МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
Національний технічний університет
„Харківський політехнічний інститут”
Кафедра „Комп’ютерна інженерія та програмування”

Методичні вказівки до виконання курсового проекту з курсу **«Теорія побудови компіляторів»**

для студентів денної та заочної форм навчання
спеціальності 123 «Комп’ютерна інженерія»

Затверджено редакційно-
видавничою радою НТУ «ХПІ»,
протокол № 2 від 27 червня 2024р.
п.561

Харків НТУ «ХПІ» 2024

Методичні вказівки до виконання курсового проекту з курсу «Теорія побудови компіляторів» для студентів денної та заочної форм навчання спеціальності 123 «Комп'ютерна інженерія», / Уклад.: С.Ю. Гавриленко, В.В. Челак. – Харків: НТУ «ХПІ», 2024. – 26 с.

Укладачі: С.Ю. Гавриленко, В.В. Челак

Рецензент В.С. Бреславець

Кафедра комп'ютерної інженерії та програмування

ЗМІСТ

ВСТУП

Дослідники, що вивчають питання появи розуму на нашій планеті, вважають, що вирішальну роль у його розвитку зіграла поява мови, що дозволила не тільки виражати і зберігати знання, але й обмінюватися ними.

Зі створенням комп'ютерів виникла потреба в спілкуванні з подібними пристроями, оскільки виявилося необхідним передавати їм накази, завдання й опис роботи, що вони повинні виконувати. Для цієї мети почали розробляти спеціальні мови, що стали називати штучними на відміну від природних мов спілкування людей. Штучні мови повинні бути, з одного боку, зручними і зрозумілими для людини, а з іншого боку – повинні сприйматися пристроями. Сполучення цих вимог в одній мові виявилося важкою задачею, тому з'явилися засоби для перетворення текстів з мови, зрозумілої людині, на мову пристрою. Такі засоби назвали **трансляторами**.

Транслятор може бути **інтерпретуючого** чи компілюючого типу. У першому випадку його називають **інтерпретатором вхідної мови**, а в другому – **компілятором**.

Інтерпретатор послідовно читає пропозиції вхідної мови, аналізує їх і відразу ж виконує, а компілятор не виконує пропозиції мови, а будує програму, що може надалі бути запущена для одержання результату.

На вхід компілятора подається текст, написаний вхідною мовою, що зрозуміла людині, а результатом роботи компілятора є текст мовою, що зрозуміла пристрою.

У даних методичних вказівках розглянуто побудову синтаксичного *LR*-аналізатора, який є однією із стадій роботи компілятора. Саме на стадії синтаксичного аналізу виявляється найбільша кількість помилок в тексті програми.

1. ЗАВДАННЯ ДО ВИКОНАННЯ КУРНОГО ПРОЕКТУ

1.1. Мета проекту

Побудова синтаксичного *LR*-аналізатора для заданого фрагменту програми.

1.2. Порядок виконання проекту

1.1.1. Залежно від номера прізвища за списком вибрати область індивідуального завдання (п.1.3). Узгодити з викладачем фрагмент програми для аналізу відповідно до індивідуального завдання.

1.1.2. Побудувати правила граматики. Перевірити, чи не містить граматика непродуктивні та недосяжні символи.

1.1.3. Побудувати множину ВПЕРШ та ВПІСЛЯ і, якщо необхідно, функцію СЛІД.

1.1.4. Побудувати таблицю переходів.

1.1.5. Побудувати керувальну таблицю.

1.1.6. Перевірити роботу розпізнавача експериментально.

1.1.7. Написати та налаштувати текст програми розпізнавача. Мова програмування та середовище обирається студентом самостійно.

1.1.8. Створити зручний інтерфейс для роботи програми.

1.1.9. Перевірити роботу програми на прикладі.

1.1.10. Оформити альбом документів згідно з п. 1.4.

1.3. Умови задач

1. Змінні цілого типу. Оператор вводу.

```
Приклад 1.
      int i = int_constant ;
      cin >> i ;
Приклад 2.
      int i = int_constant , i = int_constant ;
      cin >> i >> i;
Приклад 3.
      int i = int_constant , i = int_constant ;
      cin >> i >> i >> i;
```

2. Змінні символьного типу. Оператор виводу.

```
Приклад 1.
      char i = char_constant ;
```

```
      cout << i ;
Приклад 2.
      char i = char_constant , i = char_constant ;
      cout << i << i;
Приклад 3.
      char i = char_constant , i = char_constant ;
      cout << i << i << i;
```

3. Масиви чисел.

```
Приклад 1.
      int i[d] = {d, d, d, d, d, d, d};
Приклад 2.
      int i[] = {};
Приклад 3.
      int i[d] = {d, d, d};
```

4. Оператор циклу з параметром.

```
Приклад 1.
      for ( type i = expr ; expr ; expr )
          stmt ;
Приклад 2.
      for ( type i = expr ; expr ; expr ) {
          stmt ;
          stmt ;
      }
Приклад 3.
      for ( type i = expr ; expr ; expr ) {
          stmt ;
          stmt ;
          stmt ;
      }
```

5. Оператор циклу с передумовою.

```
Приклад 1.
      while ( expr )
          stmt ;
Приклад 2.
      while ( expr ) {
          stmt ;
          stmt ;
          stmt ;
      }
Приклад 3.
      while ( expr ) {
          stmt ;
          stmt ;
          stmt ;
      }
```

6. Оператор циклу з післяумовою.

```
Приклад 1.
      do
          stmt ;
      while ( expr ) ;
Приклад 2.
      do {
          stmt ;
          stmt ;
          stmt ;
      } while ( expr ) ;
```

Приклад 3.

```
do {
    stmt ;
    stmt ;
} while ( expr ) ;
```

7. Умовний оператор.

Приклад 1.

```
if ( expr ){
    stmt ;
}
```

Приклад 2.

```
if ( expr ){
    stmt ;
}
else {
    stmt ;
    stmt ;
    stmt ;
}
```

Приклад 3.

```
if ( expr ){
    stmt ;
    stmt ;
}
else if ( expr ){
    stmt ;
    stmt ;
}
else if ( expr ){
    stmt ;
}
else {
    stmt ;
    stmt ;
    stmt ;
}
```

8. Оператор вибору.

Приклад 1.

```
switch ( expr ) {
    case constant :
        stmt ;
        stmt ;
}
```

Приклад 2.

```
switch ( expr ) {
    default :
        stmt ;
}
```

Приклад 3.

```
switch ( expr ) {
    case constant :
        stmt ;
        stmt ;
        stmt ;
    case constant :
        stmt ;
        stmt ;
    default :
        stmt ;
}
```

9. Оператор присвоювання арифметичного виразу, до складу якого входять: ідентифікатори, операції “+”, “-“ ,”*”,”\”, вкладені дужки.

```
Приклад 1.
      i = i + ( i - i ) ;
Приклад 2.
      i = i * ( i + ( i / i ) - i ) ;
Приклад 3.
      i = i + ( i + i ) ;
```

10. Оператор присвоювання арифметичного виразу, до складу якого входять: ідентифікатори, операція “+”, дужки, математичні функції.

```
Приклад 1.
      i = i + ( i + i ) ;
Приклад 2.
      i = sin ( i ) ;
Приклад 3.
      i = sqrt ( sin ( i ) + cos ( i ) ) ;
```

11. Масиви вказівників.

```
Приклад 1.
      type* i[d];
Приклад 2.
      type*** i[d];
      type** i[d];
      type*** i[d];
Приклад 3.
      type** i[d];
      type* i[d];
      type*** i[d];
      type* i[d];
```

12. Змінні дійсного типу.

```
Приклад 1.
      float i = real_constant ;
Приклад 2.
      double i = real_constant ;
Приклад 3.
      float i = real_constant, i = real_constant ;
```

13. Строки. Оператори вводу строк.

```
Приклад 1.
      string i, i, i, i;
      getline(cin, i);
      getline(cin, i);
      getline(cin, i);
Приклад 2.
      string i;
      getline(cin, i);
Приклад 3.
      string i, i;
      getline(cin, i);
      getline(cin, i);
```

14. Строки. Вбудовані функції роботи зі строками.

```
Приклад 1.
      const char * i = string_constant ;
      strlen ( i ) ;
```

Приклад 2.
```
const char * i = string_constant , * i = string_constant ;
strcmp ( i , i ) ;
```
Приклад 3.
```
const char * i = string_constant , * i = string_constant ;
strlen ( i ) ;
strcmp ( i , i ) ;
strlen ( i ) ;
strlen ( i ) ;
```

15. Функції.

Приклад 1.
```
type i(type i, type i) {
   stmt ;
}
```
Приклад 2.
```
type i(type i, type i, type i) {
   stmt ;
   stmt ;
   stmt ;
   stmt ;
}
```
Приклад 3.
```
type i( ) {
   stmt ;
   stmt ;
}
```

16. Структури (записи).

Приклад 1.
```
struct type {
   type i ;
} ;
```
Приклад 2.
```
struct type {
   type i ;
   type i ;
} ;
```
Приклад 3.
```
struct type {
   type i ;
   type i ;
   type i ;
} ;
```

17. Файли. Введення та виведення даних в файл.

Приклад 1.
```
ifstream in;
in.open(i);
in >> i >> i >> i;
in.close();
```
Приклад 2.
```
ofstream out;
out.open(i);
out << i << i << i;
out.close();
```
Приклад 3.
```
ifstream in;
in.open(i);
in >> i;
in.close();
```

18. Файли. Функції роботи з файлами.

Приклад 1.
```
FILE * i = fopen ( string_constant , string_constant ) ;
fclose ( i ) ;
```
Приклад 2.
```
FILE * i = fopen ( string_constant , string_constant ) ;
fputs ( i , string_constant ) ;
fgets ( i , i , i ) ;
fclose ( i ) ;
```
Приклад 3.
```
FILE * i = fopen ( string_constant , string_constant ) ;
fputs ( i , string_constant ) ;
fputs ( i , string_constant ) ;
fputs ( i , string_constant ) ;
fclose ( i ) ;
```

19.Односпрямовані списки.

Приклад 1.
```
struct node {
   type i ;
   struct node * next ;
};
```
Приклад 2.
```
struct node {
   struct node * next ;
   type i ;
};
```
Приклад 3.
```
struct node {
   type i ;
   type i ;
   struct node * next ;
   type i ;
};
```

20. Двоспрямовані списки.

Приклад 1.
```
struct node {
   type i ;
   struct node * prev , * next ;
};
```
Приклад 2.
```
struct node {
   struct node * prev , * next ;
   type i ;
};
```
Приклад 3.
```
struct node {
   type i ;
   type i ;
   struct node * prev , * next ;
   type i ;
};
```

21. Використання системних та власних модулів (файлів).

Приклад 1.
```
#include<name>
#include"name.h"
#include<name>
#include<name>
```

Приклад 2.
```
#include"name.h"
#include"name.h"
#include<name>
#include"name.h"
```
Приклад 3.
```
#include<name>
```

22. Об’єкти (класи). Описання конструкцій.

Приклад 1.
```
class type {
} ;
```
Приклад 2.
```
class type {
    type i ;
    type i ;
} ;
```
Приклад 3.
```
class type {
    type i ;
    type i ;
    type i ;
    type i ;
} ;
```

23. Об’єкти (класи). Успадкування.

Приклад 1.
```
class type {};
class type : type : type {};
```
Приклад 2.
```
class type {};
class type : type {};
class type : type {};
class type : type {};
```
Приклад 3.
```
class type {};
class type : type {};
```

24. Константи.

Приклад 1.
```
const type i = expr ;
```
Приклад 2.
```
const type i = expr , i = expr ;
```
Приклад 3.
```
const type i = expr , i = expr , i = expr ;
```

25. Об’єкти (класи). Поліморфізм.

Приклад 1.
```
class_type i = (class_type)i;
class_type i = (class_type)(class_type)i;
```
Приклад 2.
```
class_type i = (class_type)(class_type)(class_type)i;
```
Приклад 3.
```
class_type i = (class_type)i;
class_type i = (class_type)i;
class_type i = (class_type)i;
```

26. Графіка. Використання графічних функцій.

Приклад 1.
```
LineTo ( i , i , i );
```

```
Приклад 2.
      Rectangle ( i , i , i , i , i );
      Ellipse ( i , i , i , i , i );
Приклад 3.
      LineTo ( i , i , i );
      LineTo ( i , i , i );
      Ellipse ( i , i , i , i , i );
      Rectangle ( i , i , i , i , i );
      Ellipse ( i , i , i , i , i );
      Rectangle ( i , i , i , i , i );
      LineTo ( i , i , i );
```

27. Унарні та бінарні операцій.

```
Приклад 1.
      i + i + i + i ^ i + ~i;
Приклад 2.
      -i + -i - i / ~i;
Приклад 3.
      ~i | i | i & i + -i * i / i;
```

28. Математичні функції.

```
Приклад 1.
      sin ( i ) ;
Приклад 2.
      cos ( i ) ;
      sin ( i ) ;
      cos ( i ) ;
Приклад 3.
      sin ( i ) ;
      cos ( i ) ;
      atan2 ( i , i );
      sqrt ( i ) ;
      pow ( i , i );
```

29. Масиви символів.

```
Приклад 1.
      char i[d] = "sssssss";
Приклад 2.
      char i[d] = {'s', 's', 's', 's', '\0'};
Приклад 3.
      char i[d] = {'s', '\0'};
      char i[d] = "ss";
      char i[d] = {'\0'};
      char i[d] = "sss";
```

30. Формати виведення.

```
Приклад 1.
      cout << fixed << i ;
Приклад 2.
      cout << fixed << i << i ;
Приклад 3.
      cout << scientific << i << fixed << i ;
```

1.4. Вимоги до оформлення звіту з виконання курсового проєкту

Курсовий проєкт оформляється у вигляді звіту. Звіт з виконання курсового проєкту містить наступні структурні елементи:

- Титульний аркуш (1 сторінка).

- Зміст (1 сторінка).
- Вступ (до 1 сторінки).
- Проєктування LR(1)-розпізнавача (5-10 сторінок, таблиці переходів та управляюча, правила граматики, опис виконання всіх дій, які треба зробити для виконання 1.2.1-1.2.6).
- Розробка симулятора LR(1)-розпізнавача (до 5 сторінок, опис головних етапів розробки такого симулятора, описи основних задач та проблем які виникали під час розробки, список додаткових структур даних, що використовувались виконавцем при розробці, тощо).
- Вихідний код (необмежена кількість сторінок, оформлення коду має бути з використанням стілистичного форматування (колір ключових слів, операторів, шрифт Courier New, розмір 10 пт, інтервал 1.0).
- Висновки (до 1 сторінки).

2. ПОРЯДОК ПОБУДОВИ ВИСХІДНОГО *LR*(1) РОЗПІЗНАВАЧА

1. Для заданого фрагмента програми побудувати правила граматики.
2. Визначити, чи не містить граматика непродуктивних та недосяжних символів. За наявності таких символів необхідно їх вилучити із правил граматики.
3. Переписати правила, застосовуючи граматичні входження.
4. Знайти для отриманої граматики функції ВПЕРШ і ВПІСЛЯ та за необхідності функцію СЛІД (якщо є правила, що анулюють).
5. Побудувати таблицю переходів, яка має по одному стовпцю для кожного граматичного символу і по одному рядку для кожного граматичного входження та маркера дна h_0. Якщо таблиця, побудована на кроці 4, недетермінована, то її необхідно перетворити в детерміновану таблицю. Стани, отримані в цій таблиці, слід використовувати як магазинні символи (при їхній наявності). Порожні стани розглядаються як заборонені.
6. Заповнити управляючу таблицю. Порожні стани таблиці розглядаються як заборонені. Якщо в результаті виконання не вдається побудувати управляючу таблицю, то задана граматика не є $LR(0)$ або $LR(1)$ граматикою і розпізнавач для неї побудувати неможливо. В такому випадку необхідно побудувати інші правила граматики.
7. Перевірити роботу автомата на прикладі розпізнавання ланцюжка.
8. Написати текст програми, що моделює роботу розпізнавача.
9. Створити зручний інтерфейс роботи програми.

3. ПРИКЛАД ПОБУДОВИ *LR*(1) - РОЗПІЗНАВАЧА

Завдання: Побудувати *LR*(1)-розпізнавач для опису арифметичного виразу, який містить: ідентифікатор *i*, =, +, (,), ;. Дужки можуть бути вкладеними.

3.1 Побудова правил граматики

Основою створення правил граматики є спосіб виділення структури заданої множини ланцюжків. Цей спосіб передбачає розчленування ланцюжків, що входять у задану множину, на їх частини таким чином, щоб виявити повторювані частини ланцюжків і частини, що входять в усі ланцюжки в незмінному вигляді. Таке розчленування на частини являє собою виявлення структури ланцюжків заданої множини.

Для кожного виявленого елемента структури введемо позначення. Множина таких позначень складає основу словника нетермінальних символів певної граматики. Наступним кроком побудови є виявлення послідовностей, у яких елементи структури можуть входити в задані ланцюжки. Такі послідовності є основою для побудови правил граматики. Щоб показати, яким чином структура ланцюжків відображається в правилах граматики, розглянемо приклади.

- Ланцюжку, що складається з заданих символів *abc*, відповідає правило

$$I \to abc.$$

- Ланцюжку, що починається з заданого символу *a*, відповідає правило

$$I \to aA.$$

- Ланцюжку, що закінчується заданим символом *a*, відповідає правило

$$I \to Aa.$$

- Ланцюжку, що починається і закінчується заданими символами *a* і *b*, відповідає правило $I \to aAb.$

- Ланцюжку, що містить усередині символ *a*, відповідає правило

$$I \to AaB.$$

- Ланцюжку заданої довжини $l = 2$ відповідають правила:

$$A \to aB \text{ і } B \to a.$$

- Ланцюжку, що складається з повторюваних символів a, відповідають правила $A \to aA$ і $A \to a$.
- Ланцюжку, що складається із символів a і b, які чергуються, відповідають правила: $A \to aB$ і $B \to bA$.

Розглянемо побудову граматик для послідовностей символів і послідовностей символів з роздільниками, тобто для списків.

Позначимо елемент послідовності a. Найпростіша послідовність може складатися з одного елемента a. Всі інші послідовності можуть бути отримані шляхом приписування до вже побудованої послідовності ще одного елемента. Якщо позначити побудовану частину послідовності нетермінальним символом R, а послідовність – символом L, то одержимо правила граматики у такому вигляді:

$$L \to aR \quad (1)$$
$$R \to aR \quad (2)$$
$$R \to \$. \quad (3)$$

У попередній задачі передбачалося, що список L має містити хоча б один елемент. Якщо ж припустити, що множина ланцюжків, породжених правилами граматики, може включати порожній символ, то до побудованих правил необхідно додати ще одне правило $L \to \$$. У цьому випадку набір правил має такий вигляд:

$$L \to aR \quad (1)$$
$$R \to aR \quad (2)$$
$$R \to \$ \quad (3)$$
$$L \to \$. \quad (4)$$

Розглянемо побудову списку, між елементами якого мають стояти роздільники. Виберемо як роздільник кому. Найпростіший список, як і в попередньому випадку, складається з одного елемента, а побудова списку з декількох елементів може бути виконана приписуванням до вже побудованої частини списку роздільника з елементом списку. Правила, що відповідають цій побудові, мають такий вигляд:

$$L \rightarrow aR \quad (1)$$
$$R \rightarrow ,aR \quad (2)$$
$$R \rightarrow \$. \quad (3)$$

Якщо список з роздільниками може бути порожнім, то наведений вище набір правил необхідно доповнити ще одним правилом з порожньою правою частиною. В результаті отримаємо

$$L \rightarrow aR \quad (1)$$
$$R \rightarrow ,aR \quad (2)$$
$$R \rightarrow \$. \quad (3)$$
$$L \rightarrow \$. \quad (4)$$

У загальному випадку, якщо описана множина ланцюжків, що являють собою певну мову, і потрібно побудувати граматику, яка породжує цю множину ланцюжків, то слід робити так:

- ✓ виписати кілька прикладів із заданої множини ланцюжків;
- ✓ проаналізувати структуру ланцюжків, виділяючи початок, кінець, що повторюються, або символи з групи символів;
- ✓ ввести позначення для складних структур, що складаються з груп символів; такі позначення є нетермінальними символами граматики;
- ✓ побудувати правила для кожної з виділених структур, використовуючи для завдання повторюваних структур рекурсивні правила;
- ✓ об’єднати всі правила;
- ✓ перевірити за допомогою виведень можливість одержання ланцюжків з різною структурою.

Для нашого завдання граматика має такий вигляд:

Γ_1: $V_т = \{i, +, =, (,), ;\}$, $V_а = \{I, A, C\}$,

1. $I \rightarrow i = A ;$
2. $A \rightarrow i\ C$
3. $A \rightarrow (\ A\ )\ C$
4. $C \rightarrow +\ A$
5. $C \rightarrow \$$

3.2. Визначення непродуктивних та недосяжних символів

Символ $x \in V_a$ називається **непродуктивним**, якщо з нього не може бути виведений кінцевий термінальний ланцюжок.

Розглядаючи правила граматики, можна зробити висновок, що коли всі символи правої частини є продуктивними, то продуктивним є і символ, що стоїть у лівій частині. Останнє твердження дозволяє організувати процедуру виявлення непродуктивних символів у такому вигляді:

- Скласти список нетермінальних символів, для яких знайдеться хоча б одне правило, права частина якого містить термінальні символи або пусто ($).
- Якщо знайдене таке правило і всі нетермінальні символи, які стоять у його правій частині, вже занесені до списку, то слід додати до списку нетермінальний символ, що стоїть у його лівій частині.
- Якщо на кроці 2 список більше не поповнюється, тоді ми отримали список усіх продуктивних нетермінальних символів граматики, а всі нетермінальні символи, які не потрапили в нього, є непродуктивними.

Визначимо непродуктивні символи для граматики. На першому кроці заносимо до списку символ *C*. Далі, відповідно до другого правила до списку заносимо символ *A*. На третьому кроці до списку заносяться символи *I*. До списку потрапили всі нетермінальні символи, отже, граматика **не місить непродуктивних символів.**

Визначимо, чи не місить граматика Γ_1 недосяжні символи.

Символ $x \in V_т \cup V_a$ називається **недосяжним** у КВ-граматиці Г, якщо *x* не з'являється в жодному виведеному ланцюжку.

Розглядаючи правила граматики, можна помітити, що якщо нетермінальний символ у лівій частині правила є досяжним, то і всі символи правої частини є досяжними. Ця властивість правил є основою процедури виявлення недосяжних символів, яку можна описати таким чином:

- Створити одноелементний список, що складається з початкового символу граматики *I*.
- Якщо знайдене правило, ліва частина якого вже є в списку, то включити до списку всі символи, які містяться в його правій частині.
- Якщо на кроці 2 нові нетермінальні символи в список більше не додаються, то отримано список усіх досяжних нетермінальних символів, а решта символів, що не потрапили в список, є недосяжними.

Визначимо недосяжні символи для граматики Γ_1 .

На першому кроці заносимо до списку символ *I*. На другому кроці додамо символ *A*. На третьому кроці список доповнимо символом *C*. До списку потрапили всі нетермінальні символи, отже, граматика **не місить недосяжних символів.**

3.3. Визначення граматичних входжень

Оскільки кожен граматичний символ може входити в правило граматики один чи декілька разів, а також входити в різні правила, то в залежності від розташування граматичних символів у правилах можуть виконуватися різні дії: перенос символу чи згортка, тому вводиться поняття *граматичного входження*. Граматичне входження символу граматики задається номером правила і номером позиції символу в цьому правилі. Крайній зліва символ – перший. Якщо символ входить у правило один раз, то він позначається номером правила. Якщо символ зустрічається в граматиці тільки один раз, то його можна не нумерувати. Початковий символ граматики позначається I_0 і називається початковим значенням.

Перепишемо правила граматики Γ_1, застосовуючи граматичні входження.

$I \to i_1 = A_1$;	(1)	$A \to (A_4)\, C_4$	(4)
$A \to i_2\, C_2$	(2)	$C \to \$$.	(5)
$C \to + A_3$	(3)		

Примітка: якщо граматика містить правила, що анулюють, то для неї будується тільки *LR*(1) розпізнавач.

3.4. Знаходження функції ВПЕРШ(*Y*) і ВПІСЛЯ(*Y*)

Під час побудові висхідного розпізнавача використовуються функції ВПЕРШ(*Y*) і ВПІСЛЯ(*Y*).

Функція ВПЕРШ(*Y*) визначає множину символів, які можуть стояти на першому місці в ланцюжках виведених з *Y*. У нього входять сам *Y* і всі символи, виведені з *Y* без правил, що анулюють. Функція ВПЕРШ(*Y*) визначається і для початкового символу граматики.

Наприклад, функція ВПЕРШ(*Y*) для першого правила граматики буде

мати вигляд.

ВПЕРШ(i_1)={i_1}
ВПЕРШ(=)={=}
ВПЕРШ(A_1)={A_1, i_2,(}
ВПЕРШ(;)={;}

Функція ВПІСЛЯ(Y) визначає множину символів, що можуть зустрічатися безпосередньо після Y у ланцюжках, виведених з початкового символу граматики.

Якщо після Y випливає символ Z, то $\alpha \rightarrow \varphi YZ$, а ВПІСЛЯ($Y$)=ВПЕРШ($Z$).

Крім того, тут визначається функція ВПІСЛЯ(h_0), що дорівнює ВПЕРШ(I_0), тобто ВПІСЛЯ(h_0)= ВПЕРШ(I_0).

Для заданої граматики будуємо функцію ВПІСЛЯ(Y).

ВПІСЛЯ (i_1) = {=};
ВПІСЛЯ (=) = {A_1, i_2, (};
ВПІСЛЯ (A_1) = {;};
ВПІСЛЯ (;) = {$};
ВПІСЛЯ (i_2) = {C_2, +};
ВПІСЛЯ (C_2) = {$};
ВПІСЛЯ (+) = {A_3, i_2,) };
ВПІСЛЯ (A_3) = {$};
ВПІСЛЯ (() = {A_4, i_2, (};
ВПІСЛЯ (A_4) = {)};
ВПІСЛЯ ()) ={C_4, +};
ВПІСЛЯ (C_4) = {$};
ВПІСЛЯ (h_0) = {I_0, i_1};
ВПІСЛЯ (I_0) = {$};.

Побудуємо функцію СЛІД для крайніх правих символів в усіх правилах:

СЛІД (A) = {), ;}; СЛІД (C) = {), ;}; СЛІД (I) = {$}.

3.5. Побудова таблиці переходів

Використовуючи функцію ВПІСЛЯ(Y), будується таблиця переходів, яка визначає зміну станів автомата. Таблиця переходів використовується для визначення граматичних входжень, що записуються у магазин. Таблиця будується в такий спосіб:

1) кожному граматичному входженню відповідає рядок таблиці, кожному граматичному символу відповідає стовпець.

2) клітинки таблиці заповнюються елементами функції ВПІСЛЯ(Y);

3) елемент $x_к$, який належить множині функції ВПІСЛЯ(Y_j), заноситься в клітку, що знаходиться на перетині рядка Y_j та стовпця x;

Наприклад, для рядка, поміченого граматичним входженням $\boldsymbol{i_1}$ функція ВПІСЛЯ (i_1) = {=}. Значить, на перетині рядка $\boldsymbol{i_1}$ і стовпця = ставимо елемент

із множини ВПІСЛЯ (i_1), тобто =. Подібні дії виконуємо із усіма рядками таблиці переходів (Табл. 3.1).

Таблиця 3.1 – Таблиця переходів

Граматичні входження	Граматичні символи								
	I	*i*	=	*A*	*C*	+	*(*	*)*	*;*
i_1			=						
=		i_2		A_1			*(*		
A_1									*;*
;									
i_2					C_2	+			
C_2									
+		i_2		A_3			*(*		
A_3									
(		i_2		A_4			*(*		
A_4								*)*	
)					C_4	+			
C_4									
h_0	I_0	i_1							
I_0									

Слід зазначити, що при побудові таблиці переходів у клітинках може бути кілька граматичних входжень відповідних символів. Така таблиця є недетермінованою, і її необхідно зробити детермінованою за допомогою засобів, що використовуються для перетворення таблиць кінцевих автоматів. У результаті отримаємо таблицю, у якій рядки помічені множинами граматичних входжень.

3.6 Побудова управляючої таблиці

Для опису порядку дій розпізнавача будується управляюча таблиця, у якій:

- буквою *П* позначається операція переносу;
- буквою *З(к)* позначається операція згортки, де *к* – номер правила;
- буквою *Д* позначається операція “Допустити”, тобто весь ланцюжок розпізнано;
- буквою *В* (або пустою клітиною*)* позначається операція “Відкинути”, тобто відбулася помилка і далі розпізнавання неможливо.

Таблиця містить рядки, які є граматичним входженням, і стовпці, які є термінальними символами (символами вхідного ланцюжка). У таблиці також присутній стовпець ⊥ або $, що позначає кінець вхідного рядка.

Заповнюється таблиця таким чином:

- На перетині рядка, позначеного символом I_0, та стовпця, позначеного маркером h_0, заносимо операцію *Д.*
- Якщо рядок позначений граматичним входженням R_{ij}, яке не є крайнім правим входженням ніякого правила і якщо елемент таблиці переходів на перетині рядка R_{ij} та стовпця S не є порожнім, то в управляючій таблицю на перетині рядка R_{ij} і стовпця S заноситься операція перенос (*П*).
- Якщо рядок позначений крайнім правим граматичним входженням p_{ij}, і є правило $A \rightarrow \alpha$ p_{ij} з номером к, то для кожного вхідного символу *x,* що належить множині СЛІД(*A*), в керувальну таблиці (на перетині рядка p_{ij} і стовпця *x)* заноситься операція згортки *З*(*к)*.
- Якщо граматика містить правило що *анулює* $A \rightarrow$ $, з *номером к*, то необхідно позначити рядки, для яких функція ВПІСЛЯ містить A_{ij}. На перетинанні даних рядків і стовпців відповідних до функції СЛІД(*A*) ставимо операцію згортки *З*(*к)*.

Управляюча таблиця, побудована для заданої граматики, наведена в таблиці 3.2.

Таблиця 3.2 – Управляюча таблиця

Граматичні входження	Термінальні символи						
	i	=	+	(	)	;	⊥
i_1		*П*					
=	*П*			*П*			
A_1						*П*	
;							*З(1)*
i_2			*П*		*З(5)*	*З(5)*	
C_2					*З(2)*	*З(2)*	
+	*П*			*П*			
A_3					*З(3)*	*З(3)*	
(	*П*			*П*			
A_4					*П*		
)			П		*З(5)*	*З(5)*	
C_4					*З(4)*	*З(4)*	
h_0	П						
I_0							*Д*

3.7. Алгоритм роботи висхідного розпізнавача

Алгоритм роботи використовує таблицю станів і управляючу таблицю і працює таким чином:

1. Прочитати черговий символ вхідного ланцюжка *x*.
2. Прочитати символ стану, що знаходиться на вершині магазина y_{ij}.
3. Прочитати значення елемента управляючої таблиці, що знаходиться в рядку y_{ij} і стовпці *x*.
4. Якщо прочитане значення є *В* (Відкинути) чи *Д* (Допустити), то роботуавтомата закінчити.
5. Якщо отримане значення визначене операцією *П* (Перенос), то прочитати в таблиці переходів елемент, що знаходиться в рядку y_{ij} і стовпці *x*. Записати отриманий символ у магазин, перейти до п.1.
6. Якщо отримане значення в керувальній таблиці визначено операцією *З(k)* (Згортки) у нетермінал *Z* (згідно з *к-тим* правилом), то необхідно в таблиці переходів прочитати елемент Z_{ij}, що знаходиться в стовпці *Z* і рядку, що відповідає верхньому символу магазина, який не брав участі в згортці. Записати Z_{ij} у магазин і перейти до п.1.

3.8. Приклад роботи розпізнавача.

Роботу розпізнавача перевіримо на прикладі розпізнавання ланцюжка: *i=((i+i+i));*. Приклад роботи наведено в табл.3.3.

Таблиця 3.3 – Приклад роботи розпізнавача

Магазин	Вхід автомата	Операція
h_0	*i=((i+i+i));*⊥	*П*
$h_0\ i_1$	*=((i+i+i));*⊥	*П*
$h_0 i_1$=	*((i+i+i));*⊥	*П*
$h_0\ i_1$=(	*(i+i+i));*⊥	*П*
$h_0\ i_1$=((	*i+i+i));*⊥	*П*
$h_0\ i_1$=((i_2	*+i+i));*⊥	*П*
$h_0\ i_1$=((i_2+	*i+i));*⊥	*П*
$h_0\ i_1$=((i_2+i_2	*+i));*⊥	*П*
$h_0\ i_1$=((i_2+i_2+	*i));*⊥	*П*
$h_0 i_1$=((i_2+i_2+i_2	*));*⊥	*З(5)*
$h_0 i_1$=((i_2+i_2+$i_2 C_2$	*));*⊥	*З(2)*
$h_0\ i_1$=((i_2+i_2+A_3	*));*⊥	*З(3)*
$h_0\ i_1$=((i_2+$i_2\ C_2$	*));*⊥	*З(2)*
$h_0 i_1$=((i_2+A_3	*));*⊥	*З(3)*
$h_0\ i_1$=$_1$(($i_2 C_2$	*));* ⊥	*З(2)*
h_0= i_1((A_4	*));* ⊥	*П*
$h_0 i_1$=((A_4)	*);* ⊥	*З(5)*
$h_0 i_1$=((A_4)C_4	*);* ⊥	*З(4)*
$h_0 i_1$=(A_4	*);* ⊥	*П*
$h_0\ i_1$=(A_4)	*;* ⊥	*З(5)*
$h_0 i_1$=(A_4)C_4	*;* ⊥	*З(4)*
$h_0 i_1$=A_1	*;* ⊥	*П*
$h_0\ i_1$=A_1;	⊥	*З(1)*
$h_0 I_0$	⊥	*Д*

СПИСОК ЛІТЕРАТУРИ

1. Гавриленко С.Ю. Формальні мови, граматики та автомати: навч. посіб., Харків: НТУ « ХПІ », 2021, 133 с. https://repository.kpi.kharkov.ua/server/api/core/bitstreams/5847efdd-6ff5-4f7f-9de8-6f6555ad4cc0/content

2. Гавриленко С.Ю., Клименко А.М. Любченко Н.Ю. Теорія цифрових автоматів та формальних мов. Рекомендовано МОН України, Харків, НТУ «ХПІ», 2011, 176 с.

https://reposit.nupp.edu.ua/bitstream/PoltNTU/4517/1/Kompyuterna_logika_2sem_posibnik.pdf

3. Спекторський І.Я., Статкевич В.М. Формальні мови та автомати . Навч. Посібник, Київ: КПІ ім. Ігоря Сікорьского, 2019. 167 с. http://spectorsky.ho.ua/files/method_grammar_automata.pdf

4. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers: Principles, Techniques, & Tools. Pearson Education, Inc., 2007. – 1009 p. https://www.dbscience.org/wp-content/uploads/2020/03/ALSUdragonbookcompilers.pdf

5. Марченко О. І., Марченко О. О. Основи проектування трансляторів: навч. посіб., Київ : КПІ ім. Ігоря Сікорського, 2021, 118 с. https://scs.kpi.ua/wp-content/uploads/2022/01/opt_metodychka_2021.pdf

Навчальне електронне видання

Методичні вказівки

до виконання курсового проекту з курсу «Теорія побудови компіляторів» для студентів денної та заочної форм навчання спеціальності 123 “Комп’ютерна інженерія”.

Укладачі: ГАВРИЛЕНКО Світлана Юріївна,
ЧЕЛАК Віктор Володимирович

Роботу до видання рекомендував Заполовський М.Й

В авторській редакції