !

Таким чином, на виході кроку 1 - процедури нормалізації ми повинні отримати "хвіст" $A_{ji1}*X_{i1} + ... + A_{jik}*X_{ik} = B_j$ нормалізованого рівняння. Основна частина алгоритму буде виглядати так:

```
{вилучення Xi з рівнянь системи}
{нормалізація і-того рівняння й отримання "хвоста"}
        EquNorm(P);
        Q := A;
        For j := 1 to n do begin
          If i <> j
    then  If Q^.LS_Mem^.Unknown = i
              {вилучення  Xi з j-того рівняння}
                then EquTrans(Q, P);
                Q := Q^.NextEqu
          end;
```

У нашій версії метода Гауса та його частина, яку називають зворотним ходом, співміщена з прямим ходом метода. Ми приводимо матрицю системи одразу до діагонального вигляду. Зазначимо, що структура даних виходу процедури SysSol змінилась: тепер і-тий член списку Y має вид <n + 1, значення X_i>. Процедура SysSol набула завершеного виду. Залишилось уточнити процедури EquNorm(P) і EquTrans(Q, P), які оброблюють окремі рівняння.

```
{нормалізація рівнянь і виділення "хвоста"}
 Procedure EquNorm(var P: LinSys);
        Var LCoef: Rational;
            TempRef: LinEqu;
      Begin
       TempRef := P^.LS_Mem;
       New(LCoef);
       LCoef^ := TempRef^.Coef^;                    {ведучий коефіцієнт }
       {установка посилання на “хвіст” рівняння}
       P^.LS_Mem := P^.LS_Mem^.NextMem;
       Dispose(TempRef);                            {збирання сміття}
       TempRef := P^.LS_Mem;                        {ініціалізація циклу}
       Repeat
         TempRef^.Coef := DivRat(TempRef^.Coef, LCoef);
         TempRef := TempRef^.NextMem                {наступний член}
       until TempRef = Nil;
       Dispose(LCoef)
   End                                                {процедури нормалізації};
```

Найбільш цікава з точки зору техніки програмування процедура EquTrans елементарного перетворення лінійного рівняння E_1 за допомогою ведучого рівняння E_2.

$E_1 := E_1 - C*E_2$

Якби рівняння були представлені масивами коефіцієнтів, це перетворення реалізувалось б за допомогою одного арифметичного циклу. У нашій же структурі даних необхідно реалізувати перетворення списків. Нехай рівняння мають вид

$E_1 = A_1*X + T_1$, $E_2 = A_2*Y + T_2$, де A_1*X, A_2*Y - перші члени списків (голови), а T_1 і T_2 - хвости списків. Можливі наступні випадки:

Випадок 1. Невідомі X і Y однакові. Тоді:

Якщо $A_1 - C*A_2 <> 0$

то $(A_1*X + T_1) - C*(A_2*X + T_2) = (A_1-C*A_2)*X + (T_1 - C*T_2)$

інакше $(A1*X + T_1) - C*(A_2*X + T_2) = (T_1 - C*T_2)$

Випадок 2. Номер X менше, ніж номер Y. Тоді:

$(A_1*X + T_1) - C*(A_2*X + T_2) = A_1*X + (T_1 - C*E_2)$

Випадок 3. Номер X більше, ніж номер Y. Тоді:

$(A_1*X + T_1) - C*(A_2*X + T_2) = (-C*A_2)*Y + (E_1 - C*T_2)$

У кожному з цих випадків права частина відповідного співвідношення - рівності є сума (голова + хвіст) результуючого рівняння, причому обчислення хвоста легко організувати за допомогою рекурсії і трохи складніше - ітерації. Ознака закінчення обчислень - вільні члени в головах списків і відсутність хвостів. Перед застосуванням описаного алгоритму (процедура EquDiff) обчислюється множник C і з рівняння, що перетворюється вилучається перший член. Всі обчислення супроводжуються “збиранням сміття” - звільненням пам’яті, що займається членами, які вилучаються.

```
        Procedure EquTrans(var Q: LinSys; P: LinSys);
            Var LCoef: Rational;
                CurP, CurQ: LinEqu;
        Procedure EquDiff(var RefQ, RefP: LinEqu; C: Rational);
            Var NextP, NextQ: LinEqu;
                NextC: Rational;
                 II ,JJ: Integer;
                Finish: Boolean;
{Вилучення члена рівняння з нульовим коефіцієнтом}
        Procedure MemDel(var RefQ: LinEqu);
             Var TempRef: LinEqu;
          Begin
              TempRef := RefQ^.NextMem;
              RefQ^.NextMem := RefQ^.NextMem^.NextMem;
              RefQ^.Coef := TempRef^.Coef;
              RefQ^.Unknown := TempRef^.UnkNown;
              Dispose(TempRef) {збирання сміття}
              End {процедури вилучення члена рівняння};
 {вставка члена рівняння}
        Procedure MemInc(var RefQ, RefP: LinEqu; C: Rational);
            Var C0: Rational;
                TempRef: LinEqu;
        Begin
              New(TempRef);
              TempRef^.Coef := RefQ^.Coef;
              RefQ^.Coef := MinusRat(MultRat(RefP^.Coef, C));
              TempRef^.Unknown := RefQ^.UnkNown;
              RefQ^.Unknown := RefP^.Unknown;
              TempRef^.NextMem := RefQ^.NextMem;
              RefQ^.NextMem := TempRef;  RefQ := TempRef
        End {процедури вставки члена рівняння};

        Begin
```

```
        NextQ := RefQ; {покажчики на голову списку}
        NextP := RefP;
        Finish := False; {признак закінчення обчислень}
    Repeat
        II := NextQ^.Unknown; {номера невідомих}
        JJ := NextP^.Unknown;
       {випадок 1 і випадок закінчення обчислень}
    If II = JJ
     then begin {обчислення коефіцієнта рівняння}
       NextC := SubRat(NextQ^.Coef, MultRat(NextP^.Coef, C));
       If II = n + 1
          then begin {випадок закінчення обчислень}
             NextQ^.Coef := NextC;
             Finish := True end
          else {випадок 1}
              if NextC^.Num <> 0
                then begin
                   NextQ^.Coef := NextC;
                   NextQ := NextQ^.NextMem;
                   NextP := NextP^.NextMem end
                else begin
                   MemDel(NextQ);
                    NextP := NextP^.NextMem end
                end
     else if II < JJ
        then {випадок 2}
           NextQ := NextQ^.NextMem
        else begin {випадок 3}
           MemInc(NextQ, NextP, C);
           NextP := NextP^.NextMem end
    until Finish
End {процедури EquDiff перетворення рівняння};
Begin
   {підготовка до виклику процедури EquDiff}
    CurQ := Q^.LS_Mem;
    CurP := P^.LS_Mem;
    New(LCoef);
    Lcoef^_:=_CurQ^.Coef^;
    Q^.LS_Mem :=_Q^.LS_Mem^.NextMem;
    Dispose(CurQ);
    CurQ := Q^.LS_Mem;
```

```
    EquDiff(CurQ, CurP, LCoef)
  End {вилучення невідомого};
```

Проектування процедур обробки рівнянь завершено з точністю до процедур арифметичних дій із раціональними числами. Наступний етап - проектування процедур модуля RAT.

5. Проектування модулів. Модуль RAT.

Модуль RAT містить всі засоби, що забезпечують застосування у програмах раціональних чисел. Сюди входять:

- описання типу Rational, констант типу Rational;
- процедури введення-виведення раціональних чисел;
- функції перетворення числових типів і типу Rational;
- функції арифметичних операцій і порівнянь раціональних чисел;
- засоби, що використовуються для реалізації вищевказаних процедур (внутрішні засоби модуля).

Проектування почнемо з визначення поняття раціонального числа. Раціональні числа - це дроби виду Num/Den, де Num - чисельник, а Den - знаменник. Для забезпечення коректності і єдиності представлення раціонального числа у виді пари <Num, Den> (дробу Num/Den) нехай виконуються наступні обмеження:

Den > 0, НОД(Num, Den) = 1, Якщо Num = 0, то Den = 1

Таке представлення ми будемо називати канонічною формою. Дії над дробами природно і зручно реалізовувати у виді функцій. Але функції мови Pascal повертають скалярні значення. Тому представлення дробу у виді запису у цьому випадку непридатне. Для того, щоб обійти цю чисто лінгвістичну перешкоду, уточнимо тип Rational як посилання на запис:

```
Type
  Rational = ^RatValue;
  RatValue = Record
  Num, {чисельник }
  Den: LongInt {знаменник}
End;
```

{Тип LongInt - один із цілочисельних типів в TP-6. Множина значень визначена константою MaxLongInt = 2^{31} - 1.}

Нижче описані деякі (далеко не всі) засоби RAT, необхідні для роботи з раціональними числами. Ключовою функцією модуля є функція CanRat, що приводить дріб до канонічної форми. У свою чергу, CanRat використовує цілочисельну функцію GCD, яка обчислює найбільший спільний дільник (НСД) двох натуральних чисел.

A / B = A div НСД(A, B) / B div НСД(A, B)

```
{----------НСД двох натуральних чисел----------}
Function GCD(u, v: LongInt): LongInt;
```

```
    Begin
      While (u <> 0) and (v <> 0) do
     If u > v
       then u := u mod v else v := v mod u;
       If u = 0 then GCD := v else GCD := u
    End;

{------------канонізація раціонального числа----------------------}
    Function CanRat(X: Rational): Rational;
        Var u, v, w: LongInt;
     Begin
       u := X^.Num; v := X^.Den;
       If  v = 0
          then CanRat := Nil {дріб із нульовим знаменником}
          else begin
            If v < 0 {знаменник > 0}
              then begin u := -u; v := -v end;
              w := GCD(Abs(u), v);
               If  w <> 1
                then {скорочення чисельника і знаменника}
                  begin u := u div w; v := v div w end;
                  New(R);
                  Z^.Num := u;
                  Z^.Den := v ;
                  CanRat := Z;
       end
    End;
```

Як уже відзначалось, процедури введення-виведення потребують дуже старанного проектування, що враховує багато факторів (зручність використання, захист від несанкціонованих дій і т.д.). Ми опишемо тільки їх тривіальні макети.

{ВВЕДЕННЯ-ВИВЕДЕННЯ}

```
    Procedure RatInp(var X: Rational);                    {Введення}
          Var Ch: Char;
       Begin
        New(X);
        Write('введення чисельника '); Read(X^.Num);
        Ch := ReadKey;
        If Ch = '/'
          then begin
            Write('введення знаменника '); Read(X^.Den);
```

```
                X := CanRat(X)
              end
              else X^.Den := 1
          End;

        Procedure RatPrint(X: Rational);                    {Виведення}
            Begin
             If  X = Nil
                  then Write('Ділення на нуль')
                  else begin
                    Write(X^.Num);
                    If X^.Den <> 1
                       then begin Write('/'); Write(X^.Den) end;
                  end;
              WriteIn
            End;
{АРИФМЕТИЧНІ ОПЕРАЦІЇ}
        Function AddRat(X, Y: Rational): Rational;          {Додавання}
          Begin
            New(Z);
            Z^.Num := X^.Num*Y^.Den + X^.Den*Y^.Num;
            Z^.Den := X^.Den*Y^.Den;
            AddRat := CanRat(Z)
          End;

        Function SubRat(X, Y: Rational): Rational;          {Віднімання}
          Begin
            New(Z);
            Z^.Num := X^.Num*Y^.Den - X^.Den*Y^.Num;
            Z^.Den := X^.Den*Y^.Den;
            SubRat := CanRat(Z)
          End;

        Function MultRat(X, Y: Rational): Rational;    {Множення}
          Begin
            New(Z);
            Z^.Num := X^.Num*Y^.Num;
            Z^.Den := X^.Den*Y^.Den;
            MultRat := CanRat(Z)
          End;
        Function DivRat(X, Y: Rational): Rational;          {Ділення}
```

```
            Begin
              If Y^.Num = 0
                    then DivRat := Nil
                    else begin
                      New(Z);
                      Z^.Num := X^.Num*Y^.Den;
                      Z^.Den := X^.Den*Y^.Num;
                      DivRat := CanRat(Z)
                    end
              End;

        Function MinusRat(X: Rational): Rational; {число з знаком мінус}
          Begin
            New(Z);
            Z^.Num := -X^.Num;
            Z^.Den := X^.Den;
            MinusRat := Z
          End;

{ПОРІВНЯННЯ}
        Function EquRat(X, Y: Rational): Boolean; {Рівність}
          Begin
             EquRat := (X^.Num = Y^.Num) and (X^.Den = Y^.Den)
           End;

Function GrtRat(X, Y: Rational): Boolean; {Порівняння на "більш"}
  Begin
             GrtRat := SubRat(X, Y)^.Num > 0
           End;

 {ПЕРЕТВОРЕННЯ ТИПІВ}
        Function RatReal(X:Rational): Real; {Раціональні в дійсні}
          Begin
             RatReal := X^.Num/X^.Den
           End;
Function IntRat(X: LongInt): Rational; {Ціле в раціональне}
          Begin
            New(Z);
            Z^.Num := X; Z^.Den := 1;
            IntRat := Z
           End;
```

Кожна з функцій, що описані вище, реалізує одну з дій у відповідності з його математичним визначенням. Так, наприклад, додавання спирається на формулу

A/B + C/D = CanRat((A*D + B*C)/(B*D))

Оскільки модуль планується використовувати при програмуванні багатьох прикладних задач, особливу увагу треба приділити його оптимізації. З цією метою необхідно, в першу чергу, виділити і оптимізувати ключеві засоби модуля. У нашому прикладі це функції GCD і CanRat.

Функцію GCD у професійних програмах реалізують звичайно мовами нижнього рівня, використовуючи так званий бінарний алгоритм Евкліда.

Ще один шлях підвищення ефективності - зменшення середньої кількості викликів функції CanRat у функціях модуля. У функції AddRat, наприклад, це можна зробити, розглянувши поряд з загальною формулою додавання її частинні випадки, в яких виклик CanRat не потрібний. Попутно зменшується і кількість арифметичних операцій. На справді, при визначенні додавання виділимо частинні випадки додавання дробу і цілого числа, а також дробів з рівними знаменниками

A/B + C/D = CanRat((A*D + B*C)/(B*D)); (*)
A/B + C/B = CanRat((A + C)/B);
A/B + C = (A + B*C)/B;
A + C/D = (A*D + C)/D

Функція AddRat, що реалізована у відповідності з цим визначенням, у середньому ефективніше, ніж , яка описана у модулі.

Оформлення модуля.

У системі програмування TP-6 поряд із стандартними модулями можна застосовувати і власні модулі. Модуль містить: заголовок модуля, інтерфейсну частину, реалізаційну частину і ініціалізаційну частину.

Заголовок модуля має вид:

Unit < Ім'я модуля >.

Ім'я модуля повинно співпадати з іменем файла, що містить цей модуль.

Інтерфейсна частина має вид:

Interface

<

- Описання констант, міток, типів і змінних, що доступні для використання зовнішніми програмами;
- Заголовки процедур і функцій, що доступні для використання зовнішніми програмами;
- Uses-директиви з іменами модулів, що доступні для використання зовнішніми програмами.

>

Все доступні зовні засоби модуля повинні бути описані в інтерфейсі цього модуля.

Реалізаційна частина має вид:

Implementation

<

- Описання всіх засобів модуля, які скриті від зовнішніх програм;
- Uses-директиви з іменами модулів, що використовуються в цьому модулі і скриті від зовнішніх програм;
- Повні описання всіх процедур і функцій модуля.

>

Ініціалізаційна частина - розділ операторів модуля. Він виконується перед виконанням зовнішньої програми, що використовує модуль. У зовнішню програму модуль включається директивою Uses.

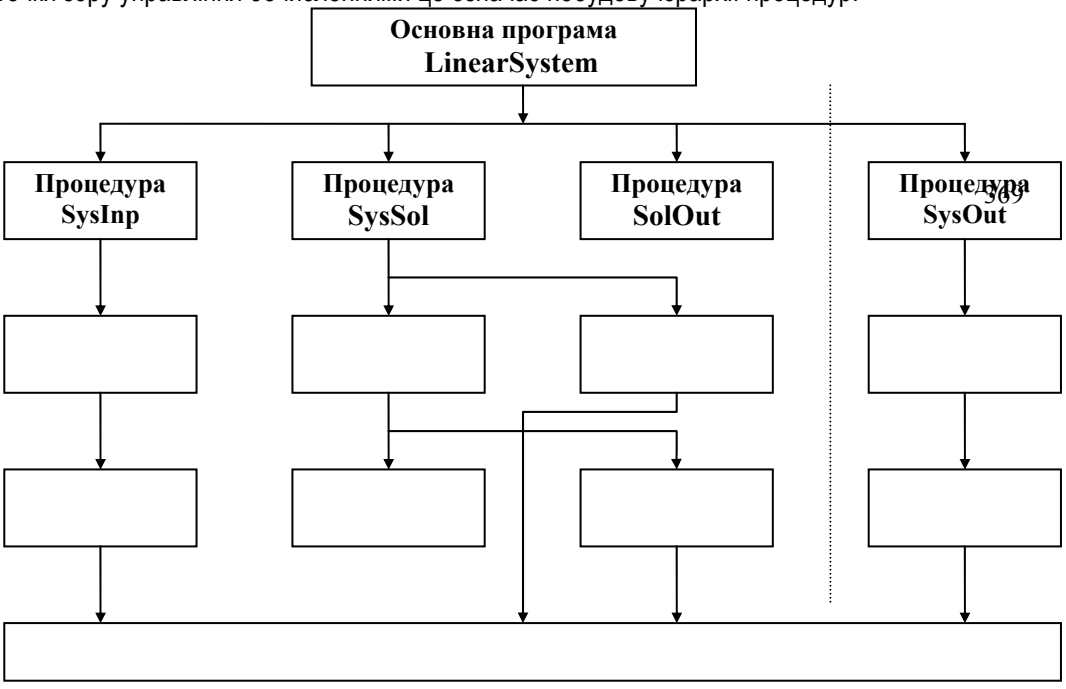
В якості приклада приведемо оформлення модуля RAT.

{заголовок модуля}

```
        Unit RAT; {<RAT.pas>}
        Uses CRT;
{інтерфейсна частина}
        Interface
           Type
             Rational = ^RatValue;
              RatValue = Record
                            Num, {чисельник }
                            Den: LongInt {знаменник}
                         End;
        Procedure RatInp(var X: Rational);
        Procedure RatPrint(X: Rational);
        Function AddRat(X, Y: Rational): Rational;
        Function SubRat(X, Y: Rational): Rational;
        Function MultRat(X, Y: Rational): Rational;
        Function DivRat(X, Y: Rational): Rational;
        Function MinusRat(X: Rational): Rational;
        Function EquRat(X, Y: Rational): Boolean;
        Function GrtRat(X, Y: Rational): Boolean;
        Function RatReal(X:Rational): Real;
        Function IntRat(X: LongInt): Rational;
{реалізаційна частина}
        Implementation
             Var Z: Rational;
 {повні описання всіх процедур і функцій модуля}
 {Ініціалізаційна частина}
 Begin
   ClrScr;
   Writeln(' Модуль Rat v.0.0 ')
 End.
```

6. Висновки.

Підведемо деякі висновки проектування програми LinearSystem. У процесі проектування "зверху-вниз" ми отримали чітко відокремлені за змістом чотири рівня програми: рівень системи, рівень рівняння, рівень члена рівняння, рівень коефіцієнтів. З точки зору управління обчисленнями це означає побудову ієрархії процедур:

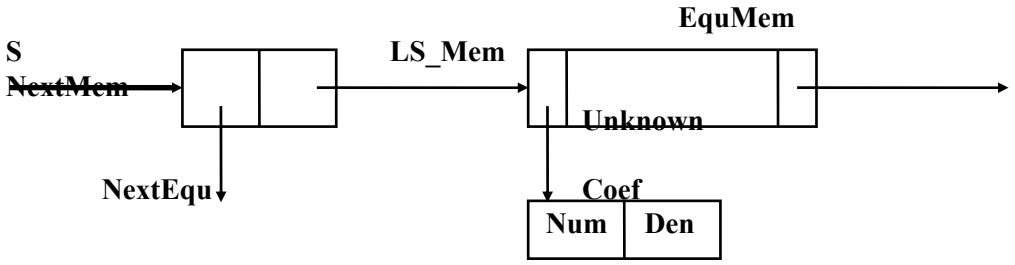


{Процедури SysOut, EquOut, MemOut використовувались для відладки і в тексті не описані.}

З точки зору даних це означає побудову ієрархії типів даних і відповідній ієрархічної структури даних:

Типи: LinSys ⟶ LinEqu {Monom} Rational

Структура даних:



{Тип Monom і поле EquMem було вирішено явним чином не вводити, оскільки на цьому рівні структура даних достатньо проста.}

Побудова програми як чотирьохрівневої ієрархії обумовлено ієрархією математичних абстракцій, що використовується в математичній теорії систем лінійних рівнянь:

Система -> Рівняння -> Член рівняння-> Коефіцієнт

Суворе слідування теорії (підкорення законам відповідній предметній області) необхідно для створення правильної (надійної) програми, а конкретна реалізація цієї ієрархії абстракцій обумовлена інженерною проблемою економії обчислювальних ресурсів.

Проектування модуля RAT представляє з себе окрему задачу - задачу з іншої предметної області. Замінивши модуль RAT, наприклад, модулем COMPLEX, у якому реалізовані дії з комплексними числами, ми отримаємо програму розв'язання системи лінійних рівнянь з комплексними коефіцієнтами. З іншого боку, наявність модуля RAT дозволяє легко програмувати і інші задачі, які потребують дій з раціональними числами.

7. Вправи.

1.Реалізувати всі функції арифметичних дій модуля RAT у відповідності з визначеннями, які аналогічні визначенню (*) функції AddRat.

2.Використовуючи модуль RAT і представлення системи лінійних рівнянь у виді n*(n+1) матриці, яка реалізована як двомірний масив, реалізувати іншу версію задачі розв'язка системи лінійних рівнянь.

3.Реалізувати модуль COMPLEX як розширення типу REAL. Підключити цей модуль замість RAT до програми LinearSystem і модифікувати цю програму для розв'язка систем лінійних рівнянь з комплексними коефіцієнтами.

4.Реалізувати модуль COMPRAT як розширення типу Rational. Використовувати цей модуль разом з RAT для модифікації програми LinearSystem, яка розв'язує системи лінійних рівнянь з комплексними раціональними коефіцієнтами.