АНОТАЦІЯ

Навчальний посібник охоплює першу частину курсів «Основи розробки трансляторів» та «Лінгвістичне забезпечення САПР». Викладено основи теорії формальних граматик, місце компілятора в програмному забезпеченні, методи побудови лексичного та синтаксичного аналізаторів. Наведені приклади та завдання для самостійної роботи студентів. Передбачається видання другої частини.

Призначений для студентів напряму підготовки «Комп'ютерні науки» та «Програмна інженерія», може бути використаний студентами інших спеціальностей.

ЗМІСТ

ВСТУП

В наш час активної комп'ютеризації суспільства мови високого рівня стали основним засобом розробки програм. Внаслідок постійного вдосконалення архітектури ЕОМ виникає необхідність розробки нових компіляторів (трансляторів) для універсальних та спеціальних мов програмування. Разом з цим ведуться розробки нових мов, розрахованих на вузькі предметні області. Майже кожна масштабна програмна система містить вбудовану мову, котра дозволяє користувачу налаштовувати функціональність системи під свої потреби. Таким чином, задача побудови мовних процесорів часто зустрічається в практичній діяльності програміста. При цьому методи розв'язання цієї практичної задачі базуються на складному теоретичному підґрунті: теорії формальних граматик, теорії автоматів, тощо. З огляду на вищесказане, вивчення теоретичних засад та методів практичної реалізації компіляторів є важливим етапом в підготовці фахівця в галузі інформаційних технологій.

Навчальний посібник розрахований на студентів, майбутніх фахівців в галузі інформатики та обчислювальної техніки. Він містить виклад перших двох розділів дисциплін «Основи розробки трансляторів» та «Лінгвістичне забезпечення САПР». Зазначені дисципліни включені до циклу «Професійної підготовки» варіативної частини навчальних планів підготовки бакалаврів з напряму «Програмна інженерія» спеціальностей «Програмне забезпечення систем» та «Інженерія программного забезпечення», а також напрямку «Комп'ютерні науки» спеціальності «Інформаційні технології проектування».

У структурно-логічній схемі навчання дисципліни рекомендується розміщувати після вивчення основ програмування та алгоритмічних мов, а також основ дискретної математики. Оскільки лабораторні роботи з дисциплін «Основи розробки трансляторів» та «Лінгвістичне забезпечення САПР» передбачають покрокову розробку досить складного програмного додатка (компілятора), студенти повинні мати певний досвід у програмуванні. З іншого боку, викладений матеріал може бути використаний при вивченні дисциплін «Програмне забезпечення інтелектуальних систем», «Моделювання складних процесів і систем», «Основи САПР складних об'єктів і систем», «Основи проектування систем штучного інтелекту».

Основною метою навчального посібника є допомога студентам в оволодінні теоретичними основами та практичними навичками в діяльності, спрямованій на розробку компілятора мови високого рівня. Для кращого сприйняття матеріалу в тексті приводиться велика кількість прикладів. Заради більшої наочності та інтуїтивної зрозумілості в посібнику випускаються складні математичні викладки, пов'язані з теоретичними засадами предмету, однак наводяться посилання на відповідні джерела.

В навчальному посібнику надаються основні теоретичні відомості, котрі повинен знати студент після вивчення дисципліни. Окреслюється місце компілятора в програмному забезпеченні ЕОМ та структура компілятора. Описуються основні поняття теорії формальних граматик, форми подання граматик мови. Розглядаються алгоритми побудови лексичного аналізатора: перегляд до роздільника, апарат скінченних автоматів. Розбираються алгоритми синтаксичного аналізу: рекурсивний спуск, використання магазинного автомату, висхідний розбір з використанням відношень простого передування. Наведені теоретичні відомості дозволяють студенту краще розуміти загальну структуру, що властива всім мовам програмуван-

ня та всім середовищам розробки програмних засобів. Це забезпечує в подальшому легку адаптацію програміста до будь-якої мови програмування та програмного середовища.

Матеріал викладається таким чином, щоб студент навчився створювати й описувати спеціалізовані мови, розробляти основні блоки транслятора: лексичний та синтаксичний аналізатор, використовуючи різні алгоритми та підходи.

РОЗДІЛ 1. МІСЦЕ КОМПІЛЯТОРА В ПРОГРАМНОМУ ЗАБЕЗПЕЧЕННІ ЕОМ

1.1. Основні функції та структура транслятора

Призначення транслятора мови високого рівня полягає в перетворенні початкового тексту програми у формат команд, котрі здатний сприймати комп'ютер [1, 2]. Наприклад для виконання простого оператора X=Y+Z необхідно перетворити його на наступну послідовність команд:

- звернення до області пам'яті, що містить Y;
- звернення до області пам'яті, що містить Z;
- додавання змінних Y і Z ;
- пересилка результату в область пам'яті, де знаходиться змінна X.

Таким чином, транслятор є програмою, котра здатна сприймати ланцюжок символів певного вигляду (тобто текст програми, написаний початковою мовою) і видавати інший рядок символів (програму на машинній мові, так звану об’єктну або цільову програму). Особливістю транслятора проблемно-орієнтованої мови є те, що його виходом може бути текст на внутрішній мові САПР: певні структури даних, які формуються з початкового тексту; команди виконавчих механізмів і таке інше. Якщо цільова програма написана машинною мовою, користувач може потім запустити її для обробки деяких вхідних даних і отримання вихідних.

Трансляторам властивий ряд загальних рис, що спрощує процес створення транслюючих програм. До складу будь-якого транслятора входять три основні компоненти:

1) лексичний аналізатор;
2) синтаксичний аналізатор;
3) генератор кодів машинних команд.

На фазі лексичного аналізу початковий текст програми, що являє собою послідовність непов'язаних один з одним символів, розбивається на одиниці, котрі називаються **лексемами**. Такими текстовими одиницями є службові слова, що використовуються в мові (наприклад, **IF**, **DO**, **BEGIN**, тощо), імена змінних, константи і знаки операцій (наприклад * або +). В подальшому ці слова розглядаються як неподільні утворення, а не як групи окремих символів.

Після розбиття тексту на лексеми іде фаза синтаксичного аналізу або граматичного розбору. В ході цієї фази розбору перевіряється правильність слідування лексем, тобто граматична правильність початкового ланцюжка. Наприклад, для оператора **IF**, що має вигляд:

IF <вираз> **THEN** <оператор>;

граматичний розбір полягає в тому, щоб переконатися, що після лексеми **IF** іде правильний вираз, а за цим виразом слідує лексема **THEN**, за котрою, в свою чергу, іде правильний оператор, що закінчується знаком «;».

Останнім виконується процес генерації коду, який використовує результати синтаксичного аналізу і формує вихід транслятора: програму на машинному коді, готову до виконання, або спеціальну структуру даних, залежно від мови та її призначення. Деякі транслятори видають об’єктну програму на асемблері, оскільки асемблерний код легше створити і відладжувати. Мова асемблера потім обробляється програмою асемблер, що видає в якості вихідних даних машинний код.

Окрім основних компонентів транслятор може містити також фази семантичного аналізу, генерації проміжного коду, різноманітні блоки оптимізації [1, 3-5].

Семантичний аналіз здійснюється зазвичай після синтаксичного, і виконує перевірку початкової програми на семантичну відповідність визначенню мови. На цьому етапі також збирається і зберігається інформація про типи змінних, виконується перевірка на узгодженість типів даних, що приймають участь в операціях. Під час перевірки типів можуть здійснюватись деякі автоматичні перетворення типів, наприклад, при складанні цілого та дійсного чисел, результат може видаватися або в цілому вигляді, або як число з плаваючою точкою, в залежності від того, який тип даних очікується на виході.

Для отримання ефективнішої об’єктної програми генерація машинного коду може здійснюватися в декілька етапів: спочатку може генеруватися проміжний код, зручний для оптимізації, а вже потім генерувати безпосередньо машинний код, котрий, в свою чергу, також може бути оптимізований. Проміжний машинний код може являти собою програму для абстрактної обчислювальної машини, і повинен легко генеруватися та легко перетворюватися в цільову машинну мову. Фаза оптимізації зазвичай має на меті скорочення часу виконання програми або скорочення вихідного коду, проте можуть бути застосовані інші критерії оптимізації. Існують так звані оптимізуючі компілятори [5], котрі витрачають багато ресурсів на оптимізацію коду, проте більшість компіляторів використовують достатньо прості алгоритми, що дозволяють швидко отримати достатньо ефективний цільовий код.

1.2. Взаємодія блоків транслятора

Хоча до складу будь-якого транслятора входять всі три описаних вище компоненти, їхня взаємодія може здійснюватися різними способами.

Трьохпрохідний транслятор

Нижче (рисунок 1.1) наведена схема взаємодії вищеописаних блоків, котра називається трьохпрохідним транслятором, оскільки програма зчитується тричі (початковий текст програми, файл лексем і файл представлення програми в деякій проміжній формі, наприклад, у постфіксній формі).

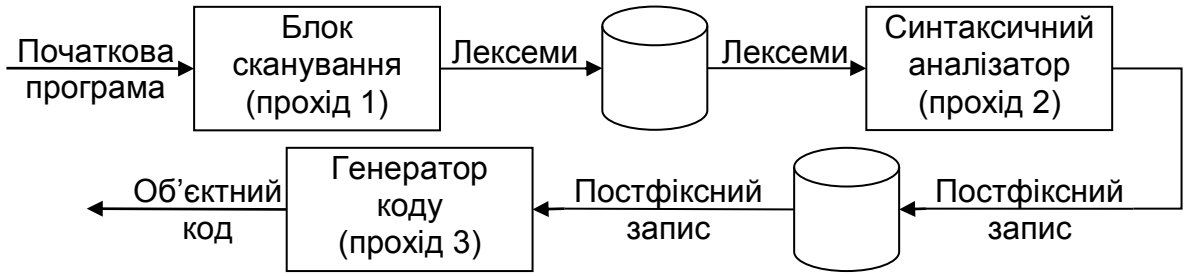


Рисунок 1.1 — Схема трьохпрохідного транслятора

Спочатку блок сканування зчитує початкову програму і подає її у формі файлу лексем. Синтаксичний аналізатор зчитує цей файл і видає нове подання програми в постфіксній формі. Нарешті, цей файл зчитується генератором коду, який створює об'єктний код програми.

Переваги такої організації:

1) відносна незалежність кожної фази трансляції: зв'язок між блоками здійснюється тільки через файли, кожен блок незалежний, що додає транслятору гнучкість – можливість зміни кожної фази;

2) використовується мінімальний об'єм оперативної пам'яті, що істотно, наприклад, при використанні міні ЕОМ.

Недолік: при такій організації швидкість транслятора не може бути високою, оскільки зазвичай операції, пов'язані із зверненням до файлів виконуються повільно.

Однопрохідний транслятор

Нижче (рисунок 1.2) наведено схему однопрохідного транслятора:

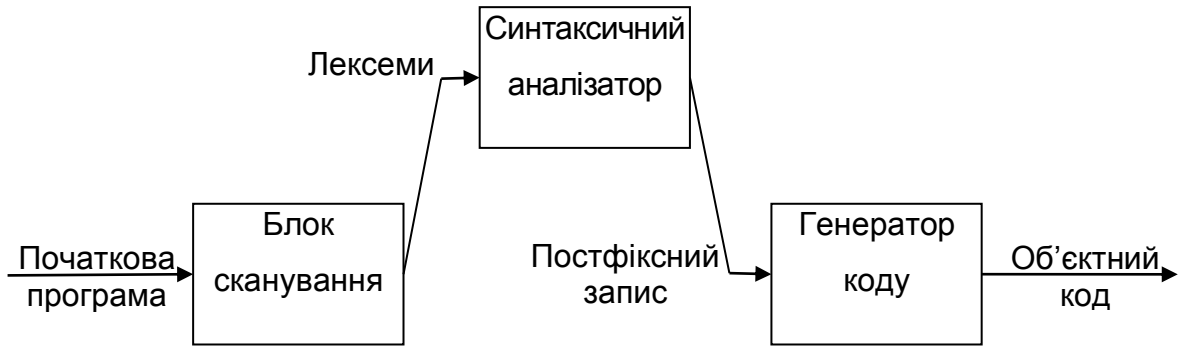


Рисунок 1.2 — Схема однопрохідного транслятора

В даному випадку синтаксичний аналізатор виступає в ролі основної керуючої програми, що викликає блок сканування і генератор коду, організовані у вигляді підпрограм. Синтаксичний аналізатор звертається до блоку сканування, отримуючи від нього лексему за лексемою з вхідної програми, доки не побудує новий елемент постфіксного запису. Після цього він звертається до генератора коду, який створює об'єктний код для цього фрагмента програми.

Переваги:

1) ефективність – програма зчитується 1 раз;
2) немає операцій звернення до файлів;

Недоліки:

1) не оптимальність створюваної об'єктної програми;

Наприклад, під час обробки фраз типу **GOTO <мітка>**; можуть виникнути труднощі, коли мітка ще не зустрічалася в програмі. Або, зустрівши текст:

A = (Y + Z)

D = (Y +Z) + (E+F)

транслятор міг би побудувати більш ефективний об’єктний код, трансформуючи програму наступним чином:

A = (Y + Z)

D = A + (E + F)

В той час, як при однопрохідній реалізації до того моменту, коли в тексті зустрінеться **(E + F),** інформація для такої підстановки вже може бути втрачена.

2) однопрохідний транслятор повинен повністю розміщуватися в пам'яті, тому висуваються підвищені вимоги до ресурсу пам'яті.

Для того, щоб об’єктна програма мала високу ефективність виконання, використовують трьохпрохідну організацію транслятора. При цьому, після роботи синтаксичного аналізатора підключають ще блоки оптимізації об'єктного та машинного коду.

Можливі й інші способи структурної організації [2]. Нижче (рисунок 1.3) показана структура **двохпрохідного компілятора**.

За такої організації синтаксичний аналізатор отримує від сканера лексеми і будує файл постфіксного запису. Генератор коду зчитує цей файл і будує об'єктний код. За такої організації виконується два звернення до

файлу замість трьох, як було в першому випадку, і легко вирішується ситуація з **GOTO**, на відміну від другого випадку.

Рисунок 1.3 — Схема двохпрохідного транслятора

В даний час ситуація з розробкою трансляторів ускладнюється тим, що центральні процесори цілого ряду моделей ЕОМ будуються на основі принципу мікропрограмного управління. Таким чином, є можливість створювати мікропрограми для нових машинних команд. Тобто до генератора коду транслятора може бути пред'явлена вимога гнучкості, а саме, уміння пристосовуватися до різних наборів машинних команд. Такої проблеми не виникає для іншого типу мовних процесорів, що називаються інтерпретаторами.

1.3. Інтерпретатор

Розглянутий підхід – трансляція, а потім виконання – характерний для побудови компіляторів. Принцип, альтернативний компіляції, реалізований в програмах, що називаються **інтерпретаторами**. В інтерпретаторі поточна інструкція мови виконується одразу після розбору [1, 7]. Наприклад, якщо є речення **X=Y+Z**, то замість побудови об'єктного коду для доступу до даних **Y** та **Z,** обчислення **Y+Z** і занесення результату в **X**,

інтерпретатор безпосередньо вибирає з пам'яті значення змінних **Y** і **Z**, сумує їх і записує отримане число в область, відведену для **X**.

Інтерпретатор і компілятор мають багато спільного. Інтерпретатор також перш за все переглядає початкову програму і виділяє в ній лексеми. При цьому використовуються блоки сканування й аналізатори, аналогічні тим, що вживаються в компіляторах. Проте генератор коду замість побудови об'єктного коду сам проводить відповідні дії.

Вимогам гнучкості відповідають транслятори, що створюють програму на деякій проміжній мові, наприклад, на асемблері. Такі транслятори так і називаються **асемблерами.**

Існують також комбіновані мовні процесори, вони можуть здійснювати трансляцію в деяке проміжне подання програми, а потім це проміжне подання виконується за принципами інтерпретатора. На цій концепції базується технологія .NET, в якості проміжного подання виступає програма на мові CIL, котре інтерпретується та виконується віртуальною машиною CLR.

1.4. Методика розробки компіляторів

Методика розробки компіляторів [7] базується на методиці розробки програмних засобів і включає 6 етапів.

1. Аналіз вимог (вхід – вихід).
2. Визначення специфікацій (швидкість, об’єм пам'яті, гнучкість застосування).
3. Проектування (загальна структура, вибір одного зі способів організації програми, принципи взаємодії компонентів).
4. Кодування.

5. Тестування.

6. Супровід (нові вимоги, усунення помилок).

Особливу роль грає перший етап: вхід і вихід. До початку інших робіт повинні бути сформовані опис вхідної мови, а також система команд або структура даних для виходу транслятора. Для опису мов програмування використовують теорію формальних граматик, що є розділом дискретної математики, цьому присвячено наступний розділ.

1.5. Завдання для самоконтролю

Завдання 1.1. Визначте функції транслятора.

Завдання 1.2. Перерахуйте основні та додаткові блоки транслятора.

Завдання 1.3. Перерахуйте види мовних процесорів.

Завдання 1.4. Визначте вхідні та вихідні дані компілятора, які дані вводить користувач, що отримує на виході.

Завдання 1.5. Побудуйте структуру двохпрохідного транслятора, в котрому в один прохід об’єднано синтаксичний аналіз та генерація коду.

Завдання 1.6. Спробуйте побудувати структуру транслятора, що містить фази лексичного, синтаксичного, семантичного аналізу, генератор коду машинних команд, таким чином, щоб він був

а) однопрохідним;

б) двохпрохідним;

в) трьохпрохідним.

Завдання 1.7. Визначити основні та додаткові задачі лексичного аналізатора.

Завдання 1.8. На якій фазі трансляції розпізнаються помилки типу:

а) неоголошений ідентифікатор;

б) невідповідність типів під час присвоєння;

в) невірно записане число;

г) неочікуваний кінець програми;

д) ділення на нуль;

е) відсутність або зайвий роздільник між командами;

ж) невизначена мітка;

з) невідповідність кількості відкриваючих та закриваючих дужок.

Завдання 1.9. Яке місце в структурі транслятора може займати блок оптимізації?

Завдання 1.10. Надайте порівняльну характеристику компілятора та інтерпретатора.

Завдання 1.11. Визначити переваги трансляторів, що видають код мовою асемблер.

Завдання 1.12. Використовуючи відповідні специфікації, визначити, до якого типу мовних процесорів відносяться: Turbo C, Borland Pascal, OpenBasic, VisualBasic, GNU Common Lisp, AutoLisp, Javac, Java SE, Factor, Active State Perl, Microsoft Fortran Power Station, TASM, COBOL.

Завдання 1.13. Перерахуйте комбіновані мовні транслятори, що Ви знаєте.

РОЗДІЛ 2. ОСНОВИ ТЕОРІЇ ФОРМАЛЬНИХ ГРАМАТИК

2.1. Граматики та мови. Символи та ланцюжки

Граматика мови програмування — це формальний опис її синтаксису, тобто форми, в якій записані окремі речення програми або вся програма. Граматика не описує семантику або зміст речень. Інформація про семантику міститься в програмах семантичного аналізу або генерації об'єктного коду.

Отже, на разі нас цікавить виключно синтаксис мови. Неформально можна визначити мову як підмножину всіх правильних речень складених зі «слів» або символів деякого базового словника або алфавіту.

Більшість концепцій, пов'язаних з організацією компіляторів, основані на формальній теорії мов [8-10]. Розглянемо основні поняття з теорії формальних граматик.

Алфавіт – це не порожня скінченна множина елементів. Будемо називати елементи алфавіту **символами**.

Будь-яка скінченна послідовність символів алфавіту A називається **ланцюжком** (рядком).

Наприклад, в алфавіті **A = {a,b,c}** ланцюжками є: **a, b**, **aaca**, **c.**

Нехай існує порожній ланцюжок, позначимо його символом $\wedge$.

Слід зауважити, що ланцюжок є впорядкованою послідовністю, тобто порядок слідування символів має велике значення: **ab** – не те саме, що **ba**.

Довжина ланцюжка x (записується |**x**|) дорівнює кількості символів ланцюжка.

Наприклад, |∧| = 0; |**a**| = 1; |**abb**| = 3.

Якщо **x** і **y** ланцюжки, то **конкатенацією xy** є ланцюжок, отриманий шляхом дописування символів ланцюжка **y** слідом за символами ланцюжка **x**.

Нехай **x** = **ab**, **y** = **ba**, тоді конкатенацією **xy** буде ланцюжок **abba**.

Оскільки ∧ – ланцюжок, що не містить символів, можна записати: **∧x = x∧ = x.**

Якщо **z** = **xy** – ланцюжок, то **x** – **голова**, **y** – **хвіст** ланцюжка **z**. Голова **x** називається правильною, якщо **y** – не порожній ланцюжок, а **y** – правильний хвіст, якщо **x** – не порожній ланцюжок.

Якщо **x** = **abc**, то, **a, ab, abc** – голови; **a, ab –** правильні голови.

Множина ланцюжків в алфавіті зазвичай позначається великими літерами **A, B ...**

Добуток двох множин ланцюжків **A** і **B** визначається як

$$\mathbf{AB = \{xy \mid x \in A,\ y \in B\}.}$$

Тобто добуток множин **A** і **B** є множиною всіх попарних конкатенацій ланцюжків **xy**, де **x** належить множині **A**, **y** належить множині **B**.

Наприклад: **A** = {**a,b**}, **B** = {**c,d**}, тоді множина **AB** = {**ac**, **ad**, **bc**, **bd**}. Очевидно {∧}**A** = **A**{∧} = **A**, де {∧} – множина, що містить лише порожній ланцюжок.

Ступінь ланцюжків визначається таким чином:

$$x^0 = \wedge$$
$$x^1 = x$$
$$x^2 = xx$$
$$\dots$$
$$x^n = \underbrace{x \dots x}_{n}$$

Визначимо тепер ступінь алфавіту A:

$\mathbf{A^0} = \{\wedge\}$,

$\mathbf{A^1} = \mathbf{A}$

$\mathbf{A^n} = \mathbf{AA^{n-1}}$

$\mathbf{A^n}$ – множина ланцюжків довжиною **n** в алфавіті **A** .

Визначимо дві останні операції: ітерацію **A*** та усічену ітерацію $\mathbf{A^+}$ над алфавітом **A**.

Ітерація (інакше, зірка Кліні або замикання Кліні) – це унарна операція над множиною символів, що записується як **A***, і визначається як множина усіх ланцюжків довжиною від 1 до ∞ над символами алфавіту **A**, включно з порожнім.

Усічена ітерація (інакше позитивне замикання або плюс Кліні) – це унарна операція над алфавітом, що записується як $\mathbf{A^+}$, і визначається як множина усіх ланцюжків довжиною від 1 до ∞ над символами алфавіту **A.**

$\mathbf{A^+} = \mathbf{A^1} \cup \mathbf{A^2} \cup ... \cup \mathbf{A^n} ...$

$\mathbf{A^* = A^0} \cup \mathbf{A^+}$.

Тобто, якщо A={a,b}, то A* = {∧, a, b, aa, ab, ba, bb, aab ...}.

Приклад 2.1

Нехай **z** = **abb**, тоді |**z**| = 3.

Голови **z:** ∧, **a, ab**, **abb**. При чому ∧, **a, ab –** правильні голови.

Хвости **z:** ∧, **b**, **bb**, **abb**. При чому ∧, **b**, **bb –** правильні хвости

Нехай **x** = **a, z** = **abb**. Тоді **zx** = **abba**, **xz** = **aabb**

$\mathbf{z^0} = \wedge$; $\mathbf{z^1}$ = **abb**; $\mathbf{z^2}$ = **abbabb**; $\mathbf{z^3}$ = **abbabbabb**;

$|\mathbf{z^0}| = 0$; $|\mathbf{z^1}| = 3$; $|\mathbf{z^2}| = 6$; $|\mathbf{z^3}| = 9$.

Нехай **S={a,b,c}**, тоді

S^+ ={a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, abc ...},

S* ={ ∧, a, b, c, aa, ab, ac, ba, bb, ...}.

Іноді в позначенні ланцюжків зручно використовувати три крапки:

z = **x...**, тут **x –** голова ланцюжка, хвіст не має значення;

z = **...x**, тут **x –** хвіст ланцюжка, голова не має значення;

z = **...x...**, означає, що **x** зустрічається десь в ланцюжку;

z = **S**..., символ **S –** перший символ ланцюжка;

z = **...S**, символ **S –** останній символ ланцюжка;

z = **...S**..., символ **S –** зустрічається десь в ланцюжку .

Мовою в алфавіті **M** називається деяка підмножина замикання Кліні базового алфавіту **L** $\subset$ **M*.**

Приклад 2.2

V = {a} – алфавіт.

V* = { $\wedge$, a, aa, aaa, aaaa ...}, нехай мова L включає ланцюжки тільки непарної довжини, тоді L = {a, aaa, aaaaa ...}.

Граматика визначає правила, за якими з **V*** можна виділити речення або ланцюжки, що належать **L**.

2.1.1. Формальна граматика

Граматика, за Хомським – це деякий алгоритм, котрий здатний визначити, чи належить даний ланцюжок до даної мови.

Існує 3 способи визначення приналежності ланцюжка мові:

1) **породжуючий** – алгоритм працює так, щоб згенерувати для заданої мови, тобто граматики, ланцюжок, що перевіряється;
2) **перераховуючий** – алгоритм генерує ланцюжки один за одним, доки випадковим чином не натрапить на потрібний ланцюжок;
3) **розпізнавальний** – за заданим ланцюжком спеціальним чином будується граматика.

На практиці використовується 1-й і 3-й варіанти, а відповідні грамати-ки називаються породжуючими та аналітичними (розпізнавальними).

Для опису формальних мов використовуються породжуючі граматики, за допомогою котрих задаються всі мови програмування.

Породжуюча граматика – це впорядкована четвірка:

$$\mathbf{G = <V_T, V_N, \sigma, P>,}$$

де

$\mathbf{V_T}$ – скінченна множина, що називається **термінальним словником** або основним алфавітом, елементи цієї множини називаються термінальними символами або терміналами;

$\mathbf{V_N}$ – скінченна множина, що називається **нетермінальним словником** або допоміжним алфавітом, її елементи називають нетермінальними симво-лами, нетерміналами, допоміжними символами або змінними;

$\mathbf{V_T \cap V_N = Ø}$ множини терміналів і нетерміналів не перетинаються.

σ – аксіома граматики, $\sigma \in \mathbf{V_N}$;

P – скінченна множина **правил підстановки** спеціального вигляду:

$$\mathbf{P = \{p: \alpha \rightarrow \beta\};\ де\ \alpha, \beta \in (V_T\ V_N)^*.}$$

Використовуючи поняття декартового добутку, можна записати, що

$$\mathbf{P} \subset (\mathbf{V_T} \cup \mathbf{V_N})^+ \times (\mathbf{V_T} \cup \mathbf{V_N})^*$$

Позначатимемо маленькими літерами символи термінального алфавіту, великими – нетермінального. Докладніше про моделі опису мови можна знайти в роботі Хомського [12].

Приклад 2.3

Граматика **G** може бути задана таким чином: **G** = <{**a, b**}, {**A, B**}, **A,** {**A→aBb, B→a, B→b**}>. Це означає, що в породженій цією граматикою мові, використовуються тільки символи з множини {**a, b**}. В граматиці наявні також два нетермінальні символи **A, B.** Аксіомою граматики **G** є нетермінал **A.** Здійснювати підстановки можна за правилами **{A→aBb, B→a, B→b**}, тобто нетермінал **A** можна замінити на послідовність **aBb,** а нетермінал **B** можна замінити або на **a** за другим правилом, або на **b** за третім правилом.

2.1.2. Класифікація граматик за Хомським

Класифікація базується на вигляді лівої та правої частини правил підстановки і включає 4 типи граматик.

ТИП 0: на ліву і праву частину правила **α→β** не накладається ніяких обмежень. Мови, що породжуються граматиками типу 0, співпадають з класом рекурсивно злічених множин. α, ß ∈ $(V_T \cup V_N)^*$.

ТИП 1: довжина лівої частини правила не більше за довжину правої частини, тобто | **α** | ≤ | **ß** |, граматика містить правила вигляду γA δ→ γ η δ, де $A \in V_N$, γ, δ, η$\in (V_T \cup V_N)^*$. Такі граматики називають контекстно-чутливими, контекстно-залежними або граматиками безпосередніх складових.

ТИП 2: ліва частина правил підстановки – довільний нетермінальний символ, на праву частину обмеження не накладаються, тобто граматика містить правила вигляду A→ ß, де **A**$\in$**V_N**, **ß** ∈ (**$V_T \cup V_N$**)*. Це **контекстно-незалежні (контекстно-вільні) граматики (КВ-граматики),** до яких відносяться граматики більшості мов програмування.

ТИП 3: ліва частина – довільний нетермінальний символ, права – один термінальний і один нетермінальний, один з яких може бути відсутнім, отже граматика містить правила вигляду A→ßB або A→Bß, де $A \in V_N$,

ß $\in (V_T \cup \{\wedge\})$, B $\in (V_N \cup \{\wedge\})$. Ці граматики називають **автоматними**, оскільки речення, породжуваних ними мов, розпізнаються скінченними автоматами.

Між мовами, що породжуються цими чотирма класами граматик, існує таке співвідношення: $L(G_3) \subseteq L(G_2) \subseteq L(G_1) \subseteq L(G_0)$.

2.1.3. Вивідність ланцюжка

Ланцюжок **η безпосередньо виводиться** з **ω** у даній граматиці **G**

$$\omega \Rightarrow \eta,$$

якщо **ω = αφß, η = αψß** та існує правило виводу **φ → ψ** $\in$ **P**.

Деяка послідовність ланцюжків називається **виводом**, якщо для будь-якої пари цих ланцюжків існує безпосередня вивідність, тобто

$$\omega_0\ \omega_1 \ldots\ \omega_n \text{ – вивід, якщо } \omega_{i-1} \Rightarrow \omega_i, \text{ де } i = 1,\ldots,n.$$

Якщо існує вивід ланцюжка **η** з **ω** у граматиці G, то говорять, що **η** виводиться з **ω**: **ω** $\mapsto$ **η**.

Вивід називається **повним**, якщо $\omega_0 = \sigma$ – аксіома, а ω_n – ланцюжок в термінальному словнику V_T^*.

Множина ланцюжків $\omega_n \in V_T^*$, котрі виводяться з аксіоми **σ,** називається **мовою**, що породжується граматикою **G,** і позначається **L(G)**.

Ланцюжок ω_i, що бере участь у повному виводі, називають **сентенціальною формою**.

Сентенціальна форма, що не містить нетермінальних елементів, називається **реченням**.

Множина всіх речень – **мова**, що породжується даною граматикою.

Фізично процес породження ланцюжка по граматиці є наступним: у множині правил **P** початкової граматики **G** обирається правило, ліва частина

котрого є аксіомою, таке правило повинно існувати. Розглядається його права частина, тобто деякий ланцюжок термінальних і нетермінальних символів. Потім в отриманому ланцюжку вибирається нетермінал і замінюється правою частиною відповідного йому правила.

Процес підстановок продовжується, доки ланцюжок не складатиметься виключно з термінальних символів. Порядок підстановки в породжуючих граматиках може бути довільним.

Визначимо, яка мова породжується граматикою, що наведена в прикладі 1.3, **G** = <{**a, b**}, {**A**, **B**}, **A, {A→aBb**, **B→a**, **B→b**}>.

Виконаємо підстановки:

0	**A**
1	**aBb**
2	**abb**

Тут для другої підстановки ми використовували правило **B→b**. Використовуючи правило **B→a,** отримаємо ланцюжок **aab**. Таким чином, мова, що задається граматикою **G**, містить два речення, а саме ланцюжки **aab** та **abb.**

Приклад 2.4

Розглянемо деяку граматику:

$$\mathbf{G = <V_T, V_N, \sigma, P>,}$$

де $\mathbf{V_T}$ = {ЗАТУЛЯЄ, СТАРИЙ, БУДИНОК, ДУБ};

$\mathbf{V_N}$ = {<речення>, <підмет>, <присудок>, <доповнення>, <прикметник>, <іменник>};

σ = <речення>;

множина правил підстановок **P** містить такі правила:

<речення>→<підмет> <присудок> <доповнення> .

<підмет>→<прикметник> <іменник>

<присудок>→ ЗАТУЛЯЄ

<доповнення>→<прикметник> <іменник>

<прикметник>→СТАРИЙ

<іменник>→БУДИНОК

<іменник>→ДУБ

Спробуємо вивести з аксіоми деяке правильне речення заданої мови, а потім визначимо множину усіх правильних речень.

Перелік сентенціальних форм або ланцюжків виводу:

0 <речення>

1 <підмет> <присудок> <доповнення>.

2 <прикметник> <іменник> <присудок> <доповнення>.

3 <прикметник> <іменник> <присудок> <прикметник> <іменник>.

4 СТАРИЙ <іменник> <присудок> <прикметник> <іменник>.

5 СТАРИЙ <іменник> ЗАТУЛЯЄ <прикметник> <іменник>.

6 СТАРИЙ <іменник> ЗАТУЛЯЄ СТАРИЙ <іменник>.

Замінивши <іменник> на одне з його визначень, отримаємо чотири варіанти правильних речень цієї мови:

- СТАРИЙ БУДИНОК ЗАТУЛЯЄ СТАРИЙ ДУБ.
- СТАРИЙ БУДИНОК ЗАТУЛЯЄ СТАРИЙ БУДИНОК.
- СТАРИЙ ДУБ ЗАТУЛЯЄ СТАРИЙ БУДИНОК.
- СТАРИЙ ДУБ ЗАТУЛЯЄ СТАРИЙ ДУБ.

Приклад 2.5

G = < $\mathbf{V_T}$, $\mathbf{V_N}$, **σ**, **P**>,

де $\mathbf{V_T}$ = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

$\mathbf{V_N}$ = {<число>,<чс>,<цифра>};

σ = <число>;

P = {

<число>→ <чс>

<чс>→<чс> <цифра>

<чс>→ <цифра>

<цифра>→ 0

<цифра>→ 1

<цифра>→ 2

…

<цифра>→ 9**}.**

Мова, що породжується цією граматикою — всі цілі десяткові числа без знаку. Побудуємо вивід числа 356.

0 <число>

1 <чс>

2 <чс> <цифра>

3 <чс> <цифра> <цифра>

4 <цифра> <цифра> <цифра>

5 3 <цифра> <цифра>

6 3 5<цифра>

7 3 5 6

2.1.4. Форми подання граматик

Поширені мови програмування, а також вхідні мови систем моделювання та проектування зазвичай відносяться до класу контекстно-незалежних мов (КН-мов), саме такі мови ми і розглядатимемо.

Іноді граматики КН-мови можуть задаватися лише множиною правил підстановок без явного вказування аксіоми, термінального і нетермінального словників. У таких випадках вважається, що ці множини ($\mathbf{V}_t$, $\mathbf{V}_n$) складаються саме з тих термінальних і нетермінальних символів, котрі зустрічаються в

правилах підстановок, при цьому в якості аксіоми виступає ліва частина найпершого правила.

Мови, за допомогою котрих описуються інші мови, називаються **метамовами**. Розглянемо одну з найпоширеніших метамов, котрі описують КН-мови.

Для опису граматик КН-мов, як правило, **застосовується форма Бекуса-Наура** [11] або **БНФ**.

У БНФ нетермінальні символи позначаються словами та записуються в кутових дужках < >, також використовуються 2 метасимволи:

1. ::= – дорівнює за означенням;

2. | – дозволяє об'єднати правила, що мають однакову ліву частину.

Всі інші символи вважаються термінальними.

Аксіомою граматики для мов програмування зазвичай виступає <програма>. Тобто якщо з аксіоми <програма> вивести сентенціальну форму, що складається виключно з терміналів, отримаємо синтаксично правильну програму. Оскільки нетермінальні символи є допоміжними, то вибір позначень для них може бути довільним. Однак кожний нетермінал визначає деяку структурну одиницю або синтаксичну категорію для всього ланцюжка, що породжується. Тому під час запису граматики (синтаксису) в якості нетермінальних символів використовуються слова, що підказують неформальне значення конструкції.

Таким чином, нетермінальний словник є **термінологічним словником** для даної мови.

Для опису синтаксису формальних мов на практиці знайшли застосування також графічні форми подання граматик: синтаксичні дерева і синтаксичні діаграми. Розглянемо ці форми на прикладі.

Приклад 2.6

Побудуємо БНФ, синтаксичне дерево та синтаксичну діаграму для граматики цілого числа.

- БНФ-граматика

<ціле число>::=**Ошибка! Ошибка связи.**<знак><ціле без знаку> | <ціле без знаку>

<ціле без знаку>::=<ціле без знаку><цифра> | <цифра>

<знак>::=+|-

<цифра>::=0|1|2|3|4|5|6|7|8|9

Очевидно, що V_N={<ціле число>, <ціле без знаку>, <знак>, <цифра>}; V_T={+,-,0,1,2,3,4,5,6,7,8,9}; σ =<ціле число>.

- синтаксичне дерево (рисунок 2.1)

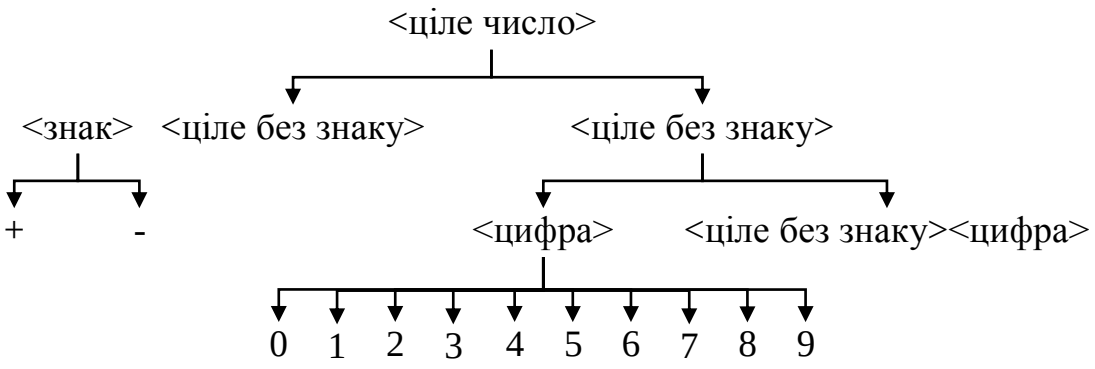


Рисунок 2.1 — Синтаксичне дерево граматики цілого числа

Пригадаємо визначення довільного дерева.

Дерево – це скінченна множина T, можливо пуста, що складається з одного або більше елементів (вузлів або вершин дерева) таких, що:

- є один спеціально позначений елемент – корінь даного дерева;
- інші елементи містяться в m > 0 множинах T1,. . . , Tm, що попарно не перетинаються, кожна з яких у свою чергу є деревом;

Дерева T1,. . . , Tm називаються піддеревами даного кореня.

Вузол *y*, який знаходиться безпосередньо під вузлом *x* має назву (безпосереднього) **нащадка** *x*. Якщо *x* знаходиться на рівні *i*, то у знаходиться на рівні *i+1*. Вузол *x* по відношенню до вузла *y* має назву (безпосереднього) **предка**.

Число піддерев деякого вузла називається **ступенем** вузла. Вузол з нульовим ступенем називається **кінцевим вузлом** (або листом або термінальним вузлом), всі інші елементи – внутрішні вузли (нетермінальні). Максимальний ступінь всіх вершин називається **ступенем дерева**. Корінь дерева має нульовий рівень.

Таким чином, коренем синтаксичного дерева є аксіома граматики. Внутрішніми вузлами можуть виступати тільки нетермінальні символи, їхніми дочірніми елементами є ланцюжки термінальних та нетермінальних символів, котрі є альтернативними визначеннями відповідного правила БНФ-граматики. Якщо правило має лише одну альтернативу, відповідний вузол має лише одного нащадка. Якщо нетермінальні символи повторюються у визначеннях (наприклад, <ціле без знаку> повторюється в нашому дереві тричі), будувати відповідні піддерева необхідно лише для одного з них.

- синтаксична діаграма (рисунок 2.2)

Синтаксична діаграма нагадує орієнтований граф. Вузлами цього графа є термінальні, нетермінальні символи та допоміжні вузли, при чому термінальні символи позначають вузлами з овальними рамками, а нетермінальні – з прямокутними, допоміжні вузли – точки. Кожному правилу відповідає зв’язний підграф, проте для зручності кілька правил можуть бути об’єднаними в один зв’язний підграф (в нашому випадку, підграф правила, що визначає нетермінал <знак>, було вставлено всередину підгафа, відповідного нетермінала <ціле число>).

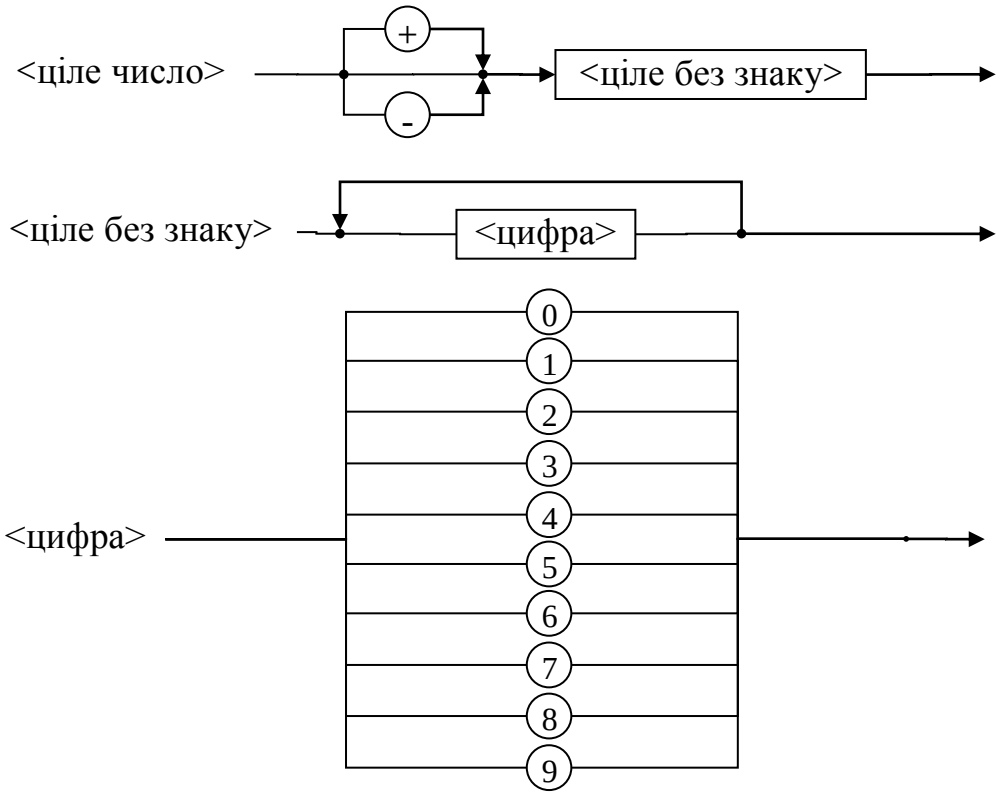


Рисунок 2.2 — Синтаксична діаграма граматики цілого числа

Початковий вузол кожного зв’язного підграфа не має рамки і містить нетермінальний символ, що визначається відповідним правилом, тобто є лівою частиною правила. Від цього вузла відходить одна або кілька дуг в залежності від кількості альтернативних визначень. Кожне визначення є ланцюжком терміналів і нетерміналів, котрий на діаграмі зображується послідовно пов’язаними вузлами. Рекурсивні правила зображуються циклами. Маршрути або шляхи в графах складають ланцюжки, котрі можна отримати при підстановці за відповідним правилом.

2.1.5. Рекурсивні правила

Правило називається **рекурсивним**, якщо воно має наступний вигляд:

$$A \rightarrow \alpha A ß$$

1. Правило називається ліворекурсивним, якщо: **α** = ∧ , тобто **A** → **A ß**.

2. Правило називається праворекурсивним, якщо: **ß**= ∧ , тобто **A** → **α A**

3. Правило називається рекурсивним з самовставкою, коли: **α ,ß** ≠ ∧, тобто **A** → **α A ß**.

Для того, щоб мова була нескінченною, її граматика повинна містити хоча б одне рекурсивне правило (див. визначення <ціле число>).

Це був випадок прямої рекурсії. Існує поняття **непрямої рекурсії**. Нехай існує послідовність правил:

1. $A_0 \rightarrow \alpha_1 A_1 ß_1$
2. $A_1 \rightarrow \alpha_2 A_2 ß_2$
...
N. $A_n \rightarrow \alpha_{n+1} A_0 ß_{n+1}$

Визначення A_0 є рекурсивним.

На практиці праву рекурсію іноді можна замінити лівою і навпаки.

КВ-граматика є автоматною, якщо вона не містить правил з самовставкою.

Необхідно знати наступний зв'язок між типами граматик і пристроями, що розпізнають відповідні мови (таблиця 2.1).

Таблиця 2.1. Відповідність мов та розпізнавальних пристроїв

Мови, що породжуються граматиками типу	**Автомат, що розпізнає**
0	Машина Тюрінга
1	Недетермінований лінійно-зв'язаний автомат
2	Автомат з магазинною пам'яттю
3	Автомат зі скінченною кількістю станів (скінченний автомат)

Приклад 2.7

Запишемо граматику, що визначає ціле невід'ємне десяткове число.

<ціле без знаку >::=<цифра> | <ціле без знаку >< цифра >

<цифра>::=0|1|2|3|4|5|6|7|8|9

або з використанням правої рекурсії:

<ціле без знаку >::=<цифра> | < цифра ><ціле без знаку >

<цифра>::=0|1|2|3|4|5|6|7|8|9

Так просто замінювати ліву рекурсію правою не завжди можливо. Наприклад, у визначенні ідентифікатора (послідовності букв і цифр, що починається з букви) допускається тільки ліва рекурсія.

Приклад 2.8

<ідентифікатор> ::= <буква> | <ідентифікатор> <буква> | <ідентифікатор> <цифра>

<буква>::=a|b|c| .|z

<цифра>::=0|1|2|3|4|5|6|7|8|9

2.2. Дерева виводу

Ми визначили КН – мову, що задається деякою граматикою, як множину термінальних ланцюжків, котрі можна вивести з початкового символу, аксіоми. Можна побудувати дерево виводу, інтерпретуючи підстановки як кроки побудови дерева.

Нехай є граматика:

S → aABc

S → ∧

A → cSB

A → Ab

B → bB

B → a

Побудуємо дерево виводу для **S** ↦ **acabac** (рисунок 2.3).

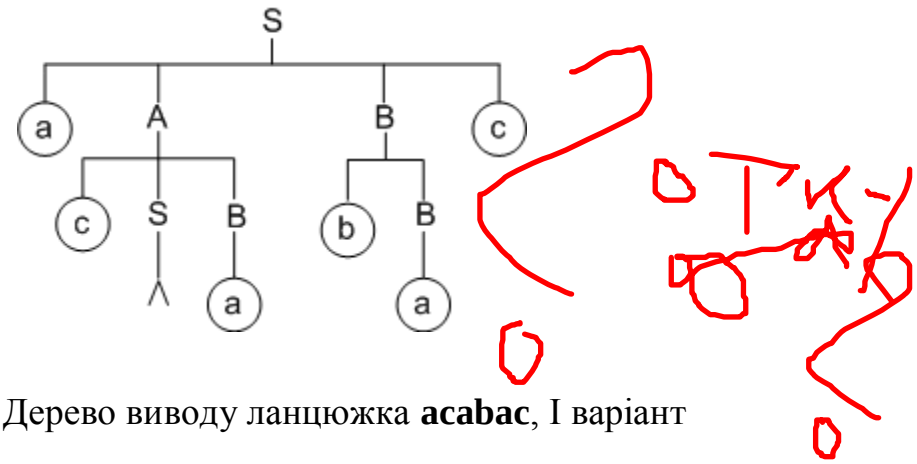


Рисунок 2.3 — Дерево виводу ланцюжка **acabac**, I варіант

Оскільки порядок підстановок в дереві прихований, то може бути багато виводів, відповідних одному дереву. Наприклад, даному дереву відповідають 3 наступні виводи:

1)	**S**	2)	**S**	3)	**S**
	aABc		**aABc**		**aABc**
	acSBBc		**aAbBc**		**aAbBc**
	acBBc		**aAbac**		**aAbac**
	acaBc		**acSBbac**		**acSBbac**
	acabBc		**acBbac**		**acSabac**
	acabac		**acabac**		**acabac**

Якщо на кожному кроці замінюється найлівіший нетермінал, вивід називається **лівостороннім** або **лівим**, якщо найправіший – **правостороннім** або **правим**.

Більшість методів обробки мов розраховані виключно на ліві або праві виводи, оскільки вони зручні для систематичного обходу дерева.

В загальному випадку одному термінальному ланцюжку може відповідати декілька дерев виводу. В цьому випадку говорять, що граматика **неоднозначна.**

Наприклад, в нашій граматиці той самий ланцюжок **acabac** можна побудувати відповідно до іншого дерева (рисунок 2.4).

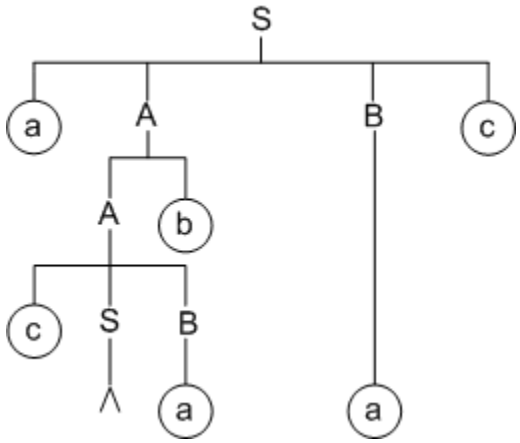


Рисунок 2.4 — Дерево виводу ланцюжка **acabac**, II варіант

Підсумуємо все вищесказане.

1. Кожному ланцюжку, що виводиться в даній граматиці, відповідає одне або кілька дерев виводу.
2. Кожному дереву відповідає декілька виводів.
3. Кожному дереву відповідає єдиний правий і єдиний лівий вивід.
4. Якщо кожному ланцюжку, що виводиться в даній граматиці КН-мови, відповідає єдине дерево виводу, то ця граматика називається однозначною, в іншому разі її називають неоднозначною.

Приклад 2.9. Граматика для константи

Для розробки штучних мов необхідно уміти складати їхню граматику. Як приклад побудуємо граматику для констант, котрі можуть бути представлені десятковими числами з фіксованою і плаваючою точкою без знаку перед числом.

<константа> ::= <дес.число> | <дес.число> E <ціле>

<дес.число> ::= <ціле> | . <цбз> |<ціле>. | <ціле>. <цбз>

<ціле> ::= + <цбз> | –<цбз>| <цбз>

<цбз>::= <цбз> <цифра> | <цифра>

<цифра>::=0|1|2|3|4|5|6|7|8|9

Отримаємо за допомогою лівого виводу константу **5.7E-15** :

<константа> ⇒ <дес.число> E <ціле> ⇒ <ціле>. <цбз> E <ціле> ⇒ <цбз> . <цбз> E <ціле> ⇒ <цифра>. <цбз> E <ціле> ⇒ 5. <цбз> E <ціле> ⇒ 5.<цифра> E <ціле> ⇒ 5.7E <ціле> ⇒ 5.7E- <цбз>⇒ 5.7E- <цбз><цифра> ⇒ 5.7E-<цифра> <цифра> ⇒ 5.7E-1<цифра> ⇒ 5.7 E–15

Побудуємо дерево виводу (рисунок 2.5).

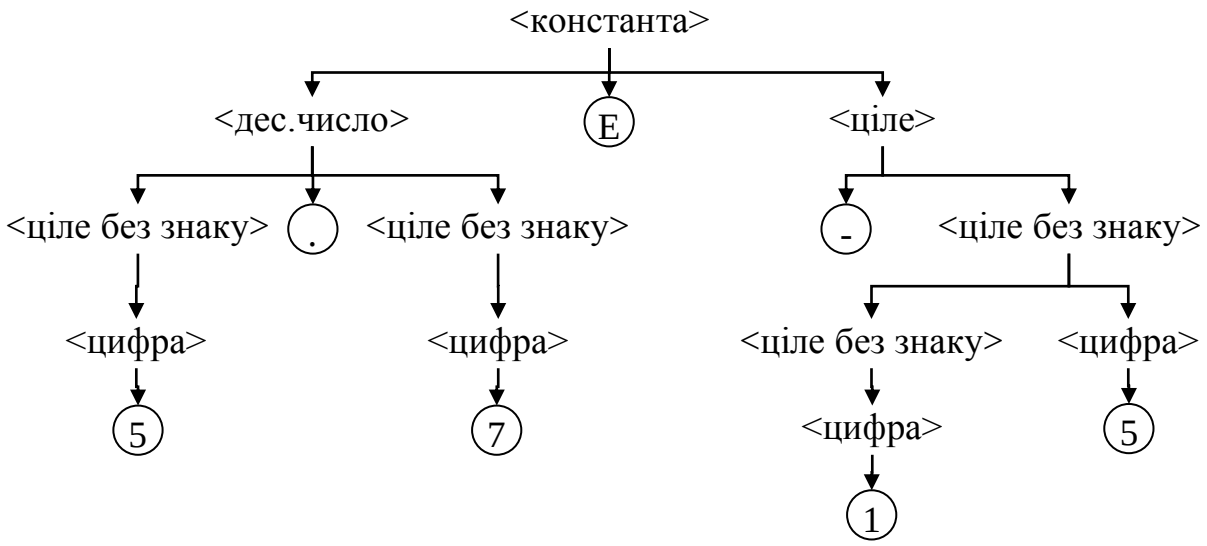


Рисунок 2.5 — Дерево виводу «5.7E-15»

Якщо прочитати листки дерева виводу (рисунок 2.5) зліва-направо, отримаємо речення **5.7E-15.**

2.3. Граматика мови програмування

Арифметичний вираз є невід'ємною частиною будь-якої мови програмування, тому розглянемо його граматику детально.

Побудуємо граматику для арифметичного виразу, в якому використовується єдиний операнд **i**, бінарні операції +, * та дужки. Її можна було б записати таким чином

<E>::=<E> + <E> | <E>*<E> | (<E>) | i

Тоді для ланцюжка **i+i*i** існують два дерева виводу (рисунок 2.6), тобто граматика – неоднозначна і неприйнятна для використання на практиці. Якщо обхід дерева асоціювати з виконанням операцій, то відповідно до першого дерева спочатку виконається операція *, потім +, а відповідно до другого – спочатку +, потім *. Приймаємо пріоритет виконання операцій, відповідний другому варіанту.

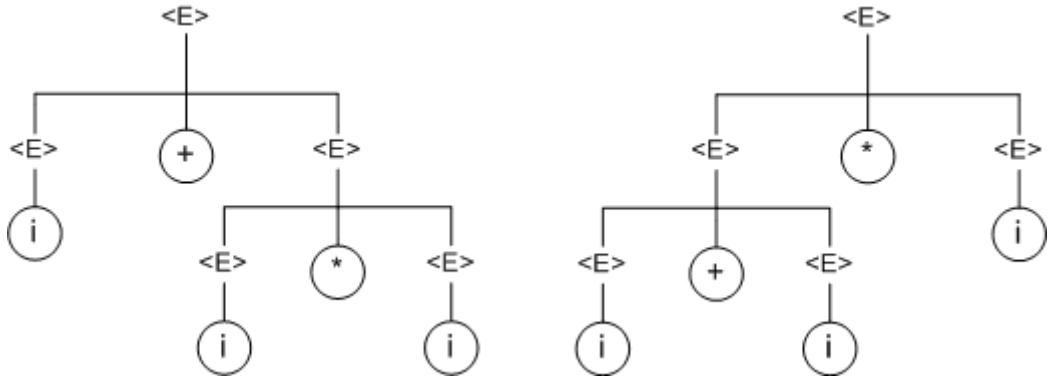


Рисунок 2.6 — Дерева виводу **i+i*i** для першої граматики

Рознесемо операції різного пріоритету по різних правилах. Представимо вираз як суму деяких доданків, що називаються термами, і зафіксуємо це

першим правилом граматики. Представимо кожний доданок як добуток елементів, що називаються множниками, і зафіксуємо другим правилом. Тоді кожний множник можна представити як операнд **i** або вираз в дужках. Отримаємо наступну граматику.

<вир> ::= <терм.> | <вир> + <терм.>| <вир>- <терм.>

<терм.> ::= <множ> | <терм.> * <множ> | <терм.> / <множ>

<множ> ::= (<вир>) | i

Отримана граматика для ланцюжка **i+i*i** дає єдине дерево виводу (рисунок 2.7).

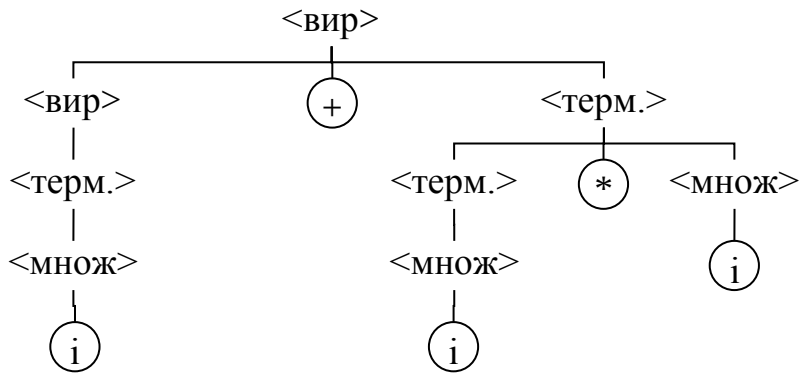


Рисунок 2.7 — Дерево виводу **i+i*i** для другої граматики

У цьому ж дереві відображено, що операндом для операції + є результат операції *, тобто операція * виконується раніше.

Нехай в арифметичному виразі допускається використовувати ще й унарний мінус та операцію піднесення до ступеня, котра виконується раніше, ніж * та / . Тоді для врахування її пріоритету необхідно ввести в граматику ще одне правило. Представимо кожен множник як послідовність деяких первинних виразів, сполучених знаками піднесення до ступеня «^». І, відповідно, первинний вираз як мінімальний елемент арифметичного виразу міститиме операнд **i** або вираз в дужках. Складемо граматику таким чином, щоб унар-

ний мінус можна було використовувати лише один раз і лише для першого доданка, таким чином уникнемо можливості написання двох мінусів підряд.

<вир> ::= <терм.> | <вир> + <терм.>| <вир>- <терм.> | - <терм.>

<терм.> ::= <множ> | <терм.> * <множ> | <терм.> / <множ>

<множ> ::= <перв.в.> | <множ> ^ <перв.в.>

<перв.в.> ::= (<вир>) | <цифра>

<цифра> ::= 0|1|2|3|4|5|6|7|8|9

Побудуємо дерево виводу для виразу 1+2*3+(5+6)*7 (рисунок 2.8).

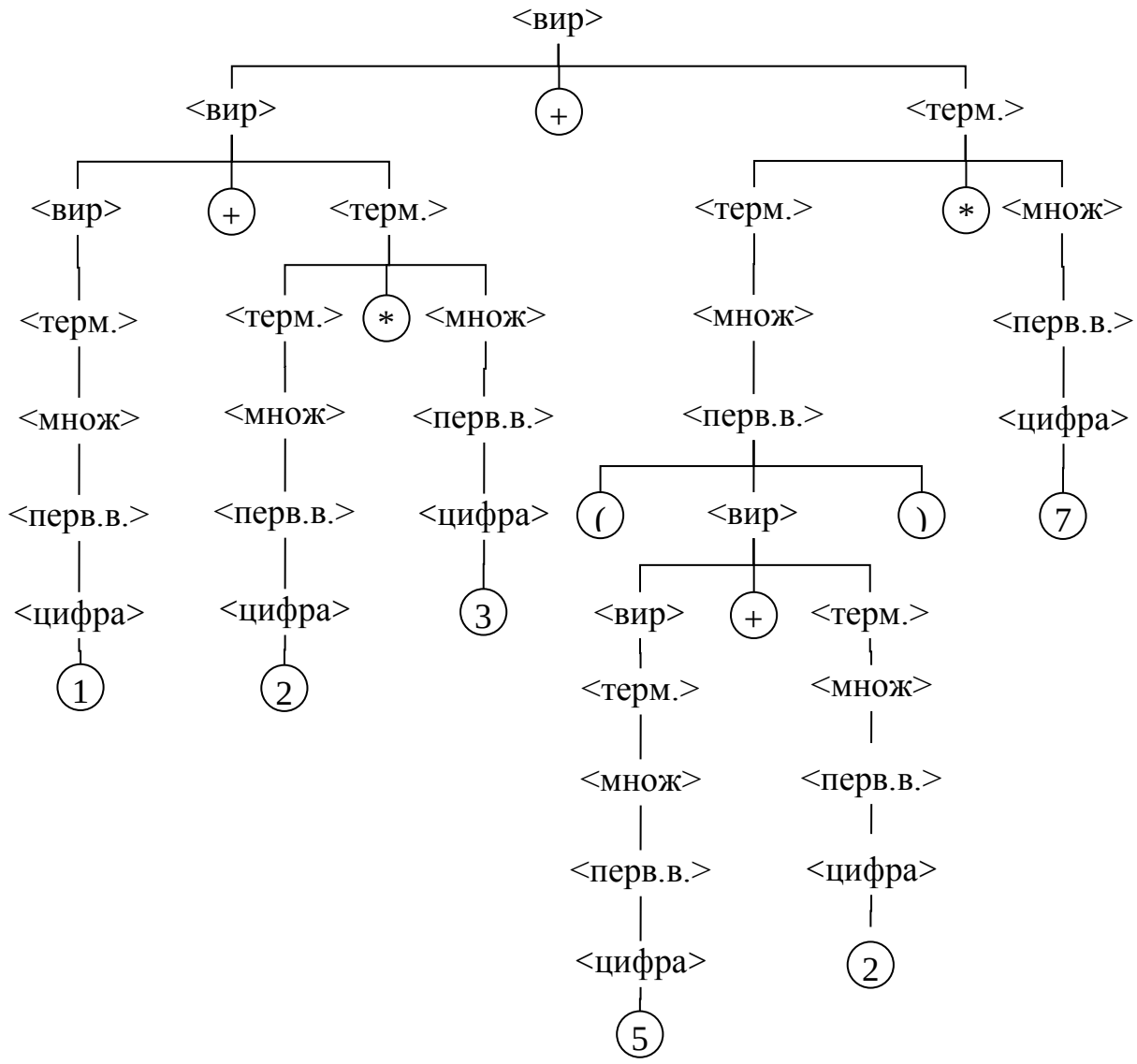


Рисунок 2.8 — Дерево виводу 1+2*3+(5+6)*7

Приклад 2.10

Визначимо оператор введення наступною граматикою:

<опер.вв.>::=READ (<список імен>)

<список імен>::=<ім’я> |<список імен>, <ім’я>

<ім’я>::=<буква>|<ім’я> <буква>|<ім’я> <цифра>

<буква> ::= A|B|C| … |Z

<цифра> ::= 0|1|2|3|4|5|6|7|8|9

Побудуємо дерево виводу для оператора введення **READ (A, BETA)** (рисунок 2.9), виведення <ім’я>↦ A та <ім’я>↦ BETA спростимо.

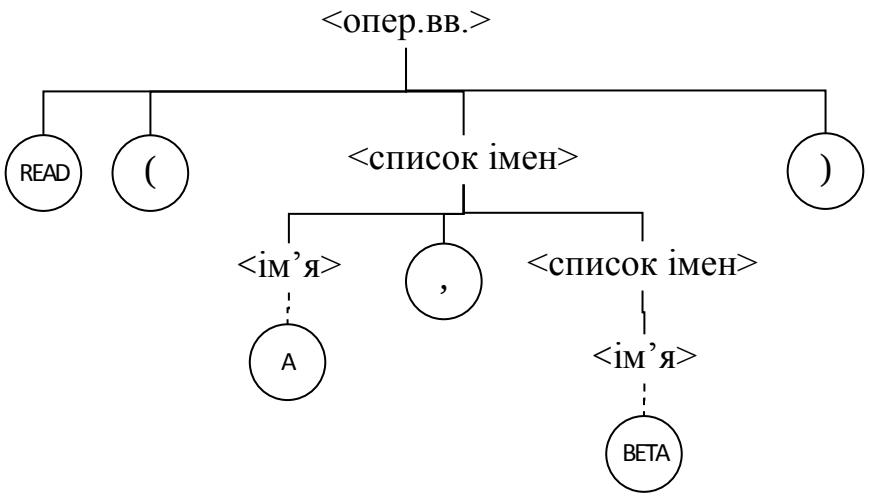


Рисунок 2.9 — Дерево виводу READ (A, BETA)

Приклад 2.11

Побудуємо граматику логічного виразу. Будемо враховувати пріоритет виконання логічних операцій аналогічно тому, як це здійснювалося в граматиці арифметичного виразу. Пригадаємо, що логічна операція **and (і)** має вищий пріоритет за операцію **or (або)**, тобто виконується раніше. Також будемо враховувати унарну операцію **not**, котра має найвищий пріоритет.

<ЛВ> ::= <ЛТ> | <ЛВ> or <ЛТ>

<ЛТ> ::= <ЛМ> | <ЛТ> and <ЛМ>

<ЛМ> ::= <відн.> | not <ЛМ> | [<ЛТ>]

<відн.>::=<вираз> <знак відн.> <вираз>

<знак відн.> ::= ≠ | ≤ | ≥ | < | > | =

Побудуємо дерево виводу для оператора **not (a>b) and c=0 or a=0**, спрощуючи виводи відношень (рисунок 2.10).

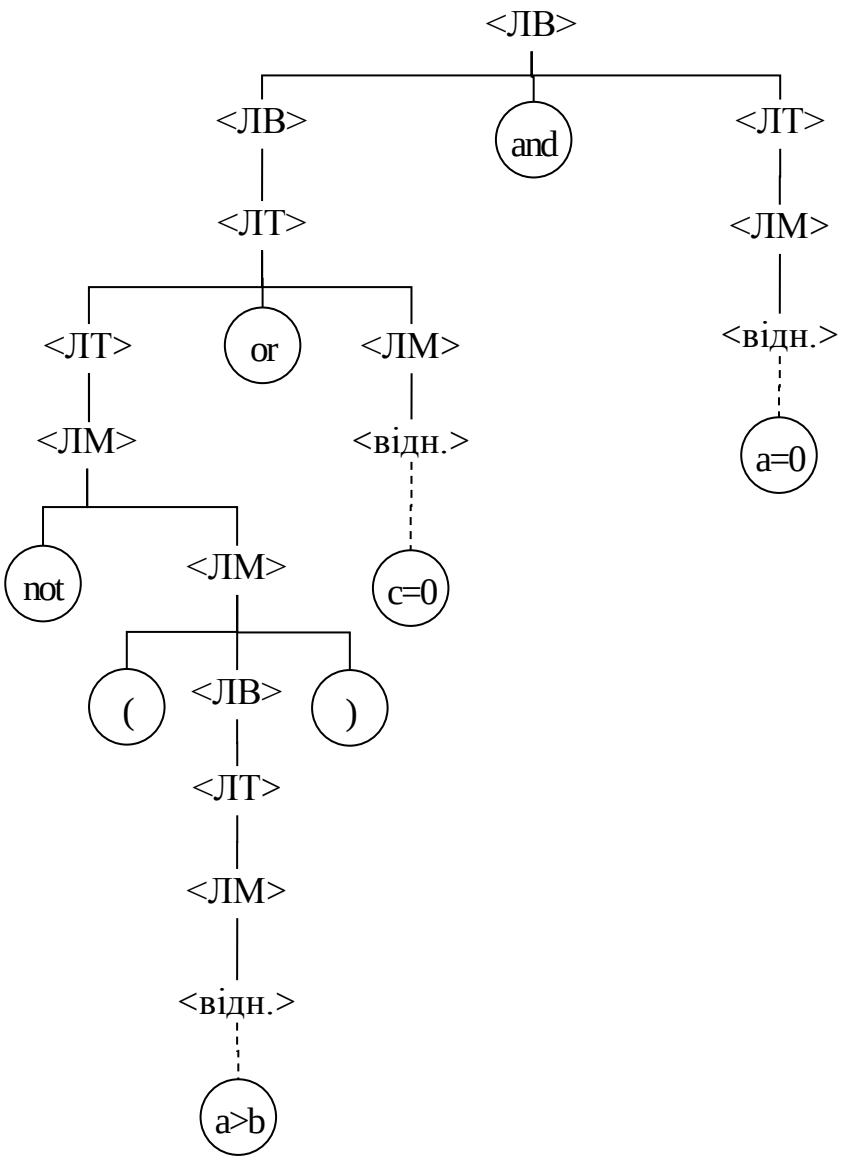


Рисунок 2.10 — Дерево виводу not (a>b) or c=0 and a=0

Отже, ми створили граматики для оператора вводу, арифметичного та логічного виразу, тепер можна розробити граматику для елементарної мови

програмування. Нехай ця мова буде схожа на Pascal, і містить оператори введення та виведення, присвоювання та цикл for. Програма буде починатися з зарезервованого слова «program», далі буде вказуватися ім'я програми. Після чого йтиме блок оголошень змінних, будемо використовувати наступні типи змінних: integer та real. Константи можна буде подавати у вигляді десяткових цифр з фіксованою точкою. Нехай арифметичний вираз містить лише оператори додавання, віднімання, множення та ділення, а також дужкові конструкції. Логічний вираз взагалі не будемо використовувати, через відсутність конструкцій умовного переходу.

Приклад 2.12

Побудуємо граматику мови, яку назвемо Міні-Паскаль.

1. <прогр> ::= program <назва> var <спис. огол.> begin <сп. опер.> end.
2. <назва>::= <ід.>
3. <спис. огол.> ::= <оголошення>| <спис. огол.>; <оголошення>
4. <оголошення> ::= <сп. ід.>: <тип>
5. <тип>::= integer | real
6. <сп. ід.> ::= <ід.> |<сп. ід.> , <ід.>
7. <сп. опер.> ::= <опер.> | <сп. опер.> ; <опер.>
8. <опер.> ::= <присв.> | <введення> | <виведення> | <цикл>
9. <присв.>::= <ід.>:= <вираз>
10. <введення>::= read (<сп. ід.>)
11. <виведення>::= write (<сп. ід.>)
12. <цикл>::= for <індекс. вир.> do <дія>
13. <вираз> ::= <терм> | <вираз> + <терм> | <вираз> – <терм>
14. <терм> ::= <множ.> | <терм> * <множ.> | <терм> / <множ.>
15. <множ.> ::= <ід.>| <кон.> | (<вираз>)
16. <індекс. вир.> ::= <ід.>:= <вираз> to <вираз>

17. <дія> ::= <опер.> | begin <сп. опер.> end

18. <ід.>::= <буква> | <ід.> <буква> | <ід.> <цифра>

19. <кон.>::= <ціле без знаку> | . <ціле без знаку> | <ціле без знаку>. | <ціле без знаку> . <ціле без знаку>

20. <ціле без знаку> ::= <ціле без знаку><цифра> | <цифра>

21. <буква>::= a | b | . | z

22. <цифра>::= 0 | 1 | . | 9

2.4. Завдання для самоконтролю

Завдання 2.1. Скласти множину символів, що складають алфавіти: української, англійської, німецької, грецької мови, шістнадцяткової та двійкової системи числення.

Завдання 2.2. Визначіть множину символів (алфавіт), з яких складається азбука Морзе (в даному випадку розглядати азбуку Морзе як мову).

Завдання 2.3. Визначити алфавіт шифру Полібія літер української абетки.

Завдання 2.4. Визначити мову, що складається з правильних українських слів над алфавітом Σ={ а, б, г, м, у}.

Завдання 2.5. Визначіть усі голови та хвости, правильні голови та хвости ланцюжків: радар, атом, рокіт.

Завдання 2.6. Для ланцюжків α, β, γ. Визначити конкатенації βαγ, βγα а також їхні довжини, голови та хвости:

а) α =ля, β = ма, γ = р;

б) α = ка, β = стари, γ = н.

в) α = и, β = порт, γ = к.

Завдання 2.7. Для ланцюжків α, β, γ. Визначити конкатенації αβγ, γβα а також їхні довжини, голови та хвости:

г) α = пір, β = а, γ = т;

д) α = бор, β = а, γ = т.

Завдання 2.8. Для ланцюжків α, β, γ. Визначити конкатенації γβα, βγα а також їхні довжини, голови та хвости:

е) α =лик, β = о, γ = р;

ж) α = к, β = ло, γ = ми;

з) α = та, β = не, γ = мо.

Завдання 2.9. Задано ланцюжки α =а, β = ба, γ = га, μ = ма, λ = л. Побудувати за допомогою конкатенації цих ланцюжків слова: агама, малага, амбал, бал, лама, абабагаламага.

Завдання 2.10. Визначити добуток множин ланцюжків:

а) A={ла, ка}, B = {ма, па};

б) A={ла, ка}, B = {к};

в) A={ста, ма, вда}, B = {ли, ло, ла};

г) A={ста, ма, вда}, B = {ли, ло, ла}, C = { сь, ся};

д) A={світло, темно, брудно}, B = {червон, біл, сір, жовт, зелен, риж}, C = {ий, а, е, і};

е) A={світло, темно, брудно}, B = {червон, біл, сір, жовт, зелен, риж}, C = {ий, ого, ому, им, ім};

ж) A={аналіза, трансля, компіля, генера, опера}, B = {тор}, C = {∧, а, ові, ом, і, е, и, ів, ам, ами, ах};

Завдання 2.11. Скласти граматику, що породжує наступні мови:

а) $a^n \mid n=1,\ldots$	б) $a^n b^n \mid n=1,\ldots$	в) $a^n b^m \mid n,m = 1,\ldots$	г) $ab^n a \mid n=0,\ldots$
д) $a^{2n} \mid n=1,\ldots$	е) $a^n b^{2n} \mid n=1,\ldots$	є) $p^n c q^n \mid n=0,\ldots$	ж) $a^n c b^m \mid n,m=1,\ldots$
з) $a^n b^m c d^m e^n \mid m,n = 0, 1, 2, \ldots$		и) $(a\mid b)^n b \mid n = 0, 1, 2,\ldots$	

Завдання 2.12. Яка мова породжується наступними граматиками?

а) A → Aa A → bA A → ^	б) A → aBb B → a B → b	в) A → a A → Ab	г) A → a A → bA
д) A → aBb B → c B → cB	е) A → aB B → b B → Bb	є) A → c A → aAb	ж) S->FF F->aFb F->ab
з) A → x A → (B) B → A B → B+A	и) S->ab S->aKSb K->bSb K->^ KS->b	і) A → BC B → a B → b C → d C → f	к) A → abAbbA bA →a

Завдання 2.13. Побудувати граматику

а) для цілих чисел, нуль на початку числа не допускається;

б) для чисел паліндромів;

в) для непарних чисел;

г) констант з фіксованою точкою;

д) ідентифікаторів мови C;

е) ідентифікаторів мови PHP;

ж) строкових констант мови C;

з) керуючих послідовностей, що задають формат виводу даних функціями printf;

и) lisp-списків вигляду (<ел-т> <ел-т> … <ел-т>), елементами можуть бути ідентифікатори або такі самі списки, також кожний елемент може мати перед собою '. Списки можуть бути пустими, тобто допускається (), елементи розділені пробілами;

к) html-списку, котрий може в довільному порядку містити вкладені теги ul, ol (елементи списків задаються тегом li). Щоб можна було вивести <OL> <LI>ЛА</LI><UL><LI> До роздільника</LI><LI> ДС </LI> <LI>КА</LI></UL> <LI> Синтаксичний аналізатор</LI> <LI> ГКМК</LI></OL>.

Завдання 2.14. До якого класу відносяться граматики з завдання 2.13? Чи існує еквівалентна автоматна граматика?

Завдання 2.15. Побудувати синтаксичну діаграму та для граматик з завдання 2.13.

Завдання 2.16.

а) Дано граматику

E::=T|E+T|-T
T::=i|(E)
Чи належить ланцюжок -i+i мові?

б) Дано граматику

E::=T|T+E
T::=i|(E)|-E
Чи належить ланцюжок -i+-i мові?

в) Дано граматику

A->aBa
A->AA,
B->b,
B->c,
B->Bb,
B->cB
Чи належить ланцюжок accaabca мові?

Завдання 2.17. Чи еквівалентні граматики:

а) A::=a|Ab|Aa та A::=aB
B::=aB|bB|^

б) A::=aa|bAb|aAa та A::=aBa|bBb
B::=aa

в) A::=ab|aBAb та A::=aBb
B::=bAb|^ B::=^|b|A|bAbA
BA::=b

Завдання 2.18. За створеними в завданні 2.13 граматиками спробувати побудувати дерева виводу для наступних ланцюжків, відповідно:

а) 756, 0, 01, 102, 100;

б) 11111, 1111, 12321, 456654, 1, 114411;

в) 1, 13, 221, 344;

г) 12.1, -0.1, -11, 54., 34.3. ;

д) First_test, FT, ft, _n1, 1_n, n_1;

е) $c, $obj, $t_1, $_w, $$com;

ж) “Помилка !\n”, “big apple’ ”;

з) “%d”, "%0*.*f", "%hhX", "%-0.12f";

л) ((lambda (x) (list x (list 'quote x))) '(lambda (x) (list x (list 'quote x)))), (format t ‘Hello), ((key1 value1) (key2 value2) ());

м) <OL> <LI>ЛА</LI><UL><LI> До роздільника</LI><LI> ДС </LI> <LI>КА</LI></UL> <LI> Синтаксичний аналізатор</LI> <LI> ГКМК</LI></OL>.

Завдання 2.19. Записати лівий та правий виводи за побудованими в завданні 2.18 деревами виводу.

Завдання 2.20. Визначити чи є граматики з завдання 2.12 рекурсивними. Якщо так, з’ясувати тип рекурсії та, чи можна в них замінити ліву рекурсію правою та навпаки.

Завдання 2.21. Написати граматику для С-подібного оператора умовного переходу, та for-циклу.

Завдання 2.22. Дано граматику S->aAbS, S->b, A->Sac, A-> ^. Вивести ланцюжок abacbb.

Завдання 2.23. Нехай $\Sigma - \{a, b, c\}$. Чи дорівнюють мови $L_1 = \{(abc)^n a \mid n > 1\}$ та $L_2 = \{ab(cab)^n ca \mid n > 0\}$?

РОЗДІЛ 3. ЛЕКСИЧНИЙ АНАЛІЗАТОР

3.1. Функції лексичного аналізатора

Основна задача лексичного аналізатора (ЛА) – розбір вхідного рядка символів на лексичні одиниці.

Лексична одиниця (ЛО) – це підрядок вхідного рядка. Вона може містити лише термінальні символи (елементи) і не може містити інших лексем. Для синтаксичного аналізу лексема є найменшою одиницею мови, а з термінальними символами працює виключно лексичний аналізатор – сканер.

Лексеми описуються **двома ознаками**: клас і номер елемента даного класу. Розпізнають **3 класи лексем**: термінальні (TRM), включають символи, службові слова та роздільники, ідентифікатори (IDN) та константи (CON).

Задачі сканера

1. Виокремлення лексем в програмі.
2. Побудова таблиць (рисунок 3.1).

Вхідні дані

1. Таблиця лексем мови (ключові слова, оператори, роздільники).
2. Початкова програма.

Вихідні дані

1. Таблиця констант.
2. Таблиця ідентифікаторів.
3. Вихідна таблиця лексем.

Для кожної мови будується одна таблиця лексем. Вона не залежить від початкової програми, і будується за граматикою мови за такими принципами:

- таблиця лексем повинна містити усі термінали, що зустрічаються в правилах граматики;
- в таблиці лексем кожний запис відповідає тільки одному терміналу;
- білі роздільники не входять в таблицю лексем;
- окремими елементами таблиці лексем є ідентифікатори та константи, складові цих класів лексем не входять до таблиці лексем.

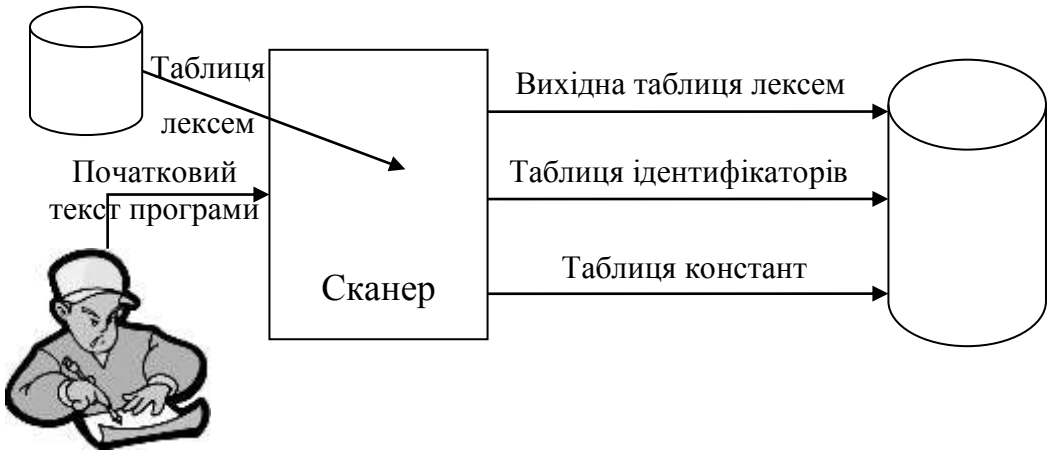


Рисунок 3.1 — Схема лексичного аналізатора

Побудуємо таблицю лексем (таблиця 3.1) для мови міні-Паскаль, наведеної в прикладі 2.12, будемо кодувати лексеми послідовними числами.

Таблиця 3.1. Таблиця лексем

Лексема	**Код**
Program	1
Var	2
Begin	3
end.	4
Integer	5
Real	6

Продовження таблиці 3.1

Лексема	**Код**
Read	7
Write	8
For	9
To	10
Do	11
End	12
/	13
;	14
:	15
,	16
:=	17
+	18
-	19
*	20
(	21
)	22
Idn	23
Con	24

Приклад 3.1

Напишемо приклад програми розробленою нами мовою програмування міні-Паскаль (приклад 2.12). Спробуємо використати усі види операторів, дозволені даною мовою. Написана програма (рисунок 3.2) буде обчислювати середнє арифметичне введених користувачем ста чисел. Отже, вона буде в циклі від 1 до 100 приймати від користувача чергове число, та додавати його до загальної суми, котру будемо зберігати в змінній під назвою sum. Для наочності ходу виконання програми в циклі буде також виводитися поточна сума та номер ітерації. Після завершення циклу загальна сума буде ділитися

на 100 (тобто кількість введених чисел), і, таким чином, в змінній sum залишиться значення середнього арифметичного, введених чисел.

```
1   program first
2   var i: integer;
3     sum, value : real
4   begin
5     sum:=0;
6     for i:=1 to 100 do
7     begin
8       read (value);
9       sum:=sum+value;
10      write (i, sum)
11    end;
12    sum := sum / 100;
13
14  End.
```

Рисунок 3.2 — Перша міні-Паскаль програма

Побудуємо таблиці, котрі повинен згенерувати лексичний аналізатор, розбираючи даний текст програми. У вихідну таблицю лексем (таблиця 3.2) будемо записувати інформацію про лексичні одиниці, котра знадобиться нам на наступних етапах розбору:

- номер рядка, в котрому зустрілася лексема;
- виокремлений підрядок (саму лексичну одиницю);
- код лексеми з таблиці лексем (таблиця 3.1);
- індекс у відповідній таблиці ідентифікаторів або констант, якщо лексична одиниця належить до класів IDN або CON.

Таблиця 3.2. Вихідна таблиця лексем

№ рядка	Підрядок	Код	Індекс idn/con
1	program	1	
1	first	23	1
2	var	2	
2	i	23	2
2	:	15	
2	integer	5	
2	;	14	
3	sum	23	3
3	,	16	
3	value	23	4
3	:	15	
3	real	6	
4	begin	3	
5	sum	23	3
5	:=	17	
5	0	24	1
5	;	14	
6	for	9	
6	i	23	2
6	:=	17	
6	1	24	2
6	to	10	
6	100	24	3
6	do	11	
7	begin	3	

№ рядка	Підрядок	Код	Індекс idn/con
8	read	7	
8	(	21	
8	value	23	4
8	)	22	
8	;	14	
9	sum	23	3
9	:=	17	
9	sum	23	3
9	+	18	
9	value	23	4
9	;	14	
10	write	8	
10	(	21	
10	i	23	2
10	,	16	
10	sum	23	3
10	)	22	
11	end	12	
11	;	14	
12	sum	23	3
12	:=	17	
12	sum	23	3
12	/	13	
12	100	24	3
14	end.	4	

Сформуємо таблицю ідентифікаторів (таблиця 3.3), в котрій будемо зберігати ім’я ідентифікатора, його індекс та тип.

Таблиця 3.3. Таблиця ідентифікаторів

Ідентифікатор	**Індекс**	**Тип**
First	1	program
I	2	integer
Sum	3	real
Value	4	real

Аналогічно побудуємо таблицю констант (таблиця 3.4). Інформацію про тип константи зберігати не будемо.

Таблиця 3.4. Таблиця констант

Константа	**Індекс**
0	1
1	2
100	3

3.2. Методи виокремлення лексем з тексту. Перегляд до роздільника

Вхідний рядок розділяється на лексеми символами роздільниками. Роздільники можуть бути значущими та незначущими символами. Значущі роздільники є лексемами, тобто містяться в таблиці лексем, вони помічаються в спеціальному полі таблиці лексем, це такі символи як «,» «:» «;» «+», тощо. Незначущі роздільники інакше називають білими роздільниками, вони не входять до таблиці лексем і не потрапляють до вихідної таблиці лексем, а

просто вилучаються з тексту, це такі символи як пробіл, знак переходу на наступний рядок, табуляція, тощо.

Коли зчитується наступний символ вхідної програми, з'ясовують, чи є він роздільником. Між двома роздільниками знаходиться лексична одиниця. Спочатку всі лексичні одиниці порівнюються з лексемами типу TRM, при збіжності вони заносяться до вихідної таблиці лексем [13]. При незбіжності класифікуються як можливий ідентифікатор або константа. Якщо лексична одиниця не підходить до жодної з цих категорій видається сигнал про помилку.

Якщо лексична одиниця класифікована як можливий ідентифікатор, то перевіряється таблиця ідентифікаторів. В разі, коли такого ідентифікатора в таблиці нема, створюється новий елемент таблиці, в котрий заноситься ім'я (інші параметри заносяться по мірі своєї появи). Не залежно від заповнення таблиці ідентифікаторів до вихідної таблиці лексем заноситься символ типу IDN з відповідним індексом. Аналогічна процедура проводиться з вірогідними константами. В мові також можуть бути наявні мітки, котрі частіше за все, відповідно до правил граматики, ідентичні ідентифікаторам. В такому разі, мітки можуть розцінюватися як ідентифікатори окремого типу і записуватись до таблиці ідентифікаторів, або можуть записуватися в окрему таблицю, тоді їх вважають лексемами окремого типу LAB.

На рівень лексичного аналізу можна віднести перевірку правильності ідентифікаторів і констант. Якщо мова потребує попереднього оголошення ідентифікаторів та міток, то при лексичному аналізі також можна виявляти наступні помилки:

- дублювання ідентифікатора або мітки в оголошенні;
- неописаний ідентифікатор або мітка.

Окрім основної функції розпізнавання лексем, сканер виявляє коментарі в програмі, при розборі вони ігноруються і видаляються з програми ще до початку синтаксичного аналізу. Також ігноруються і видаляються з програми білі роздільники, оскільки вони не несуть корисної інформації а лише розділяють лексеми. Ще однією задачею лексичного аналізатора є синхронізація повідомлень про помилки, котрі генерує компілятор, з текстом вхідної програми. Для того щоб усі помилки супроводжувалися номером рядка, в котрому вони зустрілися, ми зберігаємо (таблиця 3.2) для кожної лексеми номер рядка.

3.3. Реалізація лексичного аналізатора за допомогою діаграми станів

Розглянемо реалізацію сканера за допомогою діаграми станів (ДС), котра вказує, яким чином ведеться розбір лексеми [13]. На вхід програми надходить вхідний ланцюжок символів, на виході видається код лексеми, після чого діаграма переходить в початковий стан. Дуги діаграми помічені символами, під дією котрих здійснюється перехід з одного стану до іншого. Якщо дуга не помічена, то вона відповідає всім символам окрім тих, що вказано на альтернативних дугах, тобто тих що виходять з того самого стану.

Дуги, що ведуть на вихід і помилку («er»), свідчать про те, що виявлено кінець лексеми і необхідно взяти наступний символ і перейти на початок діаграми. Слід мати на увазі той факт, що сканування наступного символу не завжди є необхідним. При переході на «вихід», що відповідає лексемі IDN або CON, наступний символ вже считано, а при розпізнаванні односимвольного роздільника – ні. Будемо вважати, що перед виходом зі сканера наступний символ завжди зчитано.

Для того щоб зменшити кількість символів, з котрими повинен працювати сканер, усі символи поділяють на класи. Для мови, заданої в прикладі 2.12, можна визначити наступні класи (таблиця 3.5).

Таблиця 3.5. Таблиця класів

Множина символів	**Клас**
.	.
a..z	Б
0..1	Ц
;:,+-*()	ОР
:	:
=	=

В подальшому в лексичному аналізаторі використовуються або класи, або символи. Побудуємо діаграму станів (рисунок 3.3) для описаної в прикладі 2.12 мові міні-Паскаль.

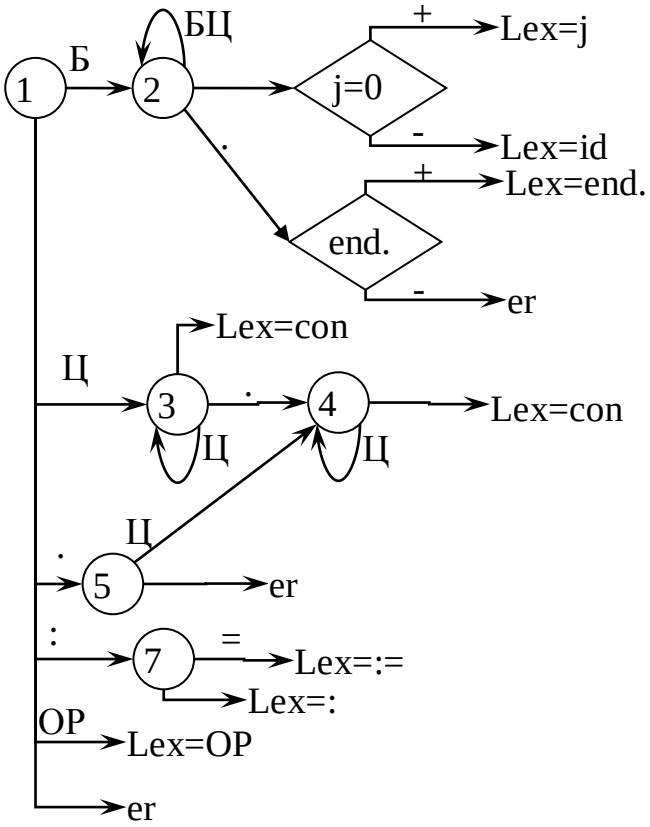


Рисунок 3.3 — Діаграма станів для міні-Паскаль

Приклад 3.2

Нехай мовою є послідовність десяткових чисел будь-якої форми, розділених комами.

<послідовність> ::= < константа> | < константа>, < послідовність>

<константа> ::=<чфт>|<чфт> E <порядок>

<чфт> ::=<цбз>| .<цбз>|<цбз>. |<цбз>. <цбз>

<цбз> ::= <цифра>|<цбз> <цифра>

<порядок> ::=<цбз>|<знак><цбз>

<знак>::= +| –

<цифра>::=0| 1| 2| 3| 4 … | 9

Тоді таблиця лексем міститиме лише дві лексеми (рисунок 3.4), а в таблиці класів необхідно виділити в окремі класи наступні символи **E .+ –** , отримаємо.

Таблиця лексем		**Таблиця класів**	
,	1	,	,
const	2	0..9	Ц
		+-	+
		E	E
		.	.

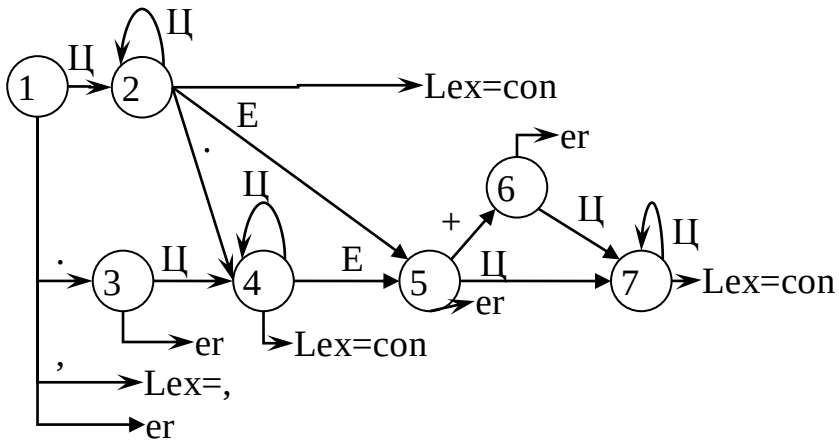


Рисунок 3.4 — Підготовка до ЛА для граматики послідовності чисел

Приклад 3.3

Нехай мова містить лише оператор вводу вигляду **read(a1,b),** в дужках може бути довільний список ідентифікаторів. Виконаємо підготовку для ЛА (рисунок 3.5).

Таблиця лексем		Таблиця класів	
read	1	(,)	ОР
(	2	a..z	Б
)	3	0..9	Ц
,	4		
id	5		

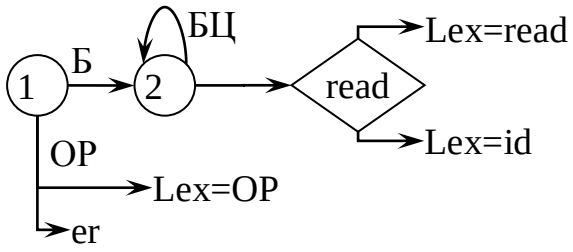


Рисунок 3.5 — Таблиці та ДС для оператора вводу

Приклад 3.4

Нехай мова містить лише оператор виводу вигляду **write(A1, 25.6E–7, 15, B)**. Тоді, якщо всі літери скласти в один клас, то при розпізнаванні констант з плаваючою точкою необхідно аналізувати і клас Б і символ Е. Якщо літеру Е помістити в окремий клас, тоді в діаграмі станів під час побудови ідентифікаторів необхідно вказувати і клас Б, і клас Е. Наведемо таблицю лексем та класів для даної мови і побудуємо відповідну діаграму станів (рисунок 3.6).

Таблиця лексем		Таблиця класів	
write	1	(,)	ОР
(	2	a..z без e	Б
)	3	0..9	Ц
,	4	+-	+
id	5	.	.
const	6	e	E

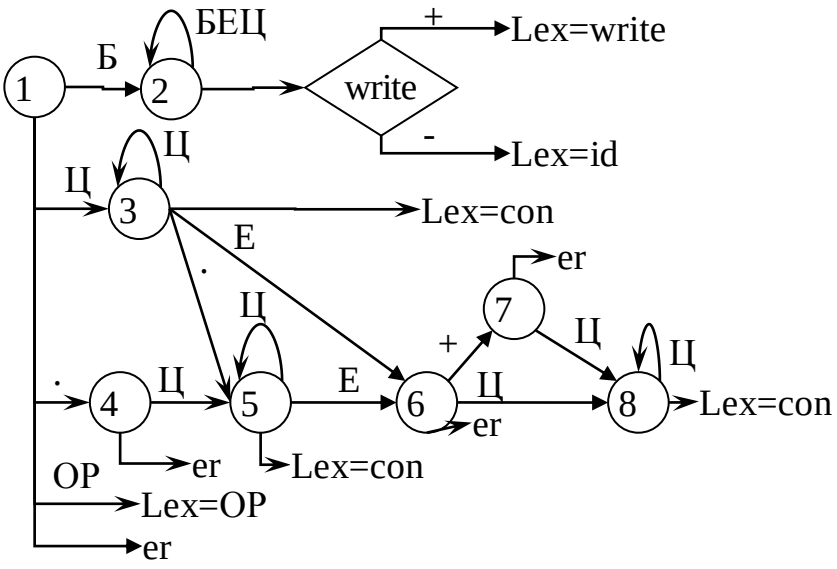


Рисунок 3.6 — Таблиці та ДС для оператора виводу

Отже, ми навчилися будувати діаграми станів та вхідні таблиці ЛА для різноманітних граматик. Ще раз зверніть увагу на деякі моменти:

- з одного стану діаграми не може виходити дві і більше дуг, помічених однаково;
- з кожного стану повинна виходити одна і тільки одна дуга без мітки;
- таблиця класів повинна охоплювати усі символи, дозволені в даній мові;
- слід пам'ятати: таблиця класів об'єднує в класи символи, не ланцюжки символів.

3.4. Скінченний автомат

Діаграму станів лексичного аналізатора можна розглядати як граф, що визначає функціонування скінченного автомата. Тоді скінченний автомат (СКА) можна вважати деревом, в якому вузли відповідають станам, а дуги – переходам. Функція переходів або правило функціонування СКА має вигляд:

$$\alpha \xrightarrow{a} \beta$$

і означає, що автомат по вхідному символу **a** переходить зі стану α до стану β. Символ **a** називається міткою переходу.

Скінченний автомат моделює обчислювальний пристрій зі скінченним і фіксованим об'ємом пам'яті, котрий зчитує послідовність вхідних символів, що належать деякій скінченній множині, і займає визначений стан. Різновиди автоматів визначаються їхніми виходами.

Скінченні автомати поділяються на два класи [1].

1. Недетерміновані скінченні автомати не мають обмежень на свої переходи. Один символ може бути міткою декількох переходів,

що виходять з одного й того ж стану, також міткою може бути порожній символ.

2. Детерміновані скінченні автомати для кожного стану і кожного символу вхідного алфавіту містять один перехід за даним символом, що виходить з цього стану.

Детерміновані і недетерміновані скінченні автомати можуть розпізнавати одній й ті самі мови, котрі називають регулярними або автоматними мовами. Автоматні мови можна також описувати за допомогою так званих **регулярних виразів**.

На практиці частіше використовують детерміновані скінченні автомати, в яких для будь-якої послідовності вхідних символів існує лише один стан, в котрий може перейти автомат з поточного стану [14]. В літературі присвяченій теорії автоматів [15, 16, 17] можна зустріти деякі відмінності у визначенні скінченних автоматів, проте доведено [15], що будь-який скінченний автомат можна представити відповідно до наступного визначення.

Скінченний автомат (СКА) – це п’ятірка

$$\mathbf{M = <Q, \Sigma, \Delta, q_0, F>,}$$

де

Q – скінченна множина станів даного автомата;

Σ – скінченний алфавіт вхідних символів (або просто алфавіт) даного скінченного автомата;

Δ – функція, що відображає множину $Q \times \Sigma$ в Q. Вона називається функцією переходів і здійснює передачу управління на новий стан з поточного стану відповідно до вхідного символу. Використовуючи поняття декартового добутку множин, можна визначити Δ як скінченну множину переходів:

$$\Delta \subseteq Q \times \Sigma \times Q = \{< \alpha, a, \beta > | \alpha \in Q, a \in \Sigma, \beta \in Q\};$$

q_0 – початкових стан скінченного автомата $q_0 \in Q$;

F – скінченна непуста множина заключних або допускаючих станів $F \subseteq Q$.

Перехід $<\alpha, a, \beta> \in \Delta$, аналогічний функції переходів $\alpha \xrightarrow{a} \beta$, котру можна записати у вигляді $\delta(\alpha, a) = \beta$. Функцію **δ** можна узагальнити, визначивши її рекурсивним чином [7]. Для ланцюжка **ax**, де $a \in \Sigma$, виконується співвідношення:

$$\delta(\alpha, ax) = \delta(\delta(\alpha, a), x).$$

Це визначення задає перехід зі стану α, відповідно до початкового символу **a**, до іншого стану, відповідно до ланцюжка **x**. Перехід здійснюється послідовно від одного стану до іншого, в міру зчитування символів вхідного ланцюжка.

Таким чином, **мова**, що задається скінченним автоматом, складається з усіх ланцюжків, що переводять автомат з початкового стану в один з кінцевих станів.

Конфігурацією або **миттєвим описом** СКА <Q, Σ, Δ, q_0, F> називається довільна впорядкована пара <q, w>, де $q \in Q$ — поточний стан автомата, а $w \in \Sigma$ — поточний символ, що подано на вхід автомата.

Скінченні автомати можна зображати у вигляді спрямованого графа, в котрому кожна дуга помічена символом. Вузли графа відповідають станам СКА і позначаються колами. Дуги графа відповідають переходам СКА. Для кожного переходу $<\alpha, a, \beta> \in \Delta$ існує дуга, що веде з вузла, відповідного стану α, до вузла, що відповідає стану β, вона помічена символом **a**. Початковому стану відповідає вузол, що містить вхідну дугу. Кінцевим станам відповідають вузли, які позначаються подвійними колами.

Приклад 3.5

Побудуємо скінченний автомат A для мови $L = \{a^n b^m \mid n \in N, m \in N\}$, де N – множина натуральних чисел. За формальним визначенням для даної мови A = <Q, Σ, Δ, q_0, F>, де Q={1, 2, 3}, Σ = {a, b}, q_0 = 1, F = {3}, Δ = {<1, a, 2>, <1, b, 4>, <2, a, 2>, <2, b, 3>, <3, b, 3>, <3, a, 4>, <4, a, 4>, <4, b, 4>}. Для наочності представимо отриманий СКА графічно (рисунок 3.7).

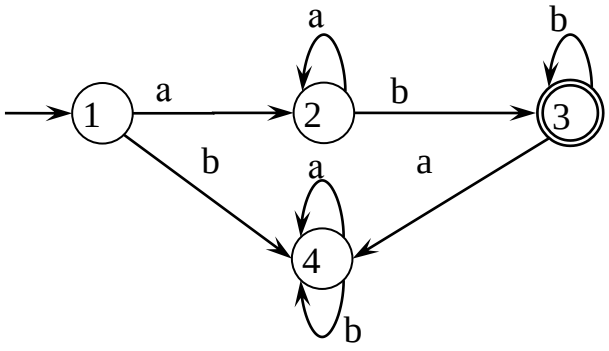


Рисунок 3.7 — Приклад графічного вигляду СКА

Якщо в скінченному автоматі існують переходи зі спільним початком та спільним кінцем, вони називаються паралельними. В прикладі 2.5 паралельними є переходи <4, a, 4>, <4, b, 4>. Іноді на діаграмах паралельні переходи зображуються однією стрілкою, а їхні мітки вказуються через кому.

Побудований автомат містить тупиковий стан 4, з котрого СКА за жодних обставин не може потрапити в кінцевий стан. Переходи <4, a, 4>, <4, b, 4> забезпечують підтвердження тупікового стану для кожного символу з алфавіту Σ.

При побудові сканера на основі скінченного автомата, заключний стан автомата відповідає видачі лексем, після чого автомат переходить в початковий стан. Очевидно, якщо автомат потрапляє в тупиковий стан [12, с. 84], подальша перевірка вхідного ланцюжка не має сенсу, оскільки він в будь-якому разі буде неприйнятим. Тому опинившись в тупиковому стані, СКА повинен

видати повідомлення про помилку і виконати аварійне завершення роботи. Дії, пов'язані з заключним або тупиковим станом автомата, можуть виконуватись як при збіганні поточного символу з указаним, так і при не збіганні. Вони визначають вміст, так званих, семантичних підпрограм.

Таким чином семантичні підпрограми обов'язково задаються для кінцевих та тупикових станів СКА. Для кінцевих станів вони містять інформацію про особливості видачі отриманої лексеми. Для тупикових – інформацію про відповідну помилку. На графіку будемо зображати семантичні підпрограми текстом без рамки, в котрий входять відповідні дуги.

Представляти скінченні автомати в канонічному вигляді, явно вказуючи усі множини Q, Σ, Δ, F та початковий стан q_0 не зручно та громіздко. Тому, надалі будемо задавати СКА таблицями переходів. Вважатимемо, що Σ та Q містять лише символи, вказані в таблиці переходів, початковому стану відповідає перший рядок таблиці, а множину заключних та тупикових станів будемо визначати за семантичними процедурами. Оскільки початкові та тупикові стани супроводжуються семантичними підпрограмами, немає необхідності задавати їх явним чином як в таблиці, так і на графіку.

Наведемо таблицю переходів для СКА з прикладу 2.5 (таблиця 3.6).

Таблиця 3.6. Приклад таблиці переходів

α	**Мітка переходу**	**β**	**Семантична підпрограма**
1	a	2	[≠] помилка
2	a	2	[≠] помилка
	b	3	
3	b	3	[≠] видача лексеми

На практиці буває важко визначити наперед множину вхідних символів, оскільки вхідний ланцюжок часто задається користувачем або деяким

стороннім пристроєм. Тому для кожного стану α необхідно вказати семантичну підпрограму [≠], що виконується по не зрівнянню вхідного символу з усіма визначеними мітками переходів $<\alpha, a, \beta> \in \Delta$. На графіку перехід по не зрівнянню позначатимемо не підписаною дугою.

Побудуємо графік для СКА з прикладу 3.5 з урахуванням наведених міркувань (рисунок 3.8).

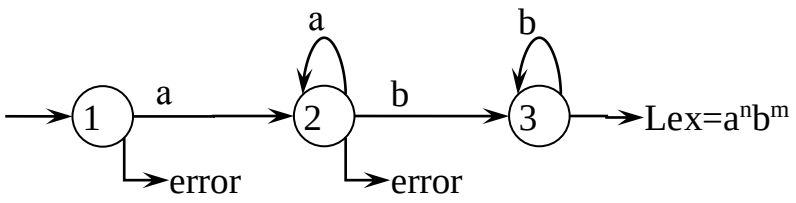


Рисунок 3.8 — Модифікований графік СКА

Приклад 3.6

Побудуємо скінченний автомат у формі списку переходів (таблиця 3.7) для мови програмування Міні-Паскаль з прикладу 2.12, в якості міток будемо використовувати визначені раніше (таблиця 3.5) класи.

Таблиця 3.7. Таблиця переходів для Міні-Паскаль

α	**Мітка переходу**	**β**	**Семантична підпрограма**
1	Б	2	[≠] помилка
	Ц	3	
	.	5	
	:	6	
	ОР		[=] виділення лексеми ОР
2	Б	2	[≠] виділення лексеми TRM або IDN
	Ц	2	
	.		[=] можливе виділення лексеми end.

Продовження таблиці 3.7

α	Мітка переходу	β	Семантична підпрограма
3	Ц	3	[≠] виділення лексеми CON
	.	4	
4	Ц	4	[≠] виділення лексеми CON
5	Ц	4	[≠] помилка
6	=		[=] виділення лексеми := [≠] лексеми :

Подана таблиця переходів скінченного автомата відповідає діаграмі станів, побудованій раніше (рисунок 3.3).

Приклад 3.7

Мова задається наступною граматикою:

<програма>::= <список операторів>**end**

<список операторів>::=<оператор>|<список операторів>;<оператор>

<оператор>::=**write**(<список виводу>)

<список виводу>::=<елемент списку> |<список виводу>,<елемент списку>

<елемент списку>::= <ідентифікатор> |<константа>

<ідентифікатор>::=<буква>|<ідентифікатор><буква>|<ідентифікатор><цифра>

<константа> ::=<чфт>|<чфт> E <порядок>

<чфт> ::=<цбз>| .<цбз>|<цбз>. |<цбз>. <цбз>

<цбз> ::= <цифра>|<цбз> <цифра>

<порядок> ::=<цбз>|<знак><цбз>

<знак>::= +| –

<цифра>::=0| 1| 2| 3| 4 … | 9

<буква>::=a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

Здійснимо підготовку до побудови лексичного аналізатора. Перш за все необхідно визначити словник лексем мови (таблиця 3.8).

Таблиця 3.8. Таблиця лексем для прикладу 3.7

Лексема	**Код**
write	1
end	2
(	3
)	4
,	5
;	6
id	7
const	8

Тепер згрупуємо символи, з яких складаються лексеми, в класи (таблиця 3.9).

Таблиця 3.9. Таблиця класів для прикладу 3.7

Символи	**Клас**
a-d, f-z	Б
0-9	Ц
.	.
, ( ) ;	ОРО
e	Е
+-	+

Символ «е» виділили в окремий клас, і для однозначності його було вилучено з класу Б. Будемо вважати, що ідентифікатор не може починатися на символ «е».

Для наочності побудуємо діаграму станів для СКА (рисунок 3.9), а також у формі списку переходів (таблиця 3.10).

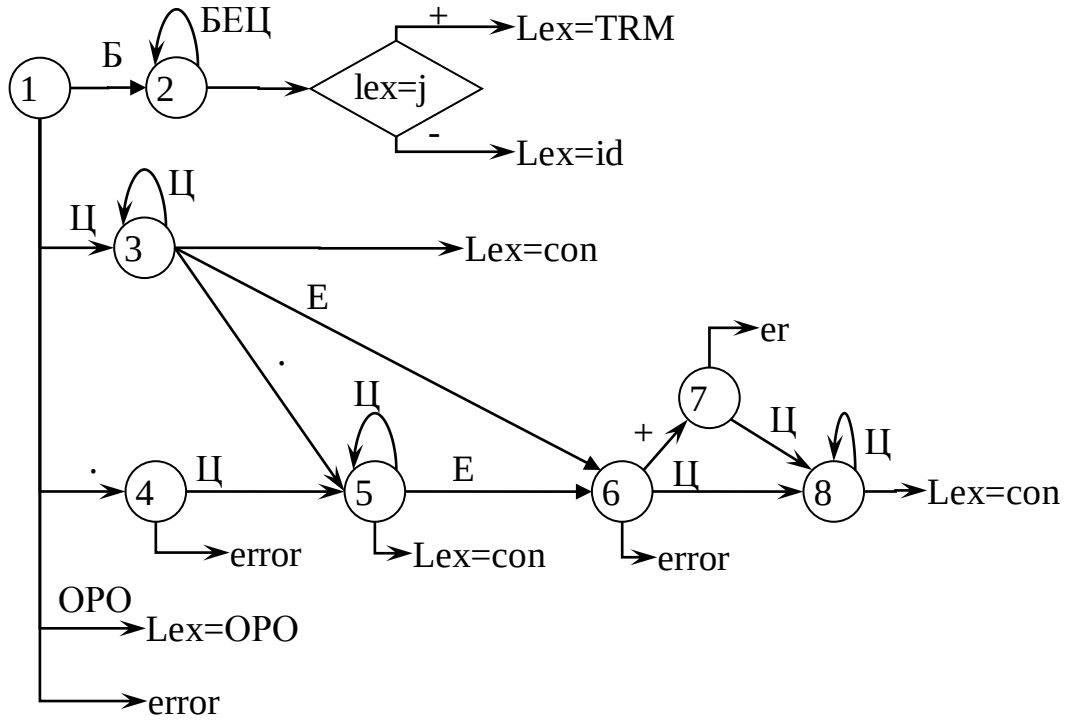


Рисунок 3.9 — Діаграма станів для прикладу 3.7

Таблиця 3.10. Таблиця переходів для прикладу 3.7

A	Мітка переходу	B	Семантична підпрограма
1	Б	2	[=] виділення лексеми ОРО [≠] помилка
	Ц	3	
	.	4	
	ОРО		
2	Б	2	[≠] виділення лексеми TRM або IDN
	Ц	2	
	Е	2	
3	Ц	3	[≠] виділення лексеми CON
	.	5	
	Е	6	
4	Ц	5	[≠] помилка
5	Ц	5	[≠] виділення лексеми CON
	Е	6	

Продовження таблиці 3.10

A	Мітка переходу	B	Семантична підпрограма
6	+	7	[≠] помилка
	Ц	8	
7	Ц	8	[≠] помилка
8	Ц	8	[≠] виділення лексеми CON

Алгоритм функціонування сканера на основі скінченного автомата

Взяти наступний символ, порівняти його з символами, що припустимі в даному стані автомата. При збіганні перейти на вказаний стан і виконати семантичну підпрограму (позначка [=]), якщо вона є. При не збіганні виконати семантичну підпрограму по не зрівнянню (позначка [≠]), котра завжди вказана. Якщо вказана семантична підпрограма відповідає видачі лексеми, виконати видачу лексеми, після цього перейти на початковий стан. Якщо вказана семантична підпрограма відповідає видачі повідомлення про помилку, повідомити про помилку і завершити роботу сканера.

Таким чином, цей алгоритм не залежить від конкретної мови. Налаштування на конкретну мову здійснюється шляхом введення відповідного списку станів СКА для сканера даної мови.

Приклад 3.8

Виконаємо тестовий розбір за визначеним алгоритмом. Нехай автомат з прикладу 3.7 знаходиться в початковому стані **1.** На вході ланцюжок **87.9)…** По першому символу **8** (клас **Ц**) автомат переходить зі стану **1** в стан **3**. Наступний символ **7** (клас **Ц**) підтверджує той самий стан. Наступний символ **.** переводить автомат в стан **5**. Далі символ **9** підтверджує стан **5.** Символ **)** не співпадає з жодним символом, що допускається в стані **5**. Необхідно виконати семантичну підпрограму по не зрівнянню [≠], генерується лексема **CON**, після чого автомат

переходить в початковий стан і розглядає символ **)** (клас **ОРО**), по зрівнянню з котрим генерується лексема, що відповідає цьому символу.

3.5. Регулярні вирази

Як було вказано раніше, автоматні мови можна задавати за допомогою регулярних виразів. Нотація регулярних виразів була введена Кліні в 1950-х роках і базувалася на таких операціях над мовами, як об'єднання, конкатенація, ітерація.

Будемо позначати регулярні вирази курсивом. Кожний регулярний вираз *r* описує мову L(*r*). Регулярні вирази рекурсивно будуються з менших регулярних виразів за наступними правилами. Нехай Σ – алфавіт, над котрим визначаються регулярні вирази, тоді

- Ø – регулярний вираз, що описує мову L(Ø)=Ø;
- $\wedge$ - регулярний вираз, що описує мову L($\wedge$) = {$\wedge$}, тобто мову, що складається з порожнього ланцюжка;
- якщо a – символ алфавіту Σ, то *a* – регулярний вираз, що описує мову L(*a*) = {a}, котра містить єдиний ланцюжок, що складається з одного символу a.

Існує чотири правила, на основі котрих регулярні вирази будуються з менших регулярних виразів:

1) *r*|*s* – регулярний вираз, що описує мову L(*r*|*s*) = L(*r*)$\cup$L(*s*);
2) *rs* - регулярний вираз, що описує мову L(*rs*) = L(*r*)L(*s*);
3) r^* - регулярний вираз, що описує мову L(r^*) = (L(*r*))*;
4) (*r*) - регулярний вираз, що описує мову L(*r*), тобто регулярний вираз можна взяти в дужки, і при цьому мова, котру він описує, не зміниться.

Ці чотири операції над регулярними виразами є лівоасоціативними. Серед них унарний оператор * має найвищий пріоритет, а оператор | - найнижчий. Дужки використовуються для зміни порядку виконання операцій.

Якщо два регулярні вирази *r* та *s* описують одну й ту саму мову, вони називаються **еквівалентними**, це записується як *r*=*s*. Визначені базові операції над регулярними виразами підкоряються наступним законам (таблиця 3.11).

Таблиця 3.11. Основні закони для операцій над регулярними виразами

Закон	**Опис**
$r\|s=s\|r$	оператор | комутативний
$r\|(s\|t)=(r\|s)\|t$	оператор | асоціативний
$r(st) = (rs)t$	конкатенація асоціативна
$r(s\|t) = rs\|rt$; $(s\|t)r = sr\|tr$	конкатенація дистрибутивна над |
$\wedge r = r \wedge = r$	$\wedge$ - одиничний елемент по відношенню до конкатенації
$r^* = (r\|\wedge)^*$	$\wedge$ завжди входить в ітерацію
$r^{**} = r^*$	оператор * - ідемпотентний

Для зручності регулярним виразам можна давати імена, а потім використовувати при побудові інших регулярних виразів. Ця дія називається **регулярним визначенням**. Слід уникати рекурсії в регулярних визначеннях.

Приклад 3.9

Визначимо ім'я *d* регурярному виразу, котрий описує мову, що складається з усіх цифр. Використовуючи *d*, визначимо регулярний вираз *number*, що описує усі цілі числа без знаку:

$d \rightarrow 0|1|2|3|4|5|6|7|8|9$

$number \rightarrow dd^*$

Приклад 3.10

Використовуючи регулярні визначення, опишемо мову чисел з плаваючою точкою над алфавітом $\Sigma = \{0,1,2,3,4,5,6,7,8,9,+,-,\mathrm{E},.\}$.

d → 0|1|2|3|4|5|6|7|8|9

number→*dd**

signed→(+|-|∧) *number*

mantissa→*signed*(.|.*number*|∧) | (+|-|∧).*number*

exponent→E *signed*|∧

float → *mantissa exponent*

Тут регулярний вираз *signed* визначає ланцюжок, котрий може починатися зі знаків «+» або «-», після чого іде послідовність цифр довжиною не менше одиниці. Вираз *mantissa* – числа з фіксованою точкою, *exponent* визначає порядок числа, котрий може бути не заданий.

Підставляючи в останній регулярний вираз попередньо визначені, можна отримати регулярний вираз виключно над початковим алфавітом. Виконаємо ці дії для описаних регулярних визначень числа з плаваючою точкою.

Приклад 3.11

Спочатку у виразі *float* розкриємо регулярні вирази *mantissa* та *exponent*:

float → (*signed*(.|.*number*|∧) | (+|-|∧).*number*) (E *signed*|∧)

Тепер розкриємо *signed*:

float → ((+|-|∧) *number* (.|.*number*|∧) | (+|-|∧).*number*) (E (+|-|∧) *number* |∧)

Замість *number* підставимо його визначення:

float → ((+|-|∧)*dd** (.|.*dd**|∧) | (+|-|∧). *dd**) (E (+|-|∧)*dd**|∧)

Нарешті розкриємо регулярний вираз *d*:

float → ((+|-|∧) (0|1|2|3|4|5|6|7|8|9) (0|1|2|3|4|5|6|7|8|9)* (. | . (0|1|2|3|4|5|6|7|8|9) (0|1|2|3|4|5|6|7|8|9)* | ∧) | (+|-|∧) . (0|1|2|3|4|5|6|7|8|9) (0|1|2|3|4|5|6|7|8|9)*) (E (+|-|∧) (0|1|2|3|4|5|6|7|8|9) (0|1|2|3|4|5|6|7|8|9)* | ∧).

Отримали досить громіздке визначення.

Для більшої зручності регулярні вирази були розширені іншими операторами. В теперішній час механізм регулярних виразів підтримуються прак-

тично усіма мовами програмування та деякими іншими програмними засобами. Розглянемо деякі особливо корисні розширення, введені в UNIX.

1. Унарний оператор $^+$ - один або кілька екземплярів: r^+ визначає мову $L(r^+) = (L(r))^+$, він має той самий пріоритет, що і *. Для нього виконуються закони $r^+ = rr^* = r^*r$, $r^* = r^+|\wedge$.
2. Унарний постфіксний оператор ? – нуль або один екземпляр: r? визначає мову $L(r?)=L(r)\cup\{\wedge\}$, має такий самий пріоритет, що і * та $^+$.
3. Квадратні дужки [] об'єднують символи в класи: регулярний вираз $a_1|a_2| \ldots |a_n$ (де $a_i \in \Sigma$, $i = \overline{1,n}$), можна записати у вигляді $[a_1a_2 \ldots a_n]$. Якщо $a_1, a_2, \ldots, a_n$ мають логічну послідовність, наприклад це послідовні літери або цифри, то їх можна замінити виразом a_1-a_n, наприклад, [02468] = 0|2|4|6|8, [a-f]=a|b|c|d|e|f.

Приклад 3.12

Використовуючи розширену нотацію регулярних виразів, перепишемо визначення, отримане в попередньому прикладі. Замінимо $.[0\text{-}9]^+|.=[0\text{-}9]^*$

float → (+|-)? ([0-9]$^+$ (.[0-9]*)? | .[0-9]$^+$) E (+|-)? [0-9]$^+$.

Як бачимо, запис значно скоротився, і легше читається.

На сьогодні, регулярні вирази стали потужним засобом обробки текстів. Найбільшого поширення вони набули в галузі WEB-програмування. Вони підтримуються багатьма програмами: редакторами, системними утілітами, ядрами баз даних, тощо; а також мовами програмування: Java, Jscript, Visual Basic, VBScript, JavaScript, C, C++, C#, elisp, Perl, Python, Tcl, Ruby, PHP та іншими. В різних програмних засобах нотації регулярних виразів можуть дещо відрізнятися, більш докладну інформацію з цього приводу можна знайти в [13, 14].

3.6. Завдання для самоконтролю

Завдання 3.1. Побудувати граматику для наступної мови і сформувати для неї таблицю лексем:

а) функції виводу даних типу printf, котра може виводити строкові літерали та ідентифікатори, передбачити можливість введення формату виводу даних та використання спеціальних символів, що позначають кінець рядка, табуляції, тощо;

б) lisp-списків вигляду (<ел-т> <ел-т> … <ел-т>), елементами можуть бути ідентифікатори або такі самі списки, також кожний елемент може мати перед собою «'». Списки можуть бути пустими, тобто допускається «()», елементи розділені пробілами;

в) html-списку, котрий може в довільному порядку містити вкладені теги ul, ol (елементи списків задаються тегом li). Щоб можна було вивести <OL> <LI>ЛА</LI><UL><LI> До роздільника</LI><LI> ДС </LI> <LI>КА</LI></UL> <LI> Синтаксичний аналізатор</LI> <LI> ГКМК</LI></OL>;

г) оператора виводу, котрий може виводити ідентифікатори та арифметичні вирази;

д) мови, що містить оператори присвоєння, при чому ідентифікатору можна присвоювати як арифметичні вирази, так і логічні, передбачити наявність констант та ідентифікаторів булевого типу, котрі можуть приймати значення true, false;

е) мови, що містить оператор присвоєння (арифметичного виразу, операндами котрого можуть бути ідентифікатори та константи з фіксованою точкою), goto <мітка> та умовного переходу виду if (<відношення>){}, передбачити можливість оголошення міток та ідентифікаторів;

ж) мови, що містить оператор присвоєння (арифметичного виразу, операндами котрого можуть бути ідентифікатори та цілі константи), а також дозволяє оголошувати та використовувати підпрограми (функції/процедури);

з) мови, що містить оператор присвоєння (арифметичного виразу, операндами котрого можуть бути ідентифікатори та константи з плаваючою точкою) та умовного переходу виду if <відношення> goto <id>, передбачити можливість оголошення ідентифікаторів типу float та спеціального типу, що відповідає міткам;

и) мови, що містить оператор присвоєння арифметичного виразу з унарним мінусом та тернарного оператора, константи можуть бути десяткові та шістнадцяткові;

к) мови, що містить С-подібний оператор циклу типу for, а також оператор присвоєння арифметичного виразу.

Завдання 3.2. Проаналізувати отриману в завданні 3.1 таблицю лексем, визначити односимвольні та багатосимвольні роздільники.

Завдання 3.3. Написати приклад програми для мови з завдання 3.1. Сформувати для нього вихідні таблиці, котрі мають бути на виході лексичного аналізатора.

Завдання 3.4. Визначити класи символів створеної за завданням 3.1 мови.

Завдання 3.5. Побудувати діаграму станів для створеної за завданням 3.1 мови.

Завдання 3.6. Побудувати скінчений автомат, що розпізнає лексеми створеної за завданням 3.1 мови.

Завдання 3.7. Опишіть за допомогою регулярних виразів мови, описані в завданні 2.11 та 2.12, а також наступні мови:

а) номери телефонів з семи цифр розділених дефісами (на кшталт 123-45-67), при чому домашні номери можуть не містити код міста, а мобільні можуть не містити код держави (код міста записується в дужках);

б) всі ланцюжки з українських букв, котрі містять голосні букви «а», «е», «и», «і», «о», «у» саме в такому порядку;

в) коментарі, що розміщуються між /* та */, в середині коментаря «/*» та «*/» може зустрічатися лише в лапках;

г) всі ланцюжки з символів a, b, що містять підланцюжок abb;

д) всі ланцюжки з символів a, b, що закінчуються на bb;

е) всі ланцюжки з символів a, b, що не містять підланцюжка abb;

ж) всі ланцюжки з символів a, b, з парною кількістю символів a і парною кількістю символів b;

з) всі ланцюжки з символів a, b, що починаються і/або закінчуються на ab;

и) непарні цифри;

к) арифметичні вирази, з бінарними операторами +,-,*,/, та унарним мінусом, в якості операндів виступають цілі числа;

л) коректні адреси електронної пошти;

м) цифри в грошовому форматі, наприклад «$12.99» або «€1,99»;

н) час в форматі 12:58am, 1:30pm;

о) правильну поштову адресу.

Завдання 3.8. Опишіть мови, що задаються наступними регулярними виразами:

а) a(a|b)*a;

б) ((∧ |a)b*)*;

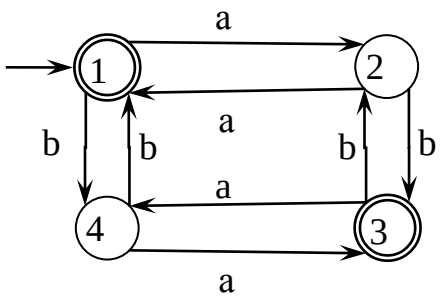
в) (a|b)*a(a|b)(a|b);

г) a*ba*ba*ba*;

д) (aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*.

Завдання 3.9. Побудуйте скінченні автомати для мов з завдань 3.7 та 3.8.

Завдання 3.10. Яку мову породжує наступний скінченний автомат:



Завдання 3.11. Розробити лексичний аналізатор, що розбирає текст до роздільника на основі отриманих даних в процесі виконання завдання 3.1, 3.2.

Завдання 3.12. Розробити лексичний аналізатор, що знаходить в довільному тексті ланцюжки, описані в завданні 3.7, використовуючи механізм регулярних виразів.

Завдання 3.13. Розробити лексичний аналізатор, що використовує діаграми станів, отримані в процесі виконання завдання 3.5. Спробуйте використовувати case-конструкції та мітки, встановіть, який механізм працює швидше.

Завдання 3.14. Розробити лексичний аналізатор, що реалізує довільний СКА, представлений у вигляді таблиці переходів. Протестувати отриманий аналізатор на прикладах СКА завдання 3.6.

РОЗДІЛ 4. СИНТАКСИЧНИЙ АНАЛІЗАТОР

4.1. Призначення і види синтаксичного аналізу

Мета синтаксичного розбору – визначення у вхідному тексті мовних конструкцій, що описуються граматикою.

На вході синтаксичний аналізатор (СА) отримує побудовані лексичним аналізатором вихідні таблиці лексем. В процесі своєї роботи СА будує дерево виводу для отриманої послідовності лексем, часто цей процес здійснюється неявно, але на ньому базуються основні методи синтаксичного розбору [22]. На виході СА повинен винести вердикт, чи є отримана послідовність лексем правильним реченням даної мови.

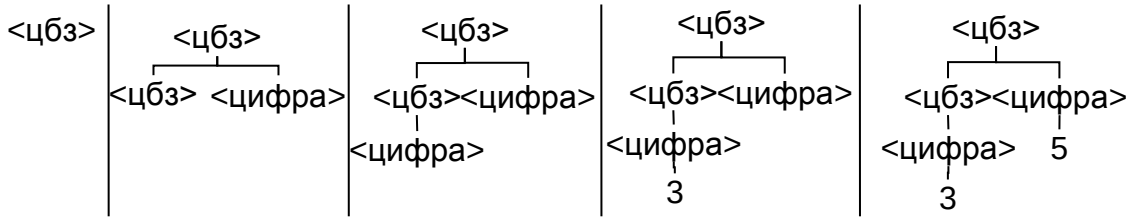
Розрізняють дві категорії алгоритмів синтаксичного розбору: низхідний (зверху-вниз) та висхідний (знизу-вверх). Ці терміни відповідають способу побудови дерев виводу. При низхідному розборі дерево будується від кореня (початкового символу, тобто аксіоми) до кінцевих вузлів (термінальних символів).

Для наступної граматики побудуємо дерево та вивід ланцюжка **35.**

<цбз>::=<цифра>|<цбз><цифра>

<цифра>::=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Нижче (рисунок 4.1) представлено поетапну побудову дерева виводу та повний вивід ланцюжка **35.**



<цбз>→<цбз><цифра>→<цифра><цифра>→3<цифра>→35

Рисунок 4.1— Приклад побудови дерева виводу зверху-вниз

На першому етапі виконується підстановка, що замінює аксіому, на кожному наступному етапі найлівіший нетермінал поточної сентенціальної форми замінюється на праву частину відповідного правила. Складність полягає в тому, що потрібно отримати саме ту сентенціальну форму, котра співпадає з заданим ланцюжком.

Метод висхідного розбору полягає в тому, що заданий термінальний ланцюжок намагаються привести до аксіоми граматики (рисунок 4.2). На першому кроці розбору ланцюжка **35** термінал **3** приводять до нетермінала **<цифра>**, в результаті чого отримується сентенціальна форма **<цифра>5**. На наступному кроці **<цифра>** приводиться до **<цбз>** і так далі, доки не буде отримано дерево, що містить аксіому. Наведемо послідовність підстановок.

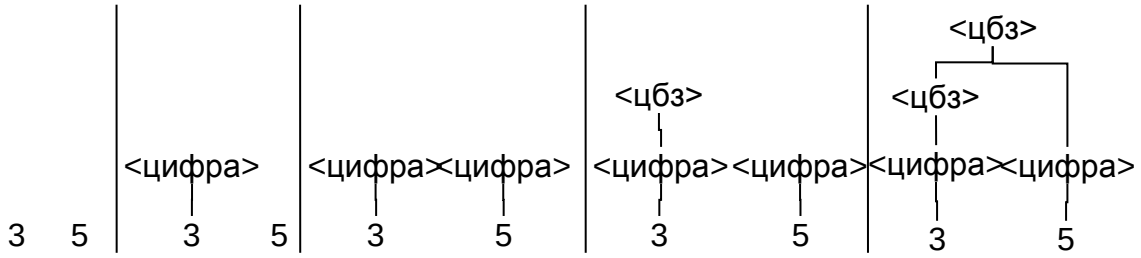


Рисунок 4.2 — Приклад побудови дерева виводу знизу-вверх

Розбір "зверху-вниз"

На практиці при низхідному розборі виникає проблема повернень. Її зміст полягає в наступному. Якщо нетермінал визначається правилом з кількома альтернативами, наприклад

$$\mathbf{V ::= X_1 \mid X_2 \mid \ldots \mid X_n},$$

то як дізнатися, котрим з ланцюжків $\mathbf{X_i}$ необхідно замінити **V**? Одним з розв'язків є вибір одного з можливих варіантів навмання, з припу-

щенням, що він вірний. Якщо пізніше виявиться помилка, необхідно повернутися назад і спробувати застосувати інший варіант. Цей прийом називається поверненням.

Іншим розв'язком є перегляд контексту навколо підланцюжка, котрий обробляється в даний момент, що дозволить зробити правильний вибір.

Низхідний розбір з поверненням

Нехай деякій людині потрібно розібрати речення A, що складається з підланцюжків a_i (i = 1, 2, …, n).

$$\mathbf{A=a_1a_2...a_n}$$

Розбір визначається правилом:

Q::=x1x2...xn| y1y2...yn| z1z2...zn

Спочатку людина намагається застосувати правило $\mathbf{Q::=x_1x_2...x_n}$. Як їй визначити, чи вірно вона обрала цю підстановку? Вважаємо, що вивід буде правильним, якщо кожне $\mathbf{x_i}$ породжує ланцюжок $\mathbf{a_i}$ в початковому реченні.

Насамперед людина бере собі названого сина (процедуру) $\mathbf{M_1}$ та доручає йому провести вивід $\mathbf{x_1|\rightarrow a_1}$. Якщо приймаку $\mathbf{M_1}$ вдається знайти такий вивід, він (і будь-який з його названих синів, онуків і т.д.) закриває підланцюжок $\mathbf{a_1}$ в реченні **A** та повідомляє батька про успіх. Тоді батько всиновлює $\mathbf{M_2}$, щоб він знайшов вивід $\mathbf{x_2|\rightarrow a_2}$, тощо.

Як вчинити, якщо $\mathbf{M_i}$ не зможе знайти вивід $\mathbf{a_i}$ з $\mathbf{x_i}$? $\mathbf{M_i}$ повідомляє про невдачу батькові. Батько зрікається його і доручає $\mathbf{M_{i-1}}$ знайти інший вивід для $\mathbf{a_{i-1}}$. Якщо $\mathbf{M_{i-1}}$ повідомляє про невдачу, батько зрікається і його і звертається до $\mathbf{M_{i-2}}$. Якщо доведеться зректися $\mathbf{M_1}$, то підстановка $\mathbf{Q::=x_1x_2...x_n}$ була невірною і людина спробує скористатися підстановкою $\mathbf{Q::=y_1,y_2,...,y_n}$.

Слід зазначити, що при виводі син завжди діє так само як батько. Привабливість методу в тому, що кожна людина повинна пам’ятати лише про свого батька та своїх синів. Імітація всиновлення і зречення здійснюється за допомоги стека.

Вилучення повернень і зациклювань

Перш за все хотілося б вилучити повернення. Для цього необхідно відсортувати всі правила, що відносяться до кожного нетермінала, таким чином, щоб одразу знаходити вірне правило. Таку операцію можна виконати над більшістю мов програмування.

Повернення можуть виникнути тому, що голови в альтернативних визначеннях одного нетермінала можуть співпадати. Якщо виділити однакові голови, повернення можна виключити. Така процедура називається **факторизацією**. Вона полягає в тому, що якщо є правило вигляду:

V::=xy|xz|xw,

його можна переписати у вигляді:

V::=x(y|z|w),

тут дужки відіграють роль метасимволів.

Подібна ситуація виникає, якщо правило має вигляд

V::=W | Z

де і **W,** і **Z –** нетермінал. В цьому випадку необхідно підставити визначення цих нетерміналів, щоб вибір був однозначним.

Другою проблемою є зациклювання. Воно виникає, коли в граматиці є ліва рекурсія, тобто правило вигляду:

X:=X…

При його розборі одразу всиновлюється той, хто буде шукати **X**. Він, в свою чергу, заведе сина, щоб шукати **X** і т.д. Тобто, потрібно позбавитися лівої рекурсії. Найкращий спосіб – записувати правила, використовуючи ітеративні позначення, а саме

E::=E+T|E-T|T можна записати як E::=T{+T|-T}

T::=F|T*F|T/F можна записати як T::=F{*F|/F}

Фігурні дужки – це метасимволи, вони означають повторення свого вмісту довільну кількість разів, у тому числі 0.

Отже, сформулюємо 2 принципи, що дозволяють перетворити правила граматики, щоб уникнути повернень та зациклювань при низхідному розборі.

1. Факторизація. Якщо існують правила виду **U::=xy| xz | … |xw**, їх необхідно замінити на **U::=x(y| z | … |w)**. Зазначимо, що **U::=x|xy** перетворюється на правило **U::=x(y|Λ)**, пустий ланцюжок завжди має бути останньою альтернативою, оскільки такий ланцюжок завжди виконується.
2. Після факторизації в граматиці залишаться правила, що містять в правій частині не більше однієї лівої рекурсії, котра замінюється ітеративним записом.

Наприклад, **U::=Ux|Uy|Z** заміниться на **U::=U(x|y)|Z,**, а потім на **U::=Z{x|y}.**

Нехай в граматиці є правило **U::=x | y | … | z | Uv** . Таке правило набуде вигляду **U::= (x | y | … | z) { v }.**

Приклад 4.1

Нехай існує правило: **A::=BC | BCD | Axz | Axy**

Після факторизації отримаємо: **A::=BC (D | Λ) | Ax (z | y)**

Після здійснення другого пункту перетворення отримаємо:

A::=BC (D | /\ ) { x (z | y) }

Приклад 4.2

Перетворимо тепер за розглянутими правилами граматику міні-Паскаля. Слід зауважити, що на рівні синтаксичного аналізу ідентифікатори та константи являють собою неподільні лексеми, будемо позначати їх **id** та **const**, відповідно. Збережемо нумерацію початкової граматики.

Перетворена граматика міні-Паскаля.

1. <прогр> ::= program id var <спис. огол.> begin <сп. опер.> end.

3. <спис. огол.> ::= <оголошення>{; <оголошення>}

4. <оголошення> ::= <сп. ід.>: <тип>

5. <тип>::= integer | real

6. <сп. ід.> ::= id {, id }

7. <сп. опер.> ::= <опер.>{ ; <опер.>}

8. <опер.> ::= id := <вираз>| read (<сп. ід.>) | write (<сп. ід.>) | for id := <вираз> to <вираз> do <дія>

13. <вираз> ::= <терм> { + <терм> | –<терм> }

14. <терм> ::= <множ.> { * <множ.> | / <множ.> }

15. <множ.> ::= id | const | (<вираз>)

17. <дія> ::= <опер.> | begin <сп. опер.> end

Деякі правила зникли внаслідок вимог до перетвореної граматики. Наприклад, для однозначного визначення правильної альтернативи нетермінала <опер.>, треба було розкрити нетермінали <присв.>, <введення>, <виведення> та <цикл>, при цьому відповідні їм правила зникли. Також було вирішено одразу розкрити нетермінали <індекс. вир.> та <назва>. Правила 18-22, котрі описували ідентифікатори та константи, не потрібні на етапі синтаксичного аналізу.

4.2. Рекурсивний спуск

Перетворена відповідно до розглянутих раніше правил граматика може використовуватися в низхідному методі граматичного розбору без повернень, котрий називається методом рекурсивного спуску.

Процесор граматичного розбору, що базується на цьому методі [21], складається з окремих процедур для кожного нетерміналу, визначеного граматикою. Кожна така процедура відображає праву частину відповідного правила і намагається знайти у вхідному потоці підрядок, що починається з поточної лексеми і може бути інтерпретований як нетермінальний символ, пов'язаний з даною процедурою. В процесі своєї роботи вона може викликати інші подібні процедури або навіть саму себе (рекурсивно) для пошуку інших нетерміналів, якщо вони зустрічаються в правій частині правила.

Якщо ця процедура знаходить (тобто визначає) відповідний їй нетермінал, вона завершує свою роботу, передає керуючій програмі ознаку успішного завершення і встановлює вказівник поточної лексеми на першу лексему після розпізнаного підрядка. Якщо ж процедурі не вдається знайти підрядок, котрий може бути інтерпретований як відповідний нетермінальний символ, вона завершується з ознакою невдачі і викликає процедуру видачі діагностичного повідомлення та процедуру відновлення.

Процедура видачі діагностичних повідомлень повинна повідомляти про місце помилки (наприклад, рядок, символ) і можливо, про зміст помилки.

Наведемо схему процедури, що реалізує аксіому граматики міні-Паскаля (рисунок 4.3) в синтаксичному процесорі, реалізованому по методу рекурсивного спуску. Змінна found є ознакою успішного або невдалого завершення процедури. Змінна lex зберігає код поточної лексеми.

```
procedure Main(var found : boolean);
begin
 found := false;
 if (lex = 1) then {program}
 begin
  Перейти до наступної лексеми
  if (lex = 23) then {id}
  begin
   Перейти до наступної лексеми
   if (lex = 2) then {var}
   begin
    Перейти до наступної лексеми
    { ListDeclaration()- функція, що відповідає <спис. огол.>,
повертає ознаку успішного або неуспішного завершення}
    if (ListDeclaration() = true) then
     if (lex = 3) then  {begin}
     begin
      Перейти до наступної лексеми
      {ListOp() - функція, що відповідає <сп.опер.>}
      if (ListOp() = true) then
      begin
       Перейти до наступної лексеми
       if (lex = 4) then {end.}
        found := true
       else
        Error('Невірно завершено програму')
      end else // of (ListOp() = true)
       Error('Невірний список операторів')
     end else // of begin
      Error('Очікується початок програми begin')
    end else // of ListDeclaration() = true
     Error('Невірний список оголошень');
   end else // of var
    Error('Очікується var')
  end else // of id
   Error('Очікується назва програми')
 end else
  Error('Пропущено ключове слово program') end;
```

Рисунок 4.3 — Схема підпрограми, що реалізує аксіому міні-Паскаля

Наведена процедура викликає підпрограми:

- ListDeclaration() - функція, що відповідає нетерміналу <спис. огол.>, повертає ознаку успішного або невдалого завершення пошуку правильного списку оголошень в тексті вхідної програми;
- ListOp() – функція, котра шукає правильний список операторів (відповідно до нетермінала <сп.опер.>) в тексті вхідної програми.

Приведемо схему підпрограми ListDeclaration() (рисунок 4.4).

```
Function: boolean; {список оголошень}
var found : boolean;
begin
  found := false;
  if Declaration() = true then // знайдено вірне оголошення
  begin
    found := true;
    while (lex = 14 and found=true) do //поки «;»
    begin
      Перейти до наступної лексеми
      if Declaration()= true then // знайдено вірне оголошення
        ; // нічого не робити
      else found=false;
    end;
  end;
  Result := found;
end; // of ListDeclaration()
```

Рисунок 4.4 — Схема підпрограми для <оголошення> Міні-Паскаля

Функція ListOp(), що відповідає нетерміналу <сп.опер.>, має аналогічну структуру, тому її схему наводити не будемо. Натомість, розглянемо реалізацію підпрограм для пошуку правильного оголошення (рисунок 4.5), та списку ідентифікаторів (рисунок 4.6) методом рекурсивного спуску. Функцію для нетерміналу <оголошення> назвемо Declaration(), вона викликається описаною раніше функцією ListDeclaration(). В правій частині правила, що відповідає нетерміналу <оголошення>, зустрічається нетермінал <тип>, і йому

має відповідати окрема підпрограма. В цілях оптимізації програмного коду розкриємо її в середині функції Declaration().

```
function Declaration(): boolean; {оголошення}
var found : boolean;
begin   found := false;
  if IdList() = true then // знайдено список ідентифікаторів
  begin
    //розкриття підпрограми для нетермінала <тип>
    if (lex = 5 or lex = 6) then // integer або real
    begin
      Перейти до наступної лексеми
      found := true;
    end else // якщо <тип> не знайдено
      Error('Не знайдено тип змінних')
  end else
    Error('Невірний список ідентифікаторів');
  Result := found;
end; // of Declaration()
```

Рисунок 4.5 — Схема підпрограми для <оголошення> Міні-Паскаля

```
function IdList(): boolean; {список ідентифікаторів }
begin   Result := false;
  if lex = 23 then // знайдено ідентифікатор
  begin
    Перейти до наступної лексеми
    Result := true;
    while (lex = 16 and found=true) do //поки «,»
    begin
      Перейти до наступної лексеми
      if lex = 23 then // знайдено ідентифікатор
        Перейти до наступної лексеми
      else found := false;
    end;
  end else // якщо ідентифікатор не знайдено
    Error('Очікується ідентифікатор')
end; // of IdList()
```

Рисунок 4.6 — Схема підпрограми для <сп. ід.> Міні-Паскаля

Побудуємо схему для оператора введення (рисунок 4.7):

<введення> ::= read(<сп. ід.>)

```
function Read(): boolean; {введення}
var found : boolean;
begin   found := false;
  if (lex = 6) then // знайдено слово read
  begin
    Перейти до наступної лексеми
    if (lex = 21) then // знайдено (
    begin
      Перейти до наступної лексеми
      if IdList() = true then // знайдено список ідентифікаторів
        if (lex = 22) then // знайдено )
        begin
          Перейти до наступної лексеми
          found := true;
        end else // якщо ) не знайдено
          Error('Очікується )')
      else // of IdList() = true
        Error('Не вірний список ідентифікаторів')
    end else
      Error('Очікується (');
  end else
    Error('Очікується слово read');
  Result := found;
end; // of Read()
```

Рисунок 4.7 — Схема підпрограми для оператора введення

Виконаємо синтаксичний розбір речення: read (VALUE), користуючись наведеними підпрограмами: Read() та IdList() (рисунок 4.8).

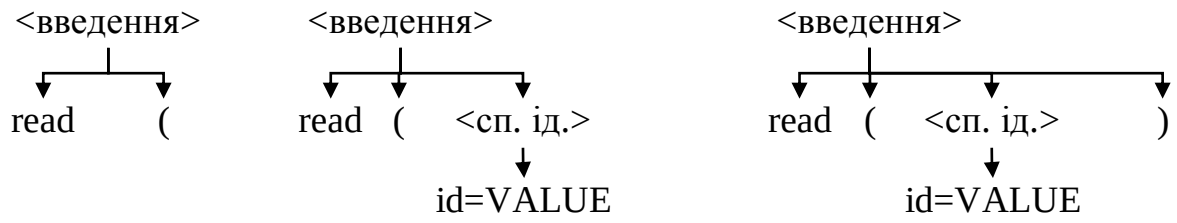


Рисунок 4.8 — Послідовність побудови дерева виводу «read (VALUE)»

Якщо список операторів організований таким чином, що роздільник (;) стоїть після кожного оператора, схема відповідної підпрограми декілька відрізняється від наведеної раніше (рисунок 4.4).

Приклад 4.3

Нехай дано граматику:

<програма>::=<список операторів> end

<список операторів>::= <список операторів> <опер.> ;| <опер.>;

Після підготовки граматики для рекурсивного спуску отримаємо:

<список операторів>::= <опер.>;{ <опер.>;}

Наведемо (рисунок 4.9) схему підпрограми для нетермінала <список операторів>. Змінна lex містить назву поточної лексеми.

```
function OPERLIST(): boolean; {список операторів }
begin Result := false;
  if PrOper()= true then // знайдено оператор
  begin
    Перейти до наступної лексеми
    if lex = ';' then
    begin
      Result := true;
      while (lex <> 'end' and found=true) do
{доки не трапиться лексема, що може йти після списку операторів}
        if PrOper()= true then
        begin
          Перейти до наступної лексеми
          if lex=; then
            Перейти до наступної лексеми;
          else Result := false;
        end else
          Result := false;
      end;//of while
    end else
      Error('Пропущено ;');
end;// of OPERLIST()
```

Рисунок 4.9 — Схема підпрограми список операторів

4.3. Магазинний автомат

4.3.1. Вступні визначення

Автомат з магазинною пам'яттю (МПА, МП-автомат, магазинний автомат, стековий автомат) – це сімка

$$M = < Q, \Sigma, Z, \delta, q_0, z_0, F >,$$

де

Q – скінченна множина станів;

Σ – вхідний алфавіт;

Z – алфавіт магазинної пам'яті;

δ – це відображення множини $Q \times (\Sigma \cup \{\wedge\}) \times Z$ в множину скінченних підмножин множини $Q \times Z^*$, тобто $Q \times (\Sigma \cup \{\wedge\}) \times Z \overset{\delta}{\to} Q \times Z^*$. Формально аргументами δ є трійки δ(α, x, γ), де $\alpha \in Q$, x – вхідний символ або пустий ланцюжок, $\gamma \in Z$. Вихід δ складають пари (β, ζ), де β – новий стан, ζ – ланцюжок магазинних символів, що замінює γ на вершині стека. Наприклад, якщо $\zeta = \wedge$, магазинний символ видаляється, якщо $\zeta = \gamma$, магазин не змінюється;

q_0 – початковий стан автомата;

z_0 – початковий символ магазину, котрий інакше називають **маркером магазинної пам'яті**;

F – скінченна множина заключних станів.

Конфігурацією МПА називається трійка $< \alpha, x, \gamma > \in Q \times \Sigma^* \times Z^*$, де α – поточний стан автомата, x – невикористана частина вхідного рядка (якщо $x = \wedge$ вважається, що вхідний ланцюжок прочитано), γ – вміст магазину, найлівіший символ ланцюжка γ називається верхнім символом магазину або вершиною магазину, якщо $\gamma = \wedge$, магазин вважається пустим.

Як і скінченні автомати, МПА поділяються на детерміновані та недетерміновані. Інтуїтивно, МПА є детермінованим, якщо в будь-якій конфігурації в нього немає можливості вибору різних переходів. Якщо вихід δ(α, x, γ) містить більше однієї пари, то МПА не є детермінованим. Однак, навіть якщо вихід δ(α, x, γ) містить одну пару, існує можливість вибору переходу по пустому ланцюжку δ(α, $\wedge$, γ). Таким чином, МПА $M =< Q, \Sigma, Z, \delta, q_0, z_0, F >$ визначається як детермінований, якщо виконуються наступні умови:

- δ(α, x, γ) містить не більше одного елемента для кожного α з Q, x з Σ та γ з множини Z;
- якщо δ(α, x, γ) не пусто для деякого x з Σ, то δ(α, $\wedge$, γ) має бути пустою.

В даному випадку МП-автомат може продовжувати свою роботу при завершенні вхідного ланцюжка, але не може продовжувати роботу, якщо вичерпано магазин [24].

МПА як і скінченний автомат можна зображати графічно, при чому усі стани розташовуються в колах, початковий стан позначається стрілкою, що входить в нього, а вихідні стани позначаються подвійними колами.

Приклад 4.4

Побудуємо МПА для мови $\{a^n b^n \mid n = \overline{0,\infty}\}$.

Нехай $M =< \{q_0, q_1, q_2\}, \{a, b\}, \{Z, A\}, \delta, q_0, Z, \{q_0, q_2\} >$, де

$\delta(q_0,a,Z)=\{(q_1,A)\}$

$\delta(q_1,a,A)=\{(q_1,AA)\}$

$\delta(q_1,b,A)=\{(q_2,\wedge)\}$

$\delta(q_2,b,A)=\{(q_2,\wedge)\}$.

Робота даного автомата полягає в копіюванні в магазин літер A з вхідного ланцюжка, та видаленні їх по одній за кожну прочитану літеру b. Зобра-

зимо отриманий МПА графічно (рисунок 4.10). Мітку переходу $\delta(\alpha,x,\gamma)=\{(\beta,\zeta)\}$ будемо формувати таким чином: $x,\gamma : \zeta$.

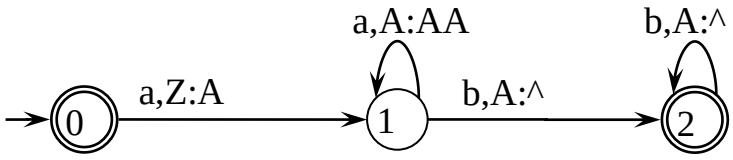


Рисунок 4.10 — МПА для мови $\{a^n b^n \mid n = \overline{0,\infty}\}$

Розберемо ланцюжок aaabbb за допомогою отриманого магазинного автомата, будемо відображати кроки розбору в горизонтальній таблиці (таблиця 4.1).

Таблиця 4.1. Розбір ланцюжка aaabbb

Поточний стан	q_0	q_1	q_1	q_1	q_2	q_2	q_2
Вхідна лексема	a	A	a	b	b	b	^
Наступний стан	q_1	q_1	q_1	q_2	q_2	q_2	
Магазин	Z	A	A A	A A A	A A	A	^

Розбір завершено успішно, оскільки після зчитування вхідного ланцюжка та вичерпання стека автомат опинився в заключному стані.

4.3.2. Використання магазинного автомата для синтаксичного розбору

При реалізації лексичного аналізу ми використовували скінченний автомат (СКА) для контролю послідовності символів в процесі побудови лексем. При реалізації синтаксичного аналізатора спробуємо використовувати такий самий автомат для контролю послідовності лексем в процесі граматичного розбору. СКА може бути використаний для синтаксичного розбору мови, що задається тільки автоматною граматикою.

Приклад 4.5

Побудуємо, наприклад, автомат для розбору речень, відповідних правилу:

<оператор>::= read(<сп. ід>);

Наведемо для наочності скінченний автомат у формі діаграми станів (рисунок 4.11) та списку переходів (таблиця 4.2).

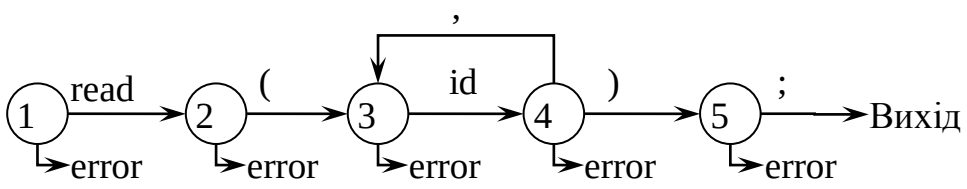


Рисунок 4.11 — Автомат у формі діаграми для розбору операторів введення

Завершальний стан відповідає виходу з вказівкою того, що розбір завершений успішно. Переходи на «error» позначають вихід з вказівкою невдалого закінчення.

Таблиця 4.2. Список переходів для розбору операторів введення

A	Мітка переходу	B	Семантична підпрограма
1	Read	2	[≠] помилка
2	(	3	[≠] помилка
3	id	4	[≠] помилка
4	,	3	[≠] помилка
	)	5	
5	;		[=] вихід, [≠] помилка

Але скінченного автомата виявляється недостатньо для розбору мови, яка описується контекстно-незалежною граматикою [25], а саме граматикою, що містить рекурсію з самовставленням, наприклад, дужкові конструкції. Для реалізації таких мов, тобто більшості мов програмування знадобиться автомат з магазинною пам'яттю.

Приклад 4.6

Спробуємо, все ж таки, побудувати автомат для розбору мови, що задається наступною граматикою:

A::=b|(A)

Отримаємо наступну діаграму (рисунок 4.12).

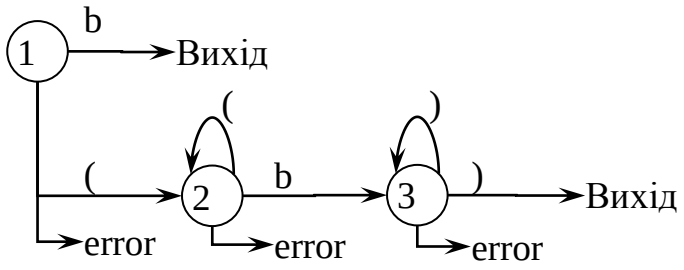


Рисунок 4.12 — Діаграма СКА для граматики дужкових конструкцій

Тут зі стану 3 по одному й тому ж символу «)» можливі різні переходи, що неприпустимо для детермінованого скінченного автомата. Для того, щоб визначитися, на який стан переходити в такій ситуації, потрібна додаткова інформація. Такою інформацією в даному випадку може бути ознака, що відрізняє пару зовнішніх дужок від будь-яких внутрішніх дужок. Введемо для зовнішніх дужок ознаку 1, а для внутрішніх – 11. Ознаки 1 і 11 у момент їх появи будемо записувати в стек, а при переході зі стану 3 будемо аналізувати вхідний символ і символ, зчитаний зі стека. Таким чином, тепер кожний перехід буде характеризуватися не тільки міткою, семантичною підпрограмою, початковим і наступним станом, але й дією по внесенню і зчитуванню даних зі стека.

Алгоритм функціонування формально описаного магазинного автомата передбачає на кожному переході вилучення символу зі стека під час його зчитування та занесення символів до стека. Однак, часто при переході не має потреби змінювати вміст стеку, в такому випадку необхідно після зчитування (вилучення) заносити в стек той самий символ, що тільки-но було вилучено.

Це провокує збільшення кількості операцій звернення до стеку. Якщо перетворювати автомат з прикладу 4.6 на магазинний, при переході зі стану 1 на стан 3 по символу b не потрібно ні зчитувати елементи зі стека, ні записувати в стек. При переході зі стану 1 на стан 2 по (, потрібно тільки записати ознаку 1 в стек, а зчитування можна не виконувати. Усі переходи зі стану 3 передбачають лише зчитування даних зі стека.

В цілях оптимізації алгоритму розбору функціонування автомата з магазинною пам'яттю можна описувати наступним набором правил:

1) $\alpha \xrightarrow{x} \beta$ — зі стану α перейти в стан β, якщо на вході символ x (при цьому зі стека символ не зчитується і до стека символ не заноситься);

2) $\alpha \xrightarrow{x\downarrow\gamma} \beta$ — зі стану α перейти в стан β і занести в стек γ, якщо на вході символ x (символ з вершини стека не зчитується);

3) $\alpha \xrightarrow{x\uparrow\gamma} \beta$ — зі стану α перейти в стан β, якщо на вході символ x, і при цьому зчитаний зі стека символ дорівнює γ.

Наведемо описаний автомат у формі діаграми (рисунок 4.13) та списку переходів (таблиця 4.3) для граматики дужкових конструкцій.

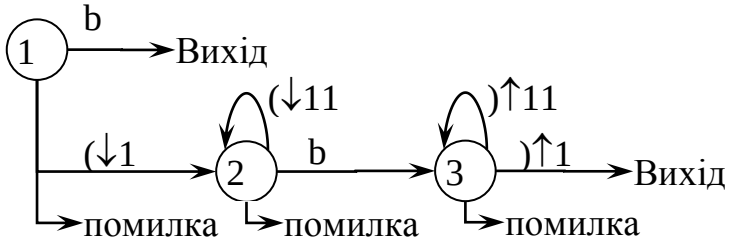


Рисунок 4.13 — Діаграма МПА для граматики дужкових конструкцій

Таблиця 4.3. Автомат з використанням стека у формі списку переходів для розбору дужкових конструкцій

A	Мітка переходу	B	Стек	Семантична підпрограма
1	B			[=] вихід
	(	2	↓1	[≠] помилка
2	(	2	↓11	[≠] помилка
	B	3		
3	)		↑1	[=] вихід
	)	3	↑11	[≠] помилка

Для кращого розуміння роботи розробленого автомата спробуємо з його допомогою розібрати речення ((b)), протокол розбору оформимо у вигляді таблиці (таблиця 4.4).

Таблиця 4.4. Розбір фрази ((b)) за допомогою МПА

Вхідна лексема	(	(	b	)	)
Стан	1	2	2	3	3
Стек		1	11 1	11 1	1

Додатково введені ознаки 1 та 11 є інтуїтивно зрозумілими, але це викликає труднощі при реалізації. Більш наглядним варіантом подання МПА є механізм використання вкладених автоматів або підавтоматів. Розглянемо його на прикладі.

Приклад 4.7

Представимо діаграму функціонування автомата для прикладу 4.6, використовуючи механізм підавтоматів (рисунок 4.14).

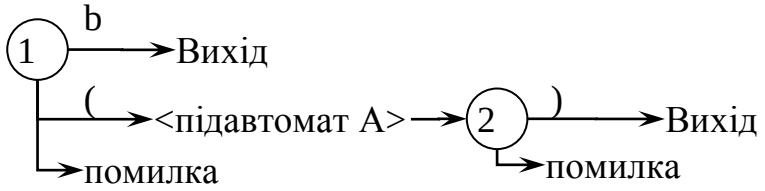


Рисунок 4.14 — Діаграма МПА дужкових конструкцій з підавтоматом

Тут зі стану 1 по символу «**(**» здійснюється перехід на деякий фрагмент — підавтомат (в даному випадку це той самий автомат А). Цей перехід відповідає правилу граматики, де після дужки «**(**» йде нетермінал А. Після закінчення роботи цього підавтомата здійснюється перехід на наступний стан 2. Перехід на підавтомат означає перехід в його початковий стан, а для збереження наступного стану головного автомата використовується стек. Після закінчення роботи підавтомата семантична процедура «вихід» означає зчитування зі стека записаної раніше адреси головного автомата.

Функціонування такого модифікованого автомата визначається наступним набором правил переходів.

1) $\alpha \xrightarrow{x} \beta$ — зі стану α перейти в стан β, якщо на вході символ x;

2) $\alpha \xrightarrow{x\downarrow\gamma} < п/а >$ — зі стану α перейти на підавтомат <п/а> і занести в стек γ, якщо на вході символ x;

3) $\alpha \xrightarrow{x\uparrow\gamma}$ — зі стану α перейти в стан γ, зчитаний зі стека, якщо на вході символ x.

Подамо автомат, що відповідає діаграмі (рисунок 4.14), у формі списку переходів (таблиця 4.5), і виконаємо розбір фрази ((b)) (таблиця 4.6).

Таблиця 4.5. МПА для розбору дужкових конструкцій з використанням підавтоматів

α	Мітка переходу	β	Стек	Семантична підпрограма
1	B			[=] вихід
	(	<п/а A>	↓2	[≠] помилка
2	)			[=] вихід, [≠] помилка

Тут правилом другого типу є перехід зі стану 1 по вхідному символу «(» на підавтомат A, зі записом в стек адреси 2. Правилом третього типу є «[=] вихід».

Таблиця 4.6. Розбір фрази ((b)) за допомогою МПА з підавтоматами

№ конфігурації	1	2	3	4	5
Вхідна лексема	(	(	b	)	)
Стан	1	1	1	2	2
Стек		2	2 2	2	

Розглянемо кожну конфігурацію автомата детальніше.

Конфігурація 1. Розбір починається, коли автомат знаходиться в стані 1, а на вхід потрапляє символ «(». Відповідно до другого рядка списку переходів, необхідно перейти на підавтомат A (тобто знову на стан 1) і занести в стек стан 2, в котрий необхідно повернутися після виходу з підавтомата. Таким чином здійснюється перший перехід.

Конфігурація 2. Другий перехід виконується аналогічно першому. Зі стану 1 по вхідному символу «(» виконується перехід в стан 1, в стек заноситься 2.

Конфігурація 3. Автомат знаходиться в стані 1, а на вході символ b, це означає, що наступні дії визначаються першим рядком списку переходів, а саме, необхідно виконати семантичну процедуру «вихід», відповідно до алгоритму розбору: якщо стек не пустий, потрібно зчитати символ зі стека і пе-

рейти на нього. В даному випадку стек не пустий, в вершині стека знаходиться 2, значить необхідно здійснити перехід на стан 2.

Конфігурація 4. Тепер автомат перебуває в стані 2, а на вхід потрапляє «)», відповідно до третього рядка списку переходів, здійснюємо «вихід»: зі стека зчитуємо наступний стан 2 і переходимо на нього.

Конфігурація 5. Знов опиняємося в стані 2, на вході «)», виконуємо «вихід»: аналізуємо стек, він пустий, отже розбір завершено, якщо вхідний ланцюжок вичерпано.

На практиці для синтаксичного розбору зручніше використовувати магазинний автомат з модифікованим набором правил. Апарат підавтоматів може бути використаний і для автоматної мови, продемонструємо це на прикладі.

Приклад 4.8

Побудуємо МПА з використанням підавтоматів для мови:

<оператор>::= read(<сп.ід.>);

<сп.ід.>::=id|id, <сп.ід.>

Зобразимо головний автомат та підавтомат **<сп.ід.>** у вигляді діаграм (рисунок 4.15, 4.16) та у вигляді списків переходів (таблиця 4.7, 4.8).

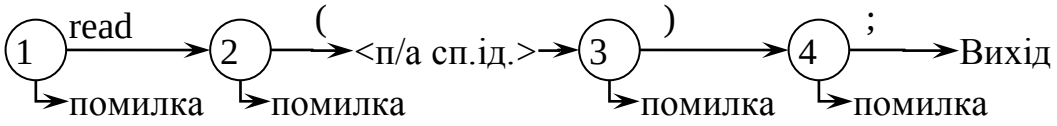


Рисунок 4.15 — МПА з підавтоматами для операторів введення

Таблиця 4.7. МПА з підавтоматами у формі списку переходів для операторів введення

A	Мітка переходу	β	Стек	Семантична підпрограма
1	read	2		[≠] помилка

Продовження таблиці 4.7

A	**Мітка переходу**	**β**	**Стек**	**Семантична підпрограма**
2	(	п/а **сп.ід.**	↓3	[≠] помилка
3	)	4		[≠] помилка
4	;			[=] вихід, [≠] помилка

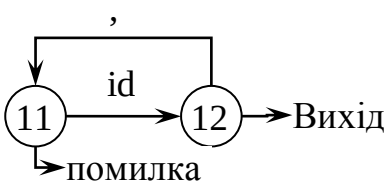


Рисунок 4.16 — Діаграма підавтомата **<сп.ід.>**

Таблиця 4.8. МПА з підавтоматами у формі списку переходів для операторів введення

α	**Мітка переходу**	**β**	**Стек**	**Семантична підпрограма**
11	id	12		[≠] помилка
12	,	11		[≠] вихід

Виконаємо розбір фрази «read(a, b, c);», де a, b, c – ідентифікатори. Наведемо протокол розбору у вигляді таблиці (таблиця 4.9).

Таблиця 4.9. Розбір фрази «read(a, b, c);»

№ конфігурації	1	2	3	4	5	6	7	8	9
Вхідна лексема	read	(	a	,	b	,	c	)	;
Стан	1	2	11	12	11	12	11	123	4
Стек			3	3	3	3	3	3	

Вхідний ланцюжок вичерпано, стек пустий і при цьому автомат перейшов на вихід, отже, розбір завершено успішно.

4.3.3. Побудова магазинного автомата для арифметичного і логічного виразів

Правила функціонування автомата визначають переходи автомата з одного стану в інший залежно від вхідного символу і стану стека. Виходи автомата несуть семантичне навантаження, що залежить від призначення і конкретного застосування автомата, і можуть працювати, як у разі збігу вхідного символу з допустимими символами в даному стані, так і в разі не збігу вхідного символу з жодним з допустимих в даному стані.

Скористаємося цією властивістю виходів для побудови магазинного автомата з підавтоматами для арифметичного виразу.

Для побудови автомата введемо поняття «хибного символу», з яким не порівняється жоден символ мови, і будемо використовувати його для переходу до підавтомата по «не зрівнянню». Позначатимемо «хибний символ» спеціальним символом «¢», а перехід по не зрівнянню з ним будемо помічати «¢≠». Перехід по не зрівнянню з «хибним символом» аналогічний переходу $\delta(\alpha, \wedge, \beta)$. Такий перехід буде здійснюватися для будь-якого вхідного символу, при цьому автомат не буде зчитувати наступний символ вхідного ланцюжка.

Побудуємо автомат для арифметичного виразу, заданого граматикою:

<вир> ::= <терм.> | <вир> + <терм.>| <вир>- <терм.>

<терм.> ::= <множ> | <терм.> * <множ> | <терм.> / <множ>

<множ> ::= (<вир>) | id | const

Приклад 4.9

Будемо розглядати всі нетермінали як підавтомати, тоді отримаємо наступні діаграми (рисунок 4.17-4.19) і списки станів (таблиця 4.10-4.12).

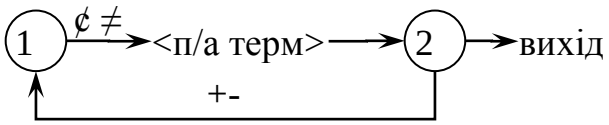


Рисунок 4.17 — Діаграма МПА для нетермінала <вир>

Таблиця 4.10. Список переходів МПА для нетермінала <вир>

A	**Мітка переходу**	**β**	**Стек**	**Семантична підпрограма**
1	¢			[=]помилка, [≠] п/а терм ↓2
2	+ -	2 2		[≠]вихід

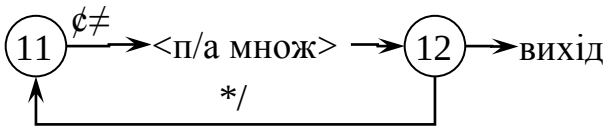


Рисунок 4.18— Діаграма підавтомата <терм.>

Таблиця 4.11. Список переходів підавтомата <терм.>

A	**Мітка переходу**	**β**	**Стек**	**Семантична підпрограма**
11	¢			[=]помилка, [≠] п/а множ ↓12
12	* /	11 11		[≠]вихід

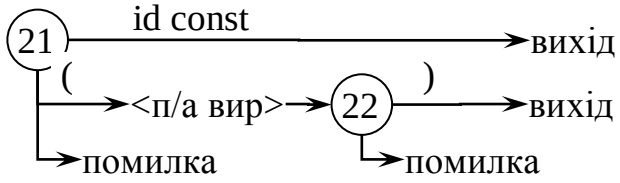


Рисунок 4.19 — Діаграма підавтомата <множ>

Таблиця 4.12. Список переходів підавтомата <множ>

A	**Мітка переходу**	**β**	**Стек**	**Семантична підпрограма**
21	id			[=]вихід
	const			[=]вихід
	(	п/а вир	↓22	[≠]помилка
22	)			[=]вихід , [≠]помилка

Використання таких дрібних підавтоматів ускладнює розбір через багатократні звернення до стека. Проілюструємо це на прикладі розбору фрази **a*(b+c);** (таблиця 4.13).

Таблиця 4.13. Розбір фрази a*(b+c);

№ конфігурації	1	2	3	4	5	6	7	8
Вхідна лексема	a	*	(	b	+	c	)	;
Стан	1 11 21	12	11 21	1 11 21	12 2	1 11 21	12 2 22	12 2 ⌂
Стек	12 2 ⌂	2 ⌂	12 2 ⌂	12 2 22 12 2 ⌂	2 22 12 2 ⌂	12 2 22 12 2 ⌂	2 22 12 2 ⌂	2 ⌂

Вважаємо, що на початок роботи автомата <вир> в стек вже занесена адреса повернення ⌂ у той автомат, котрий викликав <вир>, і в якому допустимим символом буде символ «;».

Таким чином, при використанні «хибного символу» отримаємо наступний алгоритм розбору.

1. Взяти поточний символ. Порівняти його з символами, припустимими в поточному стані автомата.
   - ➢ При зрівнянні:
     - виконати запис в стек, якщо вказано;
     - визначити наступний стан автомата:
       a) вказаний в β, якщо в β не порожньо;

б) якщо в β порожньо, за ознакою «[=] вихід» зчитати наступний стан зі стека;

- взяти наступний символ вхідного потоку.

➢ При незрівнянні:

- визначити наступний стан:
  а) за ознакою «[≠] вихід» зчитати адресу зі стека;
  б) за ознакою «[≠] п/а …» перейти на початковий стан підавтомата;
- за ознакою «[≠] помилка» видати повідомлення про помилку.

Приклад 4.10

Побудовані раніше автомати (рисунок 4.17-4.19) для розбору арифметичного виразу можна об'єднати в один. Зобразимо його графічно (рисунок 4.20), а також у вигляді списку переходів (таблиця 4.14).

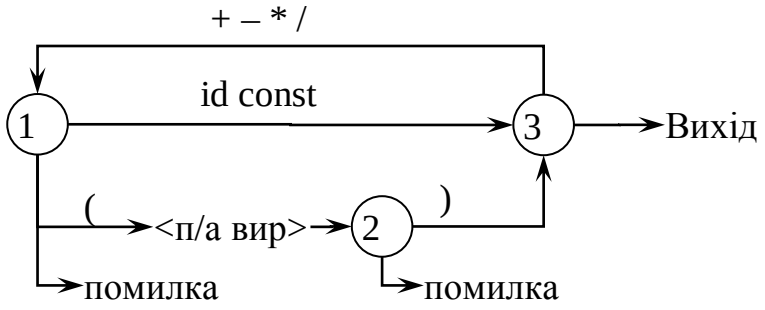


Рисунок 4.20 — Діаграма МПА для арифметичного виразу

Таблиця 4.14. Список переходів підавтомата <вир>

α	Мітка переходу	β	Стек	Семантична підпрограма
1	id	3		[≠]помилка
	const	3		

Продовження таблиці 4.14

α	Мітка переходу	β	Стек	Семантична підпрограма
	(	п/а вир	↓2	
2	)	3		[≠]помилка
3	+	1		[≠]вихід
	-	1		
	*	1		
	/	1		

Побудований автомат містить менше станів та функцій переходів, зникли переходи по не зрівнянню з «порожнім символом», а також переходи на підавтомати по не зрівнянню. При розборі він здійснює менше переходів та звернень до стеку, отже він має працювати швидше. Щоб переконатися в цьому, виконаємо розбір тієї самої фрази «a*(b+c);» за допомогою об'єднаного автомата (таблиця 4.15). Як і раніше позначатимемо символом «⌂» адресу повернення в автомат, котрий викликав <вир>.

Таблиця 4.15. Розбір фрази a*(b+c) за допомогою об'єднаного МПА

№ конфігурації	1	2	3	4	5	6	7	8
Вхідна лексема	a	*	(	b	+	c	)	;
Стан	1	3	1	1	3	1	3	2
Стек	⌂	⌂	⌂	2 ⌂	2 ⌂	2 ⌂	2 ⌂	⌂

Приклад 4.11

Очевидно, що об'єднаний автомат арифметичного виразу (рисунок 4.20) є компактнішим і зручнішим у використанні. Побудуємо на його основі автомат арифметичного виразу, в якому допускається використання одномісного знаку +,- (,).

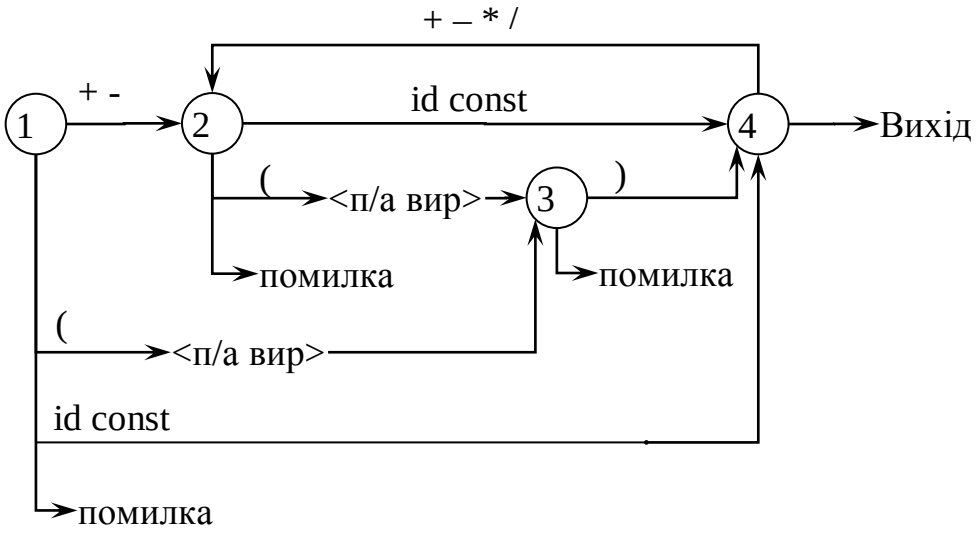


Рисунок 4.21 — Діаграма МПА для виразу з одномісними операціями +-

Таблиця 4.16. Список переходів МПА виразу з одномісними +-

α	**Мітка переходу**	**β**	**Стек**	**Семантична підпрограма**
1	id	4		[≠]помилка
	const	4		
	(	п/а вир	↓3	
	+	2		
	-	2		
2	id	4		[≠]помилка
	const	4		
	(	п/а вир	↓3	
3	)	4		[≠]помилка
4	+	2		[≠]вихід
	-	2		
	*	2		
	/	2		

Приклад 4.12

Тепер по аналогії до арифметичного виразу побудуємо автомати для логічного виразу, що задається граматикою:

<ЛВ>::=<ЛТ> {or <ЛТ>}

<ЛТ>::=<ЛМ> {and <ЛМ>}

<ЛМ>::= <вир> (<|>|=|≠|≥|≤)<вир>| [<ЛВ>] | not <ЛМ>

Одразу побудуємо об’єднаний автомат для всіх нетерміналів (рисунок 4.22, таблиця 4.17), будемо використовувати виклики описаного раніше підавтомата <вир>.

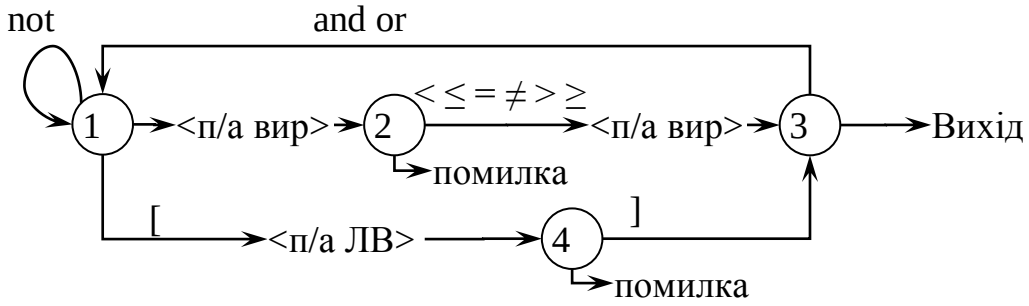


Рисунок 4.22 — Діаграма МПА для логічного виразу

Таблиця 4.17. Список переходів МПА логічного виразу

α	**Мітка переходу**	**β**	**Стек**	**Семантична підпрограма**
1	Not	1		[≠]п/а вир ↓2
	[	п/а ЛВ	↓4	
2	<	п/а вир	↓2	[≠]помилка
	≤	п/а вир	↓2	
	=	п/а вир	↓2	
	≠	п/а вир	↓2	
	>	п/а вир	↓2	
	≥	п/а вир	↓2	

Продовження таблиці 4.17

α	Мітка переходу	β	Стек	Семантична підпрограма
3	And	1		[≠] вихід
	Or	1		
4	]	3		[≠]помилка

Розглянемо побудову автоматів, що реалізують різні варіанти списку операторів і можливість використання міток. Будемо вважати, що підавтомат <оператор> вже існує, і розкривати його не будемо.

Приклад 4.13

Побудуємо автомат (рисунок 4.23, таблиця 4.18) для граматики:

<програма> ::= begin <сп. опер.> end

<сп. опер.> ::= <оператор>; {<оператор> ;}

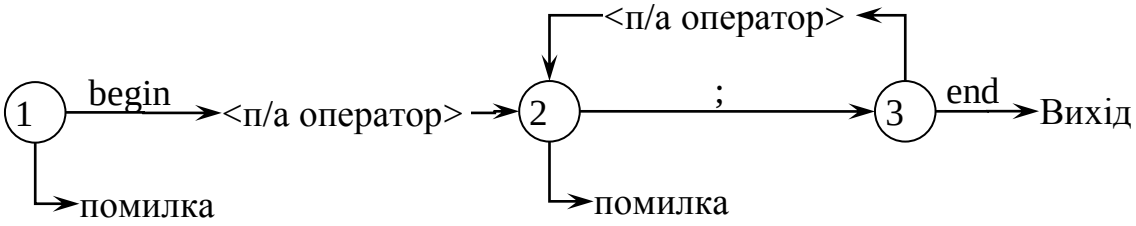


Рисунок 4.23 — Діаграма МПА (Приклад 4.13)

Таблиця 4.18. Список переходів МПА (Приклад 4.13)

α	Мітка переходу	β	Стек	Семантична підпрограма
1	begin	п/а оператор	↓2	[≠]помилка
2	;	3		[≠]помилка
3	end			[=] вихід, [≠]п/а оператор↓2

Приклад 4.14

Побудуємо автомат (рисунок 4.24) для граматики:

<програма> ::= begin <сп. опер.> end

<сп. опер.> ::= < оператор >{; < оператор >}

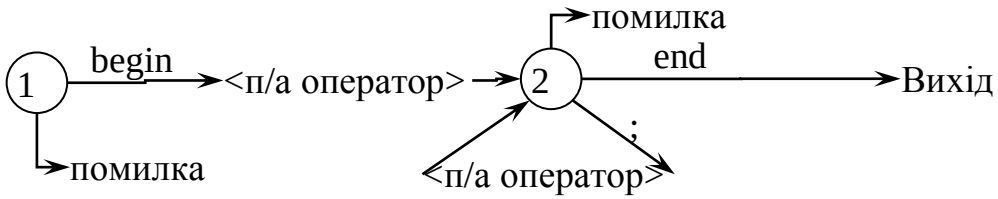


Рисунок 4.24 — Діаграма МПА (Приклад 4.14)

Таблиця 4.19. Список переходів МПА (Приклад 4.14)

α	**Мітка переходу**	**B**	**Стек**	**Семантична підпрограма**
1	begin	п/а оператор	↓2	[≠]помилка
2	end			[=] вихід,
	;	п/а оператор	↓2	[≠]помилка

Приклад 4.15

Побудуємо автомат для граматики, що містить мітки:

<програма> ::= begin <сп. опер.> end

<сп. опер.> ::= <помічений оператор>{; <помічений оператор>}

<помічений оператор>::= <оператор>| мітка : <оператор>

Для заданої граматики перші 2 правила можна описати аналогічним побудованому раніше МПА (рисунок 4.24), тільки замість виклику п/а оператор викликати п/а помічений оператор. Зобразимо графічно (рисунок 4.25) МПА, що розкриває нетермінал <помічений оператор>.

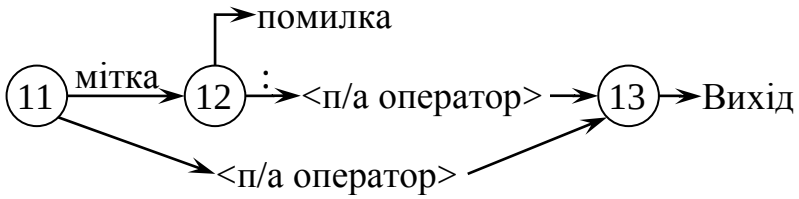


Рисунок 4.25 — Діаграма МПА <помічений оператор>

А тепер побудуємо єдиний МПА (Рисунок 4.26) для всієї граматики, наведемо також список переходів (Таблиця 4.20).

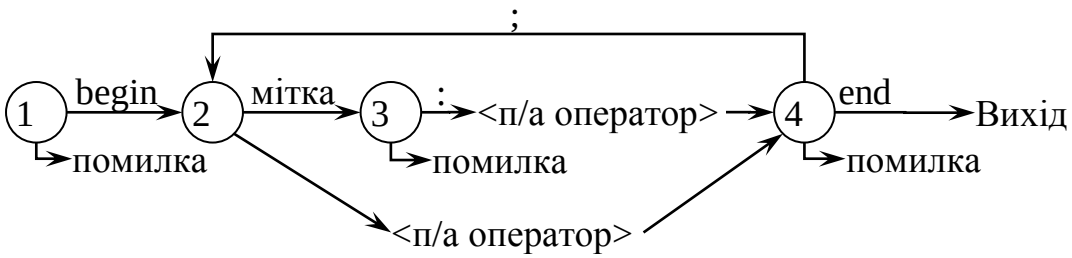


Рисунок 4.26 — Діаграма МПА (Приклад 4.15)

Таблиця 4.20. Список переходів МПА (Приклад 4.15)

α	**Мітка переходу**	**B**	**Стек**	**Семантична підпрограма**
1	begin	2		[≠]помилка
2	мітка	3		[≠]п/а оператор↓4
3	:	п/а оператор	↓4	[≠]помилка
4	end			[=]вихід
	;	2		[≠]помилка

4.3.4. Магазинний автомат для мови Mini-Паскаль

Тепер можемо побудувати магазинний автомат для тестової мови Міні-Паскаль. Будемо орієнтуватися на перетворену граматику.

1. <прогр> ::= program id var <спис. огол.> begin <сп. опер.> end.

3. <спис. огол.> ::= <оголошення>{; <оголошення>}

4. <оголошення> ::= <сп. ід.>: <тип>

5. <тип>::= integer | real

6. <сп. ід.> ::= id {, id }

7. <сп. опер.> ::= <опер.> {; <опер.> }

8. <опер.> ::= id := <вираз> | read (<сп. ід.>) | write (<сп. ід.>) | for id:= <вираз> to <вираз> do <дія>

13. <вираз> ::= <терм> { + <терм> | -<терм>}

14. <терм> ::= <множ.> { * <множ.> | / <множ.>}

15. <множ.> ::= id | const | (<вираз>)

17. <дія> ::= <опер.> | begin <сп. опер.> end

Почнемо реалізацію Mini-Паскаля з побудови головного автомата, відповідного аксіомі граматики (Рисунок 4.27, Таблиця 4.21).

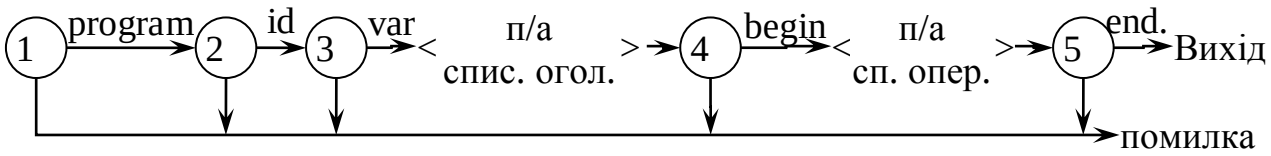


Рисунок 4.27 — Діаграма для аксіоми Міні-Паскаля

Таблиця 4.21. Список переходів МПА аксіоми Міні-Паскаля

α	**Мітка переходу**	**β**	**Стек**	**Семантична підпрограма**
1	program	2		[≠]помилка
2	id	3		[≠]помилка
3	var	п/а спис. огол.	↓4	[≠]помилка
4	begin	п/а сп. опер	↓5	[≠]помилка
5	end			[=] вихід, [≠]помилка

Список оголошень має наступне визначення:

<спис. огол.> ::= <оголошення>{; <оголошення>}

Що породжує, відповідно, діаграму (Рисунок 4.28) і список переходів (Таблиця 4.22) наведені нижче.

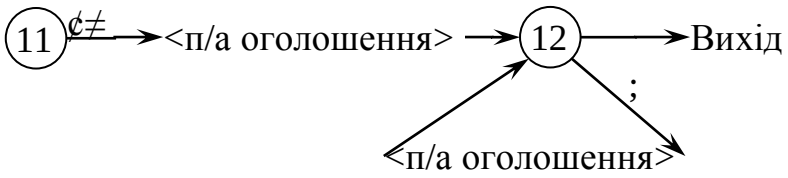


Рисунок 4.28 — Діаграма МПА для <спис. огол.>

Таблиця 4.22. Список переходів МПА для <спис. огол.>

α	**Мітка переходу**	**β**	**Стек**	**Семантична підпрограма**
11	¢			[≠]п/а оголошення ↓12
12	;	п/а оголошення	↓12	[≠] вихід

У свою чергу, оголошення визначається як:

<оголошення> ::= <сп. ід.>: <тип>

Йому відповідає наступна діаграма (Рисунок 4.29) і список переходів (Таблиця 4.23).

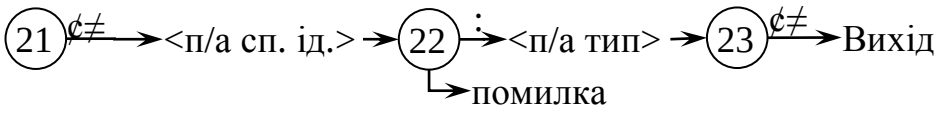


Рисунок 4.29 — Діаграма МПА для <оголошення>

Таблиця 4.23. Список переходів для <оголошення>

α	**Мітка переходу**	**β**	**Стек**	**Семантична підпрограма**
21	¢			[≠]п/а сп. ід.↓22
22	:	п/а сп. ід.	↓23	[≠] помилка
23	¢			[≠] вихід

Підавтомат для правила <тип> буде мати наступну діаграму (Рисунок 4.30) та список переходів (Таблиця 4.24).

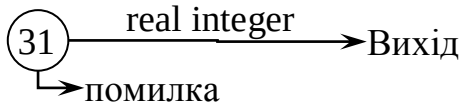


Рисунок 4.30 — Діаграма МПА для правила №5

Таблиця 4.24. Список переходів МПА для правила №5

α	**Мітка переходу**	**β**	**Стек**	**Семантична підпрограма**
31	real integer			[=] вихід [=] вихід, [≠]помилка

Список ідентифікаторів визначається наступним правилом <сп. ід.>, діаграмою (

Рисунок 4.31) і списком переходів (Таблиця 4.25).

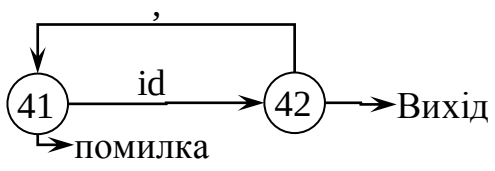


Рисунок 4.31 — Діаграма МПА для списку ідентифікаторів

Таблиця 4.25. Список переходів МПА для <сп. ід.>

α	**Мітка переходу**	**B**	**Стек**	**Семантична підпрограма**
41	id	42		[≠] помилка
42	,	41		[≠] вихід

Приклад 4.16

Наведемо приклад розбору фрагмента програми «program text var a, b: integer begin» по цих автоматах (Таблиця 4.26).

Таблиця 4.26. Розбір фрагмента програми Міні-Паскаль

№	1	2	3	4	5	6	7	8	9
Вхідна лексема	program	test	var	a	,	b	:	integer	begin
Стан	1	3	3	11 21 41	42	41	42 22	31	23 12 4
Стек				22 12 4	22 12 4	22 12 4	22 12 4	23 12 4	12 4

Очевидно, що розбір йтиме швидше, якщо зменшити число підавтоматів. Виключимо підавтомат <спис. огол.>, тобто внесемо визначення цього нетермінала до того правила, де це визначення використовується, в даному випадку в аксіому. Отримаємо наступне визначення аксіоми:

<прогр> ::= program id var <оголошення>{; <оголошення>} begin <сп. опер.> end.

Аналогічно виключимо підавтомати <оголошення> і <тип>, підставивши визначення цих нетерміналів в отримане правило.

<прогр> ::= program id var <сп. ід.> : (integer | real) {; <сп. ід.> : (integer | real)} begin <сп. опер.> end.

Так само вчинимо з визначенням <сп. ід.> і <сп. опер.>. Отримаємо наступне правило:

<прогр> ::= program id var id {, id } : (integer | real) {; id {, id }: (integer | real)} begin <опер.> { ; <опер.>} end.

Подальші вилучення автоматів недоцільні, оскільки підавтомат для <опер.> є досить масивним і рекурсивним.

Отриманому правилу відповідає наступна діаграма (Рисунок 4.32) і автомат у формі списку переходів (Таблиця 4.27)

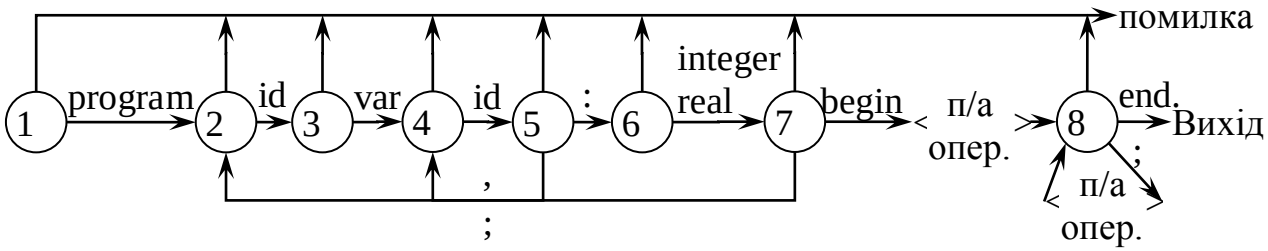


Рисунок 4.32 — Діаграма об’єднаного МПА аксіоми Міні-Паскаля

Таблиця 4.27. Список переходів об’єднаного МПА Міні-Паскаля

α	**Мітка переходу**	**B**	**Стек**	**Семантична підпрограма**
1	program	2		[≠]помилка
2	id	3		[≠]помилка
3	var	4		[≠]помилка
4	id	5		[≠]помилка
5	, :	4 6		[≠]помилка
6	integer real	7 7		[≠]помилка
7	begin ;	< п/а опер.> 2	↓8	[≠]помилка
8	end ;	< п/а опер.>	↓8	[=] вихід, [≠]помилка

Отже, в МПА аксіоми Міні-Паскаля використовується один підавтомат <опер.>. Він об’єднує оператори присвоєння, вводу, виводу та циклу:

<опер.> ::= id := <вираз> | read (<сп. ід.>) | write (<сп. ід.>) | for id:= <вираз> to <вираз> do <дія>

Наведемо його діаграму (Рисунок 4.33) і список переходів (Таблиця 4.28).

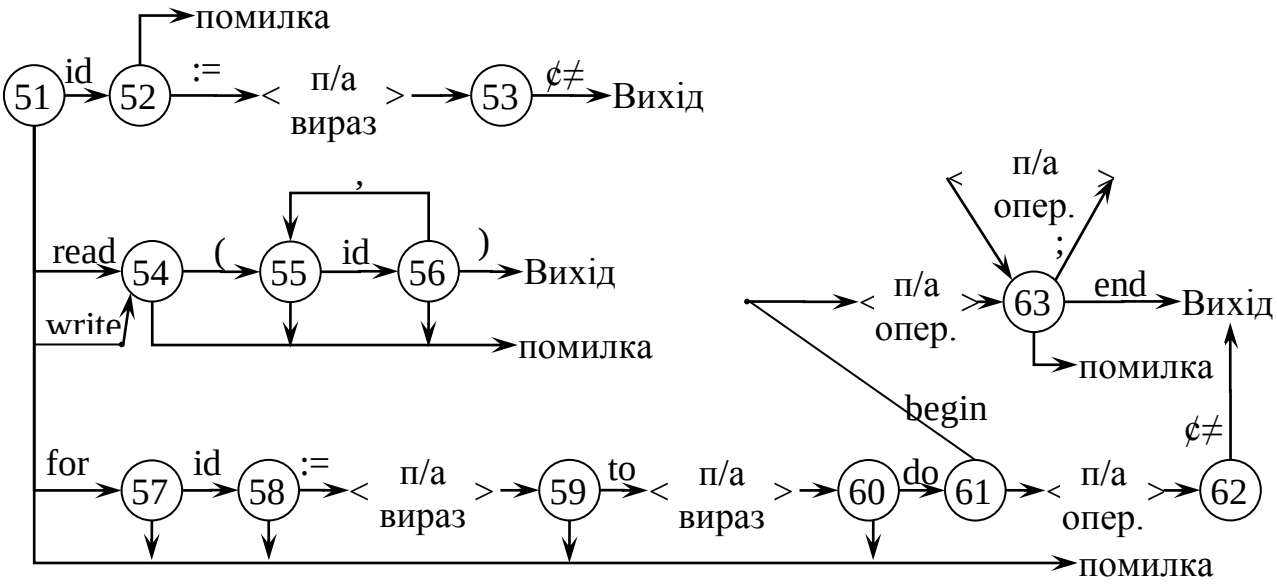


Рисунок 4.33 — Діаграма для підавтомата <опер.>

Таблиця 4.28. Список переходів МПА підавтомата <опер.>

α	**Мітка переходу**	**β**	**Стек**	**Семантична підпрограма**
51	id read write for	52 54 54 57		[≠]помилка
52	:=	<п/а вираз>	↓53	[≠]помилка
53	¢			[≠]вихід
54	(	55		[≠]помилка
55	id	56		[≠]помилка
56	) ,	 55		[=] вихід, [≠]помилка
57	id	58		[≠]помилка

Продовження таблиці 4.28

α	Мітка переходу	β	Стек	Семантична підпрограма
58	:=	<п/а вираз>	↓59	[≠]помилка
59	to	<п/а вираз>	↓60	[≠]помилка
60	do	61		
61	begin	<п/а опер.>	↓63	[≠]<п/а опер.>↓62
62	¢			[≠]вихід
63	end ;	<п/а опер.>	↓63	[=] вихід, [≠]помилка

Наведемо також (Рисунок 4.34, Таблиця 4.29) підавтомат вираз, котрий використовується в підавтоматі опер.

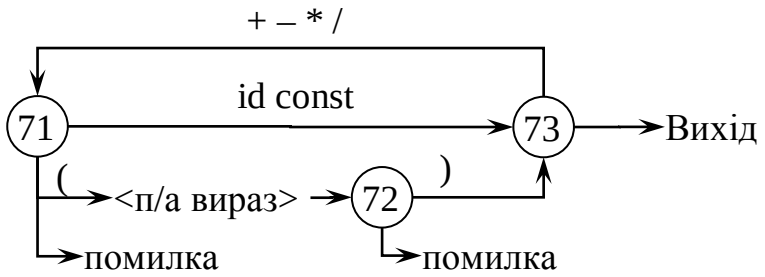


Рисунок 4.34 — Діаграма МПА для виразу Міні-Паскаль

Таблиця 4.29. Список переходів підавтомата вираз Міні-Паскаль

α	Мітка переходу	β	Стек	Семантична підпрограма
71	id	73		[≠]помилка
	const	73		
	(	<п/а вираз>	↓72	
72	)	73		[≠]помилка
73	+	71		[≠]вихід
	-	71		

α	Мітка переходу	β	Стек	Семантична підпрограма
	*	71		
	/	71		

Приклад 4.17

Виконаємо розбір простої програми (Рисунок 4.35), написаної мовою Міні-Паскаль, використовуючи наведені вище автомати.

```
program prim
var s, d: integer
begin
  s := 0;
  for i := 1 to 100 do
    begin
      read(d);
      s := s + d
    end;
  write(s);
end.
```

Рисунок 4.35 — Приклад Міні-Паскаль програми

Дана програма повинна обчислювати суму ста введених користувачем чисел, після чого отримана сума має бути виведена на екран. Для суми та введених користувачем чисел використовуються, ідентифікатори «s» і «d». Оформимо розбір наведеної програми у вигляді таблиці (Таблиця 4.30).

Таблиця 4.30. Розбір Міні-Паскаль програми

№	1	2	3	4	5	6	7	8	9	10	11	12	13
Вхідна лексема	program	prim	var	s	,	d	:	integer	begin	s	:=	0	;
Стан	1	2	3	4	5	4	5	6	7	51	52	71	73

													53 8
Стек										8	8	53 8	53 8

Продовження Таблиця 4.30

№	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Вхідна лексема	for	i	:=	1	to	100	do	begin	read	(	d	)	;	s	:=	s
Стан	51	57	58	71	73 59	71	73 60	61	51	54	55	56	63	51	52	71
Стек	8	8	8	59 8	59 8	60 8	60 8	8	63 8	63 8	63 8	63 8	8	63 8	63 8	53 63 8

Продовження Таблиця 4.30

№	30	31	32	33	34	35	36	37	38	
Вхідна лексема	+	d	end	;	write	(	s	)	end.	
Стан	73	71	73 53 63	8	51	54	55	56	8	вихід
Стек	53 63 8	53 63 8	53 63 8		8	8	8	8		

Розбір завершено успішно: вхідний ланцюжок вичерпано, стек – пустий, автомат знаходиться в кінцевому стані.

4.4. Висхідний розбір. Граматика простого передування.

Завданням висхідного розбору є приведення вхідного термінального ланцюжка до аксіоми граматики мови, котрій належить ланцюжок.

Нехай $G=\{V_T, V_N, \sigma, P\}$ – граматика, xzy – сентенціальна форма, тобто $\sigma|\rightarrow xzy$. Тоді z називається простою фразою сентенціальної форми для нетермінала Z, якщо:

1) в граматиці міститься правило Z::=z

2) ланцюжок xZy також є сентенціальною формою, тобто $\sigma|\rightarrow xZy$

Приклад 4.18

Нехай задано граматику:

<число>::=<чс>

<чс>::=<цифра>|<чс><цифра>

<цифра>::=0|1|...|9

Визначимо, що є простою фразою для сентенціальної форми **<чс>3**.

1. Чи є **<чс>** простою фразою? Перевіримо умови:
    1) в граматиці є правило, для якого ланцюжок **<чс>** збігається з правою частиною: <число>::=<чс>
    2) але після заміни в початковій сентенціальній формі **<чс>** на **<число>** отримаємо ланцюжок **<число>3**, котрий не може бути виведений з аксіоми, тобто не є сентенціальною формою. Значить, **<чс>** не є простою фразою.
2. Чи є **3** простою фразою? Перевіримо умови:
    1) **3** замінюється на нетермінал <цифра>, за правилом <цифра>::=3
    2) отриманий ланцюжок <чс><цифра> є сентенціальною формою.

Тобто **3** є простою фразою сентенціальної форми **<чс>3**.

Найлівіша проста фраза сентенціальної форми називається **основою.**

При висхідному розборі істотне значення має тип виводу. Визначимо **вивід зліва-направо** так, щоб на кожному кроці редукувалася (замінювалася) основа поточної сентенціальної форми, тоді ланцюжок праворуч від основи завжди буде містити виключно термінальні символи.

Безпосередній вивід xZy$\Rightarrow$xzy — канонічний, якщо **y** містить лише термінали. Вивід називається канонічним, якщо кожний безпосередній вивід в ньому є канонічним.

Отже, при висхідному розборі в поточній сентенціальній формі повторюється пошук основи z, яка відповідно до правила Z::= z граматики приводиться до нетерміналу Z.

При застосуванні будь-якого методу висхідного розбору виникає питання – як знайти основу і з'ясувати, до якого нетермінала її треба приводити. Це питання легко вирішується для певного класу граматик, котрі називаються **граматиками простого передування** [10].

Як же знайти основу сентенціальної форми? Хотілося б, рухаючись зліва направо і розглядаючи одночасно тільки два сусідні символи, визначити хвіст основи, а потім рухаючись справа наліво і аналізуючи знову тільки два сусідні символи сентенціальної форми, визначити голову основи.

Для цього ми повинні до початку розбору мати деякі відомості про кожну пару символів граматики, а саме відношення передування між ними.

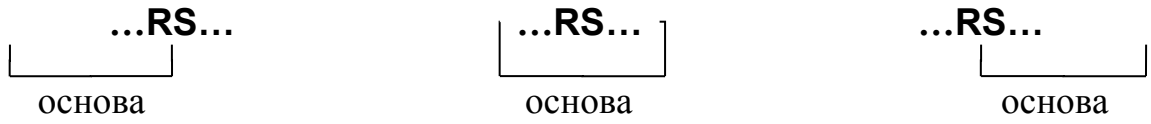
Розглянемо два символи R і S зі словника V. Припустимо, що існує канонічна сентенціальна форма …RS… . На деякому етапі розбору або R, або S, або обидва символи одночасно повинні увійти до основи. При цьому можливі наступні варіанти:

1) R – частина основи, а S – ні (Рисунок 4.36а), цю ситуацію записуємо як R •> S і говоримо, що R передує S або R більше S, оскільки

символ R буде редукований раніше, ніж S (R – останній символ основи і, відповідно, правила U::=…R, а S – термінальний символ, оскільки вивід канонічний);

2) обидва символи входять в основу (Рисунок 4.36б): записуємо $R \doteq S$, символи R і S мають однакове значення передування, вони редукуються одночасно (очевидно, в граматиці є правило U::=…RS…);

3) S – частина основи, а R – ні (Рисунок 4.36в): говорять, що R менше S, відношення записується R <• S (в цьому випадку символ S має бути першим в правій частині правила U ::=S…).



а) б) в)

Рисунок 4.36 — Варіанти розміщення основи сентенціальної форми

Якщо не існує канонічної сентенціальної форми …RS…, вважають, що між впорядкованою парою символів (R,S) не визначено жодного відношення передування.

Зауважимо, що жодне з відношень передування <•, $\doteq$, •> не є симетричним. Наприклад, з R<•S зовсім не слідує S•>R.

Приклад 4.19

Розглянемо граматику:

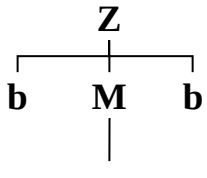
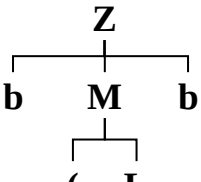
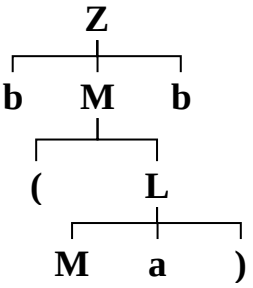
Z::=bMb

M::=(L | a

L::=Ma)

Вона породжує наступні ланцюжки: bab, b(aa) b, b((aa)a)b, тощо. В процесі виводу по цій граматиці можуть бути отримані, наприклад, такі сентенціальні форми: bab, B(Lb, b(Ma)b. Наведемо для кожної сентенціальної форми дерево виводу, основу і визначені за деревом відношення передування при редукції цієї основи (Таблиця 4.31).

Таблиця 4.31. Визначення відношень передування за деревом виводу

Сентенціальна форма	**bab**	**b (L b**	**b (Ma) b**
Дерево виводу			
Основа	**a**	**(L**	**M a)**
Відношення, визначені за деревом	**b <• a** **a •> b**	**b <• (** **($\doteq$ L** **L •> b**	**(<• M** **M $\doteq$ a** **a $\doteq$)** **) •> b**

Наведемо (Таблиця 4.32) матрицю передування, що містить усі відношення передування для даної граматики.

Таблиця 4.32. Матриця передування (Приклад 4.19)

S_i \ S_j	**Z**	**b**	**M**	**L**	**a**	**(**	**)**	**#**
Z								**·>**
b			$\doteq$		**<·**	**<·**		**·>**
M		$\doteq$			$\doteq$			**·>**

L		·>			·>			·>
a		·>			·>		$\overset{\bullet}{=}$	·>
(			<·	$\overset{\bullet}{=}$	<·	<·		·>
)		·>			·>			·>
#	<·	<·	<·	<·	<·	<·	<·	

Елемент матриці B(i, j) містить відношення між символами (S_i, S_j). Порожній елемент говорить про те, що відношення не визначене, тобто символи S_i та S_j не можуть стояти поруч в сентенціальній формі.

Як скористатися відношеннями передування при розборі речення? Якщо між будь-якою парою символів (R, S) визначено більше одного відношення, в них нема користі. Якщо ж між будь-якою парою символів визначено не більше одного відношення, можна знайти основу будь-якої сентенціальної форми за наступним визначенням.

Основою будь-якої сентенціальної форми $\mathbf{S_1 \ldots S_n}$ є найлівіший підланцюжок $\mathbf{S_j \ldots S_i}$, такий, що

$$\mathbf{S_{j-1} <\bullet S_j}$$
$$\mathbf{S_j \overset{\bullet}{=} S_{j+1} \overset{\bullet}{=} \ldots \overset{\bullet}{=} S_{i-1} \overset{\bullet}{=} S_i}$$
$$\mathbf{S_i \bullet> S_{i+1}.}$$

Скористаємося матрицею передування і проведемо граматичний розбір речення b(aa)b за граматикою G (Таблиця 4.33).

В результаті розбору термінальний ланцюжок приведений до аксіоми, тобто розбір завершений успішно.

Таблиця 4.33. Розбір речення b(aa)b

Крок	**Сентенціальна форма**	**Основа**	**Привести основу до**	**Побудований безпосередній вивід**
1	b (a a) b <• <• •> $\overset{\bullet}{=}$ •>	a	M	b(Ma)b $\Rightarrow$ b(aa) b

2	b (M a) b <• <• = = •>	Ma)	L	b(Lb ⇒ b(Ma)b
3	b (L b <• = •>	(L	M	bMb ⇒ b(Lb
4	b M b = =	b Mb	Z	Z ⇒ bMb

4.4.1. Визначення і побудова відношень

Ми визначили відношення передування в термінах синтаксичних дерев виведення сентенціальних форм. Тепер визначимо їх, виключно на основі правил граматики.

Для цього визначимо ще два додаткові відношення — множини FIRST і LAST. Відношення FIRST в словнику V граматики визначається наступним чином:

$S \in FIRST(U)$ тоді і тільки тоді, коли існує правило U ::= S… . Відповідно, $S \in FIRST^{+}(U)$ тоді і тільки тоді, коли існує не порожня послідовність правил $U::=S_1…$, $S_1::=S_2…$, …, $S_n::=S$.

Аналогічно визначимо множину символів, якими закінчуються ланцюжки, що виводяться з деякого символу U. Вона визначається відношеннями LAST і $LAST^{+}$:

$S \in LAST(U)$ тоді і тільки тоді, коли існує правило U::= …S. Відповідно, $S \in LAST^{+}(U)$ тоді і тільки тоді, коли існує не порожня послідовність правил $U::=…S_1$, $S_1::= …S_2$, …, $S_n::=…S$.

Приклад 4.20

Щоб проілюструвати ці відношення випишемо декілька правил (Таблиця 4.34) і вкажемо відношення, котрі з них виводяться (Таблиця 4.35).

Таблиця 4.34. Набір правил

Правила

A→Af
A→B
B→DdC
B→De
C→e
D→B f

Таблиця 4.35. Відношення передування (Приклад 4.20)

FIRST(A)={A,B}	$FIRST^+$(A)={A,B,D}	LAST (A) ={f,B}	$LAST^+$(A)={f,B,C,e}
FIRST(B)={D}	$FIRST^+$ (B) = {D,B}	LAST (B) ={C,e}	$LAST^+$(B) ={C,e}
FIRST(C)={e}	$FIRST^+$ (C) = {e}	LAST (C) ={e}	$LAST^+$(C) ={e}
FIRST(D)={B}	$FIRST^+$ (D) = {B,D}	LAST (D) ={f}	$LAST^+$(D) ={f}

Дамо визначення відношень передування за граматикою.

Якщо задано граматику G, то відношення передування між символами з об'єднаного словника V визначаються таким чином:

1. $R \dot{=} S$ тоді і тільки тоді, коли G містить правило U::=RS…

2. R<•S тоді і тільки тоді, коли існує правило U::=…RV… (тобто $R \dot{=} V$, і V - нетермінал) таке, що справедливо $S \in FIRST+(V)$

3. R•>S тоді і тільки тоді, коли S термінал і існує правило U::=…VS…, таке що справедливе співвідношення $R \in LAST^+(V)$ або існує правило U::=…VW… (тобто $V \dot{=} W$) таке, що справедливі співвідношення $R \in LAST^+(V)$ і $S \in FIRST^+(W)$.

Тепер можна визначити граматику простого передування.

Граматику G називають **граматикою простого передування** або **граматикою (1.1) передування**, якщо

1) між будь-якими двома символами зі словника V визначено не більше одного відношення передування;

2) жодні два правила граматики не мають однакових правих частин.

Запис **(1.1)** означає, що для прийняття рішення про те, чи дійсно ймовірна основа є основою, використовують по одному символу зліва і справа від неї, при цьому дотримання другої умови гарантує те, що основу можна однозначно привести до єдиного нетермінала.

Зауваження. Можуть виникнути труднощі, якщо в основу входить перший (останній) символ сентенціальної форми. Для вирішення цієї проблеми пропонується ввести додатковий символ, що позначатиме границю сентенціальної форми. Таким символом може бути будь-який символ, що не належить граматиці, виберемо символ #. Кожний ланцюжок, що перевіряється, будемо розміщувати між символами # і # (тобто вставимо ці символи на початку ланцюжка і в кінець). Вважатимемо, що #<•S і S•># для будь-якого символу S граматики.

4.4.2. Побудова відношень передування за граматикою

1. Будується матриця відношень $\overset{\bullet}{=}$. Для цього треба переглянути всі праві частини правил граматики і встановити відношення рівності для всіх R і S, що стоять поряд в правих частинах. Для граматики (Приклад 4.19) отримаємо наступну матрицю (Таблиця 4.36).

Таблиця 4.36. Матриця відношень $\overset{\bullet}{=}$ EQ

Z::=bMb
M::=(L | a
L::=Ma)

	Z	**b**	**M**	**L**	**a**	**(**	**)**	**#**
Z								
b			$\overset{\bullet}{=}$					
M		$\overset{\bullet}{=}$			$\overset{\bullet}{=}$			
L								
a							$\overset{\bullet}{=}$	

(				$\overset{\bullet}{=}$				
)								
#								

2. Будується матриця відношень $FIRST^+$ (Таблиця 4.37).

Таблиця 4.37. Матриця F відношень $FIRST^+$

	Z	**b**	**M**	**L**	**a**	**(**	**)**	**#**
Z		**F**						
b								
M					**F**	**F**		
L			**F**		**F**	**F**		
a								
(								
)								
#								

3. Помножити матрицю EQ на F. Таким чином, якщо є рівність $R \overset{\bullet}{=} S$, то встановлюється відношення <• між R і множиною $FIRST^+(S)$ (Таблиця 4.38).

Таблиця 4.38. Матриця відношень <•

	Z	**b**	**M**	**L**	**a**	**(**	**)**	**#**
Z								
b			$\overset{\bullet}{=}$		**<•**	**<•**		
M		$\overset{\bullet}{=}$			$\overset{\bullet}{=}$			
L								
a							$\overset{\bullet}{=}$	
(			**<•**	$\overset{\bullet}{=}$	**<•**	**<•**		

)								
#								

Оскільки $b \dot{=} M$, а $FIRST^{+}(M)=\{a, (\}$, то $b<\bullet a$, $b<\bullet($; відповідно, оскільки $(\dot{=} L$, а $FIRST+(L)= \{M, a, (\}$, то $(<\bullet M$, $(<\bullet a$, $(<\bullet($.

4. Будується матриця відношень $LAST^{+}$ (Таблиця 4.39).

Таблиця 4.39. Матриця L відношень $LAST^{+}$

	Z	**b**	**M**	**L**	**a**	**(**	**)**	**#**
Z		**L**						
b								
M				**L**	**L**		**L**	
L							**L**	
a								
(								
)								
#								

5. Якщо є відношення $R \dot{=} S$ і S — термінал, то встановлюємо відношення $LAST^{+}(R) \bullet> S$. Якщо S — нетермінал, то встановлюємо відношення $LAST^{+}(R) \bullet> FIRST^{+}(S)$. Встановлюємо відношення передування з граничним символом #, і остаточно отримаємо матрицю відношень передування (Таблиця 4.40).

Таблиця 4.40. Відношення передування, визначені за граматикою

	Z	**b**	**M**	**L**	**a**	**(**	**)**	**#**
Z								**·>**
b			$\dot{=}$		**<·**	**<·**		**·>**
M		$\dot{=}$			$\dot{=}$			**·>**

L		·>			·>			·>
a		·>			·>		$\overset{\bullet}{=}$	·>
(			<·	$\overset{\bullet}{=}$	<·	<·		·>
)		·>			·>			·>
#	<·	<·	<·	<·	<·	<·	<·	

Іншими словами, після отримання рівностей діємо таким чином.

1. Обираємо рівності, в котрих другий нетермінал. У нашому випадку це $b \overset{\bullet}{=} M$ і $(\overset{\bullet}{=} L$. Визначаємо множину $FIRST^{+}(M)=\{(, a\}$ і $FIRST^{+}(L)=\{M (, a\}$. З першої рівності отримуємо наступні відношення: **b<•(, b<•a;** з другої: **(<•M, (<•(, (<•a**.
2. Обираємо рівності, в котрих перший нетермінал. У нашому випадку це $M \overset{\bullet}{=} b$ і $M \overset{\bullet}{=} a$. Визначаємо множину $LAST^{+}(M)=\{L, a,)\}$ Отримуємо з першої рівності наступні відношення: **L•>b, a•>b,)•>b;** з другої: **L•>a, a•>a,)•>a.**

4.4.3. Синтаксичний аналізатор для граматики простого передування

Подання відношень та граматики в оперативній пам'яті

Відношення передування зазвичай подаються і зберігаються в пам’яті комп’ютера у вигляді матриці P, елементи якої мають значення:

- $P_{ij} = 0$, якщо S_i і S_j незрівняні, тобто між цими елементами не встановлено відношення передування
- $P_{ij} = 1$, якщо $S_i <\bullet S_j$

- $P_{ij} = 2$, якщо $S_i \dot{=} S_j$
- $P_{ij} = 3$, якщо $S_i \bullet> S_j$

Оскільки в граматиці простого передування між будь-якими двома символами визначено не більше одного відношення, то таке подання можливе.

Для роботи аналізатора по граматиці простого передування граматику зручно реалізовувати у вигляді таблиці, такої щоб по заданій правій частині можна було легко знайти відповідне правило і його ліву частину.

Алгоритм розбору

Для розбору використовується стек. Перед початком розбору в стек заноситься ознака границі ланцюжка #, також цей символ додають в кінець ланцюжка, що перевіряється.

Визначається відношення передування між символом на вершині стека і поточним символом вхідного ланцюжка. Якщо це відношення $\dot{=}$ або <•, символ з вхідного ланцюжка переноситься в стек. Таким чином, символи вхідного ланцюжка обробляються зліва направо і заносяться в стек до тих пір, поки не виявиться, що верхній символ стека знаходиться у відношенні •> до наступного вхідного символу. Це означає, що верхній символ стека є хвостом основи, тобто вся основа вже в стеку. Тоді стек переглядається вглиб у пошуках голови основи, тобто першої пари символів, між якими встановлено відношення <•. Підланцюжок, що знаходиться в стеку між знаками <• і •> і буде основою. Отриману основу знаходять в списку правил, і в стеку вона замінюється тим нетерміналом, до якого вона приводиться. Процес повторюється до тих пір, доки в стеку не опиниться символ, відповідний аксіомі, а наступним вхідним символом буде #.

Приклад 4.21

Проведемо ще раз розбір ланцюжка b(aa)b граматиці Z (Приклад 4.19) по даному алгоритму. Для зручності знову запишемо граматику та матрицю відношень передування (Таблиця 4.41).

Z::=bMb

M::=(L | a

L::=Ma)

Таблиця 4.41. Відношення передування граматики Z

	Z	**b**	**M**	**L**	**a**	**(**	**)**	**#**
Z								·>
b			$\overset{\bullet}{=}$		<·	<·		·>
M		$\overset{\bullet}{=}$			$\overset{\bullet}{=}$			·>
L		·>			·>			·>
a		·>			·>		$\overset{\bullet}{=}$	·>
(			<·	$\overset{\bullet}{=}$	<·	<·		·>
)		·>			·>			·>
#	<·	<·	<·	<·	<·	<·	<·	

Оформимо покроковий розбір у вигляді таблиці (Таблиця 4.42).

Таблиця 4.42. Висхідний розбір ланцюжка b(aa)b

Кроки	**Стек**	**Відношення**	**Вхідний ланцюжок**
	S(1) S(2)...		**R T(k)**
0	#	<•	b (a a) b #
1	# b	<•	(a a) b #
2	# b (	<•	a a) b #
3	# b (a <•	•>	a) b #
4	# b (M	$\overset{\bullet}{=}$	a) b #

Продовження таблиці 2.52

5	# b (M a	$\overset{\bullet}{=}$	) b #
6	# b (M a) $<\bullet \quad \overset{\bullet}{=} \quad \overset{\bullet}{=}$	•>	b#
7	# b (L $<\bullet \quad \overset{\bullet}{=}$	•>	b#
8	# b M	$\overset{\bullet}{=}$	b#
9	# b M b $<\bullet \quad \overset{\bullet}{=} \quad \overset{\bullet}{=}$	•>	#
10	# Z		#

4.4.4. Труднощі, що виникають при побудові граматик простого передування

З теоретичної точки зору метод простого передування здається бездоганним й ефективним. Проте на практиці при застосуванні цього методу можуть виникати деякі проблеми.

Дуже часто між двома символами граматики визначено більше одного відношення, це породжує конфлікт. В цьому випадку необхідно перетворити граматику, щоб обійти конфлікт. Але в результаті подібних перетворень може змінитися вся структура мови.

Конфлікт може виникати при наявності лівосторонньої рекурсії. Якщо існує правило V::=…SU…, і U має ліворекурсивне визначення U::=U…, то одночасно $S \overset{\bullet}{=} U$ і S<•U.

Такого конфлікту можна позбавитися, шляхом введення ще одного нетермінала W і проміжного правила. Тоді послідовність правил

V::=…SU…

U::=U…

замінюємо наступними правилами:

V::=…SW…

W::=U

U::=U…

Після чого отримаємо наступні відношення $S \doteq W$, S<•U. Такий прийом називається **стратифікацією** або **розділенням**.

При правосторонній рекурсії аналогічна ситуація виникає з відношеннями $\doteq$ і •>. Стратифікація не завжди допомагає, адже можуть виникнути й інші конфлікти. Якщо одночасно виникають відношення •> і <•, треба застосовувати іншу техніку розбору.

Приклад 4.22

Розглянемо граматику

<оператор>::=read (<сп.ід.>);

<сп.ід.>::= id |<сп.ід.>, id

Тут ліворекурсивне визначення <сп.ід.>. Конфлікт виникає для пари символів (і <сп.ід.>, а саме ($\doteq$<сп.ід.>і (<•<сп.ід.>. Проведемо стратифікацію:

<оператор>::=read (<сп.1>);

<сп.1> = <сп.ід.>

<сп.ід.>::= id |<сп.ід.>, id

За цією граматикою будуємо матрицю передування (Таблиця 4.43).

Таблиця 4.43. Відношення передування для граматики оператора read

	read	<сп.ід.>	<сп.1>	(	)	id	,	;	#
read				$\doteq$					•>
<сп.ід.>					•>		$\doteq$		•>
<сп.1>					$\doteq$				•>
(		<•	$\doteq$			<•			•>

)								• =	•>
id					•>		•>		•>
,						• =			•>
;									•>
#	<•	<•	<•	<•	<•	<•	<•	<•	

Конфлікт вирішено.

Приклад 4.23

Розглянемо граматику:

<оголошення>::= <сп.ід.> <тип>

<сп.ід.>::= id | <сп.ід.>, id

<тип> ::= real | integer

Тут, як і в попередньому прикладі, наявне ліворекурсивне визначення, однак воно не призводить до конфлікту, оскільки не використовується в інших правилах так, щоб перед описуваним цим правилом нетерміналом (<сп.ід.>) стояв який-небудь символ. Для того, щоб упевнитися в цьому, побудуємо відношення передування для цієї граматики за наведеним в попередній лекції алгоритмом.

1. З правил граматики отримуємо рівності:

<сп.ід.> $\dot{=}$,

, $\dot{=}$ **id**

<сп.ід.> $\dot{=}$ <тип>

2. Вибираємо рівності, що містять другий нетермінал: <сп.ід.> $\dot{=}$ <тип>. Відповідно до правил побудови відношень передування з цієї рівності вихо-

дить, що символ <сп.ід.> знаходиться у відношенні <• до всіх символів, котрі належать множині FIRST$^+$(<тип>). Отримуємо:

<сп.ід.> <• real

<сп.ід.> <• integer

3. Вибираємо рівності, що містять перший нетермінал: <сп.ід.> $\overset{\bullet}{=}$, і <сп.ід.>$\overset{\bullet}{=}$ <тип>. З першої рівності виходить, що всі символи, котрі належать множині LAST$^+$(<сп.ід.>) знаходяться у відношенні •> до символу «,» (тобто **id•>,).**

Другу рівність ми розглядаємо вже вдруге. З неї виходить, що всі символи, котрі належать множині символів LAST$^+$(<сп.ід.>) знаходяться у відношенні •> до всіх символів множини FIRST$^+$(<тип>), відповідно до правил побудови відношень передування. Оскільки множина LAST$^+$(<сп.ід.>) містить єдиний символ **id**, а множина FIRST$^+$(<тип>) символи **real** та **integer**, отримаємо наступні відношення: **id •> real; id •> integer**

Таким чином, між будь-якими двома символами даної граматики визначено не більше одного відношення передування. Сформуємо на основі отриманих відношень матрицю передування (Таблиця 4.44).

Таблиця 4.44. Відношення передування для <оголошення>

	<сп.ід.>	<тип>	real	integer	id	,	#
<сп.ід.>		$\overset{\bullet}{=}$	<•	<•		$\overset{\bullet}{=}$	•>
<тип>							•>
real							•>
integer							•>
id			•>	•>		•>	•>
,					$\overset{\bullet}{=}$		•>
#	<•	<•	<•	<•	<•	<•	

Проведемо розбір речення id, id real (таблиця 4.45).

Таблиця 4.45. Розбір фрази id, id real

Крок	**Стек**	**Відношення**	**Вхідний ланцюжок**
0	#	<•	id, id real #
1	# id <•	•>	, id real #
2	# <сп.ід.>	• =	, id real #
3	# <сп.ід.> ,	• =	id real #
4	# <сп.ід.> , id <• • • = =	•>	real #
5	# <сп.ід.>	<•	real #
6	# <сп.ід.> real <•	•>	#
7	# <сп.ід.> <тип> <• • =	•>	#
	# <оголошення>	•>	#

Розбір завершений успішно.

4.4.5. Граматика простого передування для арифметичного виразу

Розглянемо побудову відношень передування для поширеного фрагмента граматик реальних мов – арифметичного виразу, що задається наступною граматикою:

E:: =E+T | T

T:: =T*F | F

F::=(E)| i

В даній граматиці ліворекурсивними є визначення E і T. Конфлікт відношень передування може виникнути в тих місцях, де перед E або T стоять які-небудь символи в правих частинах правил граматики. А саме, в першому правилі, між символами + і T, а також в третьому, між символами (і E. Дійсно, по даній граматиці отримаємо: $+ \dot{=} T$ **та +<•({ FIRST⁺(T)}, звідки +<•T.** Аналогічно: $(\dot{=} E$ та (<•{ FIRST⁺(E)}, звідки (<•E.

Таким чином, потрібна стратифікація. Вводимо додаткові нетермінали E1 і T1. Граматика набуде вигляду:

E1:: =E

E:: =E+T1 | T1

T1:: =T

T:: =T*F | F

F::=(E1)| i

Тепер аксіомою граматики став нетермінал E1.

Побудуємо таблицю передування по отриманій граматиці за описаним раніше алгоритмом.

Серед других символів у відношеннях $\dot{=}$ зустрічаються нетермінали E1, T1, F. Для них будуємо відношення FIRST⁺ і відповідні відношення <•.

1) **FIRST⁺(E1)= {E, T1, T, F (, i}**

З відношення $(\dot{=}$ E1 отримуємо: **(<·E; (<•T1; (<•T; (<•F; (<•(; (<•i.**

2**) FIRST⁺ (T1) = {T, F (, i}**

З відношення $+ \dot{=}$ T1 отримуємо: **+<•T; +<•F; +<• (; +<•i.**

3) **FIRST⁺ (F) = {(, i}**

З відношення $* \dot{=}$ F отримуємо: ***<• (; *<•i.**

Серед перших символів у відношеннях $\overset{\bullet}{=}$ зустрічаються нетермінали E, E1, T. Для них будуємо відношення $LAST^+$ і відповідні відношення •>.

1) **$LAST^+$ (E) = {T1, T, F), i}**

З рівності $E \overset{\bullet}{=} +$ отримуємо: **T1•>+ ; T•>+ ; F•>+ ;)•>+ ; i•>+**

2) **$LAST^+$(E1)= {E, T1, T, F), i}**

З рівності $E1 \overset{\bullet}{=})$ отримуємо: **E•>) ; T1•>) ; T•>) ; F•>) ;) •>) ; i•>).**

3) **$LAST^+$(T)= {F), i}**

З рівності $T \overset{\bullet}{=} *$ отримуємо: **F•>* ;)•>* ; i•>*.**

Всі ці відношення вносимо до матриці передування (Таблиця 4.46).

Таблиця 4.46. Відношення передування для арифметичного виразу.

	E1	E	T1	T	F	i	(	)	+	*	#
E1								$\overset{\bullet}{=}$			·>
E								·>	$\overset{\bullet}{=}$		·>
T1								·>	·>		·>
T								·>	·>	$\overset{\bullet}{=}$	·>
F								·>	·>	·>	·>
i								·>	·>	·>	·>
(	$\overset{\bullet}{=}$	<·	<·	<·	<·	<·	<·				·>
)								·>	·>	·>	·>
+			$\overset{\bullet}{=}$	<·	<·	<·	<·				·>
*					$\overset{\bullet}{=}$	<·	<·				·>
#	<·	<·	<·	<·	<·	<·	<·	<·	<·	<·	

Виконаємо висхідний розбір виразу (i* (i+i)) (Таблиця 4.47).

Таблиця 4.47. Покроковий висхідний розбір виразу (i* (i+i))

Крок	**Стек**	**Відношення**	**Вхідний ланцю-**

Продовження таблиці 4.47

			жок
0	#	<·	(i*(i+i))#
1	# (	<·	i*(i+i))#
2	# (i <•	·>	*(i+i))#
3	# (F <•	·>	*(i+i))#
4	# (T	• =	*(i+i))#
5	# (T *	<·	(i+i))#
6	# (T * (	<·	i+i))#
7	# (T * (i <•	·>	+i))#
8	# (T * (F <•	·>	+i))#
9	# (T * (T <•	·>	+i))#
10	# (T * (T1 <•	·>	+i))#
11	# (T * (E	• =	+i))#
12	# (T * (E +	<·	i))#
13	# (T * (E + i <•	·>	))#
14	# (T * (E + F <•	·>	))#
15	# (T * (E + T <•	·>	))#
16	# (T * (E + T1 • • <• = =	·>	))#
17	# (T * (E <•	·>	))#
18	# (T * (E1 <•	• =	))#
19	# (T * (E1) • • <• = =	·>	)#

Продовження таблиці 4.47

20	# (T * F <• = =	·>	)#
21	# (T <•	·>	)#
22	# (T1 <•	·>	)#
23	# (E <•	·>	)#
24	# (E1 <•	• =	)#
25	# (E1) <• = =	·>	#
26	# F <•	·>	#
27	# T <•	·>	#
28	# T1 <•	·>	#
29	# E <•	·>	#
30	# E1	·>	#

Розбір завершено успішно.

4.5. Завдання для самоконтролю

Завдання 4.1. Перетворити граматику для рекурсивного спуску:

а) F::=TaB|aB|TbB|bB|B

б) <речення> ::= <підмет> <присудок> | <підмет> | <речення> , <підмет> | <речення> , <підмет> <присудок> |(<речення>)

в) <сп.ід.>:=<ід.>,<сп.ід.>|<ід.>,|<сп.ід.>:<ід.>|<ід.>:

г) S::=SSD|SSK|SK|zz|z|zzz

д) A::=KD | KDD | Aab | Aba

е) <вираз> ::= <терм> | <вираз> + <терм> | <вираз> - <терм> | - <терм> | + <терм>

ж) D:=Cba|Cbd|CbD|CDa|CDd

з) A::=ABc|ABk|Ac|Ak|x|y|z

и) <вираз> ::= <терм> | <терм> + <вираз> | <терм> - <вираз> | - <терм> | + <терм>

Завдання 4.2. Побудувати МПА для граматик з завдання 3.1.

Завдання 4.3. Перетворити наступні граматики для рекурсивного спуску і написати відповідні підпрограми синтаксичного аналізатора, вважати, що неописані нетермінали вже мають відповідні функції, котрі повертають значення true, якщо аналіз завершився успіхом

а) <вираз> ::= <терм> | <вираз> + <терм> |<вираз> - <терм> | - <терм> | + <терм>

б) <вираз>::=<терм>|<терм>+<вираз>|<терм>-<вираз>|-<терм>|+<терм>

в) <сп.ід.>:=<ід.>,<сп.ід.>|<ід.>,|<ід.>:<сп.ід.>|<ід.>:

г) <опер>::=read(<сп.ід>)
<сп.ід>::=id|<сп.ід>, id

д) <скл.оп.>::={<сп.оп.>}
<сп.оп.>::=<оп.>|lab:|<сп.оп.>;<оп.>|<сп.оп.>;lab:|<сп.оп.>;< скл.оп.>

е) <опер.>::=read(<сп.ід>)
<сп.ід>::=id,| <сп.ід> id,

ж) <сп.ід.>:=<ід.>,<сп.ід.>|<ід.>,|<сп.ід.>:<ід.>|<ід.>:

з) <скл.оп.>::={<сп.оп.>}

<сп.оп.>::=<оп.>|lab:|<оп.>;<сп.оп.>|lab:<сп.оп.>

и) <сп.ід>::=write(<сп.вив>)

<сп.вив>::=id|const|id,<сп.вив>|const, <сп.вив>

Завдання 4.4. Визначити відношення простого передування для граматик завдання 3.1, 4.3.

Завдання 4.5. Перетворити граматики завдання 3.1, 4.3 на граматики простого передування.

Завдання 4.6. Розробити програмний додаток, що реалізує алгоритмроботи довільного МПА, представленого у вигляді списку переходів. Протестувати програмний продукт на прикладі обробки побудованих за завданням 4.2 магазинних автоматів.

Завдання 4.7. Розробити програмний додаток, що встановлює відношення передування за довільною граматикою.

Завдання 4.8. Розробити синтаксичний аналізатор, що реалізує розбір знизу-вверх для граматик простого передування. Протестувати на прикладі отриманих за завданням 4.5 граматик.

ПРЕДМЕТНИЙ ПОКАЖЧИК

БІБЛІОГРАФІЯ

1. Ахо, А.В. Компиляторы: Принципы, технологии, инструментарий / А.В.Ахо, М.С. Лам, Р. Сети [и др.].– 2-е изд. : пер с анлг. – М.[и др.]: ИД Вильямс, 2010.– 1184 с.

2. Маккиман, У. Генератор компиляторов / У.Маккиман, Дж. Хорнинг, Д. Уортман; пер. с англ. С. М. Круговой;[под.ред. и с предисл. В.М. Савинкова].– М. : Статистика, 1980.– 527 с.

3. Льюис, Ф. Теоретические основы проектирования компиляторов : пер. с англ. / Ф. Льюис, Д. Розенкранц, Р. Стирнз.– М. : Мир, 1979.– 656 с.

4. Хантер, Р. Основные концепции компиляторов : пер. с англ. / Робин Хантер; [зав. ред. А.В.Слепцов].– М. : ИД Вильямс, 2002.– 256 с.

5. Пратт, Т. Языки программирования: разработка и реализация / Т. Пратт, М. Зелковиц.– СПб.: Питер, 2002.– 688 с.

6. Оптимизирующие компиляторы [Электронный ресурс] : практикум / Нижегородский государственный университет им. Н. И. Лобачевского.– Режим доступа: http://www.inf.ethz.ch/personal/rmitin/download/compiler_practice.update19.pdf.

7. Грис, Д. Построения компиляторов для цифровых вычислительных машин / Д. Грис.– М.: Мир, 1975.– 545 с.

8. Зелковиц, М. Принципы разработки программного обеспечения : пер с англ. / М. Зелковиц, А. Шоу., Дж. Гэннон.– М. : Мир, 1982.– 368 с.

9. Гросс, М. Теория формальных грамматик / М. Гросс, А. Лантен; пер. с фр. И.А. Мельничука; [под ред. А.В. Гладкого].–М. : Мир, 1971.– 294 с.

10. Ахо, А. Теория синтаксического анализа, перевода и компиляции. Т. 1: Синтаксический анализ / А. Ахо, Дж. Ульман.– М.: Мир, 1978. – 612 с.

11. Мозговой, М.В. Классика программирования: алгоритмы, языки, автоматы, компиляторы. Практический подход / М.В. Мозговой. — СПб.: Наука и Техника, 2006. — 320 с.

12. Хомский, Н. Три модели описания языка / Н.Хомский // Кибернетический сборник : сборник переводов / [под ред. А.П. Ершова и др.].– М. : Издательство иностранной литературы, 1961.– Т.2.– С.237-266.

13. Backus, J.W. The syntax and semantics of the proposed international algebraic language of Zurich ACM-GAMM Conference / J. W. Backus // Proc. International Conf. of Information Processing. – UNESCO, 1959.– P. 125-132.

14. Серебряков, В.А. Основы конструирования компиляторов / В.А. Серебряков, М.П. Галочкин.– М. : Едиториал УРСС, 2001.– 224 с.

15. Хопкрофт, Д. Введение в теорию автоматов, языков и вычислений / Д. Хопкрофт, Р. Мотвани , Дж. Ульман.– 2-е изд.: пер. с англ. - М.: Издательский дом "Вильямс", 2002. - 528 с.

16. Пентус, А.Е. Теория формальных языков / А.Е. Пентус, М.Р. Пентус.– М.: Изд-во ЦПИ при механико-математическом ф-те МГУ, 2004. — 80 с.

17. Глушков, В. М. Синтез цифровых автоматов / В.М. Глушков.– М. : Физматгиз, 1962.– 474 с.

18. Братчиков, И.Л. Синтаксис языков программирования / И. Л. Братчиков.– М. : Наука, 1975.– 232с.

19. Ginzburg, A. Algebraic Theory of Automata / A. Ginzburg.– New York : Academic Press, 1968. – 173 p.

20. Фридл, Дж. Регулярные выражения / Дж. Фридл. – 3-е изд. : пер. с англ – СПб.: Символ-Плюс, 2008.– 608 с.

21. Гойвертс, Я. Регулярные выражения. Сборник рецептов / Ян Гойвертс, Стивен Левитан; пер. с англ. А. Киселев.– М. : Символ-Плюс, 2010.– 608 с.

22. Фостер, Дж. Автоматический синтаксический анализ / Дж. Фостер.– М. : Мир, 1975.– 71 с.

23. Wirth, N. Compiler Construction / N. Wirth.– Zurich : Addison-Wesley Pub, 2005.– 176 p.

24. Зубенко, В.В. Програмування / В.В. Зубенко, Л.Л. Омельчук.– К. : ВПЦ "Київський університет", 2011.– 623 с.

25. Гинзбург, С. Математическая теория контекстно-свободных языков / С. Гинзбург.– М.: Мир, 1970. – 326 с.