Лабораторная работа № 3. “Использование метода рекурсивного спуска для построения синтаксического анализатора”

1 Цель работы

Целью выполнения лабораторной работы является

- получение навыков описания языка с использованием контекстно-свободных грамматик;
- изучение метода рекурсивного спуска.

2 Общие сведения

Не все языки можно разбирать с помощью регулярных выражений, например нельзя записать регулярное выражение для степенной функции. Для работы с такими языками используются контекстно-свободные грамматики.

В самом общем виде контекстно-свободная грамматика описывает язык как множество строк, полученных применением конечного множества продукций. Формально контекстно-свободная грамматика это четверка <V, ∑, P, S>. Рассмотрим составляющие контекстно-свободной грамматики. V – это конечное множество всех грамматических символов. ∑ - множество терминальных символов, причем ∑ является подмножеством V. Терминальные символы в грамматике будем обозначать строчными буквами. Множество нетерминальных символов N=V-∑. Нетерминальные символы в грамматике будем обозначать заглавными буквами. S – стартовый нетерминал. P – множество правил вывода. P⊆NxV*, то есть продукция – это последовательность, начинающаяся с нетерминального символа (левая часть правила), за которым следует замыкание Клини терминальных и нетерминальных символов (правая часть правила).

Рассмотрим пример грамматики, описывающей арифметические действия.

```
1. GOAL→ EXPR
2. EXPR→EXPR+TERM
3. EXPR→ EXPR-TERM
4. EXPR→ TERM
5. TERM→ TERM*FACTOR
6. TERM→ TERM/FACTOR
7. TERM→ FACTOR
8. FACTOR→ num
9. FACTOR→ id
```

Рисунок 18. – Грамматика языка арифметических действий

В данном примере стартовым является нетерминал GOAL, так как он не встречается в правой части ни одной продукции. ∑={+, -, *, /, num, id}; N={GOAL, EXPR, TERM, FACTOR}. Любая из продукций состоит из нетер-

минала в левой части и последовательности терминалов и нетерминалов в правой. Например, правило 2 утверждает, что EXPR есть последовательность из нетерминала EXPR, терминала + и нетеминала TERM.

Будем говорить, что $\alpha\gamma\beta$ выводимо за один шаг из $\alpha A\beta$ ($\alpha\gamma\beta$=>$\alpha A\beta$), если в грамматике существует правило вывода A→γ, а α и β - произвольные строки из V. Если u1=>u2=>…=>un, то можно утверждать, что u1 выводится из un за 0 или более шагов (u1=>*un). Если un нельзя вывести из u1 за 0 шагов, то говорят, что un выводимо из u1 за 1 или более шагов(u1=>+un).

Различают два вида вывода: левый и правый. Если на каждом шаге заменяют самый левый нетерминальный символ, то вывод называется левым, если самый правый, то вывод – правый.

Если дана грамматика G со стартовым символом S, то используя отношение =>+, можно определить язык L(G), порожденный грамматикой G. Строки такого языка могут содержать только терминальные символы из G. Строка терминалов w принадлежит L(G) тогда и только тогда, когда w выводимо из S за 1 или более шагов S=>+w.

Таким образом, приведенная ранее грамматика описывает язык арифметических операций. При этом строки этого языка могут содержать только терминалы +, -, *, /, num, id. Пусть есть строка “id*id+num”. Рассмотрим, принадлежит ли эта строка языку арифметических операций. Для этого проверим, можно ли вывести входную строку из стартового символа. В скобках после знака => будем указывать номер продукции, по которой осуществляется вывод.

```
GOAL=>(1) EXPR=>(2) EXPR+TERM=>(4) TERM+TERM=>(5)
TERM*FACTOR+TERM=>(7) FACTOR*FACTOR+TERM=>(9)
id*FACTOR+TERM=>(9) id*id+TERM=>(7) id*id+FACTOR=>(8) id*id+num
```

Получили входную строку, следовательно, строка “id*id+num” принадлежит языку. Заметим, что был использован левый вывод.

Вывод можно представить в виде дерева. Дерево является деревом вывода грамматики, если выполнены следующие условия

- корень дерева помечен стартовым символом
- каждый лист помечен терминалом или ε
- каждая внутренняя вершина помечена нетерминалом
- если N – нетерминал, которым помечена внутренняя вершина и X1, X2, …, Xn – метки ее прямых потомков в указанном порядке, то правило N→X1 X2 … Xn существует в грамматике.

Входная последовательность соответствует языку, определяемому грамматикой, если листья дерева вывода соответствуют этой последовательности.

Для рассмотренного ранее примера дерево вывода представлено на рисунке 19.

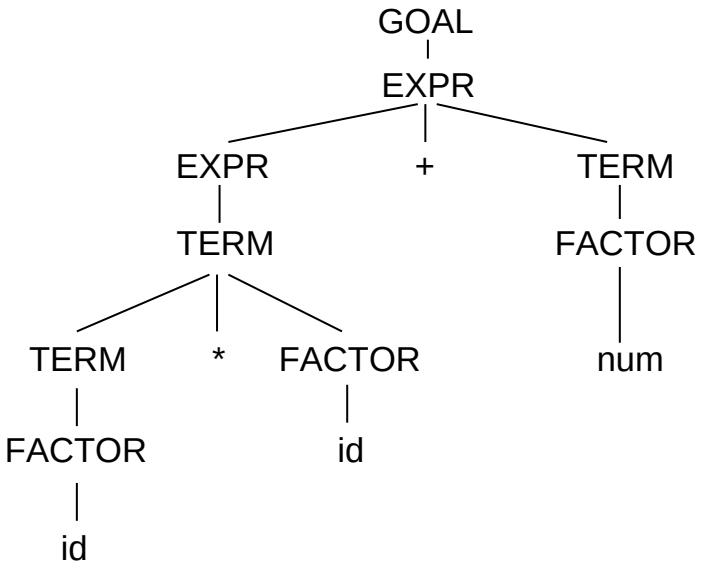

Рисунок 19.

3 Рекурсивный спуск

Существует два метода синтаксического разбора: сверху вниз и снизу вверх. В данной лабораторной работе будем рассматривать первый метод. При разборе сверху вниз начинают разбор со стартового нетерминала и, применяя правила, заменяют нетерминалы левой части правила на последовательности грамматических символов правой. Входная последовательность принимается, если она выведена из стартового сим вола.

Процедура рекурсивного разбора сверху вниз состоит из следующих шагов:

- Для узла дерева, помеченного, как нетерминал A, выбирают одну из продукций вида A→α. После этого строим от A ветви, соответствующие α.
- Если в процессе применения продукций получено обрамление, несоответствующее входной последовательности, то производится откат.
- Находим следующий узел, помеченный нетерминалом, для подстановки правила.

При таком подходе может возникнуть проблема бесконечного цикла. В грамматике для арифметических операций применение второго правила

приведет к зацикливанию процедуры разбора. Подобные грамматики называются леворекурсивными. Грамматика называется леворекурсивной, если в ней существует нетерминал A, для которого существует вывод A=>+Aα. В простых случаях левая рекурсия вызвана правилами вида

A→Aα|β

В этом случае вводят новый нетерминал и исходные правила заменяют следующими.

A→βB

B→αB|ε

Рассмотрим правила 2 и 4 грамматики с рисунка 18. Правило 2 делает грамматику леворекурсивной. Воспользуемся вышеизложенным приемом исключения левой рекурсии. α=+T, β=T. Введем новый нетеримнал EXPR1 и получим новые продукции. EXPR→TERM EXPR1; EXPR1→+TERM EXPR1; EXPR1→ε.

На рисунке 16 изображена грамматика, к которой была преобразована грамматика с рисунка 14 путем исключения левой рекурсии.

```
1. GOAL→EXPR
2. EXPR→TERM EXPR1
3. EXPR1→+TERM EXPR1
4. EXPR1→-TERM EXPR1
5. EXPR1→ε
6. TERM→FACTOR TERM1
7. TERM1→*FACTOR TERM1
8. TERM1→ /FACTOR TERM1
9. TERM1→ε
10. FACTOR→num
11. FACTOR→id
```

Рисунок 20.

Применение рекурсивного спуска в вышеизложенном виде может работать очень длительное время за счет откатов. Поэтому важно найти такой алгоритм, который мог бы однозначно выбирать продукцию на каждом шаге. Есть разновидность грамматик, которые при разборе сверху вниз позволяют выбирать продукцию на основе первых k символов входной последовательности. Будем использовать LL(1)-грамматики, которые позволяют выбирать продукцию на основе первого символа входной последовательности. Первое L означает, что сканирование осуществляется слева на право, второе L, что строиться левый вывод.

Грамматика, приведенная на рисунке 20, является LL(1), так как при выборе правила для любого нетерминала достаточно проанализировать первый символ входной последовательности.

4 Пример реализации метода рекурсивного спуска

Работа системы процедур, построенных в соответствии с методом рекурсивного спуска, начинается с главной функции main ( ), которая:

- считывает первый символ исходной цепочки (заданной во входном потоке stdin),

- затем вызывает процедуру S ( ), которая проверяет, выводится ли входная цепочка из начального символа S (в общем случае это делается с участием других процедур, которые, в свою очередь, рекурсивно могут вызывать и саму S ( )для анализа фрагментов исходной цепочки).

Считаем, что в конце любой анализируемой цепочки всегда присутствует символ ⊥ (признак конца цепочки). На практике этим признаком может быть ситуация «конец файла» или маркер «конец строки».

Пример реализации на C++ грамматики, представленной на рисунке 21.

```
int c;
void A ( );
void B ( );
void gc ( ) {
    cin >> c;
}

void S ( ) {
     A ( );
     B ( );
}

int main ( ) {
    try {   gc ( );
            S ( );
            if (c  !=  '⊥')    throw c;
            cout << "SUCCESS !!!" << endl;
            return 0;
    }
    catch ( int c ) {
            cout << "ERROR on lexeme" << c << endl;
            return 1;
    }
}
```

```
void A ( ) {
      if (c == 'a')   gc ();
      else
      if (c == 'c') {   gc ( );   A ( );  }
      else   throw c;
}

void B ( ) {
      if (c == 'b') {  gc ( );  A ( ); }
      else    throw c;
}
```

$S \rightarrow AB\perp$
$A \rightarrow a \mid cA$
$B \rightarrow bA$

Пример реализации грамматики методом рекурсивного спуска

Пример реализации на C++ синтаксического анализатора для калькулятора

```
/*
 *  Калькулятор. Пример из учебника "C++ Programming Language" (основа).
 *  Применяется алгоритм "рекурсивный спуск" (recursive descent).
 *  (Примечание: калькулятор может работать с выражениями. Например, если
 *  на входе подать r = 2.5; area = pi * r * r; то на выходе будет
 *  2.5; 19.635)
 */

#include <cctype>
#include <iostream>
#include <map>
#include <sstream>
#include <string>

enum Token_value : char {
  NAME,             NUMBER,               END,
  PLUS='+',         MINUS='-',            MUL='*',        DIV='/',
  PRINT=';',        ASSIGN='=',           LP='(',         RP=')'
};

enum Number_value : char {
  NUM0='0', NUM1='1', NUM2='2',
  NUM3='3', NUM4='4', NUM5='5',
  NUM6='6', NUM7='7', NUM8='8',
  NUM9='9', NUMP='.',
};

Token_value curr_tok = PRINT;        // Хранит последний возврат функции
get_token().
double number_value;                 // Хранит целый литерал или литерал с
плавающей запятой.
std::string string_value;            // Хранит имя.
std::map<std::string, double> table; // Таблица имён.
int no_of_errors;                    // Хранит количество встречаемых ошибок.

double expr(std::istream*, bool);    // Обязательное объявление.

/*************************************************************************
/

// Функция error() имеет тривиальный характер: инкрементирует счётчик ошибок.
double error(const std::string& error_message) {
  ++no_of_errors;
  std::cerr << "error: " << error_message << std::endl;
  return 1;
}

Token_value get_token(std::istream* input) {
  char ch;

  do {    // Пропустить все пробельные символы кроме '\n'.
    if (!input->get(ch)) {
      return curr_tok = END; // Завершить работу калькулятора.
    }
  } while (ch != '\n' && isspace(ch));

  switch (ch) {
    case 0: // При вводе символа конца файла, завершить работу калькулятора.
      return curr_tok = END;
    case PRINT:
    case '\n':
```

```
      return curr_tok = PRINT;
    case MUL:
    case DIV:
    case PLUS:
    case MINUS:
    case LP:
    case RP:
    case ASSIGN:
      return curr_tok = Token_value(ch); // Приведение к целому и возврат.
    case NUM0: case NUM1: case NUM2: case NUM3: case NUM4:
    case NUM5: case NUM6: case NUM7: case NUM8: case NUM9:
    case NUMP:
      input->putback(ch); // Положить назад в поток...
      *input >> number_value; // И считать всю лексему.
      return curr_tok = NUMBER;
    default:
      if (isalpha(ch)) {
        string_value = ch;
        while (input->get(ch) && isalnum(ch)) {
          string_value.push_back(ch);
        }
        input->putback(ch);
        return curr_tok = NAME;
      }
      error("Bad Token");
      return curr_tok = PRINT;
  }
}

/* Каждая функция синтаксического анализа принимает аргумент типа bool
 * указывающий, должна ли функция вызывать get_token() для получения
 * очередной лексемы. */

// prim() - обрабатывает первичные выражения.
double prim(std::istream* input, bool get) {
  if (get) {
    get_token(input);
  }

  switch (curr_tok) {
    case NUMBER: {
      double v = number_value;
      get_token(input);
      return v;
    }
    case NAME: {
      double& v = table[string_value];
      if (get_token(input) == ASSIGN) {
          v = expr(input, true);
      }
      return v;
    }
    case MINUS:
      return -prim(input, true);
    case LP: {
      double e = expr(input, true);
      if (curr_tok != RP) {
          return error("')' expected");
      }
      get_token(input);
      return e;
    }
    default:
      return error("primary expected");
```

```
  }
}

// term() - умножение и деление.
double term(std::istream* input, bool get) {
  double left = prim(input, get);

  for ( ; ; ) {
    switch (curr_tok) {
      case MUL:
        left *= prim(input, true);
        break;
      case DIV:
        if (double d = prim(input, true)) {
            left /= d;
            break;
        }
        return error("Divide by 0");
      default:
          return left;
    }
  }
}

// expr() - сложение и вычитание.
double expr(std::istream* input, bool get) {
  double left = term(input, get);

  for ( ; ; ) {
    switch (curr_tok) {
      case PLUS:
        left += term(input, true);
        break;
      case MINUS:
        left -= term(input, true);
        break;
      default:
        return left;
    }
  }
}

int main(int argc, char* argv[]) {
  std::istream* input = nullptr; // Указатель на поток.

  switch (argc) {
    case 1:
      input = &std::cin;
      break;
    case 2:
      input = new std::istringstream(argv[1]);
      break;
    default:
      error("Too many arguments");
      return 1;
  }

  table["pi"] = 3.1415926535897932385;
  table["e"] = 2.7182818284590452354;

  while (*input) {
    get_token(input);
    if (curr_tok == END) {
      break;
```

```
    }

    // Снимает ответственность expr() за обработку пустых выражений.
    if (curr_tok == PRINT) {
      continue;
    }

    // expr() -> term() -> prim() -> expr() ...
    std::cout << expr(input, false) << std::endl;
  }

  if (input != &std::cin) {
    delete input;
  }

  return no_of_errors;
}
```

5.5 Контрольные вопросы

- Что такое контекстно-свободная грамматика?
- Какой нетерминал является стартовым в грамматике?
- Когда можно утверждать, что одна последовательность выводим из другой за один шаг?
- В чем различие левого и правого вывода?
- Как вывод можно представить в виде дерева?
- В чем состоит метод разбора сверху вниз?
- Какова процедура рекурсивного разбора сверху вниз?
- Какие грамматики являются леворекурсивными?
- В чем состоит метод исключения левой рекурсии?
- Какие грамматики являются LL(1)-грамматиками?

5.6 Задание

1. Написать грамматику для распознавания программы на языке Си.

```
#include <stdio.h>
int main (void)
{
 return 0;
}
```

Типы данных: int, float.

Команды могут быть следующего вида:

а) Команда присвоения, состоит из идентификатора, знака присвоения (=) и арифметического выражения. Арифметическое выражение включает в

себя сложение, умножение, вычитание, деление, вычисление остатка, скобки.

б) Команда чтения значения переменной – scanf("управляющая строка", аргумент_1);.

```
scanf("%d", &a);
```

в) Команда вывода. - printf ("управляющая строка", аргумент _1);.

```
printf("Computer\n%d\n", i);
```

г) Команда ветвления, которая состоит из трех разделов: условие, список операторов при выполнении условия, список операторов при не выполнении условия.

```
if (проверка_условия) оператор_1; else оператор_2;
```

Раздел условия начинается со служебного слова If. После него следует само условие. Условие состоит из конструкций вида идентификатор|число ==|<|> идентификатор|число. Конструкции такого вида заключаются в скобки. Такие конструкции могут быть соединены в условии или операцией &&, или операцией ||. Если в условии присутствуют такие операции, то связываемые ими конструкции должны быть взяты в круглые скобки. Скобки могут встречаться в условии и для обозначения приоритета действий.

Два остальных раздела команды ветвления имеют идентичную структуру и содержать команду или список команд в операторных скобках. Второй раздел в операции ветвления может отсутствовать.

Команды могут отсутствовать.

```
#include <stdio.h>
int main()
{
  int k;          // объявляем целую переменную k
  printf("k= ");   // выводим сообщение
  scanf("%d", &k); // вводим переменную k
  if (k >= 5)      // если k>5
    printf("%d >= 5", k); // выводим "ЗНАЧЕНИЕ >= 5"
  else             // иначе
    printf("%d < 5", k);  // выводим "ЗНАЧЕНИЕ < 5"
  getchar(); getchar();
  return 0;
}
```

2 Реализовать полученную грамматику методом рекурсивного спуска.

Первая часть задания выполняется студентом при подготовке к лабораторной работе.

7 Содержание отчета

Отчет о лабораторной работе должен включать грамматику для языка, описанного в задании.