МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

О. І. Марченко

ОСНОВИ ПРОЕКТУВАННЯ ТРАНСЛЯТОРІВ
КОНСПЕКТ ЛЕКЦІЙ

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра за освітньо-професійною
програмою підготовки бакалаврів спеціальності 123 – «Комп'ютерна інженерія»

Київ
КПІ ім. Ігоря Сікорського
2021

Рецензент *Заболотня Т.М.,* канд. техн. наук, доцент, доцент, КПІ ім. Ігоря Сікорського

Відповідальний редактор *Орлова М.М.*, ., канд. техн. наук, доцент, доцент

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол № 2 від 09.12.2021 р.) за поданням Вченої ради факультету прикладної математики (протокол № 2 від 27.09.2021 р.)

Електронне мережне навчальне видання

Марченко Олександр Іванович, канд. техн. наук, доцент

ОСНОВИ ПРОЕКТУВАННЯ ТРАНСЛЯТОРІВ
КОНСПЕКТ ЛЕКЦІЙ

Основи проектування трансляторів: Конспект лекцій : [Електронний ресурс] : навч. посіб. для студ. спеціальності 123 – «Комп’ютерна інженерія» / О. І. Марченко ; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 2,74 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2021. – 108 с.

Навчальний посібник розроблено для ознайомлення студентів з теоретичним матеріалом дисципліни «Основи проектування трансляторів», Конспект лекцій містить повну теоретичну та практично-орієнтовану інформацію, необхідну для розробки всіх основних частин трансляторів: лексичних аналізаторів, синтаксичних аналізаторів, семантичних аналізаторів та генераторів вихідного коду. У навчальному посібнику розглядаються такі важливі для розробки трансляторів теми, як класифікація формальних мов Хомського, зв’язок та відповідність різних типів формальних мов до певних класів абстрактних автоматів, важливі для практичного застосування при розробці трансляторів підкласи формальних граматик та мов, практичні аспекти побудови автоматів лексичного та синтаксичного аналізу, а також семантичний аналіз та генерація коду для всіх основних конструкцій сучасних мов програмування.

Навчальний посібник призначений для студентів очної фор-ми навчання за спеціальністю 123 – «Комп’ютерна інженерія» факультету прикладної математики КПІ ім. Ігоря Сікорського.

ЗМІСТ

ВСТУП

Основні визначення

Транслятор – це програма чи комплекс програм, що здійснюють переклад тексту, написаного однією мовою програмування (вхідна мова), в текст, поданий іншою мовою (вихідна мова).

Розрізняють транслятори двох видів:

1) компілюючого типу;
2) інтерпретуючого типу;

Компілятор – це транслятор, для якого вхідною мовою є мова високого рівня (наприклад C, Pascal, Algol, та інші), а вихідною – мова асемблера чи мова машинних команд. До того ж переклад вхідної програми вихідною мовою виконується одразу цілком.

Асемблер – це компілятор, в якому вхідною мовою є мова асемблера, а вихідною – мова машинних команд.

Варто зауважити, що вхідною та вихідною програмами компілятора завжди є текст.

Інтерпретатор – це транслятор, що здійснює пооператорний переклад тексту програми вихідною мовою з одночасним виконанням цих операторів.

На виході інтерпретатора отримуємо результат роботи вхідної програми.

Структура транслятора (компілятора)

Загальна структура транслятора (компілятора) показана на рис.1.

Лексичний аналізатор **(*scanner, сканер*)** здійснює перетворення вхідного тексту програми, що подана рядком символів, в рядок лексем, поданий в цифровій формі, а також знаходить лексичні помилки.

Лексема – найменша одиниця інформації, яка обробляється синтаксичним аналізатором.

Приклади лексем:

1) односимвольні роздільники (, ; .), знаки операцій (+, -, *, / тощо), що є односимвольними роздільниками;

2) багатосимвольні роздільники (<=, <>, тощо);

3) ідентифікатори;

4) константи;

5) ключові слова (**for**, **while** та інші).

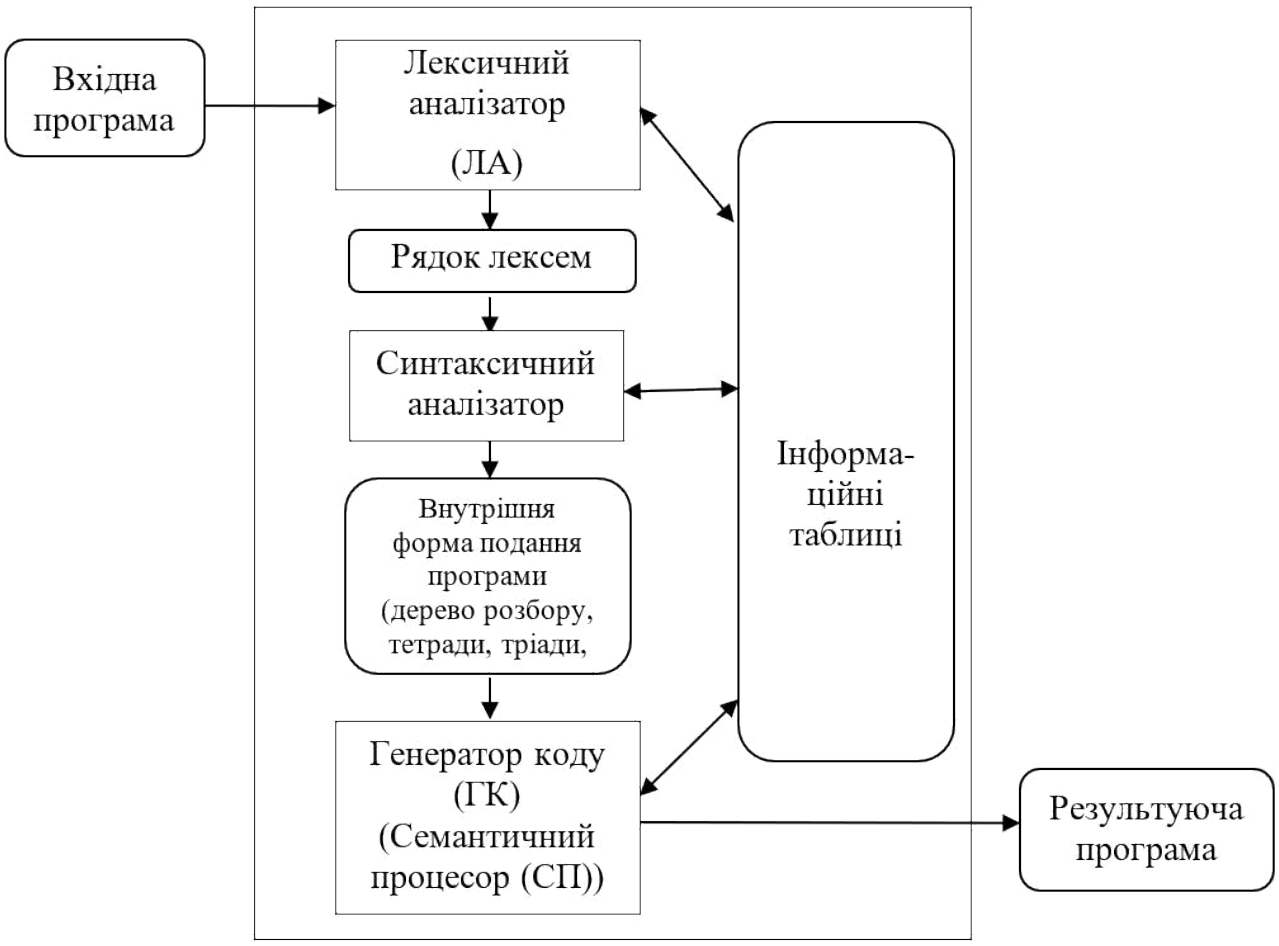


Рис.1. Структура транслятора (компілятора)

Синтаксичний аналізатор (parser, парсер) здійснює декомпозицію вхідної програми, поданої рядком лексем, у структурні одиниці мови (оператори, описи, декларації тощо) згідно граматики вхідної мови, а також знаходить синтаксичні помилки.

Дерево розбору є однією із внутрішніх форм подання вхідної програми, що містить структурні одиниці мови, а також зв'язки між ними.

Генератор коду (***семантичний процесор***) перетворює вхідну програму, подану одною із внутрішніх форм, в текст (оператори, команди) вихідною мовою на основі семантики вхідної мови.

Семантичний процесор – це інша назва генератора коду програми.

Для опису синтаксису і семантики мови використовуються так звані **метамови**, тобто мови, за допомогою яких можна описати інші мови.

Для опису синтаксису використовується метасинтаксична мова.

Для опису семантики використовується метасемантична мова.

Побудова лексичного і синтаксичного аналізаторів базується на теорії формальних граматик.

Для побудови генератора коду використовуються мови та методи опису семантики мов програмування.

Основні поняття теорії формальних граматик

Алфавіт мови (T) – попередньо визначена довільна непуста скінченна множина символів (так званих **термінальних** символів) для побудови речень (рядків) мови.

T^* – множина всіх рядків алфавіту T (тобто тих, що складаються лише із термінальних символів), включаючи порожній рядок .

T^+ – множина всіх рядків в алфавіті T, не включаючи порожній рядок .

$T^* = T^+$.

Позначимо довільну формальну граматику літерою **G**.

Формальна граматика **G** є сукупністю чотирьох понять

$$\mathbf{G = (T, N, P, S),}$$

де T – множина термінальних символів граматики (терміналів), тобто алфавіт мови;

N – множина нетермінальних символів граматики (нетерміналів).

Множина V=T N – це об'єднання множин термінальних і нетермінальних символів.

P – множина правил граматики виду $\rightarrow$,

де V^+, V^*

‘$\rightarrow$ ‘ означає «є за визначенням».

S – аксіома граматики, один із нетермінальних символів граматики (S N), з якого починається опис граматики.

Розглянемо позначення, які будуть використовуватися у конспекті надалі.

Термінали (одиничні символи алфавіту вхідної мови) позначатимемо **маленькими латинськими** літерами.

Нетермінали (одиничні нетермінальні символи мови) позначатимемо **великими латинськими** літерами чи **рядками символів у кутових дужках < і >** (наприклад: <буква>, <оператор>).

Рядки, що складаються в загальному випадку із термінальних і нетермінальних символів, тобто ті, **що належать множині V^+** , позначатимемо **маленькими грецькими літерами**.

Особливі випадки, коли рядок належить тільки T^+ або N^+, будуть зазначені окремо.

ПРИКЛАД ГРАМАТИКИ.

Розглянемо граматику, що породжує цілі двійкові числа, наприклад, 0, 10, 110 та інші.

Множина правил P може бути, наприклад, такою:

1. <рядок> → <двійковий рядок>
2. <двійковий рядок> → <цифра>
3. <двійковий рядок> → <двійковий рядок><цифра> – рекурсивне правило
4. <цифра> → 0
5. <цифра> → 1

T={0,1}

N = {<рядок>, <двійковий рядок >,
<цифра>} S = <рядок>

ВИЗНАЧЕННЯ ТА ВИДИ РЕКУРСИВНИХ ПРАВИЛ

Правило називається **рекурсивним**, якщо один і той самий нетермінальний символ знаходиться як зліва, так і справа від стрілки.

Ліва рекурсія: <X> → <X>

Права рекурсія: <X> → <X>

Центральна рекурсія: <X> → <X>

Множинна рекурсія: <X> → <X><X> тощо

Непряма рекурсія: <X> → <Y>

<Y> → <X>

де і – рядки, що належать V_+ (, V_+)

ПОНЯТТЯ ВИВОДУ (ПОРОДЖЕННЯ)

Розглянемо це поняття на прикладі: написати послідовність виводу (породження) рядка 011 за вказаною вище у цьому підрозділі граматикою цілих двійкових чисел.

<рядок> ^{1} <двійковий рядок> ^{3} <двійковий рядок><цифра> ^{3} <двійковий рядок ><цифра><цифра> ^{2} <цифра><цифра><цифра> ^{4} 0<цифра><цифра> ^{5} 01<цифра> ^{5} 011

Можна використовувати також графічне подання процесу виводу – **дерево розбору (виводу)**, де

корінь дерева – аксіома граматики;

вершини (вузли) дерева – нетермінали;

листки – термінальні символи;

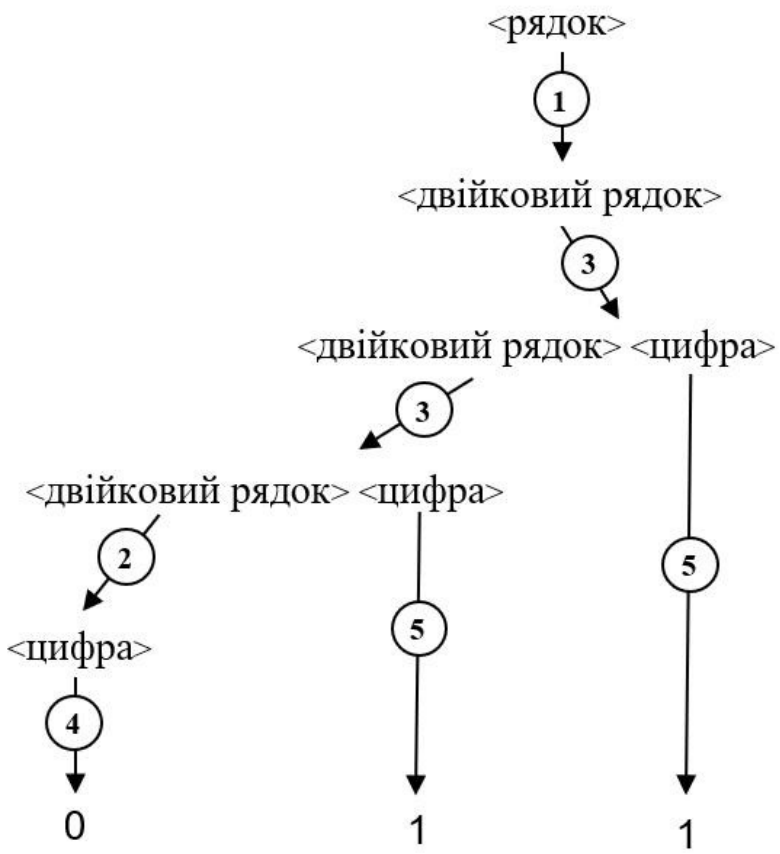


Рис.2. Дерево розбору (виводу) для рядка 011

В загальному випадку вершини (вузли) дерева можуть бути простими (складатися з одного нетермінала) і багатокомпонентними (включати декілька нетерміналів).

ЛЕКСИЧНИЙ АНАЛІЗАТОР (СКАНЕР)

Структура та функції та принцип роботи лексичного аналізатора

Структура лексичного аналізатора показана на рис.3.

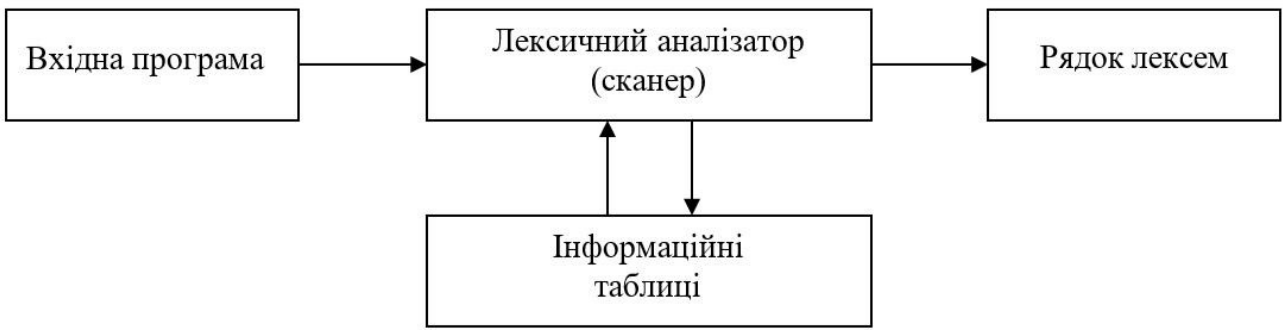


Рис. 3. Структура лексичного аналізатора

Лексичний аналізатор **(*scanner, сканер*)** здійснює перетворення вхідного тексту програми, що подана рядком символів, в рядок лексем, поданий в цифровій формі, а також знаходить лексичні помилки.

Лексема – найменша одиниця інформації, яка обробляється синтаксичним аналізатором.

Приклади лексем:

1) односимвольні роздільники (, ; .), знаки операцій (+, -, *, / тощо), що є односимвольними роздільниками;

2) багатосимвольні роздільники (<=, <>, тощо);

3) ідентифікатори;

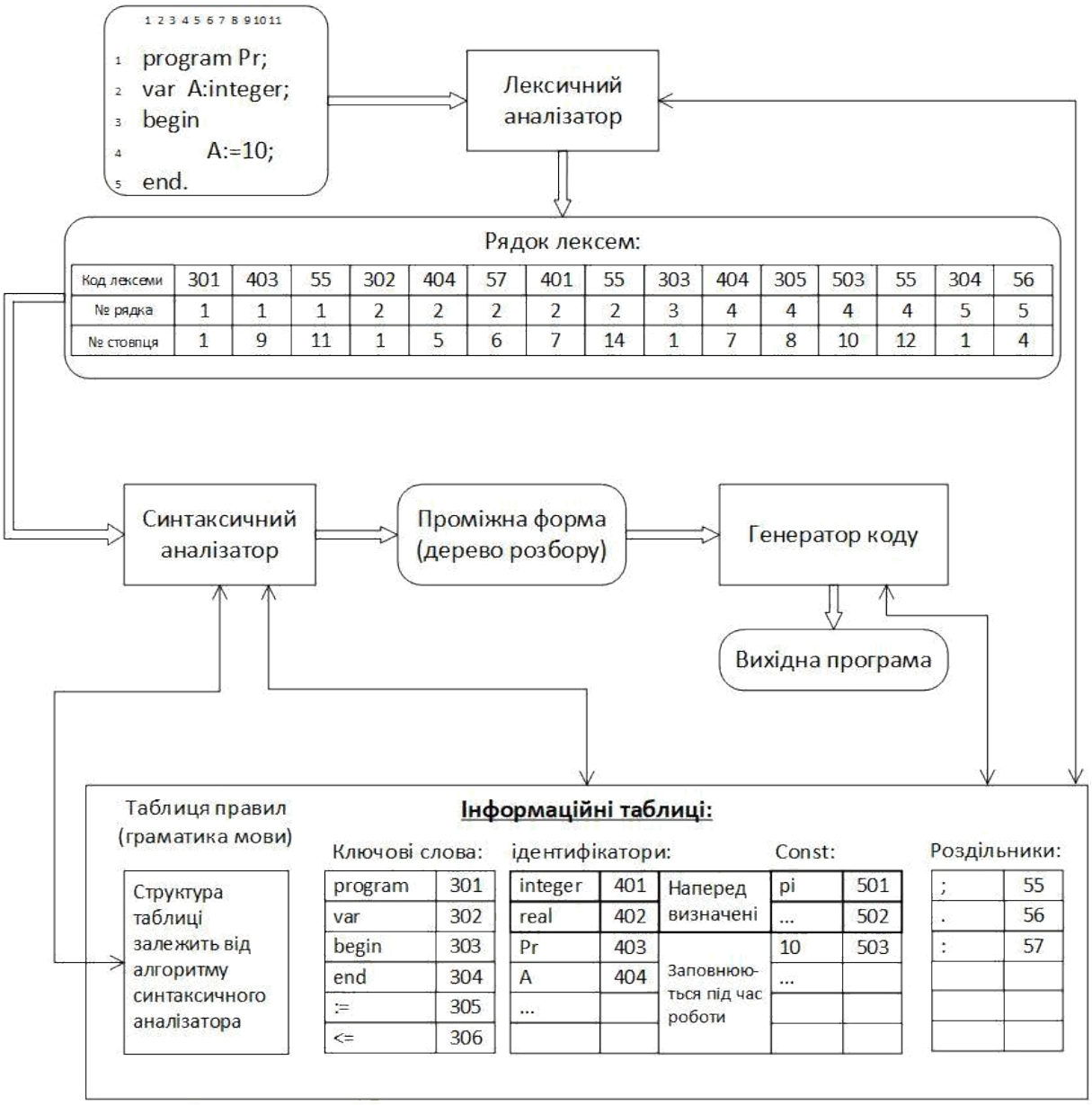
4) константи;

5) ключові слова (**for**, **while** та інші).

Функції лексичного аналізатора:

1. Виділення та згортання лексем (токенів) із заміною їх символьного виду числовими кодами.
2. Видалення коментарів.
3. Побудова інформаційних таблиць (таблиць ідентифікаторів, числових констант, рядкових констант, символьних констант).
4. Виявлення лексичних помилок (як правило виявляються усі помилки, а не тільки перша).

Принцип та схема роботи лексичного аналізатора показані в деталях на рис.4.



Масив атрибутів символів ASCII

Attributes

NUL		BEL	BS		CR		' '		'('		'0'		'9'	':'	';'	'<'		'>'		'A'		'Z'		'a'		'z'		
0		7	8		13		32		40		48		57	58	59	60		62		65		90		97		122		127
6	...	6	0	...	0	...	0	...	5	...	1	...	1	3	3	41	...	42	...	2	...	2	...	2	...	2	...	...	...

0 **Пробільні символи** (whitespace): пробіл (space) – 32; прирівняні до пробілів – 8, 9, 10, 13
1 Символи, з яких можуть починатися **константи**
2 Символи, з яких можуть починатися **ідентифікатори** та **ключові слова**
3 **Односимвольні роздільники**
4 Символи, з яких можуть починатися **багатосимвольні роздільники** (таких категорій може бути декілька)
5 Символи, з яких можуть починатися **коментарі** (таких категорій може бути декілька)
6 **Недопустимі символи**

В реальних трансляторах діапазони кодування категорій беруть великими, наприклад:
0...600 – символи
601...1000 – ключові слова
1001...1 000 000 – ідентифікатори
1 000 001...2 000 000 – константи

Рис. 4. Принцип та детальна схема роботи лексичного аналізатора

Розробка лексичного аналізатора (сканера)

Розглянемо розробку лексичного аналізатора на прикладі.

1. Допустимі лексеми, що виділяються лексичним аналізатором (ЛА) даного прикладу:
   - Ключові слова;
   - цілі десяткові константи;
   - ідентифікатор
   - одиночні роздільники і знаки операцій;

   Крім того, ЛА розпізнає ознаки початку і кінця коментарів (* текст коментаря*) і пропускає текст коментаря без формування лексем.

2. Діаграма переходів (граф) автомату ЛА (рис.5):
   - Позначення станів автомата ЛА:
     - S – початковий стан;
     - INP – стан введення поточного символу програми;
     - CNS – стан виділення константи;
     - IDN – стан виділення ідентифікатора;
     - BCOM – стан визначення символів початку коментаря;
     - COM – стан пропуску (видалення) символів коментаря;
     - ECOM – стан визначення символів кінця коментаря;
     - ERR – стан обробки помилки та видача повідомлення про помилку;
     - OUT – стан виведення лексеми;
     - EXIT – кінцевий стан.
   - Позначення вхідних символів:
     - dg – цифра (0..9), тобто вхідний символ dg встановлюється, якщо поточний прочитаний символ програми є цифрою;
     - lt – буква (A .. Z, a .. z), тобто вхідний символ lt встановлюється, якщо поточний прочитаний символ програми є буквою;
     - dm – роздільник, тобто вхідний символ dm встановлюється, якщо поточний прочитаний символ програми є роздільником;
     - er – помилковий символ, тобто вхідний символ er встановлюється, якщо поточний прочитаний символ програми є неприпустимим для даної мови (такими, як правило, є більшість керуючих (управляючих)

символів ASCII з діапазону 0..31, а також іноді деякі друковані символи);

- eof – символ кінця файлу.

Зауваження:

- У всіх станах, окрім OUT, виконується введення чергового символу програми і визначення вхідного символу автомата ЛА.
- В стані OUT виконується пошук виділеного ідентифікатора в таблицях ключових слів ідентифікаторів та константи в таблиці констант і внесення їх до цих таблиць (якщо потрібно), виведення коду сформованої лексеми у вихідний масив(файл) лексем і передача наступного символу (якщо він був вже введений) на подальший аналіз.
- В стані ERR лексема не формується, а виводиться повідомлення про помилку.

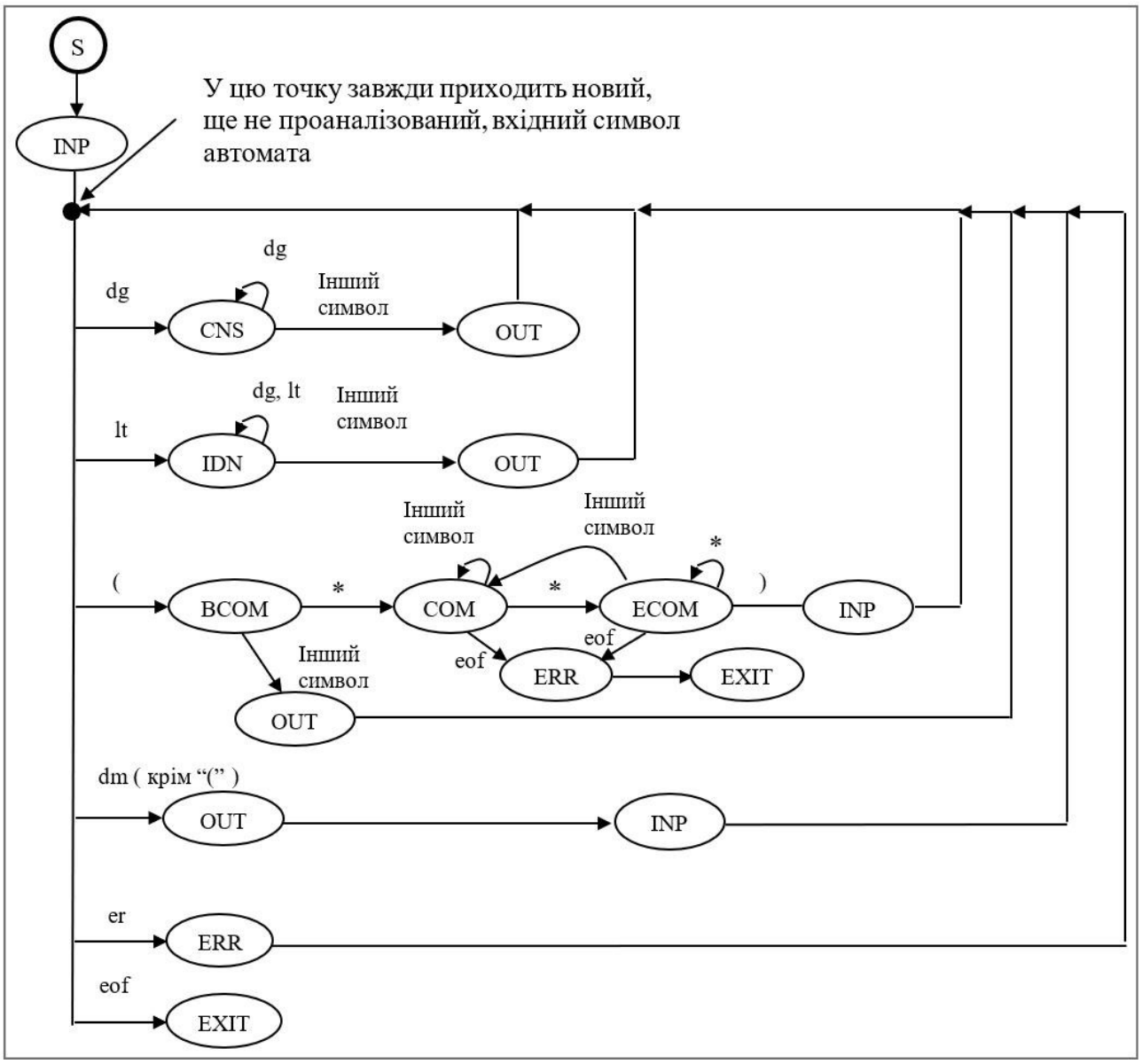


Рис. 5. Граф автомату лексичного аналізатора

3. Опис змінних, використаних в прикладі програми сканера:
    - symbol.value – код ASCII поточного символу;
    - symbol.attr – клас лексем, до якого належить поточний символ symbol.value (кожному символу таблиці ASCII заздалегідь присвоюється атрибут, що визначає лексему (токен), яка може починатися з цього символу):
      (0) пробіл і прирівняні до нього символи (whitespace);
      (1) ціла константа;
      (2) ідентифікатор;
      (3) символ початку коментаря;
      (4) роздільник;
      (5) помилковий символ;
    - buf – буфер для накопичення символів поточної лексеми;
    - lexCode – код чергової лексеми;
    - Attributes – масив значень атрибутів для кожного символу ASCII;
    - FINP – файл початкової програми;
    - SuppressOutput – ознака того, що була виявлена послідовність пробілів або коментар, які не потрібно записувати у вихідний файл.

   Приклад можливого варіанту кодування лексем:
    - діапазон кодів роздільників: 0..255 (ASCII коди);
    - діапазон кодів рядків: 301..400;
    - діапазон кодів констант: 401..500;
    - діапазон кодів ідентифікаторів: 501..600.

4. Функції, використані в прикладі програми сканера:
    - Gets – читає поточний символ з вхідної програми, визначає його атрибут по масиву Attributes і повертає запис symbol;
    - ShowError – обробка помилок, видача повідомлення про помилки;
    - IdnTabForm – формування таблиці ідентифікаторів;
    - ConstTabForm – формування таблиці констант;
    - IdnTabSearch – пошук в таблиці ідентифікаторів;
    - KeyTabSearch – пошук в таблиці ключових слів;
    - ConstTabSearch – пошук в таблиці констант.

5. Структура програми сканера на псевдокоді

```
program Scanner;

type
  TSymbol = record
    value: Char;
    attr: Byte;
  end;

{ Додати опис типів таблиць }

var
  Attributes: array [Char] of Byte;
  symbol: TSymbol;
  lexCode: Word;
  buf: string;
  SuppressOutput: Boolean;
  FINP: TextFile;

{ Додати опис таблиць }

function Gets: TSymbol;
begin
  Read(FINP, Result.value);
  Result.attr :=
Attributes[Result.value]; end;

begin
  (*відкриття файлу початкової програми*)
  (*початкове встановлення таблиць ідентифікаторів і констант*)
  FillAttributes;

  if eof(FINP) then
    ShowError('Empty file');
```

```
repeat
  symbol := Gets;
  buf := '';
  lexCode := 0;
  SuppressOutput := False;

  case symbol.attr of

    0: (*whitespace*)
    begin
      while not eof(FINP)
      do begin
        symbol := Gets;
        if symbol.attr <> 0
          then Break;
      end;
      SuppressOutput := True;
    end;

    1: (*константа*)
    begin
      while not eof(FINP) and (symbol.attr = 1)
      do begin
        buf := buf + symbol.value;
        symbol := Gets;
      end;
      if ConstTabSearch then
        lexCode := <код константи>
      else
      begin
        lexCode := <код наступної константи>
        ConstTabForm;
      end;
    end;
```

```
2: (*ідентифікатор*)
begin
  while not eof(FINP) and ((symbol.attr = 2)
    or (symbol.attr = 1)) do
  begin
    buf := buf + symbol.value;
    symbol := Gets;
  end;

  if KeyTabSearch then
    lexCode := <код ключового слова>
  else
    if IdnTabSearch then
      lexCode :=  <код ідентифікатора>
    else
    begin
      lexCode := <код наступного
      ідентифікатора> IdnTabForm;
    end;
end;

3: (*можливий коментар, тобто зустрінута '(' *)
begin
  if eof(FINP) then
    lexCode := <код відкриваючої дужки>
  else
  begin
    symbol := Gets;
    if symbol.value = '*' then
    begin
      if eof(FINP) then
        ShowError('*) expected but end of file found');
      else
      begin
        symbol := Gets;
```

```
          repeat
            while not eof(FINP) and (symbol.value <> '*')
              do symbol := Gets;
            if eof(FINP) then //якщо кінець
            файла begin
              ShowError('*) expected but end of file found');
              symbol.value = '+'; // все що завгодно, але не
              ')' Break;
            end
            else //була '*' і немає кінця файла
              symbol := Gets;
          until symbol.value = ')';
          if symbol.value = ')' then
            SuppressOutput := True;
          if not eof(FINP) then
            symbol := Gets;
        end;
      end
      else
      begin
        lexCode := <код відкриваючої
      дужки> end;
    end;
  end;
  4: (*роздільник окрім '(' *)
  begin
    symbol := Gets;
    lexCode := <ASCII код односимвольного
  роздільника> end;
  5: (*помилка*)
  begin
    ShowError('Illegal symbol');
    symbol := Gets;
  end;
end;              (*case*)
```

```
    if not SuppressOutput then
      writeln('Output: ',' ',lexCode);
  until eof(FINP);
  Readln;
end.
```

У випадку, якщо в граматиці є багатосимвольні роздільники, то вони заздалегідь вносяться до окремої таблиці і обробляються аналогічно ключовим словам. Якщо багатосимвольний роздільник не розпізнаний, то треба забезпечити повернення у тексті початкової програми до попереднього виділеного символу–роздільника.

ТРАНСЛЯТОРИ ТА ФОРМАЛЬНІ ГРАМАТИКИ

Основні визначення

У прикладах для нижченаведених визначень будемо використовувати граматику для цілого десяткового числа:

1. <число> → <рядок цифр>
2. <рядок цифр> → <цифра>
3. <рядок цифр> → <рядок цифр><цифра>
4. <цифра> → 0
5. <цифра> → 1
6. <цифра> → 2
7. <цифра> → 3
8. <цифра> → 4
9. <цифра> → 5
10. <цифра> → 6
11. <цифра> → 7
12. <цифра> → 8
13. <цифра> → 9

Визначення 1. Рядок **безпосередньо породжує** рядок **( )**,

якщо **= γ1<U> γ 2,** **= γ1 γ2**,

та існує правило **<U> → ,**

де **, , , γ1, γ2 V***, **<U> N**,

або, можна сказати інакше:

рядок **безпосередньо виводиться** з рядка **.**

Приклад. Показати процес породження числа 22 за вказаною граматикою.

Процес породження показаний в таблиці 1.

Таблиця 1.

№ правила		Дія		Лівий контекст γ 1	Правий контекст γ 2
1	<число>		< рядок цифр>		
3	< рядок цифр >		< рядок цифр><цифра>		
2	< рядок цифр><цифра>		<цифра><цифра>		<цифра>
6	<цифра><цифра>		2<цифра>		<цифра>
6	2<цифра>		22	2	

Результуючий ланцюжок виводу матиме такий вигляд:

<число> [1] <рядок цифр> [3] <рядок цифр> <цифра> [2] <цифра><цифра> [6] 2<цифра> [6] 22.

Визначення 2. Рядок **породжує** рядок , якщо існує ланцюжок виводів:

$= \gamma_0 \quad \gamma_1 \quad \gamma_2 \quad \ldots\ldots \quad \gamma_n = .$

Породження позначається + , якщо виводиться з більш ніж за один крок.

Породження позначається * , якщо може бути одна з двох ситуацій: + або .

Наприклад:

1) <число> * 22
2) <число><цифра>* 22
3) <число><цифра>+ 22

Для заданої граматики записи 2) і 3) є однаково коректними.

Визначення 3. Сентенцією або реченням граматики G називається рядок, що складається лише з термінальних символів і виводиться з аксіоми граматики.

Визначення 4. **Сентенціальною формою граматики G** з аксіомою S називається будь-який рядок термінальних та/або нетермінальних символів, що виводиться з аксіоми граматики.

Визначення 5. Мовою L граматики G називається множина всіх сентенцій (речень), які можуть бути породжені граматикою G.

Класифікація мов за Хомським.

Ноам Хомський (Аврам Ноум Хомські (Чомски)) (Avram Noam Chomsky) народився у 1928 році у Філадельфії штат Пенсильванія, у єврейській родині. Його батьки — відомий гебраїст, професор Уільям Хомський (William Chomsky, 1896—1977, народився у містечку Купель Волинської губернії) і Елсі Симоновська (народилась у Бобруйську). Фраза з інтерв'ю Хомського (Noam Chomsky): My father came from the Ukraine... — «Мій батько походить з України».

Класифікація формальних мов була вперше викладена в книзі «Синтаксичні структури» у 1957 році. Згідно з цією класифікацією існує чотири типи формальних граматик і, відповідно, формальних мов.

- **Граматика типу 0** має правила вигляду
  → ,
  де V^+, V^*.
  До даного типу відносяться всі природні мови.

- **Граматика типу 1** має правила виду
  <u> → γ ,
  де , γ, V^*, <u> N, α – лівий контекст, β – правий контекст.

Мови, що породжуються граматиками типу 1, називаються **контекстно-залежними мовами** (КЗ-мовами) або **мовами безпосередніх складових** (БС-мовами).

- **Граматика типу 2** має правила вигляду
  <u> → ,
  де <u> N, V^*.
  Такі граматики називаються **контекстно-вільними** граматиками, що породжують **контекстно-вільні мови** (КВ-мови).

До КВ-мов відносяться практично всі мови програмування. Граматики типу 2 використовується для побудови синтаксичних аналізаторів.

- **Граматика типу 3** має правила тільки одного з двох наступних видів:

  перший: <u> → a

  <u> → a <v>

  або

  другий: <u> → a

  <u> → <v>a

  де a T ,<u> N , <v> N.

Граматики типу 3 також називають **регулярними** або **автоматними** граматиками. Мови, які породжуються такими граматиками, називаються **регулярними мовами**.

Визначення деяких властивостей граматик

1. Еквівалентність граматик.

Граматики називаються **еквівалентними**, якщо вони породжують одну і ту ж саму мову.

Наведені нижче граматики 1 та 2 є еквівалентними:

1. <рядок> → <рядок символів> 2. <рядок символів> → a 3. <рядок символів> → <рядок символів> a	1. <рядок> → <рядок символів> 2. <рядок символів> → <символ> 3. <рядок символів> → <рядок символів> <символ> 4. <символ> → a

2. Однозначність граматики.

Граматика називається однозначною, якщо для будь-якого речення, породжуваного цією граматикою, всі можливі схеми його виводу приводять до одного і того ж дерева виводу.

3. Неоднозначність граматики.

Граматика називається неоднозначною, якщо для одного і того ж речення, породжуваного граматикою, існує декілька неспівпадаючих дерев виводу.

Наприклад, для граматики

1. <аксіома> → <вираз>

2. <вираз> → <вираз>+<вираз>

3. <вираз> → <вираз>*<вираз>

4. <вираз> → a

можливі два різних дерева виводу для рядка **a+a*a**, показані на рис.6.

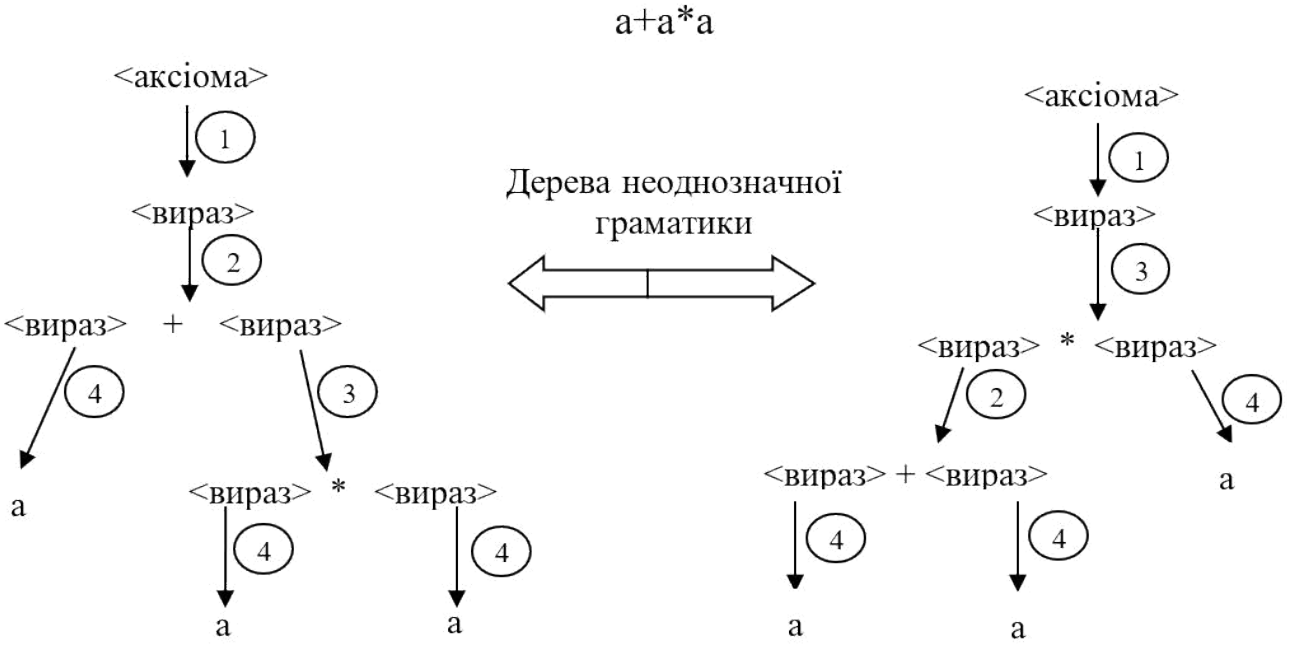


Рис. 6. Дерева розбору для неоднозначної граматики

Якщо цю граматику переписати інакше, то отримаємо однозначну граматику вигляду

1. <аксіома> → <вираз>
2. <вираз> → <терм>
3. <вираз> → <вираз>+<терм>
4. <терм> → <множник>
5. <терм> → <терм>*<множник>
6. <множник> → a

Примітка. *В термінології формальних граматик під терміном «терм» розуміють, як правило, вираз, що складається лише з множників.*

Деякі властивості КВ-граматик

1. *Будь-яку ε-вільну КВ-граматику можна привести до* ***нормальної форми Грейбах*** *з правилами вигляду*:

<x> → b ,

де b T,N, тобто - рядок нетерміналів, можливо порожній.*

2. *Будь-яку ε-вільну КВ-граматику можна привести до* ***нормальної форми Хомського*** *з правилами вигляду:*

1. <*x*> → <*A*><*B*>,

2. <*x*> → *b*,

де <*x*>,<*A*>,<*B*> *N; b T.*

Приклад. Привести до нормальної форми Хомського граматику

1. <U> → <A><b><Z><Y>
2. <b> → b

Введемо правило: <U1> → <A><b><Z>,

тоді: <U> → <U1><Y>.

Введемо правило: <U2> → <A><b>,

тоді: <U1> → <U2><Z>.

Тобто замість граматики

1. <U> → <A><b><Z><Y>
2. <b> → b

отримаємо граматику у нормальній формі Хомського:

1. <U> → <U1><Y>
2. <U1> → <U2><Z>
3. <U2> → <A><b>
4. <b> → b.

3. *Для будь-якої КВ-мови L з граматикою G існує ε-вільна КВ-граматика G′, така що*

L(G′) = L(G) \ { ε }.

СКІНЧЕННІ АВТОМАТИ ТА ГРАМАТИКИ

Формальне визначення скінченного автомату

Скінченний автомат визначається п’ятьма поняттями:

M = (Q, T, q_0, d, F), де

Q – множина всіх станів автомата;

T – множина вхідних символів автомата;

q_0 – початковий стан (q_0 Q) автомата; d

– функція переходів автомата;

F – множина кінцевих станів (F Q) автомата.

Приклад. Розглянемо скінченний автомат, що розпізнає десяткові числа з фіксованою комою зі знаком та без знаку.

Структура числа з фіксованою комою може мати одну із форм, показаних на рис.7.

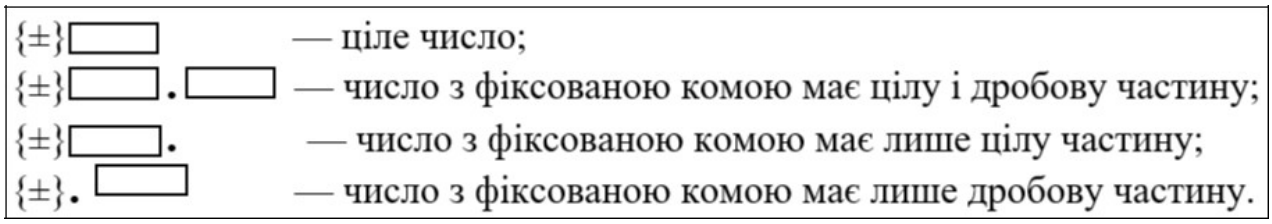


Рис. 7. Структура числа з фіксованою комою

Граф переходів цього автомату матиме вигляд, показаний на рис.8.

Зауваження. У всіх графах автоматів, що наведені далі у конспекті, прийняті такі позначення:

 – початковий стан;

 – кінцевий стан;

 – стан є одночасно і початковим, і кінцевим,

dg – десяткова цифра (0–9).

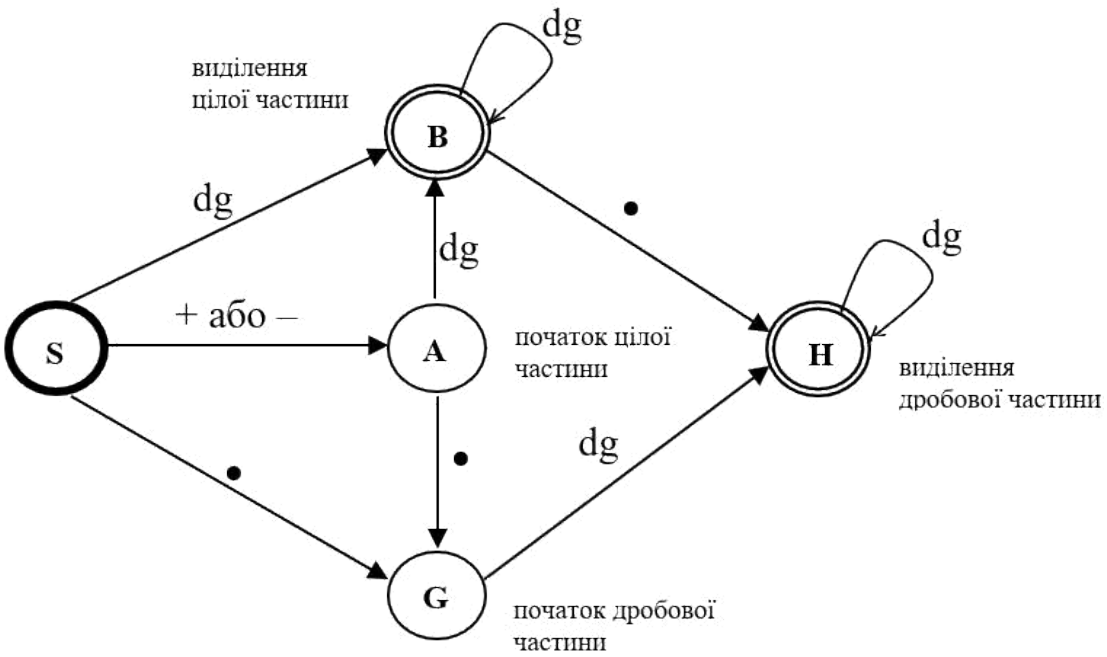


Рис.8. Граф переходів автомату числа з фіксованою комою

Множина станів автомата:

Q={S,A,B,G,H}

Множина вхідних символів автомата:

T = { dg, +, –, . }

Початковий стан автомата:

q0 = S

Функція переходів:

1. d(S, +) = {A}
2. d(S, –) = {A}
3. d(S, dg) = {B}
4. d(S, .) = {G}
5. d(A, dg) = {B}
6. d(A, .) = {G}
7. d(B, dg) = {B}
8. d(B, .) = {H}
9. d(G, dg) = {H}
10. d(H, dg) = {H}

Множина кінцевих станів автомата:

F={B,H}

Приклад детермінованого скінченного автомата

Нехай скінченний автомат задано наступним чином:

$M = (Q, T, q_0, d, F)$,

де $Q = \{A, B\}$ — множина станів автомата,

$q_0 = A$ — початковий стан автомата,

$T = \{0, 1\}$ — множина вхідних сигналів автомата,

$F = \{A\}$ — множина кінцевих станів автомата.

Функція переходів:

d(A, 1) = B

d(A, 0) = A

d(B, 1) = A

d(B, 0) = B

Цей скінченний автомат можна задати також таблицею станів чи графом переходів (рис.9).

Таблиця станів та переходів автомата даного прикладу:

	A	B
0	A	B
1	B	A

Зауваження. В таблицях станів автоматів прийнято такі позначення в заголовку таблиці:

1) початковий стан виділено **жирним шрифтом**;
2) кінцевий стан виділено *курсивним шрифтом*;
3) якщо стан одночасно являється і початковим, і кінцевим, то він виділений ***жирно-курсивним шрифтом***.

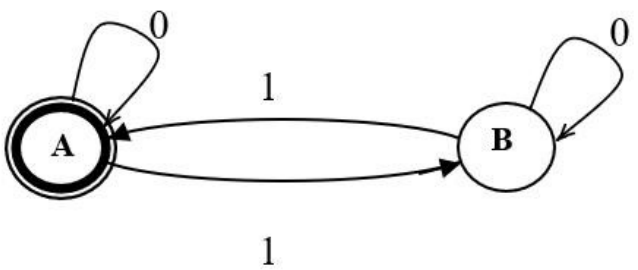


Рис.9. Граф переходів автомата прикладу

Наведений скінченний автомат розпізнає рядки цілих двійкових чисел, у яких кількість одиниць є завжди парною.

Приклади розпізнавання рядків, що поступають на вхід цього скінченного автомату:

Правильний рядок : 1001011	Неправильний рядок : 00111
BBBAABA	A A B A B (немає кінцевого стану)

Скінченний автомат називається детермінованим, якщо завжди існує однозначний перехід автомату із одного стану в інший по деякому символу.

Недетермінований скінченний автомат — це автомат, в якому можливий перехід в різні стани по одному і тому ж символу.

Приклад недетермінованого скінченного автомату та еквівалентного йому детермінованого скінченного автомату

Розглянемо недетермінований автомат із таким описом:

$M_1 = \{Q_1, T_1, q_{01}, d_1, F_1\}$

$Q_1 = \{A, B\}$

$T_1 = \{0,1\}$

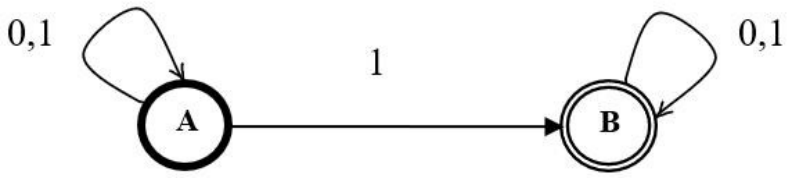
$q_{01} = A$

$F_1 = \{B\}$

Таблиця станів та переходів автомата M1:

	A	*B*
0	A	B
1	{A,B}	B

Граф автомата M2 має наступний вигляд:



Цей автомат визначає довільний рядок нулів та одиниць, який має, як мінімум, одну одиницю і мінімальний рядок складається тільки з однієї одиниці.

Послідовність переходів даного автомату для вхідного рядка 1101 може бути, наприклад, такою:

1. по ‘1’ із стану A відбувається перехід в стан A;
2. по ‘1’ із стану A відбувається перехід в стан B;
3. по ‘0’ із стану B відбувається перехід в стан B;
4. по ‘1’ із стану B відбувається перехід в стан B.

Недетермінований скінченний автомат може призводити до повернень в процесі роботи.

Еквівалентний вищенаведеному детермінований скінченний автомат M2 можна описати таким чином:

$M_2 = \{Q_2, T_2, q_{02}, d_2, F_2\}$

$Q_2 = \{A, B, C\}$

$T_2 = \{0,1\}$

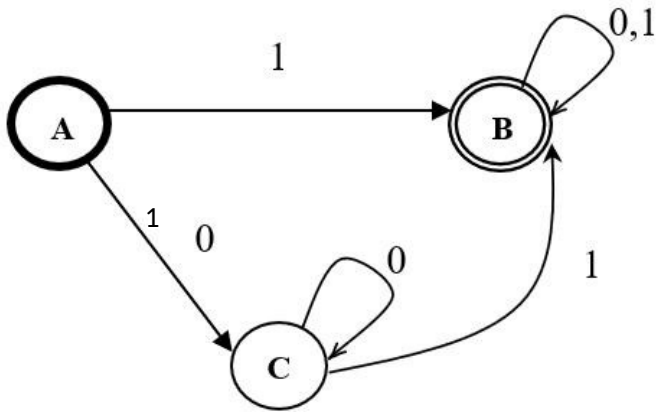
$q_{02} = A$

$F_2 = \{B\}$

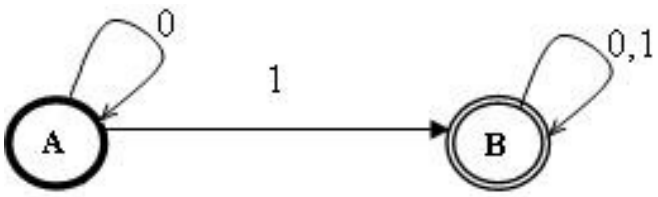
Таблиця станів та переходів автомата M2:

	A	*B*	C
0	C	B	C
1	B	B	B

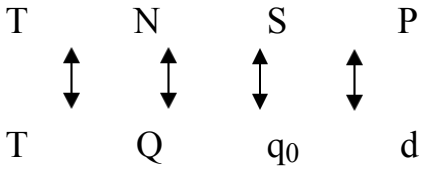
Граф автомата M2 має наступний вигляд:



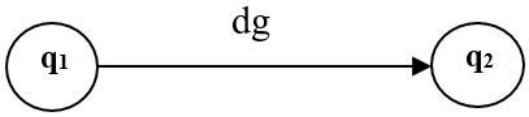
Заданому автомату відповідає також і такий граф



Відповідність між скінченним автоматом і граматикою типу 3

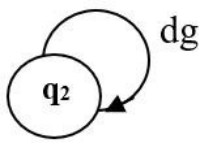


1. Переходу зі стану q1 в стан q2 по символу dg



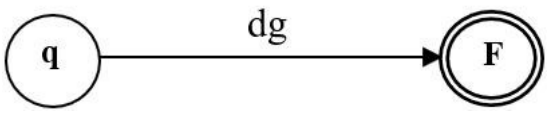
відповідає правило <q1> → dg<q2>.

2. Переходу зі стану q2 в той самий стан q2 по символу dg



відповідає правило <q2>→ dg <q2>.

3. Переходу із стану q в кінцевий стан F



відповідає правило <q>→ dg.

Граматика, що відповідає детермінованому скінченному автомату M2, матиме такий вигляд:

1. <A> → 1<B>	7. <A>→1
2. <A> → 0<C>	8. <B>→0
3. <B> → 0<B>	9. <B>→1
4. <B> → 1<B>	10.<C> → 1
5. C> → 0<C>	
6. <C> → 1<B>	

РЕГУЛЯРНІ ВИРАЗИ

Регулярні вирази базуються на теорії автоматів та теорії формальних мов. **Стівен Кліні (Клейні) (Stephen Kleene)** зробив опис **простих** автоматів (опис моделі нейрона) приблизно в 1940 році за допомогою своєї нової системи математичних позначень, яку він назвав «регулярними множинами» [згадаємо: мова – це також множина].

Ноам Хомський у 1957 році класифікував мови, що описуються математичним апаратом регулярних множин (регулярних граматик), як регулярні мови Типу 3.

Визначення регулярного виразу

Наведемо два визначення регулярного виразу: перше з них – більш строге визначення, а друге визначення орієнтоване на сучасну практику використання цього математичного апарату.

Визначення 1. Регулярний вираз є послідовністю символів (спеціальних математичних позначень), що описує множину рядків (тобто мову). Такі послідовності спеціальних символів використовують для точного описання множини рядків без фактичного перелічення всіх її елементів. *Тобто регулярний вираз є формальним способом (тобто метамовою, як і БНФ) для опису граматики деякої іншої мови.*

Фактично, регулярний вираз R визначає мову L(R), тобто множину всіх теоретично можливих рядків, що відповідають даному формальному опису (виразу R) на метамові Regexp.

Визначення 2. Регулярний вираз – це формальна мова пошуку та маніпуляцій з підрядками в тексті, що базується на використанні системи метасимволів (шаблонів, паттернів, wildcard) (система метасимволів = метамова).

Функція/метод RegExp (R, Text) виділяє/знаходить всі рядки з тексту Text, які належать теоретичній множині рядків (тобто мові), яка визначається конкретним регулярним виразом R (тобто конкретною граматикою, записаною метамовою RegExp).

Широке практичне застосування математичний апарат регулярних виразів отримав завдяки **Кену Томпсону (Kenneth Lane "Ken" Thompson, народився в 1943 р.)**, який вбудував його спершу у текстовий редактор QED (~1969 р.) для операційної системи CTSS (Compatible Time-Sharing System), а потім у редактор ed для ОС Unix (~1971 р.), де застосував цей апарат для знаходження множини підрядків у тексті, модифікувавши цю систему нотації (метамову). Майже всі сучасні програми, які працюють з регулярними виразами, використовують певний варіант нотації Томпсона для регулярних виразів.

Пізніше регулярні вирази були вбудовані (реалізовані) в awk, grep, egrep, lex, flex, Perl, Tcl, PHP, Python та інші.

Конструкція Томпсона

Конструкція Томпсона – це спосіб побудови недетермінованого скінченного автомата (НДСА), який розпізнає мову заданого регулярного виразу.

Ця конструкція придумана Кеном Томпсоном для реалізації регулярних виразів в текстовому редакторі QED для Compatible Time-Sharing System.

Перед тим як застосовувати автомат для перевірки рядків (тексту) на відповідність шаблону (регулярному виразу), цей автомат детермінують та мінімізують.

Наведемо переклад зі сторінки англійської вікіпедії «Thompson's construction» *(https://en.wikipedia.org/wiki/Thompson%27s_construction):*

«**Конструкція Томпсона (Thompson's Construction) є алгоритмом перетворення регулярного виразу у еквівалентний недетермінований скінченний автомат (НДСА).** Цей НДСА може бути використаний для співставлення рядків з регулярним виразом.

Регулярний вираз і НДСА є двома формами представлення формальних мов.

Наприклад, утиліти обробки текстів використовують регулярні вирази для опису шаблонів пошуку, в той час як НДСА краще підходять для виконання на комп'ютері.

Таким чином конструкція (алгоритм) Томпсона має практичний інтерес, оскільки цей алгоритм є алгоритмом компіляції регулярних виразів у НДСА.

З теоретичної точки зору, цей алгоритм є частиною доведення того, що як регулярні вирази, так і НДСА описують в точності ті ж самі мови, які так і називаються регулярними мовами (тип 3 за Хомським).

НДСА може бути детермінізованим (перетвореним у детермінований скінченний автомат - ДСА) і потім може бути мінімізованим з отриманням оптимального автомата, що відповідає даному регулярному виразу. Однак це не є обов'язковим і НДСА може бути реалізованим безпосередньо, без детермінізації і мінімізації.

Для того, щоб з'ясувати, чи описують два заданих регулярних вирази одну й ту ж саму мову, спочатку кожен з них повинен бути перетворений за допомогою конструкції Томпсона у НДСА, потім детермінізований у ДСА і, наостанок, мінімізований. Якщо, і тільки якщо після мінімізації обидва результуючі автомати будуть однаковими з точністю до назв станів, мови, що описуються цими регулярними виразами, будуть також однаковими».

Всі описані вище взаємозв'язки між зазначеними поняттями конструкції Томпсона показані на семі рис.10.

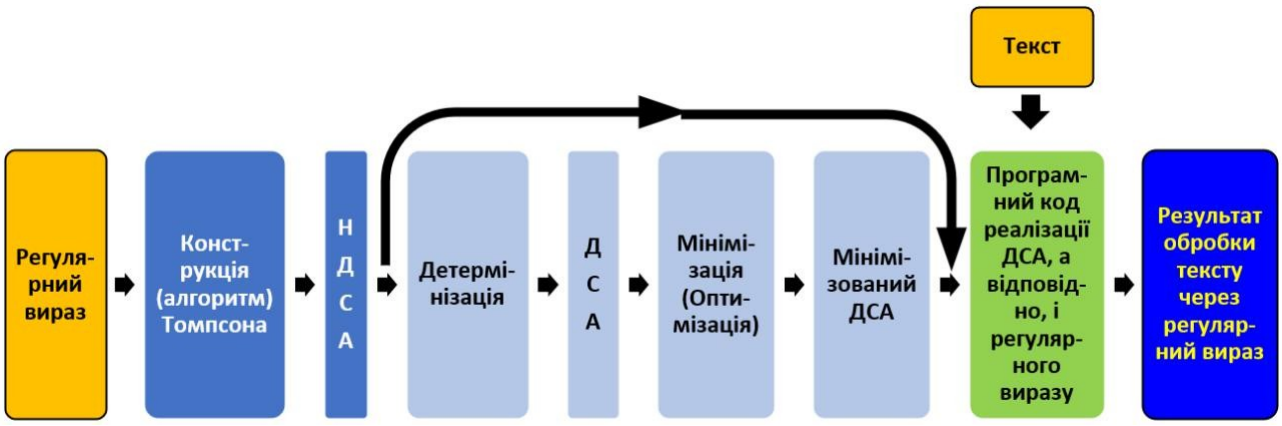


Рис.10. Конструкція Томпсона

Порівняння реалізації синтаксичного аналізатора КВ-мови та реалізації регулярних виразів (конструкції Томпсона) показане на рис. 11.

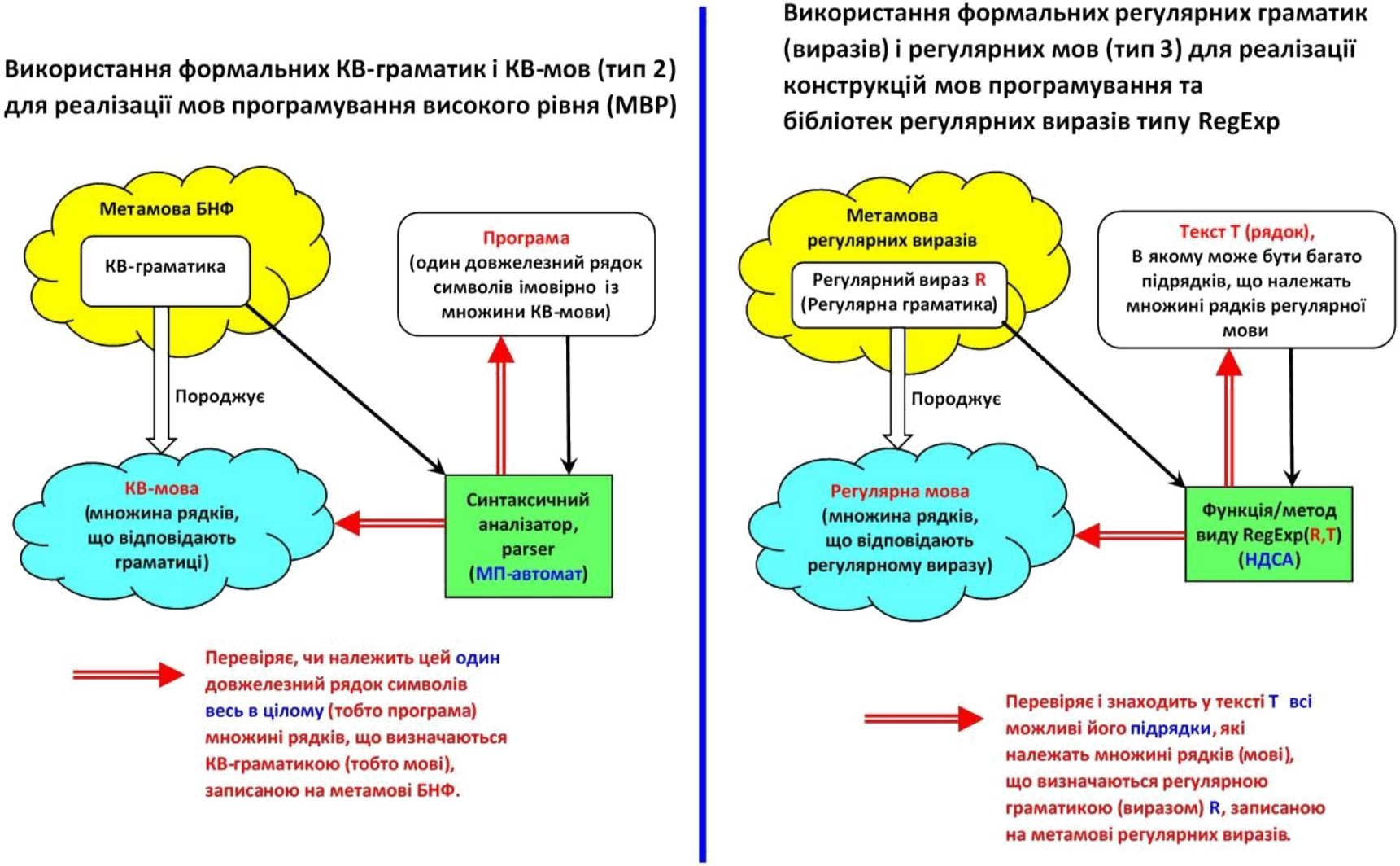


Рис. 11. Порівняння реалізації синтаксичного аналізатора КВ-мови та реалізації регулярних виразів (конструкції Томпсона)

Строге визначення регулярного виразу

Регулярний вираз **R** задає (визначає) мову **L(R)**.

Мови, які можуть бути задані регулярними виразами, називаються регулярними множинами або регулярними (автоматними) мовами.

Регулярний вираз над алфавітом T визначається наступними правилами:

1. є регулярним виразом, що визначає множину **{ ε }**, тобто множину, що містить порожній рядок.

2. Якщо *a* є символом з алфавіту **T,** то цей же символ *a* є простим регулярним виразом, що визначає множину **{a}**, тобто множину, що містить рядок **a**. У конкретних записах суть позначення **“a”** (регулярний вираз, рядок або символ) зрозуміла з контексту.

3. Для регулярних виразів визначені три операції:

   1) альтернатива;

   2) конкатенація;

   3) ітерація.

Якщо A і B – регулярні вирази, що визначають мови **L(A)** і **L(B),** тоді

1) **A | B** (інше позначення **A + B**) є регулярним виразом, що визначає мову (множину) **L(A)** U **L(B).**

2) **AB** є регулярним виразом, що визначає мову (множину) **L(A) L(B)**, тобто після речення мови **A** безпосередньо слідує речення мови **B**.

3) **A*** є регулярним виразом, що визначає мову (множину) **(L(A)) ***.

Прийнято наступні домовленості:

1. Унарний оператор * (зірочка Кліні (Клейні)) має вищий пріоритет і є лівоасоціативним.

2. Конкатенація має другий пріоритет і є лівоасоціативною.

3. Альтернатива (об'єднання) має нижчий пріоритет і є лівоасоціативною.

Алфавіт регулярного виразу складається з наступних елементів:

1) a, b, c – рядки;

2) – порожній рядок;

3) – порожня множина.

Розглянемо позначення, прийняті для регулярних виразів.

Якщо E_1 і E_2 – регулярні вирази, тоді:

1. Альтернатива двох регулярних виразів E_1 і E_2 позначається:

$E = E_1|E_2$ або $E=E_1+E_2$

2. Конкатенація двох регулярних виразів E_1 і E_2 позначається:

$E = E_1E_2$

3. Ітерація регулярного виразу E_1 – це багатократне його повторення від 0 до ∞ (можливо жодного разу) і позначається:

$E = E_1^* = \ | E_1 | E_1E_1 | E_1E_1 \ldots E_1.$

Ітерація регулярного виразу E_1 також може позначатися як $\{E_1\}$, тобто записи E_1^* та $\{E_1\}$ є еквівалентними.

Наприклад, за вищезазначених домовленостей наступні записи є еквівалентними:

(a) | ((b)* (c)) = a | b* c.

Обидва вирази визначають (задають) множину рядків, яка є або єдиним символом a, або декількома символами b (можливо жодного), за якими слідує єдиний символ c.

Визначення. Два регулярні вирази A і B називаються **еквівалентними** (A = B), якщо вони визначають одну і ту ж мову.

Приклади 1.

Розглянемо прості регулярні вирази і мови, що ними визначаються, на алфавіті T = {a, b}.

1. Регулярний вираз a | b задає мову (множину) {a, b}.

2. Регулярний вираз (a | b) (a | b) = aa | ab | ba | bb задає мову (множину) {aa, ab, ba, bb}, тобто множину всіх рядків з a і b довжиною в два символи.

3. Регулярний вираз a* задає мову (множину) всіх рядків з будь-якого числа символів a (можливо жодного), тобто { ε, a, aa, aaa ...}.

4. Регулярний вираз (a | b)* = (a* b*)* задає мову (множину) всіх рядків, що містять декілька екземплярів a і b (можливо жодного), тобто множину всіх рядків, які можна скласти з a і b.

Приклади 2.

Розглянемо приклади регулярних виразів, що визначають конструкції мов програмування.

1. Рядки цілих чисел без знаку (мінімум одна
   цифра): dd*, де d = {0, 1, ..., 9}
2. Цілі із знаком або без знаку (мінімум одна цифра):
   (+ | – | ε)dd*
3. Ідентифікатор:
   1 (1 | d)*, де 1 = {A, B, ..., Z, a, b, ..., z}
   d = {0, 1, ..., 9}

У підрозділі «Регулярні вирази» символ «d» для зручності позначає не перехідну функцію, а множину цифр.

Основні тотожності регулярних виразів

Нехай A, B, C, E – регулярні вирази, тоді мають місце наступні тотожності:

1. A | B = B | A або A + B = B + A – комутативність альтернативи.
2. * =– тобто ітерацією порожньої множини є порожній рядок.
3. A|(B|C)=(A|B)|CабоA+(B+C)=(A+B)+C –асоціативність альтернативи.
4. A (B C) = (A B) C – асоціативність конкатенації.
5. A(B|C)=AB|ACабоA(B+C)=AB+AC –дистрибутивність конкатенації над альтернативою.
6. E = E = E – є «одиничним» елементом по відношенню до конкатенації.
7. E + = E– є «нульовим» елементом по відношенню до альтернативи.
8. (E|)*=E*.
9. E|E*=E*.
10. E** = E*.
11. ЯкщоE=AE|B,тоE=A*B.

Перетворення регулярних виразів до скінченного автомата (Конструкція Томпсона)

Нехай

E, E_1, E_2 – регулярні вирази;

– порожній рядок;

a – рядок регулярного виразу; S

– початковий стан автомата; F –

завершальний стан автомата.

Тоді перехід від регулярного виразу до кінцевого автомата буде згідно правил, показаних на рис. 12.

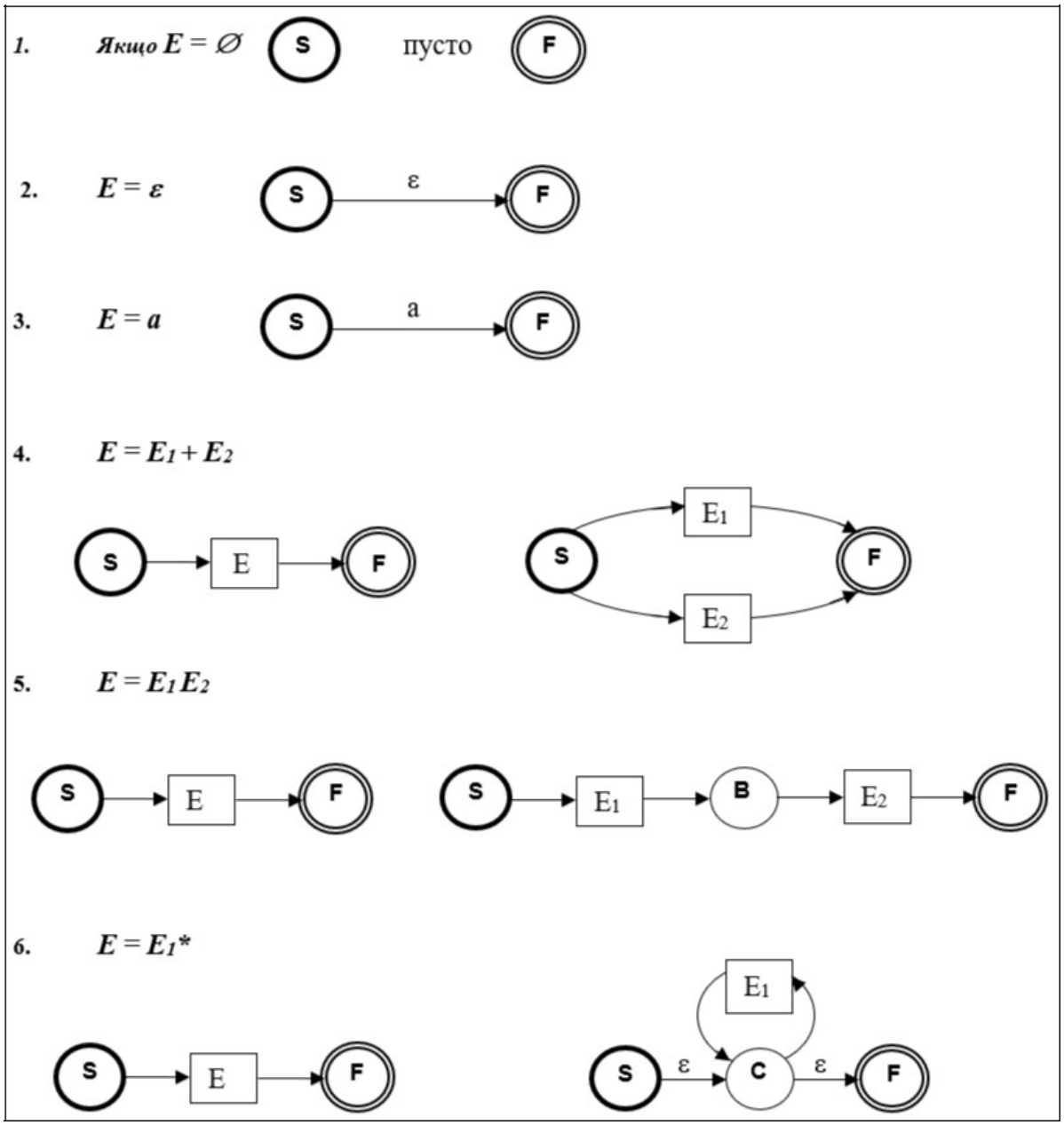


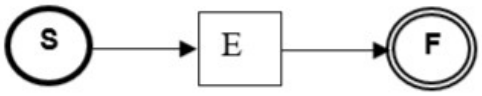
Рис.12. Правила переходу від регулярного виразу до кінцевого автомата

Приклад.

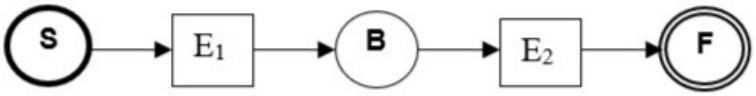
Побудуємо автомат для регулярного виразу E = (+ | – |) d d*.

Побудова автомата за цим регулярним виразом показана на рис.13.

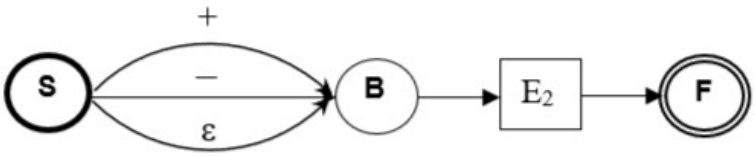
0. Позначимо частину (+ | – | ε) регулярного виразу як E_1, а частину d d* – як E_2, тобто
E_1 = (+ | – | ε),
E_2 = d d*.



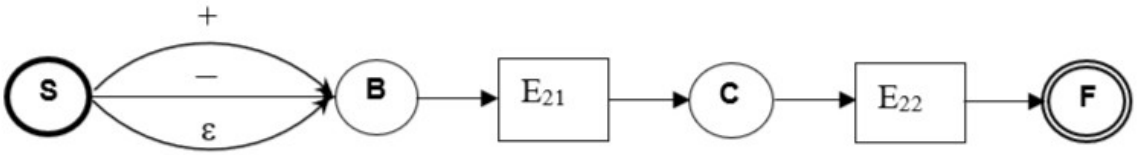
1.



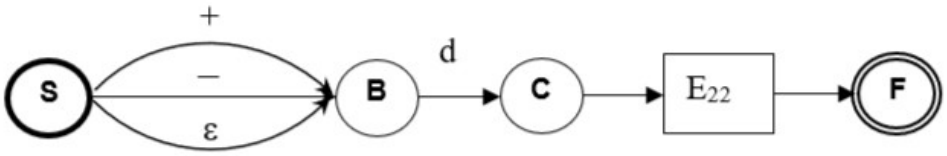
2.



3. Позначимо частину **d** регулярного виразу E_2 як E_{21}, а частину d* – як E_{22}, тобто
E_{21} = **d**,
E_{22} = **d***.



4.



5.

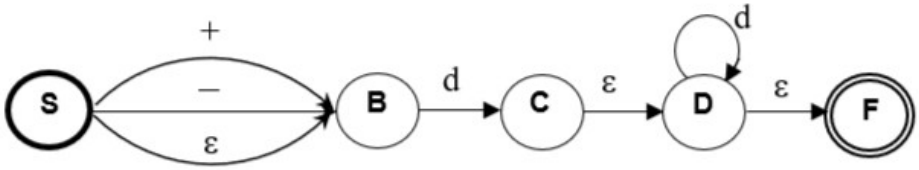


Рис. 13. Процес побудови автомату регулярного виразу E = (+ | – |) d d*

Регулярні визначення

Для зручності запису регулярним виразам можна давати імена і використовувати ці імена як символи в інших регулярних виразах.

Сукупність декількох (одного або більше) регулярних виразів, яким були дані імена, називається **регулярним визначенням**, загальний вид якого такий:

$D_1 \rightarrow R_1$

$D_2 \rightarrow R_2$

...

$D_n \rightarrow R_n$,

де D_i – імена регулярних виразів R_i.

Відмітимо, що в регулярних виразах R_i можуть використовуватися вже визначені раніше імена D_i, тобто кожне R_i визначене на множині

$T \cup \{D_1, D_2, \ldots, D_{i-1}\}$

Щоб відрізняти імена регулярних визначень від символів, записуватимемо ці імена великими буквами.

Приклад 1. Регулярне визначення ідентифікатора.

LETTER → A | B | ... | Z | a | b | ... | z

DIGIT→0|1|2|...|9

ID → LETTER (LETTER | DIGIT)*

Приклад 2. Ціле число.

DIGIT→0|1|2|...|9

INTNUM → DIGIT DIGIT*

Додаткові позначення в регулярних виразах

1. Унарний постфіксний оператор + означає "один або більше екземплярів".

Якщо R – регулярний вираз, що визначає мову L(R), то R^+ є регулярним виразом, що визначає мову $(L(R))^+$. Оператор $^+$ має той же пріоритет і асоціативність, що й оператор *.

Мають місце дві тотожності:

1) $R^* = R^+|$

2) $R^+ = RR^*$

Використовуючи цей оператор визначення цілого числа можна записати так:

DIGIT → 0|1|2|…|9

INTNUM → DIGIT +

2. Унарний постфіксний оператор ? означає "один екземпляр або жодного екземпляра".

Позначення R? є скороченим записом R | .

Якщо R – регулярний вираз, то R? – регулярний вираз, що описує мову L(R) U{ }.

3. Класи символів.

Для скороченого запису деякої множини символів можна використовувати позначення у вигляді **класів символів**.

Клас символів [abc] позначає регулярний вираз a | b | c, тобто

[abc] = a | b | c,

де a, b, c – символи алфавіту.

Клас символів [a-z] позначає регулярний вираз a | b | .... | z, тобто

[a-z] = a | b | ... |z,

де a, b … z – символи алфавіту.

Приклади.

Регулярний вираз ідентифікатора з використанням класів символів:

[A-Za-z] [A-Za-z0-9]*

Регулярний вираз цілого числа з використанням класів символів:

[0-9]+

Обмеженість регулярних виразів

Регулярні вирази мають обмежені описові можливості, тому не всі мови можуть бути описані регулярними виразами.

Характерні конструкції, які не можуть бути описані регулярними виразами:

1. Рядки зі збалансованими символами.

Наприклад, рядок, який завжди містить однакове число відкриваючих і закриваючих дужок.

2. Рядки з повторним входженням одного і того ж підрядка.

Наприклад, $\alpha\ \beta\ \alpha$.

Більш того, така конструкція не може бути описана навіть контекстно-вільною граматикою.

3. Рядки, в яких один з символів обчислюється по інших символах того ж рядка. Наприклад, так звані рядки Холлеріта:

$n\ H\ a_1\ a_2\ a_3\ ...\ a_n$,

де кількість символів a_i повинна відповідати десятковому числу n, що стоїть перед символом H.

Висновки:

1. Регулярні вирази можуть використовуватися для опису тільки фіксованої або невизначеної кількості повторень якої-небудь конструкції.

2. Два довільні числа або два довільні підрядки не можуть порівнюватися в контексті регулярних виразів для визначення, однакові вони чи ні.

СИНТАКСИЧНИЙ АНАЛІЗ

Способи синтаксичного розбору

Контекстно-вільні граматики традиційно виступають основою для створення синтаксичних аналізаторів у компіляторах. Опис вхідної мови включає правила її граматики (які також називають синтаксисом мови), відповідно до яких складаються тексти програм. Тому послідовність правил граматики, яка була застосована для написання тексту якоїсь програми, визначає структуру цієї програми.

Варіантів виводу деякого ланцюжка α за граматикою G загалом може бути досить багато. Така ситуація виникає в такому випадку, коли деякий проміжний ланцюжок виводу містить більше одного нетермінального символу, і правила підстановки можуть застосовуватись до них в будь-якій послідовності. Якщо використати ті самі правила, а змінити лише порядок їх застосування, то буде отримано вже інший вивід, проте заключний термінальний ланцюжок та його синтаксична структура залишаться незмінними. Зазвичай множину усіх можливих виводів за КВ-граматикою не використовують, розглядаючи лише, так звані, лівосторонні чи правосторонні виводи або розбори.

Лівосторонній та правосторонній виводи (розбори)

Нехай граматика визначена такими правилами:

1. <оператор> → <змінна>:= <вираз>
2. <вираз> → <терм>
3. <вираз> → <терм> + <вираз>
4. <терм> → <змінна>
5. <терм> → (<вираз>)
6. <змінна> → a
7. <змінна> → b
8. <змінна> → c

Зробимо **лівосторонній** вивід (розбір) для рядка **a:= b + c**:

<оператор> [1] <змінна>:= <вираз> [6] a:= <вираз> [3] a:= <терм> + <вираз> [4] a:= <змінна> + <вираз> [7] a:= b + <вираз> [2] a:= b + <терм> [4] a:= b + <змінна> [8] a:= b + c

Послідовність застосування правил при **лівосторонньому** розборі:
1,6,3,4,7,2,4,8

Тепер для цього ж рядка зробимо **правосторонній** вивід (розбір):

<оператор> [1] <змінна>:= <вираз> [3] <змінна>:= <терм> + <вираз> [2] <змінна>:= <терм> + <терм> [4] <змінна>:= <терм> + <змінна> [8] <змінна>:= <терм> + c [4] <змінна>:= <змінна> + c [7] <змінна>:= b + c [6] a:= b + c

Послідовність застосування правил при **правосторонньому** розборі:
1, 3, 2, 4, 8, 4, 7, 6.

Низхідний та висхідний розбір або аналіз

Різниця між низхідним та висхідним розборами показана на рис.14.

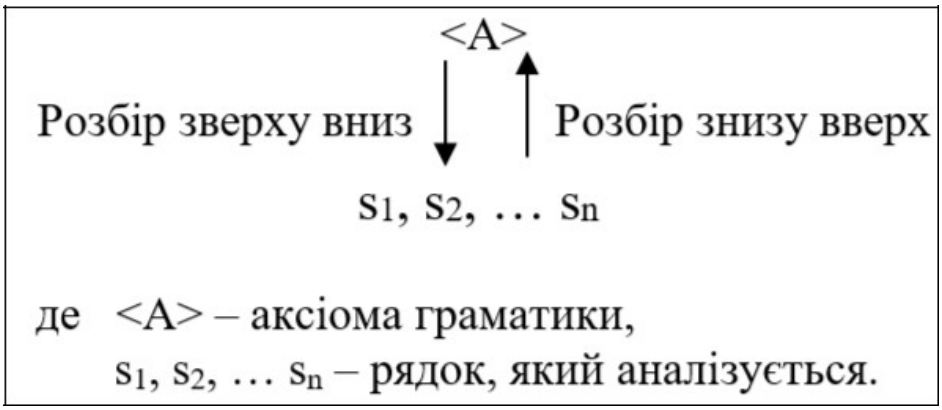


Рис.14. Різниця між низхідним та висхідним розборами

Низхідний розбір (аналіз)

Алгоритми низхідного аналізу важливе обмеження на вид допустимих у граматиці правил: **алгоритми низхідного розбору (аналізу) не працюють з граматиками, які мають ліворекурсивні правила**.

Розглянемо низхідний розбір (аналіз) для рядка **a:=b+c**, що показаний на рис.15.

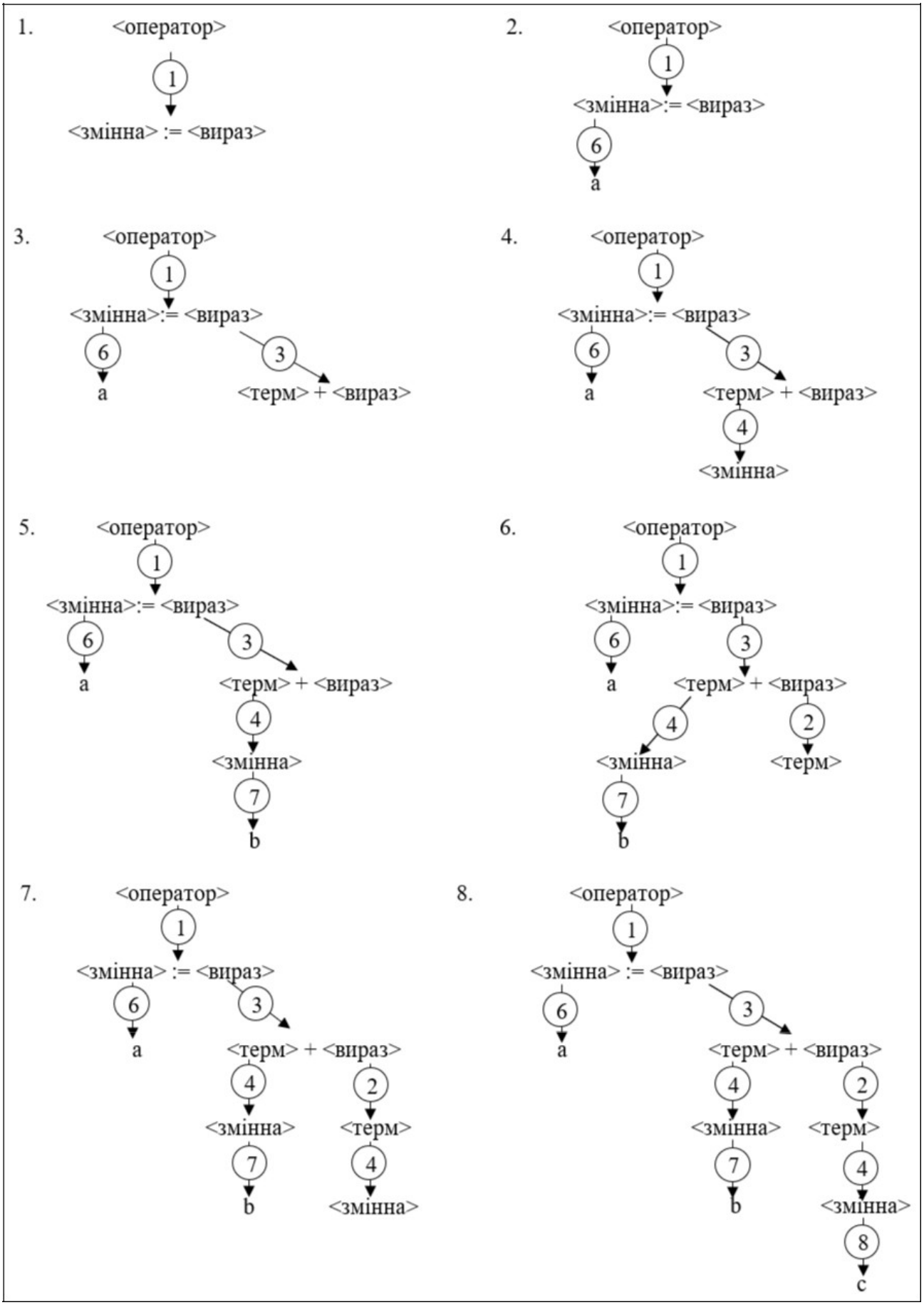


Рис.15. Низхідний розбір (аналіз) для рядка a:=b+c

Висхідний розбір (аналіз)

Висхідний розбір (аналіз) базується на понятті «***основа***».

Якщо деякий рядок терміналів та/або нетерміналів (сентенціальна форма) прямо приводиться до нетерміналу, то він називається **безпосередньо приводимою фразою.**

Найлівіша безпосередньо приводима фраза заданої сентенціальної форми (рядка термінальних та/або нетермінальних символів) називається **основою.**

Суть алгоритму аналізу полягає у пошуку основи й заміни її нетерміналом**,** до якого ця основа безпосередньо приводиться. Процес такої заміни називається **згорткою**. Розбір завершиться тоді, коли нетерміналом, до якого застосовано згортку, стане аксіома граматики.

Розглянемо висхідний розбір (аналіз) рядка a:= b + c за такою граматикою:

1. <оператор> → <змінна>:= <вираз>
2. <вираз> → <терм>
3. <вираз> → <вираз> + <терм>
4. <терм> → <змінна>
5. <терм> → (<вираз>)
6. <змінна> → a
7. <змінна> → b
8. <змінна> → c

Слід зазначити, що в цій граматиці, на відміну від граматики у прикладі низхідного розбору, у правилі 3 використовується ліва рекурсія, а не права. Це зроблено для того, щоб підкреслити, що при висхідному аналізі можна використовувати правила як з правою, так і з лівою рекурсією на відміну від низхідного аналізу.

Розглянемо процес висхідного аналізу (розбору) рядка **a:= b + c**, що показаний на рис.16.

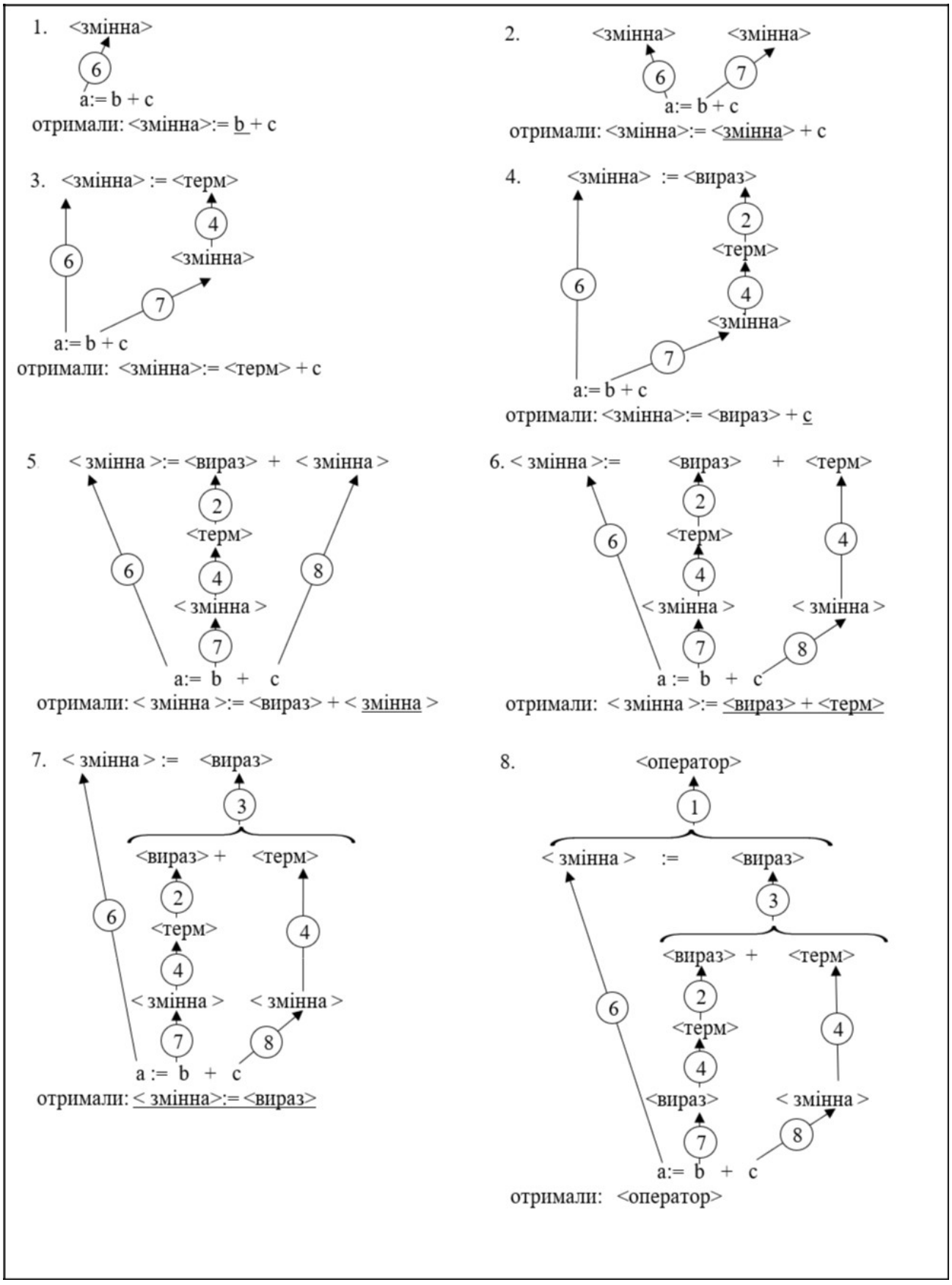


Рис.16. Висхідний розбір (аналіз) рядка a:= b + c

КВ-ГРАМАТИКИ ТА АВТОМАТИ З МАГАЗИННОЮ (СТЕКОВОЮ) ПАМ'ЯТТЮ (МП-АВТОМАТИ)

Визначення МП-автомата.

Склад МП-автомата (рис.17) включає:

1) вхідний рядок, який містить символи вхідного алфавіту;
2) читаючу голівку, яка рухається зліва направо вздовж вхідного рядка;
3) пристрій управління станами;
4) стек (магазин) зі своїм алфавітом.

Рис.17. Схема МП-автомата

Формально МП-автомат M описується наступними сімома поняттями:

M = (Q, T, H, d, q_0, z_0, F) , де

Q – множина станів,

T – множина вхідних символів, H

– множина символів магазину,

d – функція переходів, яка виконує відображення множини пар Q×(T { }) в множину пар Q×H,

q_0 – початковий стан, q_0 Q,

z_0 – граничний маркер або початковий символ магазина, z_0 H, F

– множина заключних станів, F Q .

Функція переходів d описується так:

d(q, a, z_0) ={(p_1, h_1), (p_2, h_2)…}, де

q, p_1, p_2, … – стани автомата;

z_0, h_1, h_2, … – значення у верхівці

магазина; a – вхідний символ.

Конфігурація і такт роботи МП-автомата

Конфігурація МП-автомата задається трійкою понять

(q, w, z), де

q – поточний стан;

w – непрочитана частина вхідного рядка; z

– ланцюжок символів магазину автомата.

Такт роботи МП-автомата визначається переходом автомата від однієї конфігурації до іншої.

Такт записується таким чином:

$(q, aw, hz_1) \vdash (p, w, z_2)$, якщо функція переходу d(q, a, h) містить пару (p, z_2), де

q, p Q;

a T {}; w

T*;

h H;

z_1, z_2 H*.

Позначення:

$\vdash$ – один такт;

$\vdash^+$ – один або більше тактів;

$\vdash^*$ – нуль або більше тактів.

Початкова конфігурація визначається так:

(q_0, w, z_0), де

q_0 – початковий стан;

w – непрочитаний рядок;

z_0 – початковий символ магазину.

МП-автомат завершує роботу при порожньому магазині в заключній (кінцевій) конфігурації.

При завершенні роботи МП-автомату заключні конфігурації можуть бути коректними (рядок розпізнано) і помилковими (рядок не розпізнано).

Коректні заключні конфігурації:

1. (q, , z_0)
2. (q, ,)

Помилкові заключні конфігурації:

1. (q, w,) і немає петлі на кінцевому стані q для продовження розбору.
2. (q, , z), тобто у стеку (магазині) ще є хоча б один символ, крім z_0 (інколи така заключна конфігурація є коректною для деяких автоматів).
3. (p, , z)
4. (p, , z_0)
5. (p, ,),

де q – один з кінцевих станів;

p – довільний не кінцевий стан.

Визначення: МП-автомат M розпізнає мову L, якщо

$$L(M) = \{w \mid (q_0, w, z_0) \vdash^* (q, ,)\}$$

або

$$L(M) = \{w \mid (q_0, w, z_0) \vdash^* (q, , z_0)\},$$

де q – кінцевий стан.

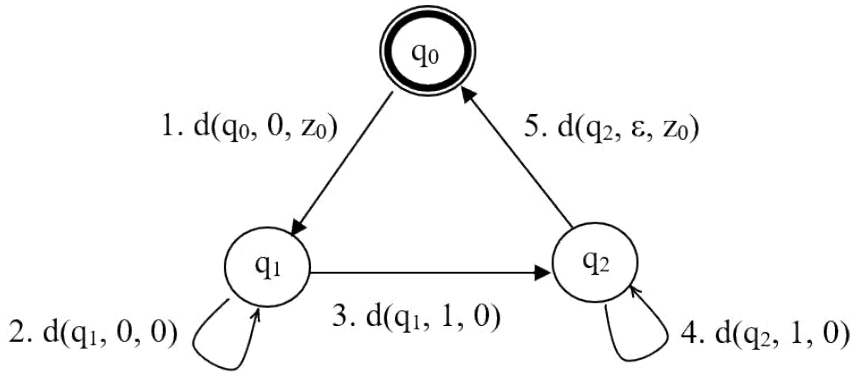
Приклад 1.

Розглянемо МП-автомат, який розпізнає мову $L=\{0^n1^n \mid n \geq 0\}$.

$M=(\{q_0, q_1, q_2\}, \{0, 1\}, \{z_0, 0\}, d, q_0, z_0, \{q_0\})$,

де функція переходів d та граф автомату мають наступний вигляд:

1. $d(q_0, 0, z_0) = \{(q_1, 0z_0)\}$ 2. $d(q_1, 0, 0) = \{(q_1, 00)\}$ 3. $d(q_1, 1, 0) = \{(q_2,)\}$	4. $d(q_2, 1, 0) = \{(q_2,)\}$ 5. $d(q_2, , z_0) = \{(q_0,)\}$



Тоді послідовність тактів для рядка 0011 буде такою:

$(q_0, 0011, z_0) \vdash (q_1, 011, 0z_0)$ по (1)
$(q_1, 011, 0z_0) \vdash (q_1, 11, 00z_0)$ по (2)
$(q_1, 11, 00z_0) \vdash (q_2, 1, 0z_0)$ по (3)
$(q_2, 1, 0z_0) \vdash (q_2, , z_0)$ по (4)
$(q_2, , z_0) \vdash (q_0, ,)$ по (5)

Приклад 2.

Розглянемо недетермінований МП-автомат, який розпізнає мову $L = \{ww^{-1} \mid w \in \{a,b\}^+\}$, де

w – непустий ланцюжок, який складається з {a,b}

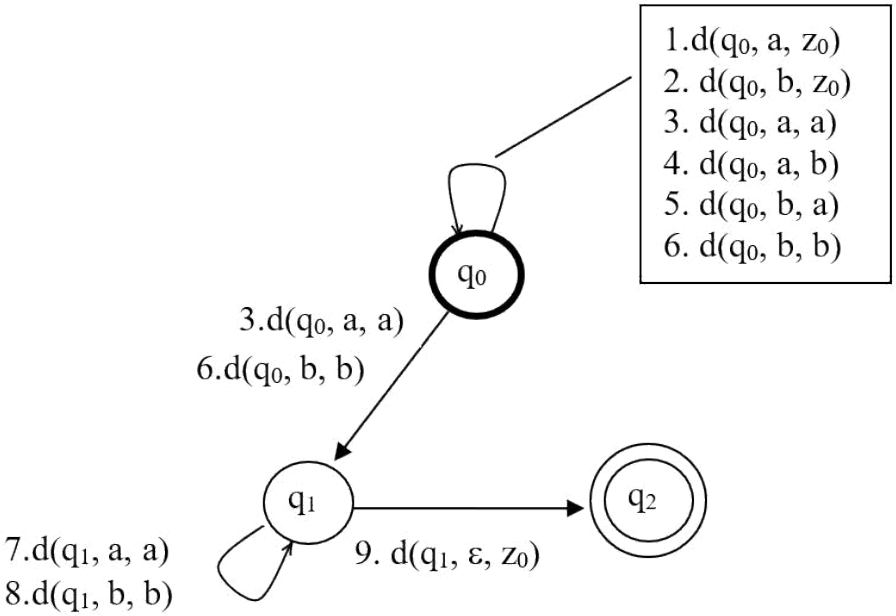
w^{-1} – «дзеркальне відображення» рядка w.

Відповідний автомат має такий вигляд:

$M = (\{q_0, q_1, q_2\}, \{a, b\}, \{z_0, a, b\}, d, q_0, z_0, \{q_2\})$,

де функція переходів d та граф автомата будуть такими:

1. $d(q_0, a, z_0) = \{(q_0, az_0)\}$
2. $d(q_0, b, z_0) = \{(q_0, bz_0)\}$
3. $d(q_0, a, a) = \{(q_0, aa), (q_1,)\}$
4. $d(q_0, a, b) = \{(q_0, ab)\}$
5. $d(q_0, b, a) = \{(q_0, ba)\}$
6. $d(q_0, b, b) = \{(q_0, bb), (q_1,)\}$
7. $d(q_1, a, a) = \{(q_1,)\}$
8. $d(q_1, b, b) = \{(q_1,)\}$
9. $d(q_1, , z_0) = \{(q_2,)\}$



Розглянемо послідовність тактів роботи МП-автомата, який:

1) завершує роботу неправильно (остання конфігурація не є заключною);
2) розпізнає вхідний рядок.

1) $(q_0, abba, z_0) \vdash (q_0, bba, az_0)$ по (1)
$(q_0, bba, az_0) \vdash (q_0, ba, baz_0)$ по (5)
$(q_0, ba, baz_0) \vdash (q_0, a, bbaz_0)$ по (6.1)
$(q_0, a, bbaz0) \vdash (q_0, , abbaz_0)$ по (4)

Вхідний рядок є пустим, а коректний заключний стан не досягнуто. => Виконується повернення до найближчої ще не використаної альтернативи в неоднозначній функції переходів.

Якщо всі можливі альтернативи вже були розглянуті, то вхідний рядок не розпізнано, тобто містить помилку.

2) $(q_0, abba, z_0) \vdash (q_0, bba, az_0)$ по (1)
$(q_0, bba, az0) \vdash (q_0, ba, baz_0)$ по (5)
$(q_0, ba, baz_0) \vdash (q_1, a, az_0)$ по (6.2)
$(q_1, a, az_0) \vdash (q_1, , z_0)$ по (7)
$(q_1, , z_0) \vdash (q_2, ,)$ по (9)

Досягнуто коректний заключний стан – вхідний рядок розпізнано.

Відповідність між КВ-граматикою та МП-автоматом

Якщо КВ-граматика G задана як

$G = (T, N, P, S)$,

а МП-автомат M задано як

$M = (Q, T, H, d, q_0, z_0, F)$,

то відповідність між їх елементами буде такою:

КВ-грамматика G	МП-автомат M
T	T
V=T N	H
P	d
S	q_0

Множина станів Q і заключних станів F МП-автомата не мають прямої відповідності в КВ-граматиці. Часто практично весь вивід по КВ-граматиці виконується з використанням тільки одного стану МП-автомата.

Сформулюємо деякі теореми (без доведення) і визначення.

Визначення 1. Детермінованим МП-автоматом називається такий МП-автомат, будь-якій конфігурації якого відповідає тільки один перехід.

Строге визначення детермінованого МП-автомату є таким:

МП-автомат $M = (Q, T, H, d, q_0, z_0, F)$ є детермінованим, якщо для довільних q Q, a T та A H виконуються наступні умови:

1) d(q, a, A) має не більше одного елемента в множині переходів;
2) d(q, , A) має не більше одного елемента в множині переходів;
3) якщо d(q, , A) ≠ , то d(q, a, A) = для довільного a T.

Третя умова іншими словами означає, що коли з деякої конфігурації автомат може виконати хоча б один -перехід, то цей перехід є єдиним для даної конфігурації.

Визначення 2. Детермінованою контекстно-вільною мовою називають мову, яка приймається детермінованим МП-автоматом.

Теорема 1. Для довільної контекстно-вільної мови існує приймаючий її недетермінований МП-автомат.

Теорема 2. Якщо деяку мову L приймає недетермінований МП-автомат M, то вона може бути прийнятою також і недетермінованим автоматом M’, який має єдиний кінцевий стан і при переході в який (і тільки в цьому випадку) стек МП-автомата стає пустим.

Теорема 3. Якщо деяку мову L приймає недетермінований МП-автомат M, то ця мова L є контекстно-вільною.

Теорема 4. Якщо мова L T* – детермінована, то мова також є детермінованою мовою. L = T* \ L **Теорема 5**. Довільна регулярна мова є детермінованою мовою, тобто для неї

можна побудувати приймаючий її детермінований МП-автомат. Зворотне твердження є невірним, тобто не для всякої детермінованої мови можна побудувати звичайний (не магазинний) детермінований автомат (оскільки детермінована мова може бути також КВ-мовою, а не тільки регулярною).

Теорема 6. Довільна детермінована мова є контекстно-вільною мовою. Зворотне твердження є невірним.

Приклад недетермінованого МП-автомата

Розглянемо граматику G = (N, T, P, S), де:

N={E,T,F}

T = {a, +, *, (,)}

S = E

P={ 1.E→T+E

2. E → T

3. T→F*T

4. T → F

5. F → a

6. F→(E)

}.

Граматика G описує простий вираз з двома операціями + та *. Відповідний до неї недетермінований МП-автомат буде таким:

M = (Q, T, H, d, q_0, z_0, F), де

Q = {q_0, q_1,

q_2}; z_0 = #;

T = {a, +, *, (,)} { };

H = {E, T, F, a, +, *, (,), #}

N T z_0

F = { q_2 }

Функція переходів d, яка визначена наступними допустимими переходами:

1. d(q_0, , #) = { (q_1, E#) }
2. d(q_1, , E) = {(q_1, T+E), (q_1, T)}
3. d(q_1, , T) = { q_1, F*T), (q_1, F) }
4. d(q_1, , F) = { (q_1, a), (q_1, (E)) }
5. d(q_1, a, a) = { (q_1,) }
6. d(q_1, +, +) = { (q_1,) }
7. d(q_1, *, *) = { (q_1,) }
8. d(q_1, (, ( ) = { (q_1,) }
9. d(q_1,),)) = { (q_1,) }
10. d(q_1, , #) = { (q_2,) }

Нехай даний недетермінований МП-автомат виконує лівосторонній вивід наступним чином:

1) при кожному переході автомата з стеку виштовхується один символ;
2) якщо виштовхнутий символ виявляється нетермінальним, то замість нього в стек заносяться символи з множини пар, які є допустимі для даної конфігурації (по черзі зліва направо);
3) якщо виштовхнутий символ виявляється термінальним, то:
   a) якщо він співпадає з поточним вхідним символом, то він видаляється зі стеку, а вказівник вхідного рядка переходить на символ далі;
   b) якщо він не співпадає з поточним символом вхідного рядка, це означає хибний шлях роботи автомата і виконується повернення до найближчої ще нерозглянутої альтернативи;
4) якщо виникає конфігурація, недопустима для визначеної вище функції переходів d, то це також означає хибний шлях роботи автомата і виконується повернення до найближчої ще нерозглянутої альтернативи;
5) якщо виникає недопустима конфігурація, а всі альтернативи вже розглянуті, то даний вхідний рядок не належить мові, визначеній заданою вище граматикою G.

Для реалізації повернень необхідно запам’ятати пройдений шлях станів. При переході з початкового стану автомата q_0 в наступний стан q_1 вміст стеку автомата приймає вигляд E#, а перехід до заключного стану q_2 можливий лише тоді, коли стек пустий (містить тільки символ #) і символи вхідного рядка вже вичерпані. Таким чином, весь процес емуляції лівостороннього виводу виконується автоматом тільки в одному стані q_1.

В таблиці 2 приведена послідовність конфігурацій даного МП-автомата при обробці вхідного рядка **a+a**.

Пояснення 1. Запис $d(q_1, , E) \vdash (q_1, T+E)$ означає перехід за такими правилами:

1) виконується перехід зі стану q_1 в стан q_1 (тобто в той самий стан);
2) вхідним символом є порожній рядок , тобто з вхідного рядка береться символів (вказівник вхідного рядка не переміщується), і перехід виконується незалежно від того, який символ стоїть на початку вхідного рядка;
3) перехід виконується, якщо в вершині стеку знаходиться символ E, при переході символ E із стеку виштовхується;
4) після виштовхування символу E в стек заштовхується символи T+E.

Пояснення 2. Запис $d(q_1, a, a) \vdash (q_1,)$ означає перехід за такими правилами:

1) якщо на початку вхідного рядка стоїть символ **a**, і він же знаходиться у вершині стека, то виконується перехід зі стану q_1 в стан q_1 (тобто в той самий стан);
2) виконується перехід на аналіз наступного після **a** символу вхідного рядка;
3) символ **a** виштовхується зі стеку;
4) в стек заштовхується символів, тобто нічого.

Як видно з таблиці 2, в результаті недетермінованості переходів при роботі МП-автомата відбуваються повернення. Тому такий синтаксичний аналізатор виходить неефективним (повільним).

Задача побудови ефективних синтаксичних аналізаторів загалом є непростою. Проте, при накладанні певних обмежень на граматику, яка породжує деяку КВ-мову, побудова ефективного синтаксичного аналізатора на основі детермінованого МП-автомата може стати можливою.

Таблиця 2.

№ кроку	№ конфігурації	Поточна конфігурація: Стан	Положення покажчика вхідного рядка	Вміст стека	Виконуваний перехід: №	Перехід	Примітки
1.	1	q_0	a + a ↑	#	1	$d(q_0, , \#) \vdash (q_1, E \#)$	
2.	2	q_1	a + a ↑	E #	2.1	$d(q_1, , E) \vdash (q_1, T + E)$	
3.	3	q_1	a + a ↑	T+E#	3.1	$d(q_1, , T) \vdash (q_1, F * T)$	
4.	4	q_1	a + a ↑	F*T+E#	4.1	$d(q_1, , F) \vdash (q_1, a)$	
5.	5	q_1	a + a ↑	a * T + E #	5	$d(q_1, a, a) \vdash (q_1,)$	
6.	6	q_1	a + a ↑	*T+E#	Перехід $d(q_1, +, *)$ недопустимий => повертаємося до найближчої альтернативної конфігурації №4 та беремо перехід 4.2		
7.	4	q_1	a + a ↑	F*T+E#	4.2	$d(q_1, , F) \vdash (q_1, (E))$	
8.	5	q_1	a + a ↑	(E)*T+E#	Перехід $d(q_1, a, ( )$ недопустимий => повертаємося до найближчої альтернативної нерозглянутої конфігурації №3 і беремо перехід 3.2		

Таблиця 2 (продовження).

№ кро-ку	№ кон-фігу-рації	Поточна конфігурація: Стан	Положення покажчика вхідного рядка	Вміст стека	Виконуваний перехід: №	Перехід	Примітки
9.	3	q_1	a + a ↑	T+E#	3.2	d (q_1, , T) ⊢ (q_1, F)	
10.	4	q_1	a + a ↑	F+E#	4.1	d (q_1, , F) ⊢ (q_1, a)	
11.	5	q_1	a + a ↑	a + E #	5	d (q_1, a, a) ⊢ (q_1,)	
12.	6	q_1	a + a ↑	+ E #	6	d (q_1, +, +) ⊢ (q_1,)	
13.	7	q_1	a + a ↑	E #	2.1	d (q_1, , E) ⊢ (q_1, T + E)	
14.	8	q_1	a + a ↑	T+E#	3.1	d (q_1, , T) ⊢ (q_1, F * T)	
15.	9	q_1	a + a ↑	F*T+E#	4.1	d (q_1, , F) ⊢ (q_1, a)	
16.	10	q_1	a + a ↑	a * T + E #	5	d (q_1, a, a) ⊢ (q_1,)	
17.	11	q_1	a + a ↑	*T+E#	Якщо б q_1 було також і завершальним станом, то іноді таку конфігурацію (q_1, , z) вважають коректною для завершення (див. помилкову заключну конфігурацію 2), оскільки, загалом рядок розпізнано як допустимий. Проте дерево виводу було отримано неправильним. Тому,пристрогомутрактуванніця конфігурація є недопустимою для даного автомата і виконується повернення до найближчої нерозглянутої альтернативної конфігурації № 9 і беремо перехід 4.2.		
18.	9	q_1	a + a ↑	F*T+E#	4.2	d (q_1, , F) ⊢ (q_1, (E))	
19.	10	q_1	a + a ↑	(E)*T+E#	Перехід d (q_1, a, ( ) недопустимий => повертаємося до найближчої альтернативної конфігурації № 8 і беремо перехід 3.2		
20.	8	q_1	a + a ↑	T+E#	3.2	d (q_1, , T) ⊢ (q_1, F)	
21.	9	q_1	a + a ↑	F+E#	4.1	d (q_1, , F) ⊢ (q_1, a)	
22.	10	q_1	a + a ↑	a + E #	5	d (q_1, a, a) ⊢ (q_1,)	
23.	11	q_1	a + a ↑	+ E #	Отримали таку ж ситуацію для 11-ої конфігурації, як і для попередньої 11-ї. Див. коментар вище. Перехід d (q_1, , +) недопустимий => повертаємося до найближчої альтернативної нерозглянутої конфігурації № 9 і беремо перехід 4.2		
24.	9	q_1	a + a ↑	F+E#	4.2	d (q_1, , F) ⊢ (q_1, (E))	
25.	10	q_1	a + a ↑	(E)+E#	Перехід d (q_1, a, ( ) недопустимий => повертаємося до найближчої альтернативної нерозглянутої конфігурації № 7 і беремо перехід 2.2		

Таблиця 2 (продовження).

№ кро-ку	№ кон-фігу-рації	Поточна конфігурація			Виконуваний перехід		Примітки
		Стан	Положення покажчика вхідного рядка	Вміст стека	№	Перехід	
26.	7	q_1	a + a ↑	E #	2.2	d (q_1, , E) ▌ (q_1, T)	
27.	8	q_1	a + a ↑	T #	3.1	d (q_1, , T) ▌– (q_1, F * T)	
28.	9	q_1	a + a ↑	F*T#	4.1	d (q_1, , F) ▌– (q_1, a)	
29.	10	q_1	a + a ↑	a * T #	5	d (q_1, a, a) ▌– (q_1,)	
30.	11	q_1	a + a ↑	* T #	Отримали таку ж ситуацію для 11-ої конфігурації, як і для попередньої 11-ї. Див. коментарій вище. Перехід d (q_1, , *) недопустимий => повертаємося до найближчої альтернативної нерозглянутої конфігурації № 9 і беремо перехід 4.2		
31.	9	q_1	a + a ↑	F*T#	4.2	d (q_1, , F) ▌– (q_1, (E))	
32.	10	q_1	a + a ↑	(E)*T#	Перехід d (q_1, a, ( ) недопустимий => повертаємося до найближчої альтернативної нерозглянутої конфігурації № 8 і беремо перехід 3.2		
33.	8	q_1	a + a ↑	T #	3.2	d (q_1, , T) ▌ (q_1, F)	
34.	9	q_1	a + a ↑	F #	4.1	d (q_1, , F) ▌ – (q_1, a)	
35.	10	q_1	a + a ↑	a #	5	d (q_1, a, a) ▌ – (q_1,)	
36.	11	q_1	a + a ↑	#	10	d (q_1, , #) ▌– (q_2,)	
37.	12	q_2			Перейшли в допустиму заключну конфігурацію d (q2, ,) => Вхідний рядок a + a розпізнано з правильним проходом за правилами граматики, при якому є можливість побудови дерева виводу (розбору)		

СИНТАКСИЧНІ АНАЛІЗАТОРИ

Існує три основних типи синтаксичних аналізаторів (англ.: parser), що реалізують три стратегії розбору:

1) стратегія аналізу зверху вниз (низхідний аналіз);
2) стратегія аналізу знизу вверх (висхідний аналіз);
3) змішана стратегія.

Як було показано раніше, загальна схема синтаксичного аналізу на основі недетермінованого МП-автомату працює з поверненнями.

Проте, якщо на граматику, що породжує мову, накласти певні обмеження, а також ефективно використати стек автомата, то можна застосовувати деякі відомі прийоми, що дозволяють побудувати ефективні синтаксичні аналізатори, що працюють без повернень.

На практиці в синтаксичних аналізаторах (СА) існуючих мов програмування використовується саме такий підхід. Зокрема було визначено чотири відносно великих класи КВ-граматик, які дозволяють побудувати СА без повернень:

1. **LL(k)**-граматики;
2. **LR(k)**-граматики;
3. **RL(k)**-граматики;
4. **RR(k)**-граматики.

Перша буква **L** чи **R** у позначенні вказує напрям перегляду вхідного рядка (зліва направо чи справа наліво).

Друга буква (**L** чи **R**) в позначенні означає вид виводу (лівий чи правий відповідно).

Буква **k** в дужках означає кількість символів вхідного рядка, що аналізуються наперед при виконанні розбору.

Іншими словами можна дати наступні спрощені визначення.

Визначення 1. LL(k)-граматика – це граматика, що допускає безповоротний синтаксичний аналіз при вводі вхідного рядка **зліва направо** і реалізує **лівий** вивід із загляданням наперед і аналізом **k** символів вхідного рядка.

Визначення 2. LR(k)-граматика – це граматика, що допускає безповоротний синтаксичний аналіз при вводі вхідного рядка **зліва направо** і реалізує **правий** вивід із загляданням наперед і аналізом **k** символів вхідного рядка.

Визначення 3. RL(k)-граматика – це граматика, що допускає безповоротний синтаксичний аналіз при вводі вхідного рядка **справа наліво** і реалізує **лівий** вивід із загляданням наперед і аналізом **k** символів вхідного рядка.

Визначення 4. RR(k)-граматика – це граматика, що допускає безповоротний синтаксичний аналіз при вводі вхідного рядка **справа наліво** і реалізує **правий** вивід із загляданням наперед і аналізом **k** символів вхідного рядка.

З використанням **LL(k)** і **RR(k)**-граматик реалізується низхідна (зверху вниз) стратегія синтаксичного розбору, а з використанням **LR(k)** і **RL(k)**-граматик – висхідна (знизу вверх) стратегія синтаксичного розбору.

На практиці використовуються в основному **LL(k)** та **LR(k)**-граматики.

Алгоритми синтаксичного розбору зверху вниз

Розглянемо три алгоритми синтаксичного аналізу, що працюють за низхідною стратегією:

1) алгоритм, що працює по методу рекурсивного спуску;
2) аналізуюча машина Кнута;
3) алгоритм таблично-керованого передбачаючого розбору.

Нагадаємо, що стратегія розбору зверху вниз може бути реалізована для **LL(k)**-граматик, причому, чим менше **k**, тим ефективніший аналізатор можна побудувати.

Тому особливої уваги заслуговують **LL(1)**-граматики і умови відповідності довільної граматики **G** класу **LL(1)**, оскільки вони допускають безповоротний аналіз зверху вниз із загляданням наперед всього на один символ.

Синтаксичний аналізатор, що реалізує низхідну стратегію методом рекурсивного спуску

Метод рекурсивного спуску реалізує безповоротний аналіз за рахунок обмежень на правила граматики:

1) правила не повинні містити лівобічної рекурсії. Якщо такі правила є, то вони замінюються правилами з правосторонньою рекурсією або представляються в ітераційній формі.
2) якщо є декілька правил з однаковою лівою частиною, то права частина повинна починатися з різних термінальних символів (нормальна форма Грейбах).

Суть методу: кожному нетерміналу граматики ставиться у відповідність окрема рекурсивна процедура або функція.

Приклад розробки синтаксичного аналізатора методом рекурсивного спуску

Приклад розглянемо для наступної граматики:

1. <блок> → begin <список операторів> end
2. <список операторів> → <оператор> } < список операторів > → <оператор> {; <оператор>}
3. <список операторів> → <оператор>;<список операторів> }
4. <оператор> → <оператор присвоєння>
5. <оператор> → <оператор введення>
6. < оператор присвоєння > → <змінна>: =<вираз>
7. <вираз> → <терм> } <вираз> → <терм>{+ <терм>}
8. <вираз> → <терм> + <вираз> }
9. <терм> → <змінна>
10. <терм> → (<вираз>)
11. <оператор введення> → Read (<список введення>)
12. <список введення> → <змінна> } <список введення> → <змінна>{, <змінна>}
13. <список введення> → <змінна>,<список введення> }
14. <змінна> → i

Визначимо назви процедур, що відповідають нетерміналам граматики таким чином:

<блок> → BLK

<список операторів> → LST

<оператор> → STM

<оператор присвоєння> → ASN

<оператор введення> → GET

<вираз> → EXP

<терм> → TRM

<змінна> → VAR

<список введення> → SPW

Допоміжні процедури:

SCN - сканування лексем (код лексеми записується у змінну TS) ERR - обробка помилок.

Примітка. У наведеній нижче програмі порівняння вигляду TS = ‘символи’ означають, що TS порівнюється з кодами лексем вказаних символів.

Структура синтаксичного аналізатора буде такою:

Program PARSER;

<опис даних>;

<опис процедур>;

begin

<початкові установки>

SCN; BLK

<завершення аналізу>

end.

Розглянемо записані псевдокодом рекурсивні процедури, що реалізують синтаксичний аналізатор:

```
Procedure BLK;
  begin
       if TS <> ‘begin’ then ERR
                          else begin
                                  SCN; LST
                               end
      if TS <> ‘end’ then ERR
  end;
```

```
Procedure LST;
   begin
      STM;
       while TS = ';' do begin
              SCN; STM
        end;
end;

Procedure STM;
   begin
      if TS = 'Read' then begin SCN; GET end;
       else ASN;
 end;

Procedure ASN;
   begin
      VAR;
    if TS <> ': =' then ERR else begin SCN; EXP; end;
end;

Procedure EXP;
   begin
      TRM;
       while TS = ' +' do begin
              SCN; TRM
        end;
 end;
Procedure TRM;
   begin
      if TS = '(' then begin
                          SCN; EXP;
                          if TS <> ')' then ERR ;
                         end
                else VAR;
end;
```

```
Procedure GET;
   begin
      if TS <> ‘(’ then ERR
                    else begin
                              SCN; SPW;
                              If TS <> ‘)’ then ERR
                             end;
       SCN;
end;

Procedure SPW;
   begin
      VAR;
       while TS = ‘ , ’ do begin
              SCN; VAR
        end;
 end;

Procedure VAR;
   begin
      if TS <> ‘ i ’ then ERR else SCN;
 end;
```

Аналізуюча машина Кнута (АМК)

Ця частина конспекту по аналітичній машині Кнута базується, як і попередня на оригінальній статті Дональда Кнута [4]. Розглянемо універсальний алгоритм низхідного синтаксичного аналізу, запропонований Дональдом Кнутом, який отримав назву «аналізуюча машина Кнута (АМК)».

Спочатку розглянемо роботу АМК з поверненнями для не **LL(k)**-граматики, а потім умови, яким повинна задовольняти граматика, щоб належати класу **LL(1)**-граматик і допускати безповоротний розбір.

Розглянемо наступну граматику мови, що нагадує мову «логічних виразів». **<логічний вираз> : : = <відношення> | (<логічний вираз>) <відношення> : : = <вираз> = <вираз> <вираз> : : = a | b |(<вираз> + <вираз>)**

Запишемо цю граматику для компактності у формі, в якій синтаксичні класи позначаються великими літерами, а не беруться в кутові дужки.

B→R|(B)

R→E=E

E → a | b | (E+E)

Відповідна програма для аналізуючої машини Кнута в таблиці 3.

Таблиця 3.

Адреса операції	**Код Операції**	**AT (Адреса True)**	**AF (Адреса False)**
B	[R]	T	
	(		F
	[B]		F
	)	T	F
R	[E]		F
	=		F
	[E]	T	F
E	a	T	
	b	T	
	(		F
	[E]		F
	+		F
	[E]		F
	)	T	F
Start	[B]		**ПОМИЛКА**
	#	**ОК**	**ПОМИЛКА**

Варто звернути увагу на відповідність між граматикою і АМК-програмою. Останні два рядки програми відповідають ще одному граматичному правилу

Start → B #

де «#» – спеціальний символ-обмежувач, який зустрічається лише в самому кінці рядка, що аналізується.

Аналізуюча машина – це абстрактна машина для аналізу рядків в деякому алфавіті. Вона читає із «вхідного рядка» кожен раз по одній літері згідно з деякою програмою. Програма аналізуючої машини – це набір процедур, що рекурсивно викликають одна одну; сама програма по суті є однією з таких процедур. Кожна процедура намагається виявити у вхідному рядку присутність деякої конкретної синтаксичної конструкції і по закінчені роботи повертає значення «true» чи «false» в залежності від того, чи був пошук успішним.

Нехай $S_1 S_2 \ldots S_n$ – вхідний рядок, і S_h – «поточна» літера, яку читає машина Кнута.

Команда цієї машини складається із трьох полів: поля коду операції і двох адрес, AT (Адреса True) і AF (Адреса False). Процедури записуються з використанням двох типів команд, що відповідають двом різним видам кодів операцій.

Тип 1: Код операції – літера *a* із алфавіту.

Тип 2: Код операції – адреса процедури, що стоїть в квадратних дужках [A].

Ці команди виконуються наступним чином:

Тип 1: якщо $S_h = a$, то пропустити *a* (тобто встановити $h := h + 1$) і перейти до AT, інакше перейти до AF.

Тип 2: викликати процедуру, що починається в комірці A (рекурсивно); якщо вона повернула значення true, то перейти до AT, інакше, якщо вона повернула значення false, то перейти до AF.

Кожне з полів AT і AF може містити або адресу команди, або один зі спеціальних символів: *T*, *F* або порожнє поле. Порожнє поле адреси відповідає адресі команди, що записана в наступному рядку. Якщо він містить *T*, здійснюється вихід із процедури із значенням «true». Якщо воно містить *F*, то відбувається вихід із процедури із значенням «false» і змінна *h* знову набуває того значення, яке вона мала до входу в процедуру.

Таким чином, мається на увазі, що при виклику процедури по коду операції типу 2, завжди разом з адресою повернення зберігається і значення змінної *h*.

Процедура Start виконає перехід на мітку «ОК», лише якщо весь рядок, що аналізується, є логічним виразом B, за яким слідує символ «#», інакше вона виконає перехід на мітку «ПОМИЛКА».

Приклад.

Використовуючи розглянуту вище АМК програму, виконаємо розбір наступного вхідного рядка:

```
(  a  =  (  b  +  a  )  )  #
S1 S2 S3 S4 S5 S6 S7 S8 S9 S10
```

Якщо почати з адреси Start, то виконається така послідовність дій (спочатку h = 1, тобто оброблюється перший символ S_1).

Виклик B (h = 1)

Виклик R (h = 1)

Виклик E (h = 1)

Пошук a: ні

Пошук b: ні

Пошук (: так, встановити h : = 2
Виклик E (h = 2).

Пошук a: так; встановити h : = 3
Повернення, true.

Пошук + : ні

Повернення, false; встановити h : = 1

Повернення, false; встановити h : = 1

Пошук (: так, встановити h : = 2

Виклик B (h = 2)

Виклик R (h = 2)

і т.д.

Подібне трасування роботи АМК машини для вказаного вхідного рядка приведене в таблиці 4.

Таблиця 4.

№ кроку	Символ, що аналізується	Адреса Операції	Операція	Результат	Дія
1	S_1 = “(“	Start	[B]		Перехід на адресу операції **B**
2	S_1 = “(“	B	[R]		Перехід на адресу операції **R**
3	S_1 = “(“	R	[E]		Перехід на адресу операції **E**
4	S_1 = “(“	E(first)	a	AF=’ ’	Перехід на наступний рядок
5	S_1 = “(“		b	AF=’ ’	Перехід на наступний рядок
6	S_1 = “(“		(	розпізнано AT = ’ ’	Перехід на наступний рядок
7	S_2 = “a“		[E]		Перехід на адресу операції **E**
8	S_2 = “a“	E(second)	a	розпізнано AT = T	Повернення в попереднє **E** з “T”
9	S_3 = “=”	E(first)	Обр. рез-ту	AT=’ ’	Перехід на наступний рядок
10	S_3 = “=”		+	AF=F	Повернення в **R** з “F”
11	S_1 = “(“	R	Обр. рез-ту	AF=F	Повернення в **B** з “F”
12	S_1 = “(“	B		AF=’ ’	Перехід на наступний рядок
13	S_1 = “(“		(	розпізнано AT = ’ ’	Перехід на наступний рядок
14	S_2 = “a“		[B]		Перехід на адресу операції **B**
15	S_2 = “a“	B(second)	[R]		Перехід на адресу операції **R**
16	S_2 = “a“	R	[E]		Перехід на адресу операції **E**
17	S_2 = “a“	E	a	розпізнано AT = ‘T’	Повернення на **R** з “T”
18	S_3 = “=”	R	Обр. рез-ту	AT=’ ’	Перехід на наступний рядок
19	S_3 = “=”		=	розпізнано AT = ’ ’	Перехід на наступний рядок
20	S_4 = “(“		[E]		Перехід на адресу операції **E**
21	S_4 = “(“	E(first)	a	AF=’ ’	Перехід на наступний рядок
22	S_4 = “(“		b	AF=’ ’	Перехід на наступний рядок
23	S_4 = “(“		(	розпізнано AT = ’ ’	Перехід на наступний рядок
24	S_5 = “b“		[E]		Перехід на адресу операції **E**
25	S_5 = “b“	E(second)	a	AF=’ ’	Перехід на наступний рядок
26	S_5 = “b“		b	розпізнано AT = ’T’	Повернення в попереднє **E** з “T”
27	S_6 = “+“	E(first)	Обр. рез-ту	AT=’ ’	Перехід на наступний рядок
	і так далі				

По завершенні роботи АМК управління буде передано на адресу «ОК», при цьому історія викликів процедур, повернення з яких відбулось із значенням «true», відповідатиме дереву «розбору» вхідного рядка, показаному на рис.18.

Це дерево можна уявити собі побудованим зверху вниз в процесі виконання програми для аналізуючої машини, де частина дерева для терміналів, що стоять зліва від нетерміналу правої частини правила будується на спуску, а для терміналів, що стоять страва від нього – на поверненні.

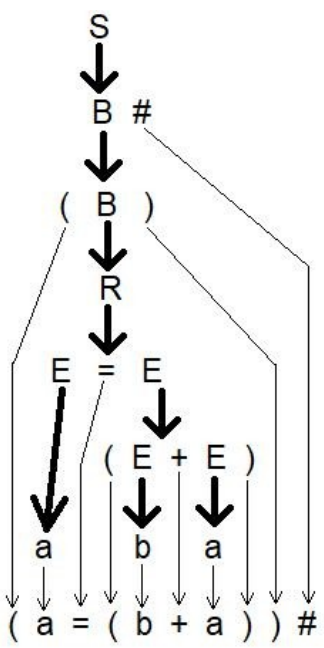


Рис.18.

Формування таблиці АМК (програмування АМК)

Припустимо спочатку, що як і в наведеному вище прикладі, всі БНФ-правила граматики записані в стандартній формі

$X \rightarrow Y_1 \mid Y_2 \mid \ldots \mid Y_m \mid Z_1 Z_2 \ldots Z_n$,

де $m, n \geq 0$, $m + n > 0$ і всі Y, Z V, тобто це або термінальні символи (тобто є буквами алфавіту), або нетермінальні символи (тобто представляють синтаксичні класи).

Права частина правила містить m+1 варіантів; якщо m = 0, то вона приймає простий вигляд:

$X \rightarrow Z_1 Z_2 \ldots Z_n$

Якщо n = 0, то рядок $Z_1 Z_2 \ldots Z_n$ розглядається як порожній рядок. АМК-програма, що відповідає правилу, записаному в стандартній формі, складається з наступних m+n інструкцій і показана в таблиці 5. Таблиця 5.

Адреса	Код операції	AT	AF
X	$[Y_1]$	T	
	$[Y_2]$	T	
	:	:	
	$[Y_m]$	T	*
	$[Z_1]$		F
	:		:
	$[Z_{n-1}]$		F
	$[Z_n]$	T	F

Якщо Y_i або Z_j є термінальними символами, то квадратні дужки при вказуванні відповідного коду операції потрібно видалити. Якщо n = 0, то адресу, що позначена через «*», потрібно замінити на F; в іншому випадку потрібно залишити пробіл. Якщо БНФ-правило записане не в стандартній формі, тоді його можна привести до стандартної форми, через введення нових нетермінальних символів $Y_1 \ldots Y_m$ і додавання правила

$Y_1 \rightarrow a_1$

....

$Y_m \rightarrow a_m$

Якщо, наприклад, наша БНФ-граматика містить правило

X→AB|CD

то ми замінюємо його двома правилами

X→Y|CD

Y→AB

Приклад побудови таблиці АМК

```
1.  <signal-program> --> <program>
2.  <program> --> PROGRAM <procedure-identifier> ; <block>.
3.  <block> --> <variable-declarations> BEGIN <statements-list> END
4.  <variable-declarations> --> VAR <declarations-list> |
    <empty>
5.  <declarations-list> --> <declaration> <declarations-list>
    | <empty>
6.  <declaration> --><variable-identifier>: INTEGER ;
```

```
1.  <signal-program> --> <program>
2.  <program> --> PROGRAM <procedure-identifier> ; <block>.
3.  <block> --> <variable-declarations> BEGIN <statements-list> END
4.  <variable-declarations> --> << VAR <declarations-list> >> ?
5.  <declarations-list> --> <declaration> { <declaration> }
6.  <declaration> --><variable-identifier>: INTEGER ;
```

АМК для цієї граматики показана в таблиці 6. Початкова установка FDone для нетерміналів = False, для терміналів завжди = True.

Таблиця 6.

				if FDone=False	if FDone=True	
	Адреса операції	**Прапорець FDone**	**Код Операції**	**AN (Адреса нетермінала)**	**AT (Адреса True)**	**AF (Адреса False)**
1	<signal-program>	F/T	<program>	2	**ОК**	
2	<program>	T	PROGRAM		3	Error
3		T	<procedure-identifier>		4	Error
4		T	;		5	Error
5		F/T	<block>	7	6	
6		T	.		FDone(1):=T; 1	Error
7	<block>	F/T	<variable-declarations>	11	8	
8		T	BEGIN		9	Error
9		F/T	<statements-list>	**???**	10	
10		T	END		FDone(5):=T; 5	Error
11	<variable-declarations>	T	VAR		12	FDone(7)=T; 7 (<empty> - безвідмовний символ)
12		F/T	<declarations-list> |	13	FDone(7)=T; 7	
13	<declarations-list>	F/T	<declaration>	14	**if** (lexem == <variable-identifier>) {FDone(13):=T; 14} **else** { FDone(12):=T; 12 (<empty> - безвідмовний символ)}	
14	<declaration>	T	<variable-identifier>		15	Error
15		T	:		16	Error
16		T	INTEGER		17	Error
17		T	;		FDone(13):=T; 13	Error
???	<statements-list>					

Строге визначення LL(k) і LR(k)-граматик

Введемо декілька позначень.

Якщо k – невід'ємне ціле число;

α V*;

$|\alpha|$ – довжина рядка α;

$^{L}_{*}$ – лівий вивід;

$^{R}_{*}$ – правий вивід;

то

$$k{:}\alpha = \begin{cases} \text{якщо } |\alpha| \geq k, \text{ то } k{:}\alpha \text{ — це перші } k \text{ символів рядка } \alpha \\ \text{інакше } k{:}\alpha \text{ – це весь рядок } \alpha \end{cases}$$

$$\alpha{:}k = \begin{cases} \text{якщо } |\alpha| \geq k, \text{ то } \alpha{:}k \text{ — це останні } k \text{ символів рядка } \alpha \\ \text{інакше } \alpha{:}k \text{ – це весь рядок } \alpha \end{cases}$$

Визначення 5. КВ-граматика є LL(k)-граматикою, якщо для будь-якого A N і будь-яких рядків β, β', γ T* і α, α', δ V* виконується наступна умова:

якщо $S \overset{L}{*}\gamma A\delta \quad \overset{L}{}\gamma\alpha\delta \quad \overset{L}{*}\gamma\beta$,

та $S \overset{L}{*}\gamma A\delta \quad \overset{L}{}\gamma\alpha'\delta \quad \overset{L}{*}\gamma\beta'$,

та $k{:}\beta = k{:}\beta'$,

то **$\alpha=\alpha'$**

Визначення 6. КВ-граматика є LR(k)-граматикою, якщо для будь-яких A і A' N і будь-яких рядків α, α', β, β' V* і γ, γ' T* виконується наступна умова:

якщо $S \overset{R}{*}\beta A\gamma \quad \overset{R}{}\beta\alpha\gamma$

та $S \overset{R}{*}\beta'A'\gamma' \quad \overset{R}{}\beta'\alpha'\gamma'$

та $(|\beta\alpha| + k){:}\ \beta\ \alpha\ \gamma = (|\beta\alpha| + k) : \beta'\ \alpha'\ \gamma'$

то **$\beta=\beta'$;$A=A'$;$\alpha=\alpha'$**.

Айронс запропонував метод розбору, який не є ні низхідним, ні висхідним і отримав назву «розбір з лівого кута», а відповідний йому клас граматик отримав назву **LC(k)**-граматики. Цей метод розбору працює з неповними поверненнями. **LC(k)**-граматики були досліджені Розенкранцем і Л'юїсом, які показали, що всі **LC(k)**-мови є **LL(k)**-мовами і навпаки.

Множини FIRST і FOLLOW

При розгляді конкретних алгоритмів синтаксичного аналізу та побудові багатьох синтаксичних аналізаторів зручно використовувати множини FIRST(A) і FOLLOW(A), що зв'язані з певною граматикою G. Ці множини дозволяють побудувати таблицю передбачаючого розбору для G, якщо, звичайно, це можливо. Крім того, ці множини можуть, крім того, бути використані при відновленні після помилок.

Якщо α – довільний рядок символів граматики, то FIRST(α) – множина терміналів, з яких починаються рядки, що виводяться із α. Якщо α * , то також належить FIRST(α).

Визначимо FOLLOW(A) для нетерміналу A як множину терміналів a, що можуть з'явитись безпосередньо справа від A в деякій сентенціальній формі, тобто множина терміналів a, таких, що для деяких α і β існує вивід виду S * α Aa β. Зазначимо, що між A і a в процесі виводу можуть з'являтись нетермінальні символи, із яких виводиться . Якщо A може бути найправішим символом деякої сентенціальної форми, то # (обмежувач) належить FOLLOW(A).

Приклад. Розглянемо граматику:

1. E→TE'	1. <вираз> → <доданок><рядок доданків>
2. E'→+TE'	2. <рядок доданків> → + <доданок><рядок доданків>
3. E' →	3. <рядок доданків> →
4. T→FT'	4. <доданок> → <множник><рядок множників>
5. T'→ FT'	5. <рядок множників> → * <множник><рядок множників>
6. T' →	6. <рядок множників> →
7. F→(E)	7. <множник> → (<вираз>)
8. F → id	8. <множник> → id

Для цієї граматики множини FIRST(A) і FOLLOW(A) будуть такими:

FIRST(E) = FIRST(T) = FIRST(F) = { (, id }
FIRST(E') = { +, }
FIRST(T') = { *, }

FOLLOW(E) = FOLLOW(E') = {), # }
FOLLOW(T) = FOLLOW(T') = { +,), # }
FOLLOW(F) = { *, +,), # }

Таблично-керований передбачаючий (прогнозуючий) синтаксичний аналізатор

На рис.19 зображена структура передбачаючого аналізатора, який визначає чергове наступне правило за один крок за допомогою таблиці. Таку таблицю можна побудувати безпосередньо за правилами заданої граматики.

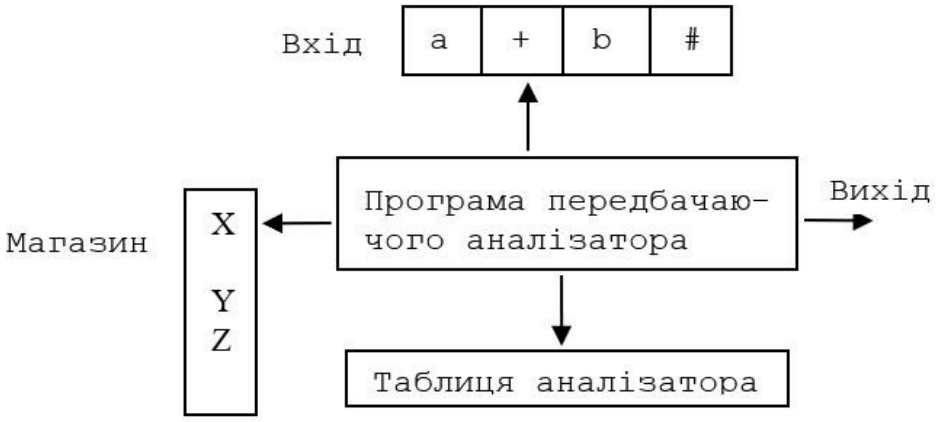


Рис 19. Структура передбачаючого синтаксичного аналізатора

Використані на рисунку символи належать таким

множинам a, +, b T

U,V,W,X,Y,Z H

– правий кінцевий маркер (ознаку кінця рядка).

Таблично-керований передбачаючий аналізатор має вхідний буфер, магазин, таблицю аналізу і вихід.

Вхідний буфер містить рядок розпізнавання, за яким йде #.

Магазин містить послідовність символів граматики з символом # на дні. Спочатку магазин містить початковий символ граматики на верхівці і символ # на дні.

Таблиця аналізу – це двовимірний масив ParserTable[A,a], де A – нетермінал, і a – термінал або символ #.

Алгоритм такого аналізатора працює за тим самим загальним принципом, що був розглянутий у прикладі недетермінованого МП-автомату, тільки з використанням таблиці і без повернень.

На кожному кроці алгоритм розглядає символ X, що знаходиться на верхівці магазину і поточний вхідний термінальний символ a. Ці два символи визначають дію аналізатора.

Загалом є три варіанти дій такого аналізатора:

1. Якщо X – термінал та X = a = #, то аналізатор зупиняється і повідомляє про успішне завершення розбору.

2. Якщо X – термінал та X = a ≠ #, то аналізатор видаляє X з магазина і просуває покажчик входу на наступний вхідний символ.

3. Якщо X – нетермінал, то алгоритм бере з таблиці елемент ParserTable[X,a]. За цими координатами в таблиці зберігається або правило для нетерміналу X, або ознака помилки.
   Якщо, наприклад, ParserTable[X,a] = {X → ABC}, то аналізатор заміняє X на верхівці магазину на символи правої частини цього правила, заштовхуючи до магазину символи правої частини правила в оберненому порядку, тобто в порядку CBA (в результаті на верхівці буде A).

Якщо ParserTable[X,a] = error, то аналізатор звертається до підпрограми аналізу помилок.

Поведінка аналізатора може бути описана в термінах конфігурацій автомата розбору. В початковому стані аналізатор знаходиться в конфігурації, в якій магазин містить символи **S#** (S – аксіома граматики, # – обмежувач), у вхідному буфері **w#** (w – вхідний рядок, # – обмежувач), а також поточний символ вхідного рядка записується до певної робочої змінної Symbol. Алгоритм використовує таблицю аналізатора ParserTable, що розглянута нижче.

Приклад 1. Розглянемо граматику арифметичних виразів у такому вигляді:

1. E→TE' FIRST(E)=FIRST(T)=FIRST(F)={(,id}
2. E'→+TE' FIRST(E')={+, }
3. E' → FIRST(T') ={*, }
4. T→FT' FOLLOW(E)=FOLLOW(E')={),#}
5. T' → * F T'FOLLOW(T)=FOLLOW(T')={+,),#}
6. T' → FOLLOW(F)={+,*,),#}
7. F→(E)
8. F → id

Таблиця передбачаючого аналізатора ParserTable показана у таблиці 7. У цій таблиці пусті клітинки відповідають помилковим ситуаціям (error), а непусті клітинки містять правила, за якими виконується розгортка нетермінала. Таблиця 7. Таблиця передбачаючого аналізатора ParserTable

Нетермінал	**Вхідний символ**					
	id	**+**	*****	**(**	**)**	**#**
E	E→ TE'			E→ TE'		
E'		E'→ +TE'			E'→	E'→
T	T→ FT'			T→ FT'		
T'		T'→	T'→ *FT'		T'→	T'→
F	F→ id			F→ (E)		

Процес трасування (послідовності кроків) роботи передбачаючого синтаксичного аналізатора для рядка **id + id * id #** показаний у таблиці 8, а відповідне дерево розбору на рисунку 20. Вказівник вхідного рядка показує на найлівіший символ в колонці «Вхід».

Якщо уважно проаналізувати дії аналізатора, то можна побачити, що він виконує лівосторонній вивід, тобто правила застосовуються у відповідності до лівостороннього виводу.

Таблиця 8. Таблиця трасування роботи передбачаючого аналізатора

Вхід	**Магазин**	**Вихід** (застосоване правило)	**№ правила**
id+id*id#	E#		
id+id*id#	TE'#	E→TE'	1
id+id*id#	FT'E'#	T→FT'	4
id+id*id#	idT'E'#	F→id	8
+id*id#	T'E'#		
+id*id#	E'#	T'→	6
+id*id#	+TE'#	E'→+TE'	2
id*id#	TE'#		
id*id#	FT'E'#	T→FT'	4
id*id#	idT'E'#	F→id	8
*id#	T'E'#'		
*id#	*FT'E'#	T'→*FT'	5
id#	FT'E'#		
id#	idT'E'#	F→id	8
#	T'E'#		
#	E'#	T'→	6
#	#	E'→	3

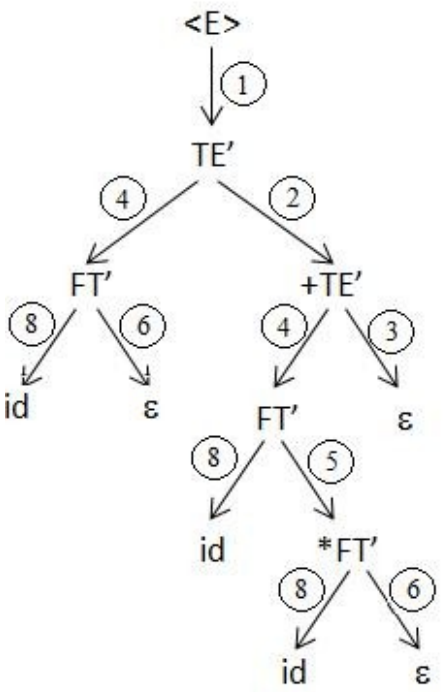


Рис. 20.

Алгоритми синтаксичного розбору знизу вгору

Як вже зазначалося раніше, висхідний розбір (аналіз) знизу-вгору базується на понятті **«основа»**.

Нагадаємо суть цього поняття. Якщо деякий рядок терміналів та/або нетерміналів (сентенціальна форма) прямо приводиться до нетерміналу, то він називається безпосередньо приводимою фразою. Найлівіша безпосередньо приводима фраза заданої сентенціальної форми (рядка термінальних та/або нетермінальних символів) називається **основою**.

Суть алгоритму аналізу полягає у пошуці основи на кожному кроці алгоритму й заміни її нетерміналом, до якого ця основа безпосередньо приводиться, доки не отримаємо аксіому граматики.

Процес такої заміни називається **згорткою (reduce)**. Розбір завершиться тоді, коли нетерміналом, до якого застосовано згортку, стане аксіома граматики.

Синтаксичний аналізатор, що реалізує стратегію знизу вгору без повернення методом граматик передування

Всі алгоритми аналізу без повернення будуються на основі спеціальних граматик з певними обмеженнями.

Розглянемо алгоритм висхідного аналізу на основі граматик простого передування. Метод граматик простого передування був розроблений Віртом і Вебером і є висхідним аналогом аналізуючої машини Кнута без повернень.

Відношення простого передування

Існує 3 види відношень передування:

а) …Si ≐ Sj …

основа

б) …Si <· Sj … Sj – голова основи

основа

в) … Si ·> Sj … Si – хвіст основи

основа

де Si, Sj V = N

Наприклад: …S_{i-1} <· S_i ≐ S_{i+1} ≐ S_{i+2} ≐ S_{i+k} ·> S_{i+k+1}

основа

Визначення граматики простого передування

Граматика G називається **граматикою простого передування**, якщо між символами граматики існують наступні відношення передування:

1) між символами граматики існують наступні відношення передування:
   а) Si ≐ Sj, якщо існує правило U → Si Sj
   б) Si ⋖ Sj, якщо існує правило U → Si D та виведення D Sj δ в) Si ⋗ Sj, якщо існує правило U → C Sj і виведення C δ Si
   або існує правило U → C D і виведення C δ Si, D Sj , де U, C, D N, тобто нетермінальні символи граматики G,
   a , , ,δ V* –рядки
2) не допускаються правила з однаковою правою частиною, а також наявність більше одного відношення передування між будь-якою парою символів.

Приклад.

Задано граматику:

T = {x, a, (,) }

N = {<A>, <терм>, <вираз>}

Множина правил G цієї граматики має наступний вигляд:

1. <A> → x<терм>x
2. <терм> → a
3. <терм> → (<вираз>
4. <вираз> → <терм>a)

При реалізації МП-автоматів синтаксичних аналізаторів використовуються обмежувальні символи, які ставляться ліворуч і праворуч від символів вхідного рядка T[1..n]. В якості такого символу візьмемо символ ‘#’.

Для обмежувальних символів T[0]=’#’ та T[n+1]=’#’ мають місце наступні відношення:

1) T[0] ⋖ «будь-який символ рядка»;
2) «будь-який символ рядка» ⋗ T[n+1].

Матриця відношень передування для цієї граматики показана к таблиці 9.

Таблиця 9. Матриця граматики передування S_j

S_i	ха		(	)	<A>	<терм>	<вираз>
x		⋖	⋖			≐	
a	⋗	⋗		≐			
(		⋖	⋖			⋖	≐
)	⋗	⋗					
<A>							
<терм>	≐	≐					
<вираз>	⋗	⋗					

Розглянемо алгоритм синтаксичного аналізатора, що працює за граматикою передування.

Для його реалізації потрібні наступні дані:

- таблиця правил;
- матриця відносин передування P;
- стек S (наприклад, у вигляді вектора);
- вхідний рядок T (вектор лексем);
- к – поточний індекс вектора T (T[k] - елемент вхідного рядка);
- i – індекс правого символу основи в стеку S;
- j – індекс лівого символу основи в стеку S;
- черговий символ R.

У цих позначеннях основа завжди матиме вигляд $\underbrace{S_j \ \ldots \ S_i}$

Алгоритм синтаксичного аналізатора, що працює за граматикою передування показаний на рисунку 21, а схема трасування роботи цього алгоритма для вищенаведеної граматики при вхідному рядку **# x (a a) x #**, показана у таблиці 10.

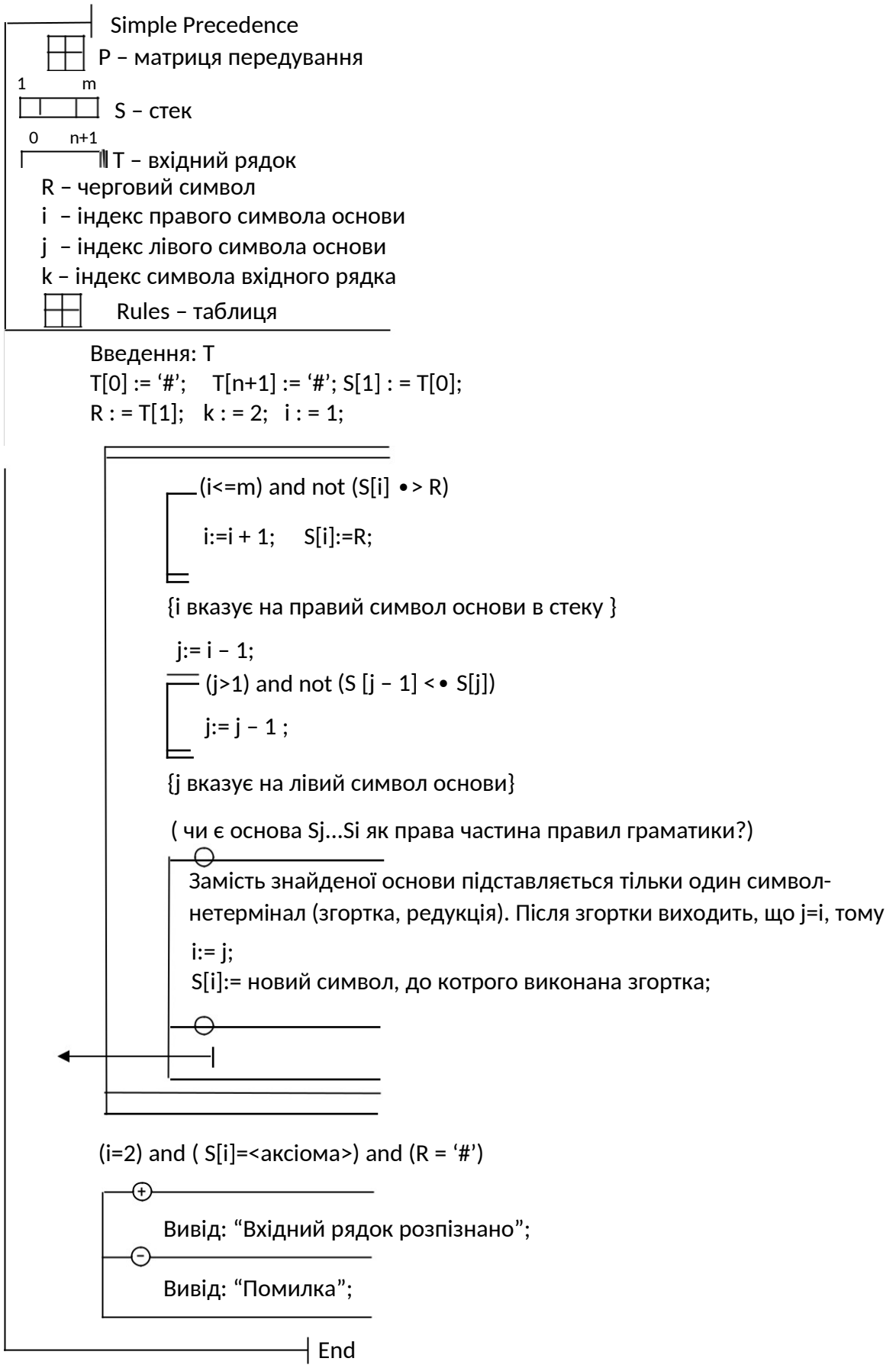


Рис. 21. Алгоритм синтаксичного аналізатора, що працює за граматикою передування

Таблиця 10. Трасування роботи алгоритму синтаксичного аналізатора, що працює за граматикою передування для рядка **# x (a a) x #**

№ кроку	S							Відношення	R + залишок вхідного рядка T	
	1	2	3	4	5	6	7		R	залишок
0	#							<·	x	(a a) x #
1	#	x						<·	(	a a) x #
2	#	x	(					<·	a	a) x #
3	#	x	(	a				·>	a	) x #
4	#	x	(	<терм>				≐	a	) x #
5	#	x	(	<терм>	a			≐	)	x #
6	#	x	(	<терм>	a	)		·>	x	#
7	#	x	(	<вираз>				·>	x	#
8	#	x	<терм>					≐	x	#
9	#	x	<терм>	x				·>	#	
10	#	<A>							#	

СИНТАКСИЧНИЙ АНАЛІЗ (ПРОДОВЖЕННЯ)

Умови безповоротного LL(1) синтаксичного аналізу

Якщо для **LL(1)**-граматики можна побудувати синтаксичний аналізатор, що працює без повернень, то отримаємо дві наступні переваги:

1) загальний час синтаксичного аналізу буде обмежений величиною, що пропорційна довжині вхідного рядка;
2) програма синтаксичного аналізатора може читати вхідні символи по одному, не зберігаючи раніше прочитані символи.

Відомо, що правила будь-якої КВ-граматики можна подати у вигляді правил, що належать лише наступним п’яти типам [4]:

Тип 1. $X_p \rightarrow X_q \mid X_r$, де $q<p$ і $r<p$

Тип 2. $X_p \rightarrow X_q \ X_r$, де $q<p$ і якщо $X_q \rightarrow^+$, то і $r<p$

Тип 3. $X_p \rightarrow X_q$, де $q<p$

Тип 4. $X_p \rightarrow a$

Тип 5. $X_p \rightarrow$

Для всіх типів правил X_p, X_q, X_r N, a T.

Відношення $q<p$ в даному випадку означає таку властивість: якщо граматика не має ліворекурсивних нетерміналів, то всі нетермінали $X_1, X_2,\ldots, X_n$ можна впорядкувати так, що

$X_p \overset{L}{+} X_q$ лише тоді, коли $q<p$, тобто X_p знаходиться вище по граматиці (ближче до аксіоми в ланцюжку виводу).

Умови безповоротного низхідного розбору формулюються наступним чином:

1. Жоден нетермінальний символ не є ліворекурсивним.
2. Для жодного правила типу 1 із X_q і X_r не виводяться рядки, що починаються з одного і того ж термінального символа, тобто $FIRST(X_q)$ $FIRST(X_r) =$.
3. Для будь-якого правила типу 1, де X_r може породжувати порожній рядок X_r * , із іншого нетерміналу X_q не може виводитись рядок, що починається з терміналу, який належить множині FOLLOW нетерміналу лівої час-тини правила X_p, тобто завжди виконується $FIRST(X_q)$ $FOLLOW(X_p)$ = . Те ж саме повинне виконуватись, якщо X_q і X_r міняються ролями.

4. В жодному правилі типу 1 одночасно обидва символи X_q та X_r не повинні бути «безвідмовними», іншими словами їм повинна відповідати процедура аналізуючої машини Кнута, яка хоча б в одному випадку повертає значення «false».

Визначення. Нетермінальний символ X_p є **безвідмовним** тоді і тільки тоді, коли відповідне йому правило:

1) або належить типу 5 ($X_p \rightarrow$);
2) або належить типу 3 і X_q є безвідмовним;
3) або належить типу 2 і X_q є безвідмовним;
4) або належить типу 1 і X_q або X_r є безвідмовним.

Перевірити умову 4 можна шляхом послідовного дослідження символів X_p на безвідмовність в порядку $X_1, X_2,\ldots,X_p$, тобто від меншого до більшого.

Інші визначення LL(1)-граматики

Визначення 1.

Теорема. Довільна LL(1) – мова може бути описана граматикою, всі правила якої належать одному з двох типів:

Тип 1. $A \rightarrow a_1 \beta 1 \mid a_2 \beta 2 \mid \ldots \mid a_m \beta m$
Тип 2. $A \rightarrow a_1 \beta 1 \mid a_2 \beta 2 \mid \ldots \mid a_m \beta m \mid$

де $a_1, a_2, \ldots a_m$ – різні термінальні символи; в правилах типу 2 жоден із символів a_i не належить множині FOLLOW(A); β_i – довільні сентенціальні форми.

Визначення 2.

1. Неоднозначна чи ліворекурсивна граматика не може бути LL(1).

2. Граматика G є LL(1)-граматикою тоді і тільки тоді, коли для двох правил вигляду

$A \rightarrow \alpha|\beta$

виконується наступне:

1) ні для якого терміналу a одночасно з α і β не виводяться рядки, які починаються з терміналу a;
2) тільки з одного із рядків α чи β може виводитися пустий рядок;
3) якщо $\beta \rightarrow^* \varepsilon$, то з α не виводиться ніякий рядок, який починається з терміналу, що належать множині FOLLOW(A).

Визначення 3.

КВ-граматика називається LL(1)-граматикою, якщо з існування двох лівих виводів

1. $S \Rightarrow^* \gamma A \alpha \gamma \beta \alpha \Rightarrow^* \gamma x$,
2. $S \Rightarrow^* \gamma A \alpha \gamma \delta \alpha \Rightarrow^* \gamma y$,

для яких з FIRST(x) = FIRST(y), слідує, що $\beta = \delta$, тобто, що для даного ланцюжка $\gamma A \alpha$ і першого символу, що виводиться з $A\alpha$ чи #, існує не більше одного правила, яке може бути застосовано до A, щоб отримати вивід деякого термінального ланцюжка, який починається з γ і продовжується цим першим символом.

Видалення лівої рекурсії

Основна складність при використанні передбачаючого аналізу – це написання такої граматики для вхідної мови, щоб за нею можна було б побудувати передбачаючий аналізатор. Іноді за допомогою деяких простих перетворень не LL(1)-граматику можна привести до LL(1)-вигляду.

Серед цих перетворень найбільш очевидні являються **ліва факторизація** і **видалення лівої рекурсії**.

Тут необхідно зробити два зауваження: по-перше, не всяка граматика після цих перетворень стає LL(1), по-друге, після видалення лівої рекурсії і лівої факторизації отримана граматика може стати важкою для розуміння.

Граматика ліворекурсивна, якщо в ній міститься нетермінал A такий, що існує вивід $A \Rightarrow^+ A\alpha$ для деякого рядка α. Ліворекурсивні граматики не можуть аналізуватися методами зверху-вниз, тому необхідно видалення лівої рекурсії.

Безпосередню ліву рекурсію (рекурсію вигляду $A \rightarrow A\alpha$) можна видалити наступним чином.

Спочатку групуємо A-правила:

$A \rightarrow A\alpha_1 \mid$
$A\alpha_2 \mid$
$\ldots \mid$
$A\alpha_m \mid$
$\beta_1 \mid$
$\beta_2 \mid$
$\ldots \mid$
β_n

де жоден з рядків β_i не починається з A.

Потім заміняємо A-правила на

$$A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$

$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid$$

Нетермінал A породжує ті ж рядки, що і раніше, але тепер немає лівої рекурсії. За допомогою цієї процедури видаляються всі безпосередні ліві рекурсії, але не видаляється ліва рекурсія, яка включає два чи більше кроки. Приведений нижче алгоритм 3 дозволяє видалити всі ліві рекурсії з граматики.

Ліва факторизація

Основна ідея лівої факторизації в тому, коли незрозуміло яку з двох альтернатив потрібно використовувати для розгортки нетерміналу A, потрібно переробити A-правила так, щоб відкласти рішення до того часу, коли не буде достатньо інформації, щоб прийняти правильне рішення.

Якщо

$$A \to \alpha\beta_1 \mid \alpha\beta_2$$

тобто два A-правила і вхідний рядок починається з непустого рядка, який виводиться з α, ми не знаємо, чи розгортати по $\alpha\beta_1$ чи по $\alpha\beta_2$. Проте можна відкласти рішення, розгорнувши $A \to \alpha A'$.

Тоді після аналізу того, що виводимо з α, можна розгорнути $A' \to \beta_1$ або $A' \to \beta_2$.

Лівофакторизовані правила приймають вигляд:

$$A \to \alpha A'$$

$$A' \to \beta_1 \mid \beta_2$$

Відновлення після синтаксичних помилок

В наведених програмах використовувалась процедура реакції на синтаксичні помилки error(). В найпростішому випадку ця процедура видає діагностику и завершує роботу аналізатора. Але можна спробувати деяким чином продовжити роботу. Для розбору зверху вниз можна запропонувати наступний простий алгоритм.

Якщо в момент виявлення помилки на верхівці магазину опинився нетермінальний символ A і для нього нема правила, яке відповідає вхідному символу, то скануємо вхід доти, доки не зустрінемо символ або з FIRST(A), або з FOLLOW(A). В першому разі розгортаємо A за відповідним правилом, в другому – видаляємо A з магазину.

Якщо на верхівці магазину термінальний символ, то можна викинути всі термінальні символи з верхівки магазину аж до першого (зверху) нетермінального символу і продовжувати так, як це було описано вище.

СЕМАНТИКА КОНТЕКСТНО-ВІЛЬНИХ МОВ І ГЕНЕРАЦІЯ ВИХІДНОГО КОДУ

Способи визначення семантики мов програмування

Розрізняють формальну і неформальну семантику.

Формальна семантика (для побудови компіляторів) – це зміст (сенс) синтаксичних правил вхідної мови, що виражені в термінах вихідної мови та в діях по формуванню значень вихідної мови.

Неформальна семантика (для користувача) – це зміст (сенс) речень вхідної мови, що виражений в термінах речень вихідної мови.

В якості метасемантичної мови може бути або спеціалізована метасемантична мова, або інша мова програмування. Серед відомих метасемантичних мов можна назвати:

3) дворівневі або w-граматики;
4) системи продукцій;
5) віденську метамову;
6) атрибутні граматики (розширений атрибутний метод).

1. Дворівневі або w-граматики використовувалися для опису семантики мови АЛГОЛ-68 на основі використання метаправил, які дозволяють описати контекстно-залежні характеристики мови у вигляді правил КВ-граматики.
2. Система продукцій дозволяє сформувати породжуюче правило, яке визначає семантику на основі контекстно-залежних характеристик мови.
3. Віденська метамова дозволяє описати процедуру, що породжує речення на деякій об’єктній мові абстрактної машини, інтерпретуючи яку можна згенерувати код для конкретного комп’ютера. Така метамова використовувалася для реалізації компілятора мови PL-1.
4. Метод атрибутних граматик оснований на тому, що правила КВ-мови доповнюються атрибутами, здійснивши обчислення яких можна отримати вихідну програму. Атрибути приписуються нетерміналам і можуть бути *успадкованими* (спадкування атрибутів відбувається зверху донизу) і *синтезованими* (синтез атрибутів відбувається знизу вгору).

Внутрішнє представлення дерева розбору

Розглянемо на прикладі оператора присвоєння та спрощеного виразу:

1) <оператор> → <змінна>:=<вираз>
2) <вираз> → <вираз ><зод><терм>
3) <вираз> → <терм>
4) <зод> → +
5) <зод> → –
6) <терм> → <змінна>
7) <терм> → (<вираз>)
8) <змінна> → A
9) <змінна> → B
10) <змінна> → C
11) <змінна> → D

При побудові дерева розбору розрізняють прості вузли, що складаються із одного терміналу або нетерміналу, і складні, що складаються із декількох компонент.

Розглянемо дерево розбору для оператора **A:=B+C**, яке показане на рис. 22.

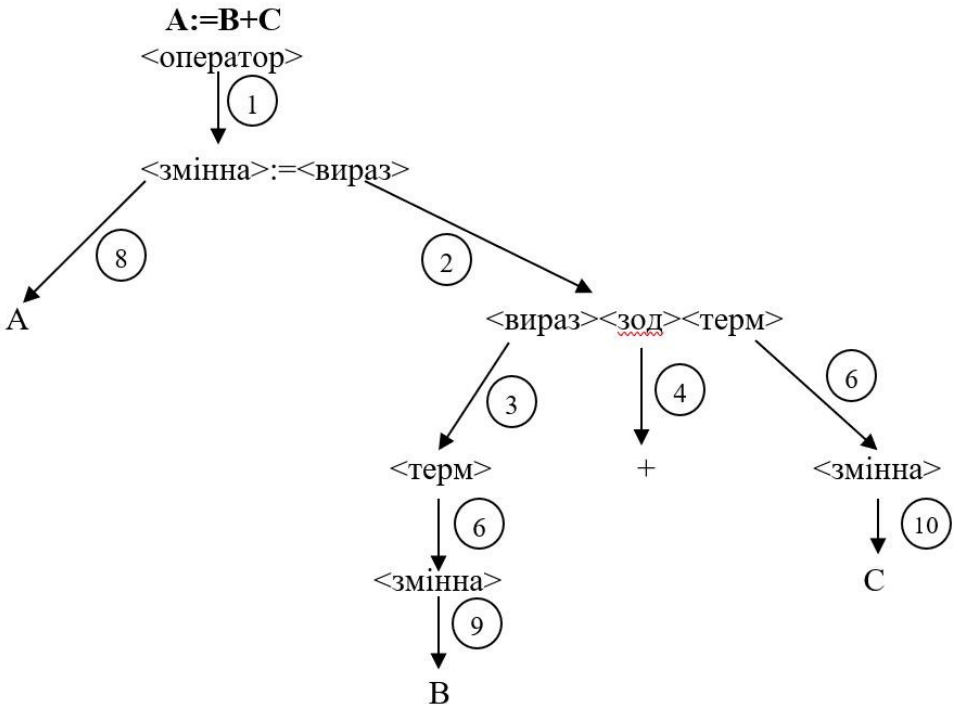


Рис. 22. Дерево розбору для оператора **A:=B+C**

Для внутрішнього представлення дерева розбору в трансляторі іноді використовується форма у вигляді вектора цілих чисел. Елементи такого вектора мають наступний сенс:

– додатне ціле число є номером правила;

– від'ємне ціле число є посиланням (індексом елемента вектора, в якому розміщується номер потрібного правила).

Представлення дерева розбору у вигляді числового вектора називається **лінійною формою представлення дерева розбору**.

Порядок побудови лінійної форми дерева розбору з посиланнями:

1. Дерево розбору проходять зліва-направо, знизу-вверх по ланцюжкам, які ведуть до нелівих нетерміналів.
2. При досягненні нелівого нетерміналу в вектор розбору (назвемо його вектором RAS) виписуються номери правил, що йдуть зверху-вниз. Якщо інших нелівих нетерміналів у даному правилі більше немає, то подальший підйом вверх можливий тільки в тому випадку, коли шлях веде до нелівої компоненти або аксіоми граматики.
3. Якщо складний вузол містить декілька нелівих компонент, то кожній нелівій компоненті ставиться у відповідність посилання, причому порядок посилань нумерується справа наліво.
4. Останнім елементом вектора RAS є посилання на аксіому граматики.

Лінійна форма представлення дерева розбору для наведеного вище прикладу оператора **A:=B+C** показана на рис.23.

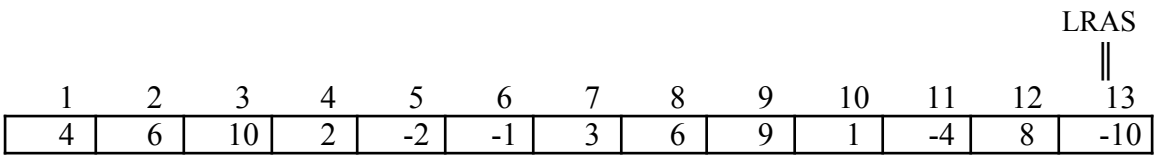


Рис. 23. Лінійна форма представлення дерева розбору оператора **A:=B+C**

Види семантичної відповідності

1. Проста відповідність – на вихід передається один символ або рядок символів вихідної мови.

2. Відповідність типу посилання (клауза) – рекурсивний спуск по дереву розбору.

3. Проста функціональна відповідність – значення або спосіб обробки нетермінала є функцією його входження в праву частину правила.
   Приклад.
   Для нетерміналу <зод> при обробці правил
   <множник> → <зод><число> та <вираз>
   → <вираз><зод><терм>

   повинні бути згенеровані різні команди вихідної мови.

4. Складна функціональна відповідність – семантика обробки нетерміналу визначається дальнім контекстом.
   Приклад.

В правилі <змінна>:=<вираз>, змінна і вираз можуть бути різних типів (дійсних, цілих, тощо) і, відповідно до цих типів, повинні бути згенеровані різні команди вихідної програми.

Проста метасемантична мова

Проста метасемантична мова – це множина семантичних визначень, які описуються трьома простими видами об’єктів і які ставляться у відповідність кожному синтаксичному правилу.

Семантичне визначення – це опис відповідності між кожним правилом (іноді конструкцією) вхідної мови та конструкцією(-ями) (фрагментом коду) вихідної мови, а також опис дій, що виконують семантичний аналіз та генерацію коду, для даного правила граматики.

Розглянемо граматику простої метасемантичної мови.

1. <семантичне визначення> → {<сукупність об’єктів>}
2. <сукупність об’єктів> → <об’єкт>
3. <сукупність об’єктів> → <об’єкт><сукупність об’єктів>
4. <об’єкт> → <простий об’єкт>
5. <об’єкт> → <посилання>
6. <об’єкт> → <розділювач>
7. <простий об’єкт> → <рядок символів>
8. <рядок символів> → <символ>
9. <рядок символів> → <символ><рядок символів>
10. <посилання> → [<клауза>]
11. <клауза> → <ціле>
12. <ціле> → <номер нетермінальної компоненти правої частини правила, рахуючи справа наліво>
13. <розділювач> → _
14. <розділювач> → α

Порядок слідування клауз (посилань для спуску по дереву) в семантичному визначенні задає порядок розбору.

Приклад 1.

<вираз> → <вираз><знак><терм>

[3] [2] [1]

Семантичне визначення для даного правила може бути, наприклад, таким: { [3][2][1] } або { [1][2][3] } або { [3][1][3][1][2] }.

Повторення клаузи у семантичному визначенні означає повторний спуск по піддереву вказаного нетерміналу.

Призначення розділювачів:

_ – розділення вихідного тексту в межах рядка;

α – перехід на новий рядок.

Приклад фрагменту вихідного коду на мові асемблера і відповідного йому семантичного визначення:

MOV AX,A {αMOV_AX,AαMOV_B,AX}

MOV B,AX

Приклад 2.

Для граматики ідентифікатора описати семантику, яка копіює вхід на вихід в оберненому порядку.

Граматика ідентифікатора визначена наступними правилами:

1. <ідентифікатор> → <рядок символів> {[1]}
2. <рядок символів> → <буква> {[1]}
3. < рядок символів> → < рядок символів> <буква> {[1][2]}
4. < рядок символів> → < рядок символів> <цифра> {[1][2]}

5÷30. <буква> → A| B| C| ..| Z {Buf=Buf+<буква>}

31÷40. <цифра> → 0| 1| 2| 3| 4| 5| 6| 7| 8| 9 {Buf=Buf+<цифра>}

Дерево розбору для ідентифікатора **AB1** показане на рис.24. При обробці цього дерева згідно зазначеної семантики, генератор коду на виході згенерує рядок **1BA**.

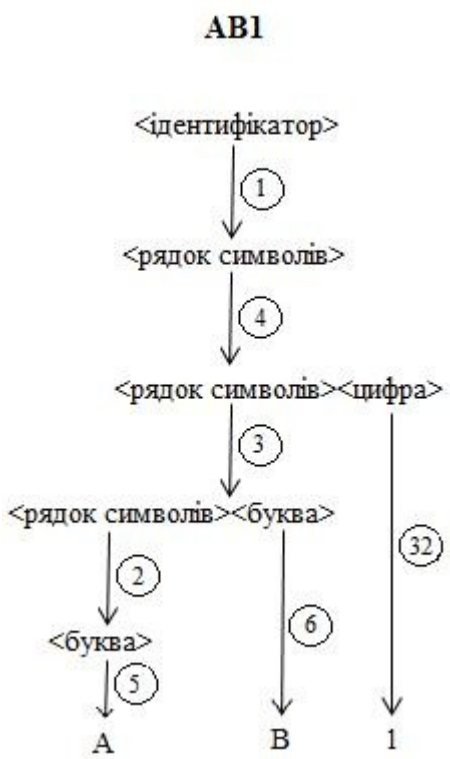


Рис.24. Дерево розбору для ідентифікатора **AB1**

Генерація коду для оператора присвоєння та арифметичних операцій

Неформальна семантика

Нехай операція присвоювання визначена тільки для цілих чисел і визначені тільки операції додавання і віднімання.

Семантичні відповідності між компонентами оператора присвоювання і командами асемблера показані у таблиці 11.

Таблиця 11.

Операції	**Команди**
:=	MOV <змінна>,AX
+	ADD AX,<змінна>
–	SUB AX,<змінна>
Отримання значення змінної	MOV AX,<змінна>

Приклад 1.

Розглянемо граматику простого оператора присвоювання:

1. <оператор> → <змінна> := <вираз>
2. <вираз> → <змінна>
3. <вираз> → <вираз> + <змінна>
4. <змінна> → a1
5. <змінна> → a2
6. <змінна> → a3
7. <змінна> → a4
8. <змінна> → a5

Нехай потрібно виконати трансляцію згідно опису наступної неформальної семантики:

1. Всі змінні мають тип integer (int);
2. Виходом повинні бути команди асемблера;
3. Результат обчислення повинен бути записаний до регістру AX;
4. Відповідність операцій і команд має бути такою:
   1) <змінна>:=<вираз> → MOV_<змінна>,AX;
   2) <змінна> → MOV_AX,<змінна>;
   3) <вираз> + <змінна> → ADD_AX,<змінна>.
   4) id (a1 | a2 | … | a5) → записати конкретний ідентифікатор до внутрішньої змінної Buf, що відповідає нетерміналу <змінна>.

Опишемо формальну семантику (семантичні визначення) для кожного правила граматики:

1. <оператор> → <змінна> := <вираз> {[1][2]αMOV_Buf,AX}
2. <вираз> → <змінна> {[1]αMOV_AX,Buf}
3. <вираз> → <вираз> + <змінна> {[2][1]αADD_AX,Buf}
4. <змінна> → a1 {Buf:=’a1’}
5. <змінна> → a2 {Buf:=’a2’}
6. <змінна> → a3 {Buf:=’a3’}
7. <змінна> → a4 {Buf:=’a4’}
8. <змінна> → a5 {Buf:=’a5’}

Розглянемо процес трансляції вхідного рядка a1 := a1 + a2 для заданої граматики.

Синтаксичний аналізатор побудує для цього рядка дерево розбору (воно, як правило, повністю зберігається в оперативній пам’яті), що показане на рис.25.

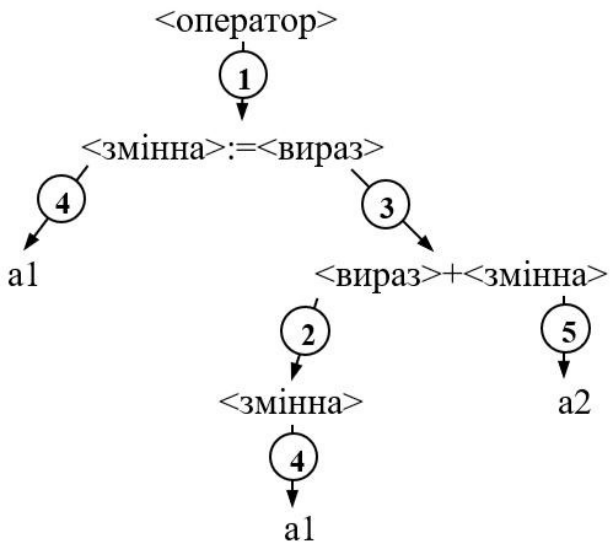


Рис.25. Дерево розбору для вхідного рядка a1 := a1 + a2

Лінійна форма для цього дерева розбору буде такою:

1	2	3	4	5	6	7	8	9
5	3	-1	2	4	1	-2	4	-6

Розглянемо процес трансляції на схемі дерева розбору (рис.26), в якій цифри у кружечках є порядковими номерами дій процесу генерації коду, а не номерами правил, як на попередніх схемах.

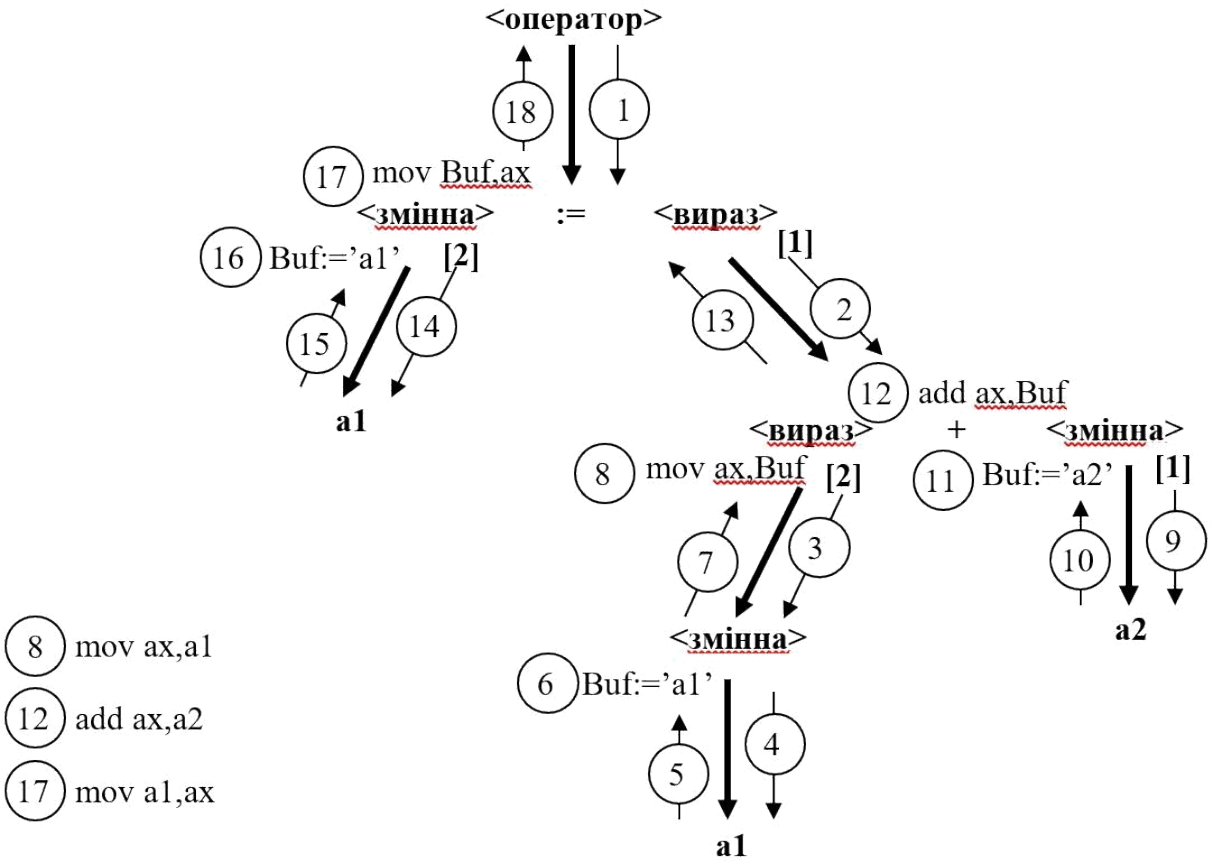


Рис. 26. Схема процесу трансляції на дереві розбору

Розглянемо процес трансляції ще на одній схемі дерева розбору (рис.27), в якій замість правил граматики в вузлах стоять відповідні їм семантичні визначення.

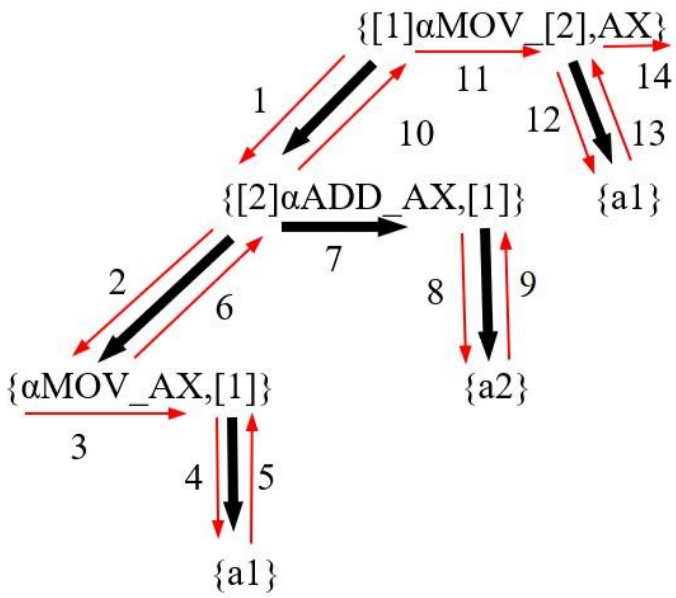


Рис.27. Схема трансляції на дереві семантичних визначень

Приклад 1. Розглянемо семантику і генерацію коду для трохи більш складної граматики оператора присвоювання:

1. <оператор> → <змінна> := <вираз>
2. <вираз> → < змінна>
3. <вираз> → <вираз> <зод> <змінна>
4. <змінна> → a1
5. <змінна> → a2
6. <змінна> → a3
7. <змінна> → a4
8. <змінна> → a5
9. <зод> → +
10. <зод> → –

Дерево розбору для вхідного рядка **a1 := a2 + a1 – a3** показане на рис. 28.

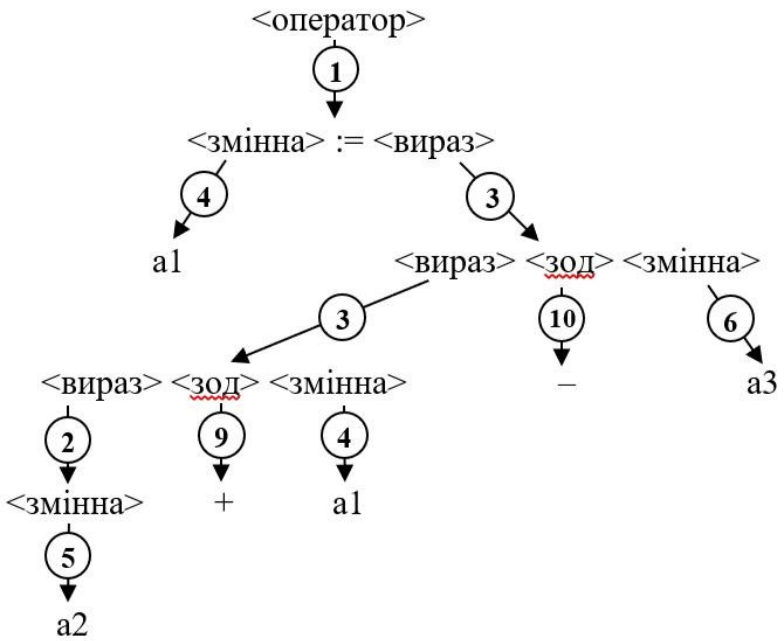


Рис.28. Дерево розбору для вхідного рядка **a1 := a2 + a1 – a3**

Відповідна цьому дереву лінійна форма має такий вигляд.

LRAS

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
9	4	10	6	3	-4	-3	3	-2	-1	2	5	1	-5	4	-13

В результаті роботи генератора коду по цьому дереву розбору буде згенерований такий асемблерний код

```
MOV  AX,a2
ADD  AX,a1
SUB  AX,a3
MOV a1,AX
```

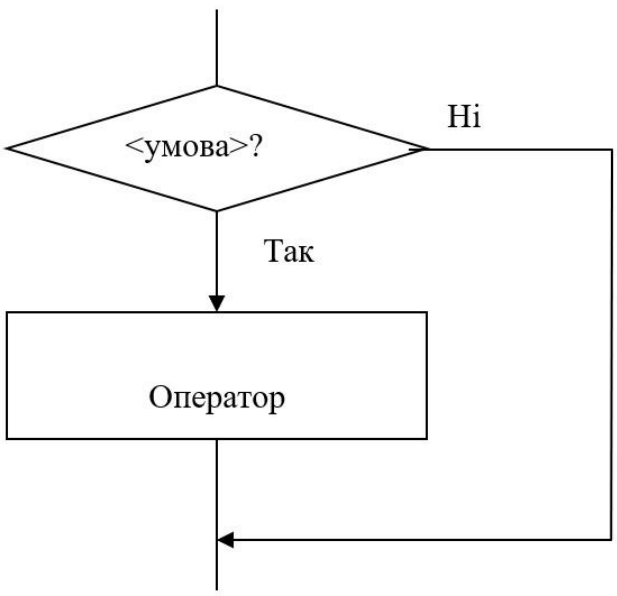
Генерація коду для конструкцій розгалуження

Генерація коду для неповної умовної конструкції if-then

Синтаксис:

1. <умовна конструкція> → if <умова> then <оператор>
2. <умова> → <вираз1><операція порівняння><вираз2>
3. <операція порівняння> → >
4. <операція порівняння> → >=
5. <операція порівняння> → <
6. <операція порівняння> → <=
7. <операція порівняння> → =
8. <операція порівняння> → !=

Неформальна семантика:



Команди умовного переходу мови асемблера Intel, що відповідають операціям порівняння:

>	→	JG	<мітка>
	→	JGE	<мітка>
<	→	JL	<мітка>
	→	JLE	<мітка>
=	→	JE	<мітка>
!=	→	JNE	<мітка>

Приклад. Розглянемо семантичне визначення для неповної умовної конструкції **if-then**.

if a>b then x:=y

Припустимо, що всі змінні мають тип integer. Тоді повинен бути згенерований такий асемблерний код

```
        MOV   AX,a
        MOV   BX,b        } a>b
        CMP   AX,BX       }
        JLE   ?L0         }, якщо a<=b, то перехід ?L0
        MOV   AX,y        }
                          } x:=y
        MOV   x,AX        }
?L0:    NOP
```

Зовнішні мітки – це мітки, які програміст вказує в тексті вхідної програми.

Внутрішні мітки не вказуються в початковому тексті, а генеруються компілятором для реалізації управляючих конструкцій.

? – символ в іменах асемблера Intel

Структура внутрішньої мітки може бути, наприклад, такою:

?<буква><рядок цифр>

'?'+'L'+'0' → ?L0

'? ' + 'M' + '21' → ?M21

Розглянемо порядок обходу дерева розбору і генерації коду для умовної конструкції if-then:

1. Спуск по піддереву нетерміналу <умова> правила 1 (* SPR(k2) *) під час якого генеруються команди:

   MOV AX,<вираз1>

   MOV BX,<вираз2>

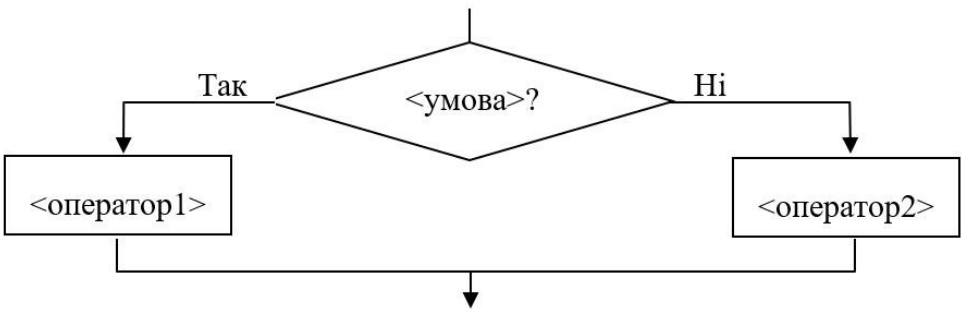
   CMP AX,BX
2. Генерація внутрішньої мітки, наприклад, ?L0 .
3. Генерація команди умовного переходу за умовою, оберненою заданій, на внутрішню мітку: JLE ?L0
4. Спуск по піддереву нетерміналу <оператор> правила 1 (* SPR(k1) *), під час якого генерується команди реалізації цього оператора.
5. Генерація команди NOP з внутрішньою міткою ?L0: ?L0: NOP

Генерація коду для повної умовної конструкції if-then-else

Синтаксис:

1. <умовна конструкція> → if <умова> then <оператор1> else <оператор2>
2. <умова> → <вираз1><операція порівняння><вираз2>
3. <операція порівняння> → >
4. <операція порівняння> → >=
5. <операція порівняння> → <
6. <операція порівняння> → <=
7. <операція порівняння> → =
8. <операція порівняння> → !=

Неформальна семантика:



Приклад. Розглянемо семантичне визначення для повної умовної конструкції **if-then-else**.

if a>b then x:=y

else y:=x

Згенерований асемблерний код буде таким

```
        MOV   AX,a
        MOV   BX,b      } a>b
        CMP   AX,BX
        JLE   ?L1
        MOV   AX,y      } x:=y
        MOV   x,AX
        JMP   ?L2
?L1:    NOP
        MOV   AX,x      } y:=x
        MOV   y,AX
?L2:    NOP
```

Розглянемо порядок обходу дерева розбору і генерації коду для повної умовної конструкції if-then-else (рис.29):

<умовна конструкція> → if <умова> then <оператор>else<оператор>

{[3] Jxx ?L1 [2] JMP ?L2 ?L1:NOP [1] ?L2:NOP}

<умова> → <вираз><операція порівняння><вираз> {Reg:=’AX’ [3] Reg:=’BX’ [1] [2] CMP AX,BX}

<операція порівняння> → >

<операція порівняння> → >=

<операція порівняння> → <

<операція порівняння> → <=

<операція порівняння> → =

<операція порівняння> → <>

{Zn=’символи операції’}

<вираз> → <змінна> {[1] MOV Reg,BF}

<змінна> → a

<змінна> → b

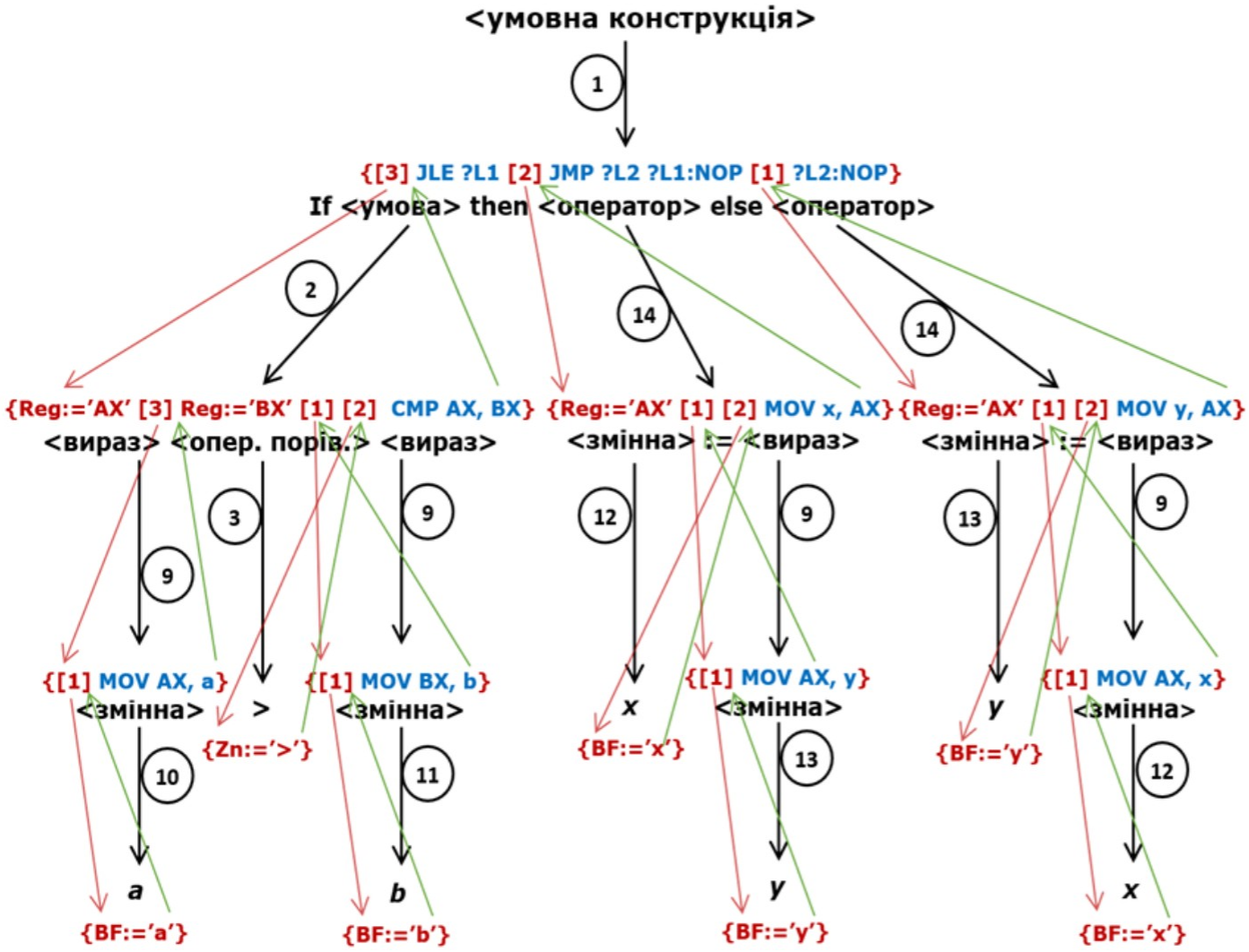
<змінна> → x

<змінна> → y

{BF:=’змінна’}

<оператор> → <змінна>:=<вираз> {Reg:=’AX’ [1] [2] MOV BF,Reg}

1. Спуск по піддереву <умова>, під час якого генеруються команди:
   ```
   MOV AX,<вираз1>
   MOV BX,<вираз2>
   CMP AX,BX
   ```
2. Генерація внутрішньої мітки, наприклад, **?L1** .
3. Генерація команди переходу на внутрішню мітку **?L1**, наприклад, **JLE ?L1**
4. Вибір команди порівняння залежить від <операції порівняння>.
5. Спуск по піддереву <оператор1> гілки then, під час якого генеруються команди реалізації цього оператора:
   ```
   MOV   AX,y
   MOV   x,AX
   ```
6. Генерація внутрішньої мітки для виходу із умовного оператора, напр. **?L2**.
7. Генерація команди безумовного переходу **JMP** на мітку **?L2, JMP ?L2**
8. Генерація команди **NOP** з внутрішньою міткою **?L1, ?L1: NOP**
9. Спуск по піддереву <оператор2> гілки else, з генерацією його команд :
   ```
   MOV AX,x
   MOV y,AX
   ```
10. Генерація команди **NOP** з внутрішньою міткою **?L2, ?L2: NOP**




```
        MOV  AX,a
        MOV  BX,b
        CMP  AX,BX      } a>b
        JLE  ?L1
        MOV  AX,y
        MOV  x,AX       } x:=y
        JMP  ?L2
?L1:    NOP
        MOV  AX,x
        MOV  y,AX       } y:=x
?L2:    NOP
```

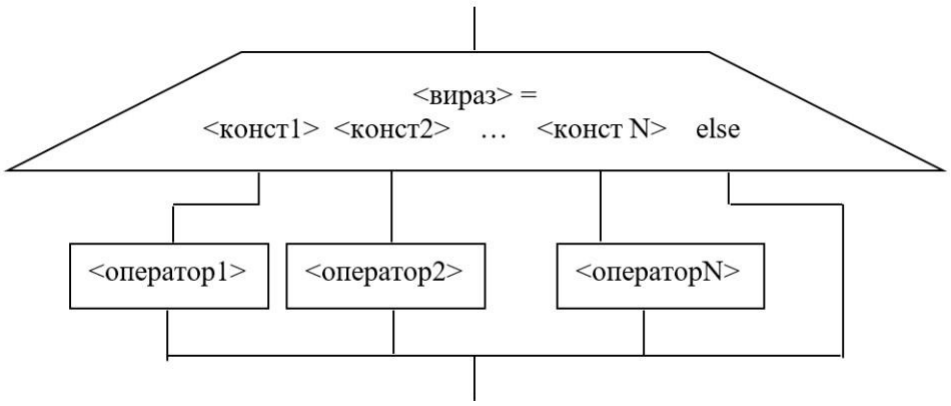
Рис. 29. Генерація коду для повної умовної конструкції **if-then-else**

Генерація коду для конструкції вибору

Синтаксис:

1. <конструкція case> → case <вираз> of <список альтернатив>
   <альтернатива else> end
2. <список альтернатив> → <список альтернатив> ; <альтернатива>
3. <список альтернатив> → <альтернатива>
4. <альтернатива> → <константа> : <оператор>
5. <альтернатива else> → <оператор>

Неформальна семантика:



Приклад.

Розглянемо семантичне визначення для конструкції вибору case такого вигляду:

case x of
  101: <оператор1>;
  201: <оператор2>;
  301: <оператор3>
end;

Для генерації команд, відповідних конструкції case, необхідний попередній прохід по піддереву списку альтернатив для того, щоб:

1) визначити кількість альтернатив;

2) сформувати масив, що містить мітки констант.

	1	2	3
CaseConst	’101’	’201’	’301’

Власне генерація команд, відповідних кожній альтернативі, здійснюється на другому проході по дереву розбору.

Згенерований код може бути, наприклад, таким:

```
        MOV     AX,x
        CMP     AX,101      ┐
        JE      ?101        ┘  x = 101 ?
        CMP     AX,201      ┐
        JE      ?201        ┘  x = 201 ?
        CMP     AX,301      ┐
        JE      ?301        ┘  x = 301 ?
        JMP     ?L1
?101:   <оператор1>
        JMP     ?L1
?201:   < оператор2>
        JMP     ?L1
?301:   < оператор3>
?L1:    NOP
```

Розглянемо порядок обходу дерева розбору і генерації коду для конструкції вибору case:

1. Спуск по піддереву <вираз>, під час якого генеруються команди обчислення цього виразу (результат записується у AX):

```
MOV   AX,x
```

2. Перший прохід по піддереву <список альтернатив> (* Fl:=true;*).

Під час першого проходу визначається кількість альтернатив і формується масив CaseConst.

3. Цикл генерації команд порівняння значення <виразу> з елементами масиву CaseConst:

```
CMPAX,101
JE ?101 CMP
AX,201
JE      ?201
CMP     AX,301
JE      ?301
```

4. Генерація команди переходу на оператор, наступний за case оператором:

```
JMP  ?L1
```

5. Другий прохід по піддереву <список альтернатив> (* Fl:=false; *).

Під час другого проходу виконується генерація команд <оператора> для кожної альтернативи і подальша генерація команди JMP ?L1 для переходу на оператор, наступний за case оператором:

```
?101:    <оператор1>
         JMP    ?L1
?201:    < оператор2>
         JMP    ?L1
?301:    < оператор3>
```

6. Генерація команди NOP, з внутрішньою міткою ?L1:

```
?L1:NOP
```

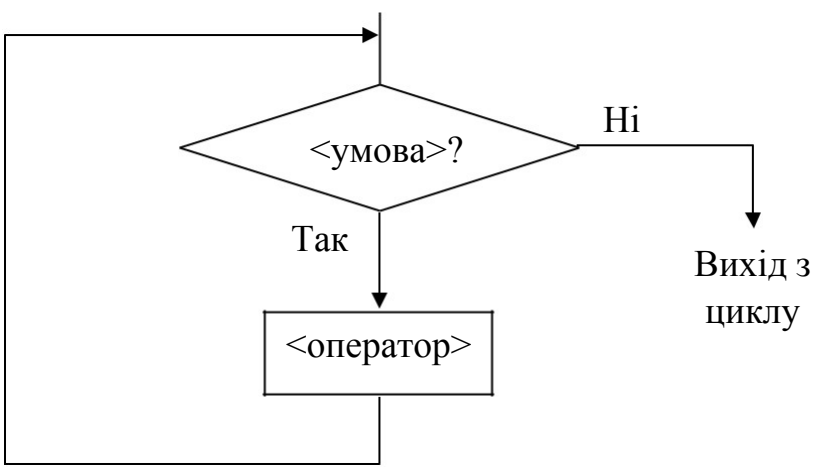
Генерація коду для циклічних конструкцій

Генерація коду для конструкції циклу з передумовою while

Синтаксис:

1. <цикл-while> → while <умова> do <оператор>
2. <умова> → <вираз1><операція порівняння><вираз2>
3. <операція порівняння> → >
4. <операція порівняння> → >=
5. <операція порівняння> → <
6. <операція порівняння> → <=
7. <операція порівняння> → =
8. <операція порівняння> → <>

Неформальна семантика:



Розглянемо семантичне визначення для конструкції циклу з передумовою while на прикладі наступного оператора:

while a>b do begin x:=y; y:=z end

```
?L1:    NOP
        MOV   AX,a
        MOV   BX,b      } a>b
        CMP   AX,BX
        JLE   ?L2
        MOV   AX,y      } x:=y
        MOV   x,AX
        MOV   AX,z      } y:=z
        MOV   y,AX
        JMP   ?L1
?L2:    NOP
```

Розглянемо порядок обходу дерева розбору і генерації коду для конструкції циклу з передумовою while:

1. Генерація внутрішньої мітки, наприклад, ?L1 .
2. Генерація команди NOP з внутрішньою міткою ?L1: ?L1: NOP
3. Спуск по піддереву <умова>, під час якого генеруються команди обчислення цієї умови:

```
MOV   AX,<вираз1>
MOV BX,<вираз2>
CMPAX,BX
```

4. Генерація внутрішньої мітки для виходу із циклу, наприклад, ?L2 .
5. Генерація команди умовного переходу на внутрішню мітку ?L2 за умовою, оберненою заданій, наприклад, JLE ?L2.
6. Спуск по піддереву <оператор>, під час якого генеруються команди реалізації цього оператора:

```
MOV   AX,y
MOV   x,AX
MOV   AX,z
MOV y,AX
```

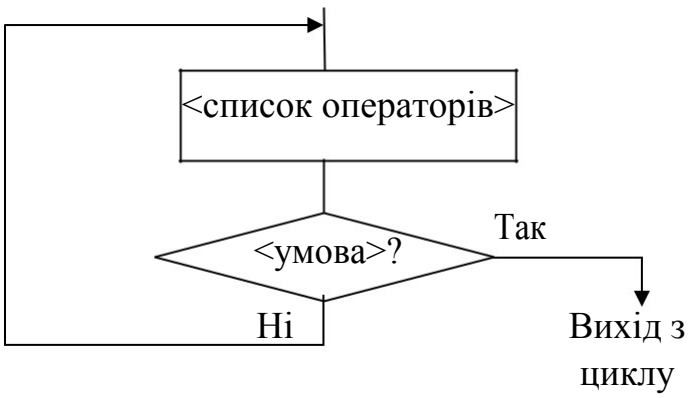
7. Генерація команди безумовного переходу JMP на початок циклу, тобто на мітку ?L1: JMP ?L1
8. Генерація команди NOP з внутрішньою міткою ?L2: ?L2: NOP

Генерація коду для конструкції циклу з післяумовою do-while

Синтаксис:

1. <цикл do-while> → do <список операторів> while <умова>
2. <умова> → <вираз><операція порівняння><вираз>
3. <список операторів> → <оператор>
4. <список операторів> → <список операторів>;<оператор>
5. < операція порівняння > → >
6. < операція порівняння > → >=
7. < операція порівняння > → <
8. < операція порівняння > → <=
9. < операція порівняння > → =
10. < операція порівняння > → <>

Неформальна семантика:



Розглянемо семантичне визначення для конструкції циклу з післяумовою do-while на прикладі наступного оператора:

do x:=y; b:=b+1 while a>b

```
?L1:    NOP
        MOV   AX,y      ┐
        MOV   x,AX      │
        MOV   AX,b      ├  x:=y;  b:=b+1
        ADD   AX,1      │
        MOV   b,AX      ┘
        MOV   AX,a      ┐
        MOV   BX,b      ├  a>b
        CMP   AX,BX     ┘
        JG    ?L1
```

Розглянемо порядок обходу дерева розбору і генерації коду для конструкції циклу з післяумовою do-while:

1. Генерація внутрішньої мітки, наприклад ?L1 .
2. Генерація команди NOP з внутрішньою міткою початку циклу ?L1, тобто ?L1: NOP
3. Спуск по піддереву <список операторів>, під час якого генеруються команди реалізації цього списку операторів:

```
MOV   AX,y
MOV   x,AX
MOV   AX,b
ADD   AX,1
MOV b,AX
```

4. Спуск по піддереву <умова>, під час якого генеруються команди реалізації цієї умови:

```
MOV   AX,a
MOV BX,b
CMPAX,BX
```

5. Генерація команди умовного переходу з внутрішньою міткою ?L1 на початок циклу:

```
JG?L1
```

Генерація коду для конструкції циклу з лічильником (параметром) for

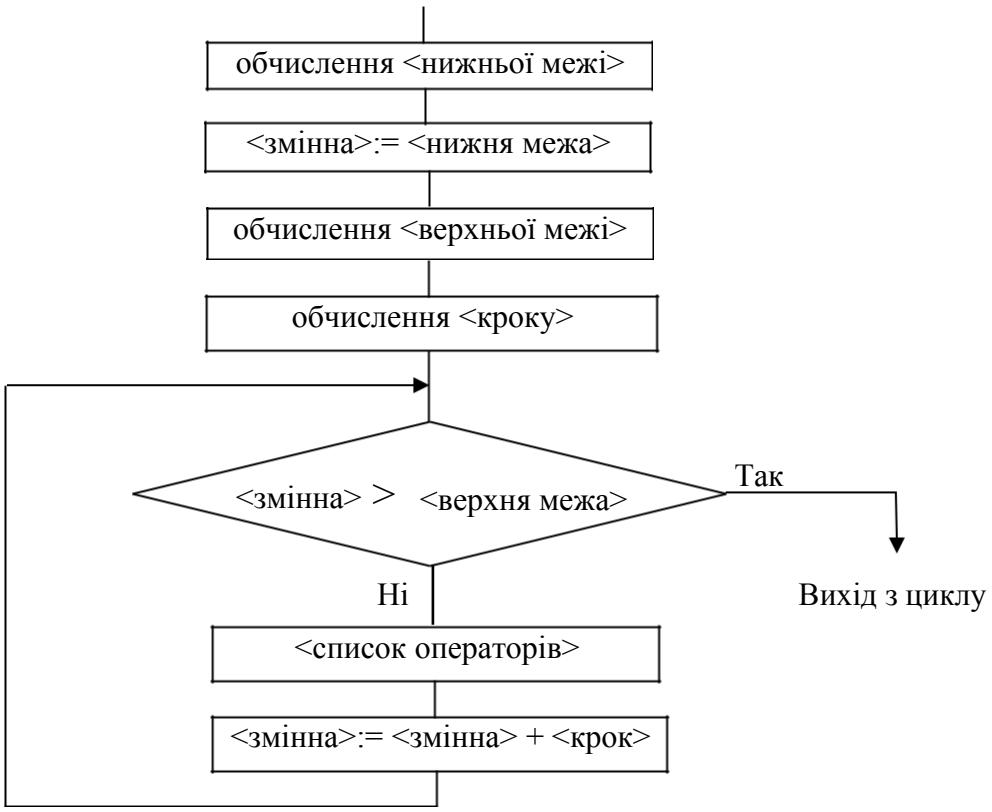
Синтаксис:

1. <цикл for> → for <змінна>:= <нижня межа> to <верхня межа> by <крок> do <оператор>
2. <нижня межа> → <вираз>
3. <верхня межа> → <вираз>
4. <крок> → <вираз>

При реалізації компілятора приймається ряд внутрішніх угод (погоджень) для нього.

Однією з внутрішніх угод для конструкції циклу for приймемо таку: кількість повторень циклу зберігатиметься в регістрі CX.

Неформальна семантика (якщо крок додатний):



Розглянемо семантичне визначення для конструкції циклу з лічильником (параметром) for наступного вигляду:

for i:= LowBound to HighBound by Step do <оператор>

```
        MOV   AX,LowBound   ; <нижня межа> → у AX
        MOV   i,AX          ; <нижня межа> через AX → в i
        MOV   AX,HighBound  ; <верхня межа> → у AX
        CMP   AX,i    ; порівняння (віднімання) верхньої та нижньої меж
        JL    ?L1     ; якщо <верхня межа> < <нижня межа>, то вихід з циклу
        MOV   DX,Step       ; <крок> → у DX
        MOV   .S+0,DX       ; <крок> → у робочу комірку.
; обчислити кількість ітерацій циклу (<верхня межа> –<нижня межа>) /<крок>+1
        SUB   AX,i    ; <верхня межа> – <нижня межа> → у AX
        CWD           ; розширити AX до DX:AX
        DIV   .S+0    ; (<верхня межа> – <нижня межа>) / <крок> → у AX
        INC   AX      ; (<верхня межа> – <нижня межа>) / крок> +1 → у AX
        MOV   CX,AX ; кількість повторень → у CX
?L2:    NOP
        <оператор>  ; команди, що були згенеровані при обробці <оператора>
        MOV   AX,i
        ADD   AX,.S+0   нарощування змінної i на величину <кроку>
        MOV   i,AX
        LOOP  ?L2
?L1:    NOP
```

Для зручності розгляду наступної частини пояснень наведемо синтаксис оператора for ще раз.

1. <цикл for> → for <змінна>:= <нижня межа> to <верхня межа> by <крок> do <оператор>
2. <нижня межа> → <вираз>
3. <верхня межа> → <вираз>
4. <крок> → <вираз>

Порядок обходу дерева розбору і генерації коду для конструкції циклу з лічильником (параметром) for:

1. Спуск по піддереву <нижня межа>
2. Спуск по піддереву <змінна>
3. Генерація команди пересилки <нижня межа> → <змінна>

```
MOV AX,LowBound
MOV   i,AX
```

4. Спуск по піддереву <верхня межа>
5. Генерація команди завантаження <верхньої межі>

```
MOV     AX,HighBound
```

6. Генерація команди порівняння і команди умовного переходу з внутрішньою міткою ?L1 для виходу з циклу

```
CMP     AX,i
JL      ?L1
```

7. Спуск по піддереву <крок>
8. Генерація команд пересилки <кроку> в робочу комірку

```
MOV DX,2
MOV .S+0,DX
```

9. Генерація команд розрахунку числа повторень циклу

```
SUBAX,i
CWD
DIV.S+0
INC     AX
```

10. Пересилка числа повторень в CX

```
MOV     CX,AX
```

11. Генерація команди NOP з внутрішньою міткою ?L2 (початок тіла циклу)

```
?L2:    NOP
```

12. Спуск по піддереву <оператор>, де генеруються команди його реалізації
13. Генерація команд зміни значення змінної циклу на величину

```
<кроку> MOV AX,i
ADD     AX,.S+0
MOV     i,AX
```

14. Генерація команди LOOP з переходом на початок циклу

```
LOOP ?L2
```

15. Генерація команди NOP з міткою виходу з циклу.

```
?L1:    NOP
```

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

Основна література

1. Ахо А., Сети Р., Ульман Д. Компиляторы: принципы, технологии и инструменты.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003, – 768 с. : ил.

2. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. В 2-х томах. – М.: Мир, 1978. – 1105 с.

3. Грисс Д. Конструирование компиляторов для цифровых вычислительных машин. – М.: Мир, 1975. – 544 с.

4. Д.Э.Кнут. Нисходящий синтаксический анализ. В кн.: Кибернетический сборник. Выпуск 15. – М.: Мир, 1978. – с.101-142.

5. Бек Л. Введение в системное программирование. – М.: Мир, 1988. – 448с.

6. Опалева Э.А., Самойленко В.П. Языки программирования и методы трансляции. – СПб: БХВ-Петербург, 2005. – 476 с.

7. Компаниец Р.И., Маньков Е.В., Филатов Н.Е. Системное программирование. Основы построения трансляторов./Учебное пособие для высших и средних учебных заведений – СПб.: КОРОНА принт, 2000. – 256 с.

8. Льюис Ф., Розенкранц Д., Стирнз Р. Теоретические основы проектирования компиляторов. – М.: Мир, 1979. – 656 с.

9. Касьянов В.Н., Поттосин И.В. Методы построения трансляторов. – Новосибирск: Наука, 1986. – 344 с.

Додаткова література

10. Глушков В.М., Цейтлин Г.М., Ющенко Е.Л. Алгебра, языки, программирование. – К.: Наукова думка, 1978. – 320 с.

11.Янг С. Алгоритмические языки реального времени. Конструирование и разработка. – М.: Мир, 1985. – 400 с.

12. Ингерман П. Синтаксически ориентированный транслятор. – М.: Мир, 1969. – 176 с.

13. Маккиман У., Хорнинг Дж., Уортман Д. Генератор компиляторов. – М.: Статистика, 1980. – 528 с.

14. Языки и автоматы. Сборник переводов статей. – М.: Мир, 1975. – 360 с.

15. Кибернетический сборник. Выпуск 15. – М.: Мир, 1978. – 224 с.