Лабораторная работа № 3

Синтаксический разбор с использованием метода рекурсивного спуска

Теоретическая часть

Одним из наиболее простых и потому одним из наиболее популярных методов нисходящего синтаксического анализа является *метод рекурсивного спуска (recursive descent method).* Метод основан на «зашивании» правил грамматики непосредственно в управляющие конструкции распознавателя.

Синтаксические анализаторы, работающие по методу рекурсивного спуска без возврата, могут быть построены для класса грамматик, называющегося LL(1). Первая буква L в названии связана с тем, что входная цепочка читается слева направо, вторая буква L означает, что строится левый вывод входной цепочки, 1 означает, что на каждом шаге для принятия решения используется один символ непрочитанной части входной цепочки. Для строгого определения LL(1) грамматики потребуются две функции - FIRST и FOLLOW.

Определение. Пусть $\alpha \in (N \cup \Sigma)^+$, $x \in \Sigma^+$, $\beta \in (N \cup \Sigma)^*$. Для КС-грамматики $G = (N, \Sigma, P, S)$ определена функция

$FIRST_k(\alpha)=\{x \mid \alpha =>^* x\beta$ и $|x| = k$ или $\alpha =>^* x$ и $|x| < k \} \cup \{ \varepsilon$ if $\alpha =>^* \varepsilon \}$

Иначе говоря, множество $FIRST_k(\alpha)$ состоит из всех терминальных префиксов длины k (или меньше, если из α выводится терминальная цепочка длины, меньшей k) терминальных цепочек, выводимых из α. По определению полагают, что $FIRST_k (\varepsilon) = \{ \varepsilon \}$.

Определение. Пусть $\alpha, \gamma \in (N \cup \Sigma)^*$, $\beta \in (N \cup \Sigma)^+$, $x \in \Sigma^+$. Для КС-грамматики $G = (N, \Sigma, P, S)$ определена функция

$FOLLOW_k(\beta) = \{ x \mid S =>^* \alpha\beta\gamma$ and $x \in FIRST_k (\gamma) \} \cup \{ \varepsilon$ if $S =>^* \alpha\beta \}$

Иначе говоря, множество $FOLLOW_k(\beta)$ состоит из всех цепочек длины k (или меньше) терминальных цепочек, которые могут встречаться непосредственно справа от β в каких-нибудь цепочках, выводимых из аксиомы, причем если $\alpha\beta$ выводимая цепочка, то ε тоже принадлежит $FOLLOW_k(\beta)$.

Для грамматики LL(1) k=1, и имеет смысл говорить только о функциях $FIRST_1(\alpha)$ и $FOLLOW_1(\beta)$, а вместо фразы «терминальных цепочек» говорить «терминальных символов».

Теорема. КС-грамматика $G = (N, \Sigma, P, S)$ является LL(1)-грамматикой тогда и только тогда, когда для каждых двух различных правил $A \to \beta$ и $A \to \gamma$ выполняется условие

$FIRST_1 (\beta\ FOLLOW_1 (A)) \cap FIRST_1 (\gamma\ FOLLOW_1 (A)) = \varnothing$

при всех $A \in N$.

Алгоритм вычисления $FIRST_1$ для символов грамматики

Вход. Символы грамматики $X \in (N \cup \Sigma)$.
Выход. Семейство множеств $FIRST_1(X)$
Метод.

```
for X ∈ (N ∪ Σ ∪ {ε}):
        if X = ε:
                FIRST1(X) = {ε}
        elif X ∈ Σ:
                FIRST1(X) = {X}
        else:
```

$FIRST_1(X) = \varnothing$
if $\exists\ (X \to \varepsilon) \in P$:
    $FIRST_1(X) = FIRST_1(X) \cup \{\varepsilon\}$
for $(X \to \alpha) \in P$:
    let $\alpha = Y_1 Y_2 \ldots Y_k$
    for i $\in$ *range*(1, *len*(α)+1):
        if $\neg\ (Y_i =>^* \varepsilon)$:
            $FIRST_1(X) = FIRST_1(X) \cup FIRST_1(Y_i)$
            break
        else:
            $FIRST_1(X) = FIRST_1(X) \cup (FIRST_1(Y_i) \backslash \{\ \varepsilon\})$
            continue
    if $Y_1 Y_2 \ldots Y_k =>^* \varepsilon$:
        $FIRST_1(X) = FIRST_1(X) \cup \{\varepsilon\}$

Пример. Для G'_0 с правилами

E → T E'	(1)
E' → + T E' | ε	(2, 3)
T → F T'	(4)
T' → * F T' | ε	(5, 6)
F → (E) | **a**	(7, 8)

Будем иметь

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

Алгоритм вычисление FOLLOW для нетерминалов грамматики

Вход. Символы грамматики $A \in$ N.
Выход. Семейство множеств $FOLLOW_1(A)$
Метод.

for $A \in$ N:
    $FOLLOW_1(A) = \varnothing$
    if A = S:
        $FOLLOW_1(A) = FOLLOW_1(A) \cup \{\$\}$
    for $(A \to \gamma) \in P$:
        if $(A \to \alpha B \beta) \in P \wedge \beta \neq \varepsilon \wedge \neg (\beta =>^* \varepsilon)$:
            $FOLLOW_1(B) = FOLLOW_1(B) \cup (FIRST_1(\beta) \backslash \{\ \varepsilon\})$
        if $(A \to \alpha B) \in P \vee ((A \to \alpha B \beta) \in P) \wedge (\varepsilon \in FIRST_1(\beta))$:
            $FOLLOW_1(B) = FOLLOW_1(B) \cup FOLLOW_1(A)$

Пример. Для грамматики G'_0 будем иметь

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

Практическая часть

В методе рекурсивного спуска полностью сохраняются идеи нисходящего разбора, принятые в LL(1)-грамматиках:

- происходит последовательный просмотр входной строки слева-направо;
- очередной символ входной строки является основанием для выбора одной из правых частей правил группы при замене текущего нетерминала;
- терминальные символы входной строки и правой части правила «взаимно уничтожаются»;
- обнаружение нетерминала в правой части рекурсивно повторяет этот же процесс.

В методе рекурсивного спуска эти идеи претерпевают следующие изменения:

- каждому нетерминалу соответствует отдельная процедура (функция), распознающая (выбирающая и «закрывающая») одну из правых частей правила, имеющего в левой части этот нетерминал (т.е. для каждой группы правил пишется свой распознаватель);
- во входной строке имеется указатель (индекс) на текущий «закрываемый символ». Этот символ и является основанием для выбора необходимой правой части правила. Сам выбор «зашит» в распознавателе в виде конструкций **if** или **switch**. Правила выбора базируются на построении множеств выбирающих символах, как это принято в LL(1)-грамматике;
- просмотр выбранной части реализован в тексте процедуры-распознавателя путем сравнения ожидаемого символа правой части и текущего символа входной строки;
- если в правой части ожидается терминальный символ, и он совпадает с очередным символом входной строки, то символ во входной строке пропускается, а распознаватель переходит к следующему символу правой части;
- несовпадение терминального символа правой части и очередного символа входной строки свидетельствует о синтаксической ошибке;
- если в правой части встречается нетерминальный символ, то для него необходимо вызвать аналогичную распознающую процедуру (функцию).

Использование рекурсивного спуска позволяет достаточно быстро и наглядно писать программу распознавателя на основе имеющейся грамматики. Главное, чтобы последняя соответствовала требуемому виду. Естественно, возникает вопрос: если грамматика не является LL(1), то существует ли эквивалентная КС-грамматика, для которой метод рекурсивного спуска применим? К сожалению, нет алгоритма, отвечающего на поставленный вопрос, т.е. это алгоритмически неразрешимая проблема.

Чтобы применить метод рекурсивного спуска, необходимо преобразовать грамматику к виду, в котором множества FIRST не пересекаются. Этот процесс может оказаться сложным. Поэтому на практике часто используется прием, называемый *рекурсивным спуском с возвратами*. Для этого лексический анализатор представляется в виде объекта, у которого помимо традиционных методов `scan, next` и т. п., есть также копирующий конструктор. Затем во всех ситуациях, где может возникнуть неоднозначность, перед началом разбора запоминается текущее состояние лексического анализатора (т.е. заводится копия лексического анализатора) и делается попытка продолжить разбор текста, считая, что рассматривается первая из возможных в данной ситуации конструкций. Если этот вариант разбора заканчивается неудачей, то восстанавливается состояние лексического анализатора и делается попытка заново разобрать тот же самый фрагмент с помощью следующего варианта грамматики и т. д. Если все варианты разбора заканчиваются неудачно, то вызывается функция обработки и нейтрализации ошибки. Такой метод разбора потенциально медленнее, чем рекурсивный спуск без возвратов, но в этом случае удается сохранить грамматику в ее оригинальном виде и сэкономить усилия программиста.

Именно таким образом реализован синтаксический разбор в демонстрационном компиляторе С-бемоль [2].

Пример. Рассматривается текст незавершенной программы распознавания вложенности скобок, использующей метод рекурсивного спуска.

```
// Рекурсивный распознаватель вложенности круглых скобок.
// Построен на основе правил следующей q-грамматики:
// 1. S -> (B)B
// 2. B -> (B)B
```

```
// 3. B -> empty

#include "stdafx.h"
#include <iostream>
#include <string>
#include <vector>
#include <stack>

using namespace std;

string str; // Строка с входной цепочкой, имитирующая входную ленту
int i;  // Текущее положение входной головки
int erFlag;  // Флаг, фиксирующий наличие ошибок в середине правила
// Функция, реализующая чтение символов в буфер из входного потока.
// Используется для ввода с клавиатуры распознаваемой строки.
// Ввод осуществляется до нажатия на клавишу Enter.
// Символ '\n' яляется концевым маркером входной строки.
void GetOneLine(istream &is, string &str) {
  char ch;
  str = "";
  for(;;) {
    is.get(ch);
    if(is.fail() || ch == '\n') break;
    str += ch;
  }
  str += '\n'; // Добавляется концевой маркер
}
// Функция, реализующая распознавание нетерминала B.
bool B() {
  if(str[i] == '(') {
    i++;
    if(B()) {
      if(str[i] == ')') {
        i++;
        if(B()) { // Если последний нетерминал корректен,
                  // то корректно и все правило.
          return true;
        } else { // Ошибка в нетерминале B
          erFlag++;
          cout << "Position " << i << ", "
               << "Error 1: Incorrect last B!\n";
          return false;
        }
      } else { // Это не ')'
        erFlag++;
        cout << "Position " << i << ", "
             << "Error 2: I want \')\'!\n";
        return false;
      }
    } else {
      erFlag++;
      cout << "Position " << i << ", "
           << "Error 3: Incorrect first B!\n";
      return false;
    }
  } else  // Смотрим другую альтернативу:
    return true; // Пустая цепочка допустима
}
// Функция, реализующая распознавание нетерминала S.
bool S() {
  if(str[i] == '(') {
    i++;
    if(B()) {
      if(str[i] == ')') {
        i++;
        if(B()) {
          if(str[i] == '\n') {
            return true; // за последним нетерминалом - конец
          } else { // Там ничего не должно быть
```

```
            erFlag++;
            cout << "Position " << i << ", "
              << "Error 4: I want string end after last right bracket!\n";
            return false;
          }
        } else {
          erFlag++;
          cout << "Position " << i << ", "
               << "Error 4: Incorrect last B!\n";
          return false;
        }
      } else {
        erFlag++;
        cout << "Position " << i << ", "
             << "Error 5: I want \')\'!\n";
        return false;
      }
    } else {
      erFlag++;
      cout << "Position " << i << ", "
           << "Error 6: Incorrect first B!\n";
      return false;
    }
  }
  return false; // Первый символ цепочки некорректен
  // то, что это ошибка, лучше определить снаружи
}
// Функция запускающая разбор и определяющая корректность его завершения,
// если первый символ не принадлежит цепочке
bool Parser() {
  // Начальная инициализация.
  erFlag = 0;
  i = 0;
  // Процесс пошел!
  if(S())
    return true; // Все прошло нормально
  else {
    if(erFlag)
      cout << "Position " << i << ", "
           << "Error 7: Internal Error!\n";
    else
      cout << "Position " << i << ", "
           << "Error 8: Incorrect first symbol of S!\n";
    return false; // Есть ошибки
  }
}

int _tmain(int argc, _TCHAR* argv[])
{
      ...
      return 0;
}
```

Варианты грамматик

Вариант 1. Грамматика G1

Рассматривается грамматика выражений отношения с правилами

<выражение> **->**
<простое выражение> **|**
<простое выражение> <операция отношения> <простое выражение>

<простое выражение> **->**
<терм> **|**
<знак> <терм> **|**
<простое выражение> <операция типа сложения> <терм>

<терм> **->**
<фактор> **|**
<терм> <операция типа умножения> <фактор>

<фактор> **->**
<идентификатор> **|**
<константа> **|**
(< простое выражение >) **|**
not <фактор>

<операция отношения> **->**
= **|** <> **|** < **|** <= **|** > **|** >=

<знак> **->**
+ **|** -

<операция типа сложения> **->**
+ **|** - **|** **or**

<операция типа умножения> **->**
* **|** / **|** **div** **|** **mod** **|** **and**

Замечания.

1. Нетерминалы <идентификатор> и <константа> **-** это лексические единицы (лексемы), которые оставлены неопределенными, а при выполнении лабораторной работы можно либо рассматривать их как терминальные символы, либо определить их по своему усмотрению и добавить эти определения.
2. Терминалы **not**, **or**, **div**, **mod**, **and -** ключевые слова (зарезервированные).
3. Терминалы ( ) **-** это разделители и символы пунктуации.
4. Терминалы = <> < <= > >= + **-** * / **-** это знаки операций.
5. Нетерминал <выражение> **-** это начальный символ грамматики.

Вариант 2. Грамматика G2

Рассматривается грамматика выражений отношения с правилами

<выражение> **->**
<арифметическое выражение> <операция отношения> <арифметическое выражение> **|**
<арифметическое выражение>

<арифметическое выражение> **->**
<арифметическое выражение> <операция типа сложения> <терм> **|**
<терм>

<терм> **->**
<терм> <операция типа умножения> <фактор> **|**

<фактор>

<фактор> ->
<идентификатор> |
<константа> |
(<арифметическое выражение>)

<операция отношения> ->
< | <= | = | <> | > | >=

<операция типа сложения> ->
+ | -

<операция типа умножения> ->
* | /

Замечания.

1. Нетерминалы <идентификатор> и <константа> - это лексические единицы (лексемы), которые оставлены неопределенными, а при выполнении лабораторной работы можно либо рассматривать их как терминальные символы, либо определить их по своему усмотрению и добавить эти определения.
2. Терминалы ( ) - это разделители и символы пунктуации.
3. Терминалы < <= = <> > >= + - * / - это знаки операций.
4. Нетерминал <выражение> - это начальный символ грамматики.

Вариант 3. Грамматика G3

Рассматривается грамматика выражений отношения с правилами

<выражение> ->
<арифметическое выражение> <знак операции отношения> <арифметическое выражение>

<арифметическое выражение> ->
<терм> |
<знак операции типа сложения> <терм> |
<арифметическое выражение> <знак операции типа сложения> <терм>

<терм> ->
<множитель> |
<терм> <знак операции типа умножения> <множитель>

<множитель> ->
<первичное выражение> |
<множитель> ^ <первичное выражение>

<первичное выражение> ->
<число> |
<идентификатор> |
(<арифметическое выражение>)

<знак операции типа сложения> ->
+ | -

<знак операции типа умножения> ->
* | / | %

<знак операции отношения> ->
< | <= | = | >= | > | <>

Замечания.

1. Нетерминалы <идентификатор> и <число> - это лексические единицы (лексемы), которые оставлены неопределенными, а при выполнении лабораторной работы можно либо рассматривать их как терминальные символы, либо определить их по своему усмотрению и добавить эти определения.
2. Терминалы ( ) - это разделители и символы пунктуации.
3. Терминалы + - * / % < <= = >= > <> - это знаки операций.
4. Нетерминал <выражение> - это начальный символ грамматики.

Вариант 4. Грамматика G4

Рассматривается грамматика логических выражений с правилами

<выражение> ->
    <логическое выражение>

<логическое выражение> ->
    <логический одночлен> |
    <логическое выражение> ! <логический одночлен>

<логический одночлен> ->
    <вторичное логическое выражение> |
    <логический одночлен> & <вторичное логическое выражение>

<вторичное логическое выражение> ->
    <первичное логическое выражение> |
    ~ <первичное логическое выражение>

<первичное логическое выражение> ->
    <логическое значение> |
    <идентификатор>

<логическое значение> ->
    true | **false**

<знак логической операции> ->
    ~ | & | !

Замечания.

1. Нетерминал <идентификатор> - это лексическая единица (лексемы), которая оставлена неопределенной, а при выполнении лабораторной работы можно либо рассматривать ее как терминальный символ, либо определить ее по своему усмотрению и добавить это
2. определение.
3. Терминалы **true**, **false** - ключевые слова (зарезервированные).
4. Терминалы ~ | & | ! - это знаки операций.
5. Нетерминал <выражение> - это начальный символ грамматики.

Вариант 5. Грамматика G5

Рассматривается грамматика выражений с правилами

<выражение> ->
    <отношение> { <логическая операция> <отношение> }

<отношение> ->
    <простое выражение> [<операция отношения> <простое выражение>]

<простое выражение> ->
    [<унарная аддитивная операция>] <слагаемое> { <бинарная аддитивная операция> <слагаемое> }

<слагаемое> ->

<множитель> { <мультипликативная операция> <множитель> }

<множитель> ->
<первичное> { ** <первичное> } |
abs <первичное> |
not <первичное>

<первичное> ->
<числовой литерал> |
<имя> |
(<выражение>)

<логическая операция> ->
and | **or** | **xor**

<операция отношения> ->
< | <= | = | /> | > | >=

<бинарная аддитивная операция> ->
+ | - | &

<унарная аддитивная операция> ->
+ | -

<мультипликативная операция> ->
* | / | **mod** | **rem**

<операции высшего приоритета> ->
** | **abs** | **not**

Замечания.

1. Нетерминалы <имя> и <числовой литерал> - это лексические единицы (лексемы), которые оставлены неопределенными, а при выполнении лабораторной работы можно либо рассматривать их как терминальные символы, либо определить их по своему усмотрению и добавить эти определения.
2. Терминалы ( ) - это разделители и символы пунктуации.
3. Терминалы < <= = /> > >= + - * / & ** - это знаки операций.
4. Терминалы **and**, **or**, **xor**, **mod**, **rem**, **abs**, **not** – это знаки операций (зарезервированные).
5. Нетерминал <выражение> - это начальный символ грамматики.

Задание на лабораторную работу

Дополнить грамматику блоком, состоящим из последовательности операторов присваивания. Для реализации предлагаются два варианта расширенной грамматики.

Вариант в стиле Алгол-Паскаль.

<программа> ->
<блок>

<блок> ->
begin <список операторов> **end**

<список операторов>
<оператор> | <список операторов> **;** <оператор>

<оператор> ->
<идентификатор> = <выражение>

Вариант в стиле Си.

<программа> ->
<блок>

<блок> ->
{ <список операторов> }

<список операторов>
<оператор> <хвост>

<хвост> ->
; <оператор> <хвост> | ε

Первый вариант содержит левую рекурсию, которая должна быть устранена. Второй вариант не содержит левую рекурсию, но имеет ε-правило. В обоих вариантах точка с запятой (;) ставится между операторами. Теперь начальным символом грамматики становится нетерминал <программа>. Оба варианта содержат цепное правило <программа> -> <блок>. Можно начальным символом грамматики назначить нетерминал <блок>. А можно <блок> считать оператором, т. е.

<оператор> ->
<идентификатор> = <выражение> |
<блок>

В последнем случае возможна конструкция с вложенными блоками. Если между символом присваивания (=) и символом операции отношения (=) возникает конфликт, то можно для любого из них ввести новое изображение, например, :=, <-, == и т. п.

Для модифицированной грамматики написать программу нисходящего синтаксического анализа с использованием метода рекурсивного спуска.

Рекомендуемая литература

1. АХО А.В, ЛАМ М.С., СЕТИ Р., УЛЬМАН Дж.Д. Компиляторы: принципы, технологии и инструменты. – М.: Вильямс, 2008.
2. Разработка компиляторов на платформе .NET / А. Терехов, Н. Вояковская, Д. Булычев, А. Москаль.
3. D. Grune, C. H. J. Jacobs "Parsing Techniques - A Practical Guide", Ellis Horwood, 1990.
   URL: http://www.cs.vu.nl/~dick/PTAPG.html
4. Introduction to Recursive Descent Parsing.
   URL: http://ag-kastens.uni-paderborn.de/lehre/material/uebi/parsdemo/recintro.html
5. Parsing Expressions by Recursive Descent.
   URL: http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm
6. Recursive descent parsing - add expressions.
   URL: http://pages.cpsc.ucalgary.ca/~robin/class/411/intro.3.html