Министерство образования и науки РФ
Севастопольский государственный университет

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению лабораторных работ
по дисциплине “Системное программное обеспечение ”

для студентов по направлению подготовки 09.03.01
«Информатика и вычислительная техника»
всех форм обучения

Севастополь
2015

Методические указания содержат краткое изложение основных теоретических положений, задания на лабораторные работы, порядок их выполнения, список рекомендованной литературы.

Методические указания утверждены на заседании кафедры информационных технологий и компьютерных систем __28 января__ 2015г, протокол N_1_

Методические указания разработали к.т.н., доцент Максимова Т.М., к.т.н., доцент Фисун С.Н.., ст.преподаватель Шалимова Е.М.
Рецензент: д.т.н., профессор Цуканов А.В.

СОДЕРЖАНИЕ

1. ЛАБОРАТОРНАЯ РАБОТА № 1. ПРОГРАММИРОВАНИЕ СКАНЕРА

Цель работы: Изучение организации данных лексического анализатора, приобретение навыков в построении регулярных грамматик и их автоматных моделей и программировании алгоритмов разбора с помощью таких грамматик.

1.1 Теоретические сведения

Лексический анализ представляет собой первую фазу процесса компиляции, при которой отдельные литеры входной цепочки группируются в слова (лексемы). Каждому распознанному слову входного языка ставится в соответствие некоторое внутреннее представление. Часто термин "лексема" относят и к этому внутреннему представлению. Во избежание двусмысленности будем называть внутреннее представление лексемы кодом лексемы.

Например, если предложения входного языка представляют собой списки идентификаторов, разделенных запятыми, то результатом лексического анализа предложения abc,d1,ef2 при условии, что коды лексем зафиксированы в таблице лексем (таблица 1), будет следующий список кодов: 2 1 3 1 4 .

Таблица 1.1

Лексема	Код лексемы
,	1
abc	2
d1	3
ef2	4

Множество различных лексем языка программирования обычно бесконечно. Поэтому формирование таблицы лексем выполняется в процессе анализа конкретной входной цепочки. При этом каждый экземпляр определенной лексемы, присутствующий во входной цепочке, должен получить в выходном списке один и тот же код, что обеспечивается наличием вышеуказанной таблицы.

Код лексемы обычно представляет собой пару вида (тип лексемы, некоторые данные) /1/. Первая компонента пары является синтаксической категорией, указывая принадлежность лексемы какой-либо непосредственной составляющей грамматики. В грамматиках языков программирования, как правило, к таким непосредственным составляющим относятся: идентификатор, константа, разделитель и др. Вторая компонента кода может быть указателем на месторасположение информации об этой конкретной лексеме, в частности, - просто номером соответствующей записи в таблице лексем.

Обозначим в вышерассмотренном примере тип лексемы символом "r", если она является запятой, и символом "i", если она принадлежит категории <идентификатор>. Тогда код лексемы "," приобретет вид: r1, а коды лексем "abc", "d1" и "ef2", соответственно: i2, i3 и i4. При этом вторая компонента кода является указателем на соответствующую запись таблицы лексем.

Синтаксис лексем, как правило, описывается в рамках автоматной грамматики, или грамматики типа 3 в соответствии с классификацией Хомского /2/. Это означает, что лексический анализатор (сканер) может быть организован в виде модели конечного автомата. Так, если порождающие правила грамматики имеют вид: <U>::=<W>N и <U>::=N, где <U>,<W> - нетерминальные, а N - терминальный символы грамматики, то соответствующий автомат определяется пятеркой /2/:

1) конечное множество V внутренних состояний, каждое из которых, за исключением одного (обозначим его - F), соответствует нетерминальному символу грамматики;

2) конечный входной алфавит T, каждый символ которого соответствует терминальному символу грамматики;

3) множество P переходов; при этом правилу грамматики <U>::=<W>N в автоматной модели соответствует переход P(W,N)=U, а правилу <U>::=N - переход P(F,N)=U;

4) начальное состояние F;

5) множество S последних состояний, соответствующее множеству нетерминалов грамматики, являющихся альтернативными определениями ее аксиомы (начального символа). Попадание автомата в одно из состояний множества S означает, что процесс распознавания входного слова закончен. Можно сказать, что каждому из последних состояний соответствует свой выходной символ (свойство модели автомата Мура /3/), при появлении которого на выходе автомата запускается соответствующая семантическая подпрограмма сканера, формирующая код распознанной лексемы.

Например, грамматике G[<S>] с порождающими правилами:

<S>::=<X>$
<X>::=<U>0|<V>1
<U>::=<X>1|1
<V>::=<X>0|0

соответствует автоматная модель, представленная на рис.1 в виде графа переходов и выходов и на рис.2 в виде матрицы переходов и выходов, где y1 - символ, сигнализирующий об успешном окончании распознавания слова, поступившего на вход автомата.

Граф переходов и выходов конечного автомата Мура

0100090000038d00000002001c0000000000040000000301080005000000 0b020000000050000000c02a2070907040000002e0118001c000000fb02100007000000000bc02000000cc0102022253797374656d00774c9ab8006ae6f876803203770100000 0f0c4ca090b000000040000002d010000040000002d0100000400000000020101001c000000fb02a4ff000000000000900100000 0cc0440002243616c69627269000000000000000000000000000000000000000000000000000000000040000002d010100040000002d010100040000002d010100500000009020000000 20d000000320a57000000010004000000000008079e0720003600050000000902000000020400000 02d010000040000002d010000030000000000

Рис.1.1

Матрица переходов и выходов конечного автомата Мура

	F	U	V	X	S
0	V	X	-	V	-
1	U	-	X	U	-
$	-	-	-	S	-
	y0	y0	y0	y0	y1

Рис.1.2

Символами "-" в матрице переходов отмечены запрещенные переходы в автомате, попадание в соответствующую клетку матрицы переходов означает наличие ошибки во входной цепочке терминальных символов. Можно ввести дополнительное внутреннее состояние автомата - E("ошибка") - и заменить в матрице переходов символы "-" символами состояния E, снабдив его выходом yE, по которому запускается семантическая подпрограмма обработки ошибочной ситуации.

Построенный автомат является автоматом однократного действия, т.е. предназначен для распознавания одного слова входной цепочки. Для преобразования его в автомат многократного действия можно, полагая состояние S неустойчивым, описать переход из него при неизменном входном символе в начальное состояние F. Другой вариант преобразования - построить автомат Мили /3/, в котором состояния F и S исходного автомата будут объединены, и переход в это объединенное состояние будет сопровождаться появлением выходного символа y1. Граф переходов для такой автоматной модели изображен на рис.3.

Граф переходов и выходов конечного автомата Мили

0100090000038d00000002001c0000000000040000000301080005000000 0b020000000005000000 0c02a2070907040000002e0118001c000000fb02100007000000000bc02000000cc0102022253797 374656d00774c9ab8006ae6f876803203770100000f0c4ca090b000000040000002d010000040000 002d010000400000002010100 1c000000fb02a4ff000000000000900100000cc0440002243616c69 6272690000000000000000000000000000000000000000000000000040000002d01010004000000 2d010100040000002d0101000500000009020000000 20d000000320a57000000010004000000000 08079e0720003600050000000902000000020400000 2d010000040000002d010000030000000000

Рис. 1.3

В ЭВМ автоматная модель может быть представлена /2/ двумерным массивом, соответствующим матрице переходов, или списочной структурой. Во втором случае каждое состояние с k дугами, исходящими из него, занимает 2*k+2 слов. Первое слово - имя состояния, второе - значение k. Каждая последующая пара слов содержит терминальный символ из входного алфавита и указатель на начало представления состояния, в которое надо перейти по этому символу. Для вышерассмотренного графа переходов список имеет вид:

F 2 1 7 0 11 U 1 0 15 V 1 1 15 X 3 0 11 1 7
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

$ 23 S 0
21 22 23 24

1.2.Порядок выполнения работы

1.2.1.Построить регулярную грамматику для заданного языка(в соответствии с вариантом).При распознавании лексемы выбирается самое короткое слово входной цепочки.
1.2.2.Построить конечный автомат для полученной грамматики.
1.2.3.Разработать и отладить лексический анализатор - препроцессор на основе построенной автоматной модели. Лексический анализатор должен быть оформлен в виде отдельной процедуры (подпрограммы).

1.3.Задание на лабораторную работу

Варианты заданий приведены в таблице 2.

Таблица 1.2

номер варианта	код задания	номер варианта	код задания
1	П 1	11	С 1
2	П 2	12	С 2
3	П 3	13	С 3
4	П 4	14	С 4
5	П 5	15	С 5
6	П 6	16	С 6
7	П 7	17	С 7
8	П 8	18	С 8
9	П 9	19	С 9
10	П 10	20	С 20

Код задания состоит из двух компонент. Первая - П или С - определяет язык программирования, ПАСКАЛЬ или СИ соответственно; вторая - номер варианта описания языка.

Варианты описаний языков.

1) Цепочка символов "a" произвольной длины, после которой следует символ "b";
   цепочка символов "a" произвольной длины, после которой следует символ "c";
   цепочка символов "b" произвольной длины, после которой следуют "a"или "c".
2) Цепочка пар символов "a""b"произвольной длины, после которой следует "b";
   цепочка пар символов "b""a"произвольной длины, после которой следует "c";
   символ "c".
3) Произвольная цепочка символов из "a","b","c", заканчивающаяся "a","b","c";
   произвольная цепочка символов из "a","b","c", заканчивающаяся "c","b","a".
4) Три подряд пришедших символа "a" в произвольной цепочке из "a" и "b", после которых следует "b";
   три подряд пришедших символа "b", после которых следует "a";
   три подряд пришедших символа "b", после которых следует "c".
5) Произвольное число символов "a" между двумя символами "b"; произвольное число символов "b" между двумя символами "c"; три подряд пришедших символа "c".
6) Произвольная цепочка из 0 и 1 между /* и */; последовательность двух пар 01; символ *.
7) Произвольная цепочка из 0 и 1,после которой следует "."; цепочка четной длины из 0 и 1 между двумя символами "."; два символа ".".
8) Цепочка четной длины из 0 между двумя 1; цепочка нечетной длины из 1 между двумя 0; две 1 подряд.
9) 1 между двумя цепочками из 0, четной длины каждая, после которых следует символ «**.**»;
   0 между двумя цепочками из 1, четной длины каждая, после которых следует символ «**.**».
10) Две 1 ,за которыми следует два 0;
    цепочка чередующихся 0 и 1 нечетной длины, за которой следует ".".

1.4.Содержание отчета

1) Постановка задачи.
2) Регулярная грамматика.
3) Граф автоматной модели грамматики и соответствующая таблица переходов и выходов.
4) Текст программы.

5) Описание тестовых примеров.
6) Распечатка результатов.

Библиографический список

1.А.Ахо, Дж. Ульман. Теория синтаксического анализа, перевода и компиляции. Т.1. Синтаксический анализ. - М.: Мир,1978. - 612с.
2.Д.Грис. Конструирование компиляторов для цифровых вычислительных машин. - М.: Мир, 1975. - 544с.
3.Р.Миллер. Теория переключательных схем. Т.1. - М.: Наука. Главная редакция физико-математической литературы, 1970. - 416с.

2. ЛАБОРАТОРНАЯ РАБОТА № 2. СИНТАКСИЧЕСКИЙ АНАЛИЗ МЕТОДОМ РЕКУРСИВНОГО СПУСКА

Цель лабораторной работы.

Изучение методов синтаксического анализа, получение навыков в программировании алгоритмов синтаксического разбора.

2.1.Теоретическое введение

2.1.1.Постановка задачи синтаксического анализа. Организация данных. Общая схема алгоритмов детерминированного разбора

Синтаксический анализ (разбор) - это процесс, в котором исследуется цепочка лексем и устанавливается, удовлетворяет ли она условиям, сформулированным в определении синтаксиса языка /1/. Выходом синтаксического анализатора является синтаксическое дерево разбора, которое представляет синтаксическую структуру исходной программы.

Пример.

Зададим грамматику G(V,T,P,S) следующим набором правил (полагая, что терминальные символы в правых частях правил представлены соответствующими кодами лексем):

1) <S> ::= <E>
2) <E> ::= <T> + <E>
3) <E> ::= <T>
4) <T> ::= <F> * <T>
5) <T> ::= <F>
6) <F> ::= i

В результате синтаксического анализа цепочки i*i+i может быть построено синтаксическое дерево разбора (рис. 1) .

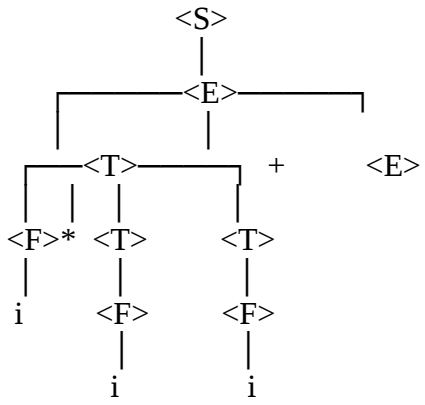


Рис.2.1. Синтаксическое дерево разбора

2.1.2 Метод рекурсивного спуска

Пример: пусть дана грамматика G =({a,b,c, ⊥}, {S,A,B}, P, S), где

P: S AB⊥
A a | cA
B bA

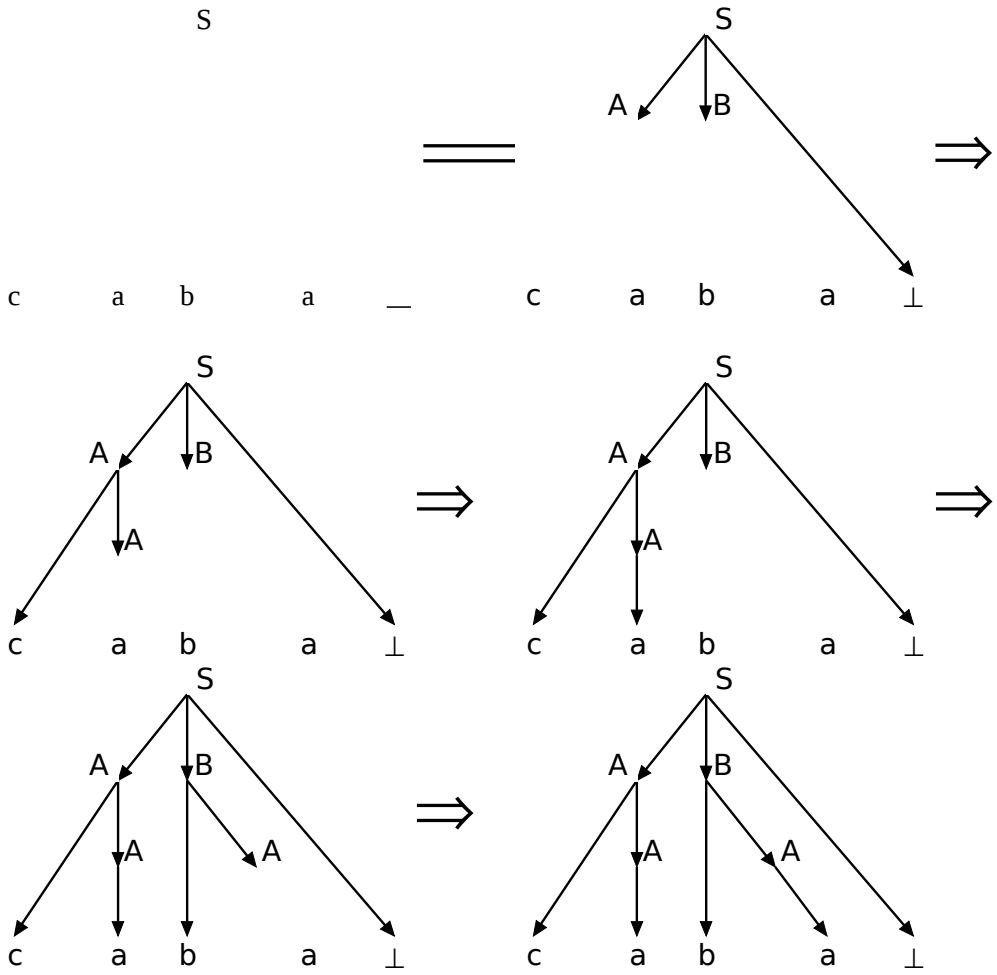
и надо определить, принадлежит ли цепочка caba языку L(G).

Построим вывод этой цепочки:

S AB⊥ cAB⊥ caB⊥ cabA⊥ caba⊥

Следовательно, цепочка принадлежит языку L(G).

Последовательность применений правил вывода эквивалентна построению дерева разбора методом "сверху вниз":



Метод рекурсивного спуска (РС-метод) реализует этот способ практически "в лоб": для каждого нетерминала грамматики создается своя процедура, носящая его имя; ее задача - начиная с указанного места исходной цепочки найти подцепочку, которая выводится из этого нетерминала. Если такую подцепочку считать не удается, то процедура завершает свою работу вызовом процедуры обработки ошибки, которая выдает сообщение о том, что цепочка не принадлежит языку, и останавливает разбор. Если подцепочку удалось найти, то работа процедуры считается нормально завершенной и осуществляется возврат в точку вызова. Тело каждой такой процедуры пишется непосредственно по правилам вывода соответствующего нетерминала: для правой части каждого правила осуществляется поиск подцепочки, выводимой из этой правой части. При этом терминалы распознаются самой процедурой, а нетерминалы соответствуют вызовам процедур, носящих их имена.

Пример: совокупность процедур рекурсивного спуска для грамматики

G = ({a,b,c,⊥}, {S,A,B}, P, S), где

P: S AB⊥
A a | cA
B bA

будет такой:

#include <stdio.h>

```
int c;
FILE *fp;/*указатель на файл, в котором находится анализируемая цепочка */
void A();
void B();
void ERROR(); /* функция обработки ошибок */
void S() {A(); B();
       if (c != '⊥') ERROR();
       }
void A() {if (c=='a') c = fgetc(fp);
       else if (c == 'c') {c = fgetc(fp); A();}
          else ERROR();
       }
void B() {if (c == 'b') {c = fgetc(fp); A();}
        else ERROR();
       }
void ERROR() {printf("ERROR !!!\n"); exit(1);}
main() {fp = fopen("data","r");
      c = fgetc(fp);
      S();
      printf("SUCCESS !!!\n"); exit(0);
     }
```

2.1.3 О применимости метода рекурсивного спуска

Метод рекурсивного спуска применим в том случае, если каждое правило грамматики имеет вид:

a) либо A α, где $\alpha \in (VT \cup VN)^*$ и это единственное правило вывода для этого нетерминала;

b) либо A $a_1\alpha_1 \mid a_2\alpha_2 \mid ... \mid a_n\alpha_n$, где $a_i \in VT$ для всех i = 1,2,...,n; $a_i \neq a_j$ для $i \neq j$; $\alpha_i \in (VT \cup VN)^*$, т. е. если для нетерминала A правил вывода несколько, то они должны начинаться с терминалов, причем все эти терминалы должны быть различными.

Ясно, что если правила вывода имеют такой вид, то рекурсивный спуск может быть реализован по выше изложенной схеме.

Естественно, возникает вопрос: если грамматика не удовлетворяет этим условиям, 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.**

Изложенные выше ограничения являются достаточными, но не необходимыми. Попытаемся ослабить требования на вид правил грамматики:

(1) при описании синтаксиса языков программирования часто встречаются правила, описывающие последовательность однотипных конструкций, отделенных друг от друга каким-либо знаком-разделителем (например, список идентификаторов при описании переменных, список параметров при вызове процедур и функций и т.п.).

Общий вид этих правил:

L a | a,L (либо в сокращенной форме L a {,a})

Формально здесь не выполняются условия применимости метода рекурсивного спуска, т.к. две альтернативы начинаются одинаковыми терминальными символами.

Действительно, в цепочке a,a,a,a,a из нетерминала L может выводиться и подцепочка a, и подцепочка a,a , и вся цепочка a,a,a,a,a. Неясно, какую из них выбрать в качестве подцепочки, выводимой из L. Если принять решение, что в таких случаях будем выбирать самую длинную подцепочку (что и требуется при разборе реальных языков), то разбор становится детерминированным.

Тогда для метода рекурсивного спуска процедура L будет такой:

```
void L()
{ if (c != 'a') ERROR();
  while ((c = fgetc(fp)) == ',')
    if ((c = fgetc(fp)) != 'a') ERROR();
}
```

Важно, чтобы подцепочки, следующие за цепочкой символов, выводимых из L, не начинались с разделителя (в нашем примере - с запятой), иначе процедура L попытается считать часть исходной цепочки, которая не выводится из L. Например, она может порождаться нетерминалом B - ”соседом” L в сентенциальной форме, как в грамматике

$S \rightarrow LB\perp$

$L \rightarrow a \{, a\}$

$B \rightarrow ,b$

Если для этой грамматики написать анализатор, действующий РС-методом, то цепочка a,a,a,b будет признана им ошибочной, хотя в действительности это не так.

Нужно отметить, что в языках программирования ограничителем подобных серий всегда является символ, отличный от разделителя, поэтому подобных проблем не возникает.

(2) если грамматика не удовлетворяет требованиям применимости метода рекурсивного спуска, то можно попытаться преобразовать ее, т.е. получить эквивалентную грамматику, пригодную для анализа этим методом.

a) если в грамматике есть нетерминалы, правила вывода которых леворекурсивны, т.е. имеют вид

$A \quad A\alpha_1 \mid ... \mid A\alpha_n \mid \beta_1 \mid ... \mid \beta_m,$

где $\alpha_i \in (VT \cup VN)^+$, $\beta_j \in (VT \cup VN)^*$, i = 1, 2, ..., n; j =1, 2 ,..., m, то непосредственно применять РС- метод нельзя.

Левую рекурсию всегда можно заменить правой:

$A \quad \beta_1 A' \mid ... \mid \beta_m A'$

$A' \quad \alpha_1 A' \mid ... \mid \alpha_n A' \mid \varepsilon$

Будет получена грамматика, эквивалентная данной, т.к. из нетерминала A по-прежнему выводятся цепочки вида $\beta_j \{\alpha_i\}$, где i = 1,2,...,n; j = 1,2,...,m.

b) если в грамматике есть нетерминал, у которого несколько правил вывода начинаются одинаковыми терминальными символами, т.е. имеют вид

$A \quad a\alpha_1 \mid a\alpha_2 \mid ... \mid a\alpha_n \mid \beta_1 \mid ... \mid \beta_m,$

где $a \in VT$; $\alpha_i, \beta_j \in (VT \cup VN)^*$, то непосредственно применять РС-метод нельзя. Можно преобразовать правила вывода данного нетерминала, объединив правила с общими началами в одно правило:

$A \quad aA' \mid \beta_1 \mid ... \mid \beta_m$

$A' \quad \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$

Будет получена грамматика, эквивалентная данной.

c) если в грамматике есть нетерминал, у которого несколько правил вывода, и среди них есть правила, начинающиеся нетерминальными символами, т.е. имеют вид

$A \quad B_1\alpha_1 \mid ... \mid B_n\alpha_n \mid a_1\beta_1 \mid ... \mid a_m\beta_m$

$B_1 \quad \gamma_{11} \mid ... \mid \gamma_{1k}$

. . .

$B_n \quad \gamma_{n1} \mid ... \mid \gamma_{np}$,

где $B_i \in VN$; $a_j \in VT$; α_i, β_j, $\gamma_{ij} \in (VT \cup VN)^*$, то можно заменить вхождения нетерминалов B_i их правилами вывода в надежде, что правило нетерминала A станет удовлетворять требованиям метода рекурсивного спуска:

$A \quad \gamma_{11}\alpha_1 \mid ... \mid \gamma_{1k}\alpha_1 \mid ... \mid \gamma_{n1}\alpha_n \mid ... \mid \gamma_{np}\alpha_n \mid a_1\beta_1 \mid ... \mid a_m\beta_m$

d) если допустить в правилах вывода грамматики пустую альтернативу, т.е. правила вида

$A \quad a_1\alpha_1 \mid ... \mid a_n\alpha_n \mid \varepsilon$,

то метод рекурсивного спуска может оказаться неприменимым (несмотря на то, что в остальном достаточные условия применимости выполняются).

Например, для грамматики G = ({a,b}, {S,A}, P, S), где

P: S bAa

A aA | ε

РС-анализатор, реализованный по обычной схеме, будет таким:

```
void S(void)
{if (c == 'b') {c = fgetc(fp); A();
          if (c != 'a') ERROR();}
 else ERROR();
}

void A(void)
{if (c == 'a') {c = fgetc(fp); A();}
}
```

Тогда при анализе цепочки baaa функция A() будет вызвана три раза; она прочитает подцепочку aaa, хотя третий символ a - это часть подцепочки, выводимой из S. В результате окажется, что baaa не принадлежит языку, порождаемому грамматикой, хотя в действительности это не так.

Проблема заключается в том, что подцепочка, следующая за цепочкой, выводимой из A, начинается таким же символом, как и цепочка, выводимая из A.

Однако в грамматике G = ({a,b,c}, {S,A}, P, S), где

P: S bAc

A aA | ε

нет проблем с применением метода рекурсивного спуска.

Выпишем условие, при котором ε-правило вывода делает неприменимым РС-метод.

Определение: *множество FIRST(A)* - это множество терминальных символов, которыми начинаются цепочки, выводимые из A в грамматике

G = (VT, VN, P, S), т.е. $FIRST(A) = \{ a \in VT \mid A \Rightarrow a\alpha, A \in VN, \alpha \in (VT \cup VN)^* \}$.

Определение: *множество FOLLOW(A)* -это множество терминальных символов, которые следуют за цепочками, выводимыми из A в грамматике

G = (VT, VN, P, S), т.е. $FOLLOW(A) = \{ a \in VT \mid S \Rightarrow \alpha A \beta, \ \beta \Rightarrow a\gamma, A \in VN, \alpha, \beta, \gamma \in (VT \cup VN)^* \}$.

Тогда, если $\mathbf{FIRST(A) \cap FOLLOW(A) \neq \varnothing}$ **,** то метод рекурсивного спуска неприменим к данной грамматике.

Если

A $\alpha_1 A \mid ... \mid \alpha_n A \mid \beta_1 \mid ... \mid \beta_m \mid \varepsilon$

B $\alpha A \beta$

и FIRST(A) $\cap$ FOLLOW(A) $\neq \varnothing$ (из-за вхождения A в правила вывода для B), то можно попытаться преобразовать такую грамматику:

$\alpha A'$

$A' \; \alpha_1 A' \mid ... \mid \alpha_n A' \mid \beta_1 \beta \mid ... \mid \beta_m \beta \mid \beta$

A $\alpha_1 A \mid ... \mid \alpha_n A \mid \beta_1 \mid ... \mid \beta_m \mid \varepsilon$

Полученная грамматика будет эквивалентна исходной, т.к. из B по-прежнему выводятся цепочки вида $\alpha \{\alpha_i\} \beta_j \beta$ либо $\alpha \{\alpha_i\} \beta$.

Однако правило вывода для нетерминального символа A’ будет иметь альтернативы, начинающиеся одинаковыми терминальными символами, следовательно, потребуются дальнейшие преобразования, и успех не гарантирован.

Метод рекурсивного спуска применим к достаточно узкому подклассу КС-грамматик. Известны более широкие подклассы КС-грамматик, для которых существуют эффективные анализаторы, обладающие тем же свойством, что и анализатор, написанный методом рекурсивного спуска, - входная цепочка считывается один раз слева направо и процесс разбора полностью детерминирован, в результате на обработку цепочки длины n расходуется время cn. К таким грамматикам относятся LL(k)-грамматики, LR(k)-грамматики, грамматики предшествования и некоторые другие (см., например, [2], [3]).

2.2. Порядок выполнения работы

2.2.1 Произвести проверку применимости метода рекурсивного спуска, при необходимости выполнить существующие преобразования грамматики.

2.2.2 Разработать и отладить синтаксический анализатор, который должен быть оформлен в виде отдельной процедуры (подпрограммы).

2.3. Задание на лабораторную работу

Разработать синтаксический анализатор подмножества заданного языка программирования на основе метода рекурсивного спуска. Варианты заданий приведены в таблице 2.1. Код задания на лабораторную работу состоит из трех компонент.

Первая цифра варианта определяет язык программирования (1- PL0, 2- ASPLE). Описание синтаксиса языков программирования PL0 и ASPLE приведены соответственно в приложениях 1 и 2.

Вторая цифра варианта определяет подмножество языка программирования. Описание подмножеств языков программирования PL0 и ASPLE приведены соответственно в приложениях 3 и 4.

Таблица 2.1

Номер варианта	1	2	3	4	5	6	7	8
Код задания	11П	12С	13П	14С	21П	22С	23П	24С
Номер варианта	9	10	11	12	13	14	15	16
Код задания	11С	12П	13С	14П	21С	22П	23С	24П

Третья цифра-язык программирования: П- Паскаль, С -Си.

2.4.Содержание отчета

4.1.Постановка задачи.
4.2.Исходная и преобразованная грамматики.
4.3.Текст программы.
4.4.Описание тестовых примеров.
4.5.Распечатка результатов.

Библиографический список

1. А.Ахо, Дж.Ульман. Теория синтаксического анализа, перевода и компиляции. Т.1. Синтаксический анализ. - М.: Мир, 1978. - 612с.
2. Р.Хантер. Проектирование и конструирование компиляторов. - М.: Финансы и статистика, 1984. - 232с.
3. Д.Грис. Конструирование компиляторов для ЦВМ. - М.: Мир, 1975.- 544 с.
4. Волкова И.А., Руденко Т.В. Формальные грамматики и языки. Элементы теории трансляции. –МГУ,1998.

3. ЛАБОРАТОРНАЯ РАБОТА № 3. ПРОГРАММИРОВАНИЕ НИСХОДЯЩЕГО СИНТАКСИЧЕСКОГО LL(1) - АНАЛИЗАТОРА

Цель работы: Изучение нисходящего синтаксического анализа методом **LL(1)**, получение навыков в программировании алгоритмов синтаксического разбора

3.1. Постановка задачи синтаксического анализа. Организация данных. Общая схема алгоритмов детерминированного разбора

Синтаксический анализ (разбор) - это процесс, в котором исследуется цепочка лексем и устанавливается, удовлетворяет ли она условиям, сформулированным в определении синтаксиса языка /1/. Выходом синтаксического анализатора является синтаксическое дерево разбора, которое представляет синтаксическую структуру исходной программы.

Пример.

Зададим грамматику G(V,T,P,S) следующим набором правил (полагая, что терминальные символы в правых частях правил представлены соответствующими кодами лексем):

1) <S> ::= <E>
2) <E> ::= <T> + <E>
3) <E> ::= <T>
4) <T> ::= <F> * <T>
5) <T> ::= <F>
6) <F> ::= i

В результате синтаксического анализа цепочки i*i+i может быть построено синтаксическое дерево разбора (рис. 1) .

0100090000038d00000002001c0000000000040000000301080005000000b020000000050000000c0
2a2070907040000002e0118001c000000fb02100007000000000bc02000000cc0102022253797374656
d00774c9ab8006ae6f876803203770100000f0c4ca090b000000040000002d010000040000002d010000
0400000020101001c000000fb02a4ff000000000000900100000cc0440002243616c696272690000000
0000000000000000000000000000000000000000000040000002d010100040000002d01010004000000
2d0101000500000009020000000020d000000320a57000000010004000000000008079e0720003600050
000000902000000020400000002d010000040000002d010000030000000000

Рис.3.1. Синтаксическое дерево разбора

Прежде, чем приступить к изложению методов решения задачи синтаксического разбора, зафиксируем формы представления ее входных и выходных данных, их выбор существенно отражается на содержании алгоритма.

Рассмотрим возможную организацию структур перечисленных данных синтаксического анализатора. Входная цепочка лексем, очевидно, должна быть представлена последовательностью кодов фиксированной длины. Для представления грамматики удобно использовать списковую структуру /3/, каждая запись которой представляет символ С из правой части правила и состоит из четырех компонент (рис.3.2).

ИМЯ		
Опр.	Алт.	Преем.

Имя - это сам символ С (в некоторой внутренней форме);

Опр - указатель на запись, соответствующую первому символу в правой части для С (0, если С - терминал);

Алт - указатель на первый символ альтернативной правой части, которая следует за правой частью, содержащей данный символ (используется только в записях первых символов правых частей);

Преем - указатель на следующий символ правой части.

Рис.3.2.Структура записи

Кроме того, для каждого нетерминала необходимо иметь указатель на первый символ в первой правой части правила этого нетерминала. Для облегчения поиска правила по его номеру эта дополнительная запись для каждого нетерминала помимо указателя может быть снабжена номером, который присвоен первому правилу данного нетерминала.

Списковая структура для грамматики вышеприведенного примера представлена на рис.3.

0100090000038d00000002001c0000000000040000000301080005000000b020000000050000000c02a
2070907040000002e0118001c000000fb021000070000000000bc02000000cc0102022253797374656d00
774c9ab8006ae6f8768032037701000000f0c4ca090b000000040000002d010000040000002d0100000400
0000020101001c000000fb02a4ff000000000000090010000000cc0440002243616c6962726900000000000000
0000000000000000000000000000000000000000040000002d010100040000002d010100040000002d01010
00500000009020000000020d000000320a57000000010004000000000008079e0720003600050000000902
0000000204000002d010000040000002d010000030000000000Рис.3.3. Списковая структура для заданной грамматики Синтаксическое дерево разбора может быть представлено с помощью стека, каждая запись которого содержит символ вершины дерева и указатель на запись, являющуюся ее предком. Дерево, полученное в вышерассмотренном примере, будет представлено стеком (таблица 3.1).

Таблица 3.1

14	i , 13
13	<F>, 12
12	<T>, 5
11	i , 10
10	<F>, 8
9	i , 6
8	<T>, 3
7	*, 3
6	<F>, 3
5	<E>, 2
4	*, 2
3	<T>, 2
2	<E>, 1
1	<S>, 0

При выполнении нисходящего разбора начальный символ грамматики будет размещаться на дне стека, а при выполнении восходящего разбора - в вершине стека.

Алгоритмы, предлагаемые для изучения в данной лабораторной работе, положены в основу

функционирования однопроходных (детерминированных) синтаксических анализаторов. Возможность построения детерминированного анализатора обеспечивается ограничениями на класс контекстно-свободных (КС) грамматик, для которого проектируется этот анализатор. К наиболее распространенным таким классам грамматик, которые практически отражают все синтаксические черты языков программирования, определяемых с помощью КС-грамматик, относятся /1/:

1) LL(k)-грамматики, для которых левый анализатор работает детерминировано, если позволить ему принимать во внимание k входных символов, расположенных справа от текущей входной позиции;

2) LR(k)-грамматики, для которых правый анализатор работает детерминировано, если позволить ему принимать во внимание k входных символов, расположенных справа от текущей входной позиции;

3) грамматики предшествования, для которых правый анализатор может находить основу правовыводимой цепочки, учитывая только некоторые отношения между парами ее смежных символов.

Рассмотрим алгоритмы функционирования синтаксического анализатора в применении к LL(1)-, LR(1)-грамматикам и грамматикам простого предшествования.

Помимо вышеперечисленных структур данных (входные: цепочка лексем, описание грамматики языка; выходные: стек, содержащий синтаксическое дерево разбора), этими анализаторами используются еще две структуры: управляющая таблица и рабочий стек, предназначенный для хранения текущей сентенциальной формы в процессе выполнения разбора. Все три алгоритма работают по одной общей схеме, которая коротко может быть описана следующим образом. Входная цепочка просматривается слева направо, символ за символом. В зависимости от содержимого вершины рабочего стека (определяющего строку таблицы) и текущего входного символа (определяющего столбец таблицы) с помощью управляющей таблицы решается вопрос о выполнении одной из процедур: занесение входного символа в рабочий стек или преобразование содержимого вершины рабочего стека в соответствии с каким-либо правилом грамматики и с переносом прежнего содержимого вершины этого стека в выходной стек. Успешное завершение разбора входной цепочки происходит по окончании ее просмотра при условии, что рабочий стек оказывается пустым. Другой вариант окончания разбора - получение сообщения о том, что входная цепочка не принадлежит языку, описанному заданной грамматикой. Появление такого сообщения предусматривается управляющей таблицей, которая содержит его в своих элементах, соответствующих несочетаемым парам "содержимое вершины стека - входной символ".

Построение управляющей таблицы (автоматическое или вручную) выполняется во время проектирования синтаксического анализатора, и для алгоритма разбора она является постоянной таблицей. Ее построение выполняется в результате анализа порождающих правил грамматики языка, определения свойства этой грамматики (LL(k),LR(k) или предшествования) и, возможно, преобразования ее правил, если они не удовлетворяют желаемому свойству. Строки управляющей таблицы ставятся в соответствие возможным символам в вершине рабочего стека, столбцы - символам входного алфавита языка. Элемент таблицы, находящийся на пересечении заданной строки и заданного столбца, содержит инструкцию о следующем действии алгоритма.

Остановимся теперь на особенностях каждого из трех вышеперечисленных алгоритмов синтаксического разбора.

3.2.1 Синтаксический анализ для LL(1)-грамматики

Алгоритм синтаксического анализа на основе LL(K)-грамматики относится к классу алгоритмов нисходящего разбора.

Строкам управляющей таблицы M для грамматики G(V,T,P,S) /1/ ставятся в соответствие элементы множества VUTU$ ($-маркер дна стека), столбцам-элементы множества T U "lambda"("lambda" - пустая строка). Перед началом работы алгоритма в рабочий стек заносятся символы $ и S. Возможные значения элементов таблицы и их интерпретация алгоритмом

разбора приведены в таблице 3.2.

Таблица 3.2

N п/п	Значение элемента таблицы	Интерпретация алгоритмом разбора
1	Номер i порождающего правила грамматики	Удаление символа из рабочего стека; запись этого символа в выходной стек; запись правой части правила номер i в рабочий стек справа налево, начиная с последнего символа
2	"сдвиг"	Удаление символа из рабочего стека; запись его в выходной стек; считывание следующего символа входной цепочки
3	допуск	конец работы
4	"ошибка"	Вывод сообщения об ошибке; конец работы

Для построения управляющей таблицы M по заданной LL(1)-грамматике G(V,T,P,S) можно воспользоваться следующим алгоритмом /1/.

1.Если <A>::=r-правило номер i заданной грамматики, то M(<A>,a)=i для всех a (кроме "lambda"), являющихся терминальными префиксами цепочек, выводимых из r. Если таким префиксом может быть "lambda", то M(<A>,b)=i для всех b, являющихся терминальными символами, которые могут встречаться непосредственно справа от <A>.

2.M(a,a)="сдвиг" для всех a, принадлежащих T.

3.M($,"lambda")="допуск".

4.Оставшиеся незаполненными элементы таблицы M получают значение "ошибка".

Пример.

Грамматика G(V,T,P,S) рассмотренного выше примера не обладает свойством LL(1),поскольку обе правые части для нетерминала <E> (правила 2 и 3) порождают цепочки, начинающиеся одним и тем же терминалом i. То же можно сказать и о правилах для терминала <T>.Преобразуем грамматику G(V,T,P,S) к грамматике G1(V1,T,P1,S), обладающей свойством LL(1).Правила последней примут вид :

1) <S>::=<E> 2) <E>::=<T><X> 3) <X>::=+<E> 4) <T>::=<F><Y>

5) <Y>::=*<T> 6) <F>::=i 7) <X>::="lambda" 8) <Y>::="lambda"

В соответствии с приведенным алгоритмом теперь можно построить управляющую таблицу для LL(1)-анализатора (таблица 3). Таблица 3

	i	+	*	$
<S>	1			
<E>	2			
<T>	4			

<F>	6			
<X>		3		7
<Y>		8	5	8
i	сдвиг			
+		сдвиг		
*			сдвиг	
$				сдвиг

Незаполненные элементы таблицы имеют значение "ошибка". Проанализируем входную цепочку i*i с помощью алгоритма LL(1)-анализатора. При этом, для облегчения формирования выходного стека, при записи символа в рабочий стек будем снабжать его указателем на символ, являющийся его предком в синтаксическом дереве разбора. Процесс анализа проиллюстрируем таблицей 4.

Таблица 3.4

Рабочий стек	Вх. символ	M(A,a)
<S>,0 $	i	M(<S>,i)=1
<E>,1 $	i	M(<E>,i)=2
<T>,2 <X>,2 $	i	M(<T>,i)=4
<F>,3 <Y>,3 <X>,2 $	i	M(<F>,i)=6
i,4 <Y>,3 <X>,2 $	i	M(i,i)="сдвиг"
<Y>,3 <X>,2 $	*	M(<Y>,*)=5

*,6 <T>,6 <X>,2 $	*	M(*,*)="сдвиг"
<T>,6 <X>,2 $	i	M(<T>,i)=4
<F>,8 <Y>,8 <X>,2 $	i	M(<F>,i)=6
i,9 <Y>,8 <X>,2 $	i	M(i,i)="сдвиг"
<Y>,8 <X>,2 $	"lambda"	M(<Y>,"lambda")=8
<X>,2 $	"lambda"	M(<X>,"lambda")=7
$	"lambda"	M($,"lambda"= "допуск"

В результате анализа получено синтаксическое дерево разбора (рис.3.4.).

0100090000038d00000002001c0000000000040000000301080005000000b020000000050000000c02a2070907040000002e0118001c000000fb02100007000000000bc02000000cc0102022253797374656d00774c9ab8006ae6f8768032037701000000f0c4ca090b000000040000002d010000040000002d010000040000002010100 1c000000fb02a4ff000000000000900100000cc0440002243616c6962726900000000000000000000000000000000000000000000000000000000040000002d010100040000002d010100040000002d010100050000000902000000020d000000320a5700000010004000000000008079e0720003600050000009020000000204000002d01000004000002d010000030000000000

Рис.3.4. Синтаксическое дерево разбора

В выходном стеке это дерево представляется таблицей 3.5.

Таблица 3.5

12	<X>,2
11	<Y>,8
10	i,9

9	<F>,8
8	<T>,6
7	*, 6
6	<Y>,3
5	i,4
4	<F>,3
3	<T>,2
2	<E>,1
1	<S>,0

3.2. Порядок выполнения работы

2.1.Проверить является ли заданная грамматика грамматикой LL(1), при необходимости выполнить соответствующие преобразования.
2.2.Построить управляющую таблицу разбора.
2.3.Разработать и отладить синтаксический анализатор, который должен быть оформлен в виде отдельной процедуры (подпрограммы).

3.3. Задание на лабораторную работу

Разработать синтаксический анализатор подмножества заданного языка программирования на основе метода LL(1). Варианты заданий приведены в таблице 3.6. Код задания на лабораторную работу состоит из трех компонент.
Первая цифра варианта определяет язык программирования (1- PL0, 2- ASPLE). Описание синтаксиса языков программирования PL0 и ASPLE приведены соответственно в приложениях 1 и 2.
Вторая цифра варианта определяет подмножество языка программирования. Описание подмножеств языков программирования PL0 и ASPLE приведены соответственно в приложениях 3 и 4.

Таблица 3.6

Номер варианта	1	2	3	4	5	6	7	8
Код задания	11П	12С	13П	14С	21П	22С	23П	24С
Номер варианта	9	10	11	12	13	14	15	16
Код задания	11С	12П	13С	14П	21С	22П	23С	24П

Третья цифра-язык программирования: П- Паскаль, С -Си.

3.4. Содержание отчета

3.4.1.Постановка задачи.
3.4.2.Управляющая таблица разбора.
3.4.3.Текст программы.
3.4.4.Описание тестовых примеров.
3.4.5.Распечатка результатов.

Библиографический список

1. А.Ахо, Дж.Ульман. Теория синтаксического анализа, перевода и компиляции. Т.1. Синтаксический анализ. - М.: Мир, 1978. - 612с.
2. Р.Хантер. Проектирование и конструирование компиляторов. - М.: Финансы и статистика, 1984. - 232с.
3. Д.Грис. Конструирование компиляторов для ЦВМ. - М.: Мир, 1975.- 544 с.

4. ЛАБОРАТОРНАЯ РАБОТА № 4. Восходящий синтаксический анализ методом предшествования

Цель работы: Изучение восходящего синтаксического анализа методом предшествования, получение навыков в программировании алгоритмов синтаксического разбора

4.1 Синтаксический анализ для грамматики простого предшествования

Алгоритм синтаксического анализа на основе грамматики предшествования относится к классу алгоритмов восходящего разбора.

Строкам и столбцам управляющей таблицы M (матрицы отношений предшествования) ставятся в соответствие элементы множества T U V U $ (в строке - символ в вершине рабочего стека, в столбце - входной символ).

Каждую запись рабочего стека представим парой: (символ, знак отношения предшествования символа предыдущей записи стека с данным символом). Перед началом работы алгоритма в стек записывается пара ($,0).

Возможные значения элементов таблицы и их интерпретация алгоритмом разбора приведены в таблице 4.1.

Таблица 4.1

Nп/п	Значение элемента таблицы	Интерпретация значения алгоритмом разбора
1	<.	Запись текущего входного символа в выходной стек и в паре со знаком "<." - в рабочий стек; если этот символ - нетерминал, установка указателей на него в ближайших к нему n записях выходного стека с пустыми указателями, где n - количество символов в правой части правила для этого нетерминала.
2	=.	Запись текущего входного символа в выходной стек и в паре со знаком "=." - в рабочий стек; если этот символ - нетерминал, установка указателей на него в ближайших к нему n записях выходного стека с пустыми указателями, где n - количество символов в правой части правила для этого нетерминала.
3	>.	Удаление из рабочего стека записей (символы которых образуют цепочку w) со знаками отношения =. до первого знака <. включительно; если <A>::=w - правило заданной грамматики, имитация считывания <A> в качестве следующего входного символа.
4	"допуск”	Конец работы
5	"ошибка”	Вывод сообщения об ошибке, конец работы.

Отношения предшествования Вирта-Вебера <., =., >. для КС- грамматики G(V,T,P,S) определяются на множестве V U T следующим образом /1/:

1) X<.Y, если в P есть такое правило <A>::=rX<B>q, что Y является головным символом хотя бы

одной из цепочек, выводимых из <B>;

2) X=.Y, если в P есть правило <A>::=rXYq;

3) X>.a, если a - терминал, и в P есть правило <A>::=r<B>Yq такое, что X является хвостовым символом хотя бы одной из цепочек, выводимых из <B>, а Y=at, или a является головным символом хотя бы одной из цепочек, выводимых из Y.

Для анализа входной цепочки методом предшествования удобно добавлять к ней левый и правый концевые маркеры ($). Будем считать, что $<.X для всех X, являющихся головными символами цепочек, выводимых из <S>, и Y>.$ для всех Y, являющихся хвостовыми символами таких цепочек.

КС-грамматика G(V,T,P,S) называется грамматикой предшествования, если она приведенная, не содержит "lambda"-правил, и для любой пары символов из T U V выполняется не более одного отношения предшествования Вирта-Вебера. Обратимая грамматика предшествования называется грамматикой простого предшествования.

Пример.

Анализируя грамматику G(V,T,P,S) из п.1.1, заметим, что она не удовлетворяет определению грамматики предшествования, поскольку <T> =. + в соответствии с правилом 2) и <T> >. + в соответствии с правилами 2) и 4). Чтобы избавиться от этого конфликта, преобразуем эту грамматику к грамматике G2(V2,T,P2,S) с правилами:

1) <S>::=<E>
2) <E>::=<X>+<E>
3) <E>::=<X>
4) <T>::=<F>*<T>
5) <T>::=<F>
6) <F>::=i
7) <X>::=<T>

Матрица отношений предшествования для грамматики G2 представлена таблицей 4.2.

Таблица 4.2

	<S>	<E>	<T>	<F>	<X>	i	+	*	$
<S>									>.
<E>									>.
<T>							>.		>.
<F>							>.	=.	>.
<X>							=.		
i							>.	>.	>.
+		=.	<.	<.	<.	<.			
*			=.	<.		<.			
$	<.	<.	<.	<.	<.	<.			

Незаполненные элементы матрицы соответствуют парам символов грамматики, для

которых не определены отношения предшествования. При использовании этой матрицы в качестве управляющей таблицы в алгоритме синтаксического анализа можно считать, что эти элементы содержат значение "ошибка", а M(<S>,$)="допуск".

Проанализируем входную цепочку i*i с использованием метода предшествования. Результат анализа представлен в таблице 4.3.

Таблица 4.3

Рабочий стек	Вх.символ	M(A,a)
$,0	i	M($,i)=<.
i,<. $,0	*	M(i,*)=>.
$,0	<F>	M(<F>,$)=<.
<F>,<. $,0	*	M(<F>,*)==.
*,=. <F>,<. $,0	i	M(7,i)=<.
i,<. *,=. <F>,<. $,0	$	M(i,$)=>.
*,=. <F>,<. $,0	<F>	M(*,<F>)=<.
<F>,<. *,=. <F>,<. $,0	$	M(<F>,$)=>.
*,=. <F>,<. $,0	<T>	M(*,<T>)==.
<T>,=. *,=. <F>,<. $,0	$	M(<T>,$)=>.
$,0	<T>	M($,<T>)=<.
<T>,<. $,0	$	M(<T>,$)=>.
$,0	<X>	M($,<X>)=<.

<X>,<. $,0	$	M(<X>,$)=>.
$,0	<E>	M($,<E>)=<.
<E>,<. $,0	$	M(<E>,$)=>.
$,0	<S>	M($,<S>)=<.
<S>,<. $,0	$	M(<S>,$)="допуск"

В результате анализа получено синтаксическое дерево разбора (рис.4.1).

0100090000038d00000002001c0000000000040000000301080005000000b0200000000050000000c02a2070907040000002e0118001c000000fb02100007000000000bc02000000cc0102022253797374656d00774c9ab8006ae6f8768032037701000000f0c4ca090b000000040000002d010000040000002d010000040000000201010001c000000fb02a4ff000000000000900100000cc0440002243616c6962726900000000000000000000000000000000000000000000000000000000000000040000002d010100040000002d010100040000002d0101000500000009020000000020d000000320a57000000010004000000000008079e0720003600050000000902000000020400000002d010000040000002d010000030000000000

Рис.4.1. Синтаксическое дерево разбора

В выходном стеке это дерево представляется таблицей 4.4.

Таблица 4.4

10	<S>,0
9	<E>,10
8	<X>,9
7	<T>,8
6	<T>,7
5	<F>,6
4	i,5
3	*,7
2	<F>,7
1	i,2

4.2. Порядок выполнения работы

2.1.Проверить является ли заданная грамматика грамматикой предшествования, при необходимости выполнить соответствующие преобразования.

2.2.Построить управляющую таблицу разбора.
2.3.Разработать и отладить синтаксический анализатор, который должен быть оформлен в виде отдельной процедуры (подпрограммы).

4.2.1. Задание на лабораторную работу

Разработать синтаксический анализатор подмножества заданного языка программирования на основе метода предшествования. Варианты заданий приведены в таблице 4.5. Код задания на лабораторную работу состоит из трех компонент.
Первая цифра варианта определяет язык программирования (1- PL0, 2- ASPLE). Описание синтаксиса языков программирования PL0 и ASPLE приведены соответственно в приложениях 1 и 2.
Вторая цифра варианта определяет подмножество языка программирования. Описание подмножеств языков программирования PL0 и ASPLE приведены соответственно в приложениях 3 и 4.

Таблица 4.5

Номер варианта	1	2	3	4	5	6	7	8
Код задания	11П	12С	13П	14С	21П	22С	23П	24С
Номер варианта	9	10	11	12	13	14	15	16
Код задания	11С	12П	13С	14П	21С	22П	23С	24П

Третья цифра-язык программирования: П- Паскаль, С -Си.

4.3 Содержание отчета

4.3.1.Постановка задачи.
4.3.2.Исходная и преобразованная грамматики.
4.3.3.Текст программы.
4.3.4.Описание тестовых примеров.
4.3.5.Распечатка результатов.

Библиографический список

1. А.Ахо, Дж.Ульман. Теория синтаксического анализа, перевода и компиляции. Т.1. Синтаксический анализ. - М.: Мир, 1978. - 612с.
2. Р.Хантер. Проектирование и конструирование компиляторов. - М.: Финансы и статистика, 1984. - 232с.
3. Д.Грис. Конструирование компиляторов для ЦВМ. - М.: Мир, 1975.- 544 с.
4. Волкова И.А., Руденко Т.В. Формальные грамматики и языки. Элементы теории трансляции. –МГУ,1998.

5. ЛАБОРАТОРНАЯ РАБОТА № 5. Восходящий синтаксический анализ методом LR(1)

Цель работы: Изучение восходящего синтаксического анализа методом **LR(1)** предшествования, получение навыков в программировании алгоритмов синтаксического разбора

5.1. Синтаксический анализ для LR(1)-грамматики

Алгоритм синтаксического анализа на основе LR(k)-грамматики относится к классу алгоритмов восходящего разбора.

Строкам управляющей таблицы M (LR-таблицы разбора) /2/ ставятся в соответствие состояния, в которых может находиться анализатор, столбцам - элементы множества VUTU$.

Каждая запись рабочего стека представляет собой пару: (символ, номер состояния). Перед началом работы алгоритма в рабочий стек заносится пара ($,1).

Возможные значения элементов таблицы и их интерпретация алгоритмом разбора приведены в таблице 5.1.

Таблица 5.1

N п/п	Значение элемента таблицы	Интерпретация алгоритмом разбора
1	Номер i порождающего правила грамматики	Удаление из рабочего стека n записей (n - количество символов в правой части правила номер i); имитация считывания в качестве следующего входного символа нетерминала левой части правила номер i.
2	("Сдвиг", номер j состояния)	Запись текущего входного символа в выходной стек и в паре с номером j - в рабочий стек; если этот символ нетерминал, установка указателей на него в ближайших к нему n записях выходного стека с пустыми указателями.
3	“допуск”	конец работы
4	"ошибка"	Вывод сообщения об ошибке; Конец работы.

Для построения управляющей таблицы M может быть выполнена разметка порождающих правил грамматики номерами состояний анализатора. Номера состояний устанавливаются в правой части каждого правила: перед первым символом, между любыми двумя символами и после последнего символа. При этом номер состояния, непосредственно справа от которого находится нетерминал, следует распространить на позиции перед первыми символами всех правых частей правил для данного нетерминала (и т. д. рекурсивно). А если непосредственно слева от одного и того же символа в каких-либо правилах установлены одинаковые множества меток, то и непосредственно справа от этого символа в этих правилах следует поставить одну и ту же метку. После разметки грамматики выполняется построение таблицы M по следующему алгоритму /2/.

1. Если символ A в правой части правила имеет непосредственно слева от себя метку k, а непосредственно справа от себя - метку j, то

M(k,A) = ("сдвиг",j).

2. Если метка j размещается за последним символом правой части правила номер i (если правая часть правила - "lambda", j совпадает с

меткой состояния, непосредственно справа от которого находится нетерминал левой части правила i), то определяется множество Q символов, которые в какой-либо сентенциальной форме могут следовать за нетерминалом левой части правила номер i, и M(j,q)=i для всех q, принадлежащих Q.

3. M(1,<S>)="допуск".

4. Оставшиеся незаполненными элементы таблицы M получают значение "ошибка".

Пример.

Разметим грамматику G(V,T,P,S):

1) <S>::= <E>
1 2

2) <E>::= <T> + <E>
1,4 3 4 5

3) <E>::= <T>
1,4 3

4) <T>::= <F> * <T>
1,4,7 6 7 8

5) <T>::= <F>
1,4,7 6

6) <F>::= i
1,4,7 9

Управляющая таблица M будет представлена таблицей 5.2.

Таблица 5.2

	<S>	<E>	<F>	<T>	i	+	*	$
1	“допуск	сдв,2	сдв,6					
2	”							1
3						сдв,4		3
4		сдв,6	сдв,3	сдв,9				
5	сдв,5							2
6						5	сдв,7	5
7		сдв,6	сдв,8	сдв,9				
8						4		4
9						6	6	6

Незаполненные элементы таблицы имеют значение "ошибка". Проанализируем входную цепочку i*i с помощью алгоритма LR(1) - анализатора. Последовательность изменения состояний стека представлена таблицей 5.3.

Таблица 5.3

Рабочий стек	Вх.символ	M(A,a)
$,1	i	M(1,i)=сдв,9
i,9 $,1	*	M(9,*)=6

$,1	<F>	M(1,<F>)=сдв,6
<F>,6 $,1	*	M(6,*)=сдв,7
*,7 <F>,6 $,1	i	M(7,i)=сдв,9
i,9 *,7 <F>,6 $,1	$	M(9,$)=6
*,7 <F>,6 $,1	<F>	M(7,<F>)=сдв,6
<F>,6 *,7 <F>,6 $,1	$	M(6,$)=5
*,7 <F>,6 $,1	<T>	M(7,<T>)=сдв,8
<T>,8 *,7 <F>,6 $,1	$	M(8,$)=4
$,1	<T>	M(1,<T>)=сдв,3
<T>,3 $,1	$	M(3,$)=3
$,1	<E>	M(1,<E>)=сдв,2
<E>,2 $,1	$	M(2,$)=1
$,1	<S>	M(1,<S>)=допуск

В результате анализа получено синтаксическое дерево разбора (рис.5.1).

0100090000038d00000002001c0000000000040000000301080005000000b0200000000050000000c02a2070907040000002e0118001c000000fb021000070000000000bc02000000cc0102022253797374656d00774c9ab8006ae6f8768032037701000000f0c4ca090b000000040000002d010000040000002d010000040000000020101001c000000fb02a4ff000000000000090010000000cc0440002243616c6962726900000000000000000000000000000000000000000000000000000000000040000002d010100040000002d010100040000002d010100050000000902000000020d000000320a57000000010004000000000808079e0720003600050000000902000000020400000002d010000040000002d0100000300000000000

Рис.5.1 Синтаксическое дерево разбора

В выходном стеке это дерево представляется таблицей 5.4.

Таблица 5.4

9	<S>,0
8	<E>,9
7	<T>,8
6	<T>,7
5	<F>,6
4	i,5
3	*,7
2	<F>,7
1	i,2

5.2. Задание на лабораторную работу

Разработать синтаксический анализатор подмножества заданного языка программирования на основе метода LR(1). Варианты заданий приведены в таблице 5.5. Код задания на лабораторную работу состоит из трех компонент.
Первая цифра варианта определяет язык программирования (1- PL0, 2- ASPLE). Описание синтаксиса языков программирования PL0 и ASPLE приведены соответственно в приложениях 1 и 2.
Вторая цифра варианта определяет подмножество языка программирования. Описание подмножеств языков программирования PL0 и ASPLE приведены соответственно в приложениях 3 и 4.

Таблица 5.5

Номер варианта	1	2	3	4	5	6	7	8
Код задания	11П	12С	13П	14С	21П	22С	23П	24С
Номер варианта	9	10	11	12	13	14	15	16
Код задания	11С	12П	13С	14П	21С	22П	23С	24П

Третья цифра-язык программирования: П- Паскаль, С -Си.

5.3.Содержание отчета

5.3.1.Постановка задачи.
5.3.2.Управляющая таблица разбора.
5.3.3.Текст программы.
5.3.4.Описание тестовых примеров.
5.3.5.Распечатка результатов.

Библиографический список

1. А.Ахо, Дж.Ульман. Теория синтаксического анализа, перевода и компиляции. Т.1. Синтаксический анализ. - М.: Мир, 1978. - 612с.
2. Р.Хантер. Проектирование и конструирование компиляторов. - М.: Финансы и статистика, 1984. - 232с.
3. Д.Грис. Конструирование компиляторов для ЦВМ. - М.: Мир, 1975.- 544 с.

6. ЛАБОРАТОРНАЯ РАБОТА № 6 ОПТИМИЗАЦИЯ ОБЪЕКТНОГО КОДА

Цель работы: изучение методов оптимизации объектного кода, получение навыков в программировании алгоритмов оптимизации.

6.1 Теоретическое введение

Под оптимизацией будем понимать ряд преобразований, которые можно выполнять над программой с целью повышения эффективности объектного кода. Такие преобразования можно выполнять в различных блоках компилятора, но наиболее сложные из них обычно выполняются отдельным блоком оптимизации, включаемым в работу после синтаксического анализа до генерации кода(1,2,3).

Рассмотрим подмножество оптимизирующих преобразований, которые предлагается реализовать в рамках данной лабораторной работы. Объектный код, над которым выполняются оптимизирующие преобразования, будем считать заданным в виде последовательности триад, состоящих из полей:
<метка><код операции><операнд 1><операнд 2>.
Поле <метка> предназначено для идентификации составных частей циклов и может содержать один из следующих четырёх идентификаторов:INIT(инициализация цикла), INCR(изменение параметра цикла), TEST(проверка условия выхода из цикла) , LOOP(тело цикла).

Удаление общих подвыражений.

Некоторая i-ая триада линейного списка считается лишней, если существует более ранняя идентичная ей j-ая триада, и никакая переменная, от которой зависит результат этой триады, не изменяется третей триадой, лежащей между i-ой и j-ой (1-3).

Для того чтобы следить за внутренней зависимостью переменных и триад, поставим им в соответствие поставим им числа зависимости по следующим правилам.
1.Вначале для всех переменных объектного кода числа зависимости dep(<переменная>) равны 0.
2.После обработки i-ой триады, в которой переменной A присваивается некоторое значение, dep(A) меняется на i.
3.При обработке i-ой триады ее число зависимости dep(Ti) равно 1+<максимальное из чисел зависимости ее операндов>.
Полученные числа зависимости используются следующим образом:
если i-ая триада идентична j-ой триаде (j<i), тогда i-ая триада считается лишней в том и только том случае, когда dep(i)=dep(j).

Пример.

Об'ектный код	dep	A	B	C	D	Ti
* C B		0	0	0	0	1
+ D T1		0	0	0	0	2
:= D T2		0	0	0	0	3
* C B		0	0	0	3	1
+ D T4		0	0	0	3	4
:= A T5		0	0	0	3	5
		6	0	0	3	

Триада T4 идентична триаде T1 и dep(T4)=dep(T1)=1. После преобразования получим объектный код:

	dep(Ti)
* C B	1
+ D T1	2

:= D T2 3
+ D T1 4
:= A T4 5

6.1.1. Удаление мертвых переменных

Переменная называется мертвой в программе, если ее значение после определения нигде в программе не используется. Из объектного кода следует удалить триаду, присваивающую значение мертвой переменной, и все триады, вычисляющие это значение. Во избежание удаления триад, на которые есть ссылки в других участках объектного кода, такое преобразование рекомендуется выполнять после удаления общих подвыражений /1-3/.

6.1.2. Распространение констант

Процедура распространения констант заключается в выполнении тех операций, значения операндов которых известны во время компиляции /1/.
Для выполнения такой процедуры организуется вспомогательная таблица Т, содержащая пары (А,К) для всех переменных А, для которых известно текущее значение К. Каждая свертываемая (вычисляемая) триада заменяется на С К, где С- новая "операция", для которой не нужно генерировать команды, а К- результат вычисления триады.

Алгоритм распространения констант последовательно просматривает триады линейного участка и для каждой триады делает следующее.
1.Если операнд-переменная, которая содержится в Т, то он заменяется на константу К.
2.Если операнд-ссылка на триаду типа С К _ , то он заменяется на константу К.
3.Если все операнды являются константами, то выполняется указанная в данной триаде операция, и триада заменяется на С К _ , где К-результат выполнения операции.
4.Если триада реализует присвоение вида А:=В, то:
- если В-константа, то А со значением В заносится в таблицу Т (старое значение А, если оно было, исключается);
-если В- не константа, то А своим значением исключается из Т, если она там была.
Пример:

Обьектный код			Результат операции				
			Обьектный код			Таблица Т	
+	1	1	C	2		A	2
:=	A	T1	:=	A	2	I	3
:=	I	3	:=	I	3	B	9.2
+	6.2	I	C	9.2			
:=	B	T4	:=	B	9.2		

Удаление лишних переменных

Переменная называется лишней, если после своего определения она используется в программе один раз(1-3).

Для выявления лишних переменных каждой переменной объектного кода ставится в соответствие троичный признак использования, который устанавливается в 0 при обнаружении триады, присваивающей значение переменной, в 1 и 2 при первом и втором появлении этой переменной в качестве операнда в триадах линейного участка программы соответственно. Выявление лишней переменной происходит в момент установки признака использования в 0: если его предыдущее значение было равно 1, то переменная является лишней, и ссылку на неё следует заменить присвоенным ей значением или ссылкой на триаду, вычисляющую её значение.

Пример.

Обьектный код	Признаки использования переменных		
	A	B	C
:= A B	0	x	X
+ A 1	1	x	X
:= C 0	1	x	0
+T2 C	1	x	1
+ C T4	1	x	2
:= A T5	0	x	2

Переменная A в приведённом линейном участке объектного кода является лишней. В триаде T2 её следует заменить присвоенным ей значением, а в триаду T1-удалить:

```
+ B  1
:= C  0
+ T1 C
+ C  T3
:= A  T4
```

Удаление индуктивных переменных

Переменная X называется индуктивной, если принимаемые ею последовательно значения x1,x2, … на участках цикла INIT и LOOP образует арифметическую прогрессию(1-3).

Индуктивные переменные i1 и i2 одного участка цикла связываются друг с другом линейными преобразованиями вида i1=i2+const. Поэтому внутри области цикла можно использовать только одну из индуктивных переменных, а все остальные, если их значения требуются за пределами цикла, вычислять один раз при выходе из этого цикла.

Пример 1.

```
 INIT  := I 1
:= L J
        := K 0
LOOP   + L I
        := S T4
        + K S
        := K T6
INCR   + I 1
        := I T8
TEST   = I 1000
```

JFALSE T4

В приведённом объектном коде индуктивными переменными являются I (принимает значения 1,2, …..1000) и S(принимает значения J+1,J+2,….,J+1000).

Задача нахождения всех индуктивных переменных в области является нетривиальной, и алгоритма её решения не существует. В рамках лабораторной работы предлагается составить программу преобразования заданного участка объектного кода при заданном списке индуктивных переменных, вычисляемых этим участком.

Одна из переменных этого списка может быть выбрана в качестве базовой. Тогда для всех остальных переменных списка следует определить значения констант, связывающих эти переменные с базовой, собрав эти значения в таблицу. При построении такой таблицы формирование записи для одной переменной потребует выполнения, возможно, нескольких шагов, поскольку переменная может быть связана с базовой не явно, а через другие индуктивные переменные.

Выберем, например, в качестве базовой такую переменную из списка индуктивных, которой присваивается значение на участке INIT раньше других. Таблицу индуктивных переменных построим за несколько шагов. На каждом шаге для любой индуктивной переменной будем определять имя текущей переменной (или триады), относительно которой определено значение данной индуктивной переменной, и вычислять константу, связывающую индуктивную переменную с текущей.Изменение записи таблицы, относящейся к некоторой индуктивной переменной, будем выполнять до тех пор, пока имя переменной, относительно которой определена данная индуктивная переменная, не совпадет с именем базовой переменной.

Пример 2.

```
  Об'ектный код
 INIT := I  1
 LOOP  + I  1          Список индуктивных переменных:
       + T2 2          I,K,S.
      := K  T3
       - K  3
       + T5 4
      := S  T6
```

Выберем I в качестве базовой переменной. Построим таблицу связей индуктивных переменных.

	Шаг 1	Шаг 2	Шаг 3	Шаг 4
I	1,0	1,0	1,0	1,0
K	T3,0	T2,2	I,3	I,3
S	T6,0	T5,4	K,1	I,4

Для объектного кода из примера 1,если выбрать базовой переменную I, значение S=I+L определяется за два шага.

Теперь цикл может быть преобразован вынесением за его пределы вычислений всех индуктивных переменных, кроме базовой:

```
 INIT := I 1
       := L J
        := K 0
 LOOP  + L I
         + K T4
         := K T5
  INCR  + I 1
         := I T7
 TEST  = I 1000
JFALSE T4
       + L I
```

```
        := S T11
```

Отметим, что в рассмотренном примере было бы выгоднее в качестве базовой выбрать переменную S, поскольку она используется для вычислений внутри участка LOOP, и, следовательно, триаду + L I из этого участка удалить нельзя, в то время как переменная I используется только для управления циклом. Если управление циклом возложить на переменную S, то от I можно совсем избавиться:

```
  INIT := L J
        := K 0
         + L 1
        := S T3
         + S 1000
        := R T5
 LOOP  + K S
        := K T7
 INCR  + S 1
        := S T9
  TEST  = S R
JFALSE T7
```

6.1.3. Вынесение инвариантных вычислений за пределы области цикла

Операция называется инвариантной в цикле, если ни один из ее операндов не изменяется во время работы этого цикла /1-3/.

Упростим выполнение преобразования этого типа в рамках лабораторной работы тем, что будем считать известным список переменных, инвариантных в оптимизируемом цикле. Тогда для выявления инвариантных операций (триад) можно воспользоваться алгоритмом распространения констант на LOOP-участке цикла. Поскольку вычисление значения инвариантной триады не всегда представляется возможным, вместо замены ее триадой типа C K _ будем помечать ее символом "C". Таблица T значений переменных, используемая алгоритмом распространения констант, очевидно, будет в этом случае представлять собой просто список обнаруженных инвариантных переменных.

Пример.

```
  Об'ектный код
  INIT := I 1      Заданный список инвариантных
        := S 0     переменных: A,B.
 LOOP  + A B
        := K T3
         * K I
         + S T5
        := S T6
   INCR  . . .
```

В процессе преобразований список инвариантных переменных пополнится переменной K, а триады T3 и T4 будут отмечены символом "C".После их вынесения за LOOP-участок получим:

```
  INIT := I  1
        := S  0
         + A  B
 LOOP  * T3 I
       + S  T4
       := S  T5
  INCR   . . .
```

6.2. Задание на лабораторную работу

Разработать процедуру оптимизации объектного кода, представленного в виде последовательности триад. Способ оптимизации и язык программирования выбрать из таблицы 1 в соответствии с заданным вариантом.

Таблица 6.1.

Номер варианта	1	2	3	4	5	6	7	8	9	10	11	12
Код задания	1п	2п	3п	4п	5п	6п	1с	2с	3с	4с	5с	6с

_____

Код задания состоит из двух компонент. Первая определяет вид оптимизирующего преобразования:

1-удаление общих подвыражений;
2-удаление мертвых переменных;
3-распространение констант;
4-удаление лишних переменных;
5-удаление индуктивных переменных;
6-вынесение инвариантных вычислений за пределы цикла;

вторая-язык программирования: П-паскаль, С-си.

6.3. Содержание отчета

6.3.1.Постановка задачи.
6.3.2.Спецификация процедуры.
6.3.3.Текст программы.
6.3.4.Описание тестовых примеров.
6.3.5.Распечатка результатов.

7. ЛАБОРАТОРНАЯ РАБОТА № 7 ТРАНСЛЯЦИЯ ВЫРАЖЕНИЙ

Цель работы: изучение методов трансляции выражений в польскую инверсную запись (ПОЛИЗ).

7.1 Теоретические сведения

В настоящее время широко используется в трансляторах метод трансляции выражений, основанный на использовании ПОЛИЗ, названный так в честь польского математика Я. Лукашевича.

Эта запись обладает следующими свойствами:

1. Операнды в ПОЛИЗ располагаются в том порядке, что и в обычной (инфиксной) записи.
2. Знаки операций встречаются в том порядке, в котором нужно выполнить соответствующие действия.
3. Знак каждой операции записывается после соответствующих операндов.

Например, выражение (1) в ПОЛИЗ представляется выражением (2):

*a + Ь * c – d / (a + Ь)* *(1)*
*a Ь c * + d a Ь + / -* *(2)*

7.1.1. Перевод простых арифметических и логических выражений в ПОЛИЗ.

Известно несколько методов получения ПОЛИЗ. Обзор различных методов приведен в [1]. Один из наиболее аффективных методов предложен в 1960 г. голландским ученым Е.В.Дейкстрой [2]. Этот метод основан на использовании стека с приоритетами, позволяющего изменить порядок следования знаков операций в выражении так, что получается ПОЛИЗ. Простейший вариант этого метода применим только к простым арифметическим и логическим выражениям, содержащим простые переменные, знаки арифметических и логических операций, знаки операций отношения и круглые скобки.

Каждому ограничителю, входящему в выражение, присваивается приоритет. Для знаков операции приоритеты возрастают в порядке, обратном старшинству операций. Скобки имеют низший приоритет.

Арифметическое или логическое выражение рассматривается как входная строка символов. Входная строка рассматривается слева направо. Операнды переписываются в выходную строку, а знаки операций помещаются вначале в стек операций.

Если приоритет входного знака равен нулю или больше приоритета знака, находящегося в вершине стека, то новый знак добавляется к вершине стека. В противном случае из стека "выталкивается" и переписывается в выходную строку знак, находящийся в вершине, а также следующие за ним знаки с приоритетами большими или равными приоритету входного знака. После этого входной знак добавляется к вершине стека.

Особенности имеет лишь обработка скобок. Открывающаяся круглая скобка, имеющая "особый" приоритет нуль, просто записывается в вершину стека и не выталкивает ни одного знака. В то же время ее не может вытолкнуть ни один знак, кроме закрывающей скобки.

Закрывающая скобка имеет приоритет 1, не превосходящий приоритет любой операции. Поэтому появление закрывающей скобки вызывает выталкивание всех знаков до ближайшей открывающей скобки включительно. В стек закрывающая скобка не записывается. Открывающая и закрывающая скобки как бы взаимно уничтожаются и в выходную строку не переносятся.

После просмотра всех символов входной строки происходит выталкивание всех оставшихся в стеке символов и дописывание их к выходной строке.

Пример 1. Перевести в обратную польскую запись выражение

a + Ь * c – d / (a + Ь)

Решение показано в табл.7.1.

Таблица 7.1

ПЕРЕВОД В ОБРАТНУЮ ПОЛЬСКУЮ ЗАПИСЬ АРИФМЕТИЧЕСКОГО ВЫРАЖЕНИЯ

Выходная строка	***a***	***a***	***a***	***a***	***a***	***a***	***a***	***a***	***a***	***a***	***a***	***a***	
	a	***a***											
			b	***b***	***b***	***b***	***b***	***b***	***b***	***b***	***b***	***b***	
	b	***b***											
					c	***c***	***c***	***c***	***c***	***c***	***c***	***c***	
	c	***c***											
						*******	*******	*******	*******	*******	*******	*******	
	*******	*******											
						+	+	+	+	+	+	+	
	+	+											
							a	***d***	***d***	***d***	***d***	***d***	
	d	***d***											
										a	***a***	***a***	
	a	***a***											
												b	
	b	***b***											
	+	+											
	-												
Стек	+												+
	(										***(***	***(***	***(***
						*******	*******			***/***	***/***	***/***	***/***
	/	***/***											
				+	+	+	+	-	-	-	-	-	-
	-	-											
	a	**+**	**b**	*****	**c**	**-**	**d**	***/***	**(**	**a**	**+**	**b**	**)**
	Входная строка												

Пример 2. Перевести в обратную польскую запись простое логическое выражение

a + Ь > - 5 ^ z - d = 1 + q ^ 2 (3)

Решение приведено в табл.2.

7.1.2. Переменные с индексами

Пусть требуется вычислить выражение

(a + Ь [i + 1 , j]) * c + d

Для выполнения вычислений на машине необходимо сначала найти рес переменной с индексами. Адрес этой переменной дает функция упорядочения [2]. Коэффициенты функции упорядочения хранятся в группе ячеек памяти, на начало которой указывает адрес, назначенный

идентификатору массива в результате обработки описаний и распределения памяти. Аргументами функции упорядочения являются значения индексов (индексных выражений) элемента массива.

Таблица 7.2

ПЕРЕВОД В ОБРАТНУЮ ПОЛЬСКУЮ ЗАПИСЬ ЛОГИЧЕСКОГО ВЫРАЖЕНИЯ

Выходная строка	**a a a b b b b b b b b b b** **b b b + + + + + + + + + +** **+ + + 5 5 5 5 5 5 5 5** **5 5 5 из из из из из из из из** **из из из > > > > > > > >** **> > > z z z z z z z** **z z z d d d d d** **d d d - - - -** **- - - 1 1 1** **1 1 1 q** **q q q 2 2** **+** **=** **^**
Стек	**+** **+ + + из из - - = = =** **= = = + + > > > ^ ^ ^ ^ ^ ^ ^ ^** **^ ^**
	a + b > - 5 ^ z - d = 1 + **q ^ 2**
	Входная строка

Введем операцию АДРЕС ЭЛЕМЕНТА МАССИВА (АЭМ), результат выполнения которой - адрес элемента массива, а операнды - идентификатор массива (точнее, назначенный ему адрес) и значения индексных выражений. Тогда рассматриваемое выражение [3] можно представить деревом, показанным на рис.7.1.

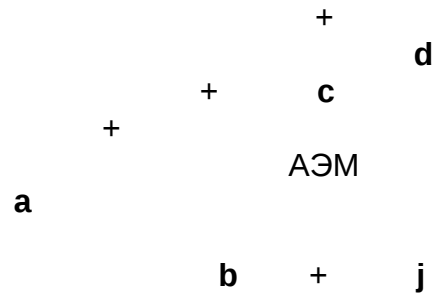


Рисунок7.1 - Дерево выражения, содержащего переменную с индексами

Обратная польская запись, полученная обходом дерева, имеет вид

a Ь i + j A Э M + c * d + **(4)**

Как обычно, в обратной польской записи левее операции АЭМ расположены операнды. Однако в отличие от логических и арифметических операций количество операндов операции АЭМ переменно. Оно зависит от размерности массива. Это вынуждает вместе со знаком операции АЭМ явным образом задать количество операндов.

Будем обозначать операцию АЗМ парой символов:

K] ,

где K - целое число, равное количеству операндов, а] - символ закрывающей индексной скобки, используемый в качестве знака операции АЭМ.

Очевидно, если n - число индексов, то

K = n + 1

Используя новое обозначение операции АЭМ, обратную польскую запись (4) можно переписать в виде

a Ь i 1 + j 3] + c * d + **(5)**

Для перевода выражений, содержащих переменные с индексами, также применим стек с приоритетами. Индексные скобки [и] в некотором смысле играют ту же роль, что и круглые скобки. Действие этих скобок на стек, как и действие круглых скобок, состоит прежде всего в следующем. Открывающая индексная скобка всегда записывается в вершину стека, а закрывающая индексная скобка выталкивает из стека все знаки до ближайшей открывающей индексной скобки включительно.

Однако в отличие от круглых скобок одновременно с записью в стек открывающей индексной скобки туда записывается также начальное (минимальное) значение счетчика операндов операции АЭМ, равное 2. Кроме того, после выталкивания из стека знаков закрывающей индексной скобкой и переписи их в выходную строку туда же переписывается значение счетчика операндов, а затем и сама закрывающая скобка.

Запятая, разделяющая индексные выражения, играет одновременно роль закрывающей скобки для предыдущего индексного выражения и роль открывающей скобки - для последующего. Поэтому запятой можно назначить приоритет 1 как закрывающей скобке, дополнив алгоритм условием, что запятая выталкивает из стека все знаки операции до ближайшей открывающей индексной скобки исключительно. Сама запятая, как и любая закрывающая скобка, в стек не записывается. Появление запятой равносильно появлению еще одного индекса, поэтому каждая запятая добавляет в счетчик операндов операции АЭМ единицу.

Пример. Перевести в обратную форму польскую инверсную запись выражение (3).

Решение приведено в табл.7.3.

табл.3 знак [сопровождается значением счетчика операндов операции АЭМ. Напомним, что минимальное значение этого счетчика равно 2. Нетрудно видеть, что окончательно выходная строка в табл.7.3 совпадает с записью в (5), полученной обходом дерева на рис.7.1.

Таблица 7.3

Перевод в обратную польскую запись выражения, Содержащего переменную с индексами

Вых. строка	a a b i + j 3] + c * d +	a b i + j 3] + c *	a	a b	a b	a b i	a b i +	a b i + j	a b i + j 3	a b i + j 3] +	a b i + j 3] + c	a b i + j 3] + c *
Стек	(*	(+	(	(	[2 (	[2 (	+ [2 (	+ [2 (	[3 (	[3 (	+ (	*

	(a + b [i + 1 , j]) * c d
	Входная строка

7.1.3. Указатели функций

Помимо простых переменных и переменных с индексами выражение может содержать также указатели функций. Рассмотрим наиболее простой случай, когда параметры функции вызываются по значению. В частности, этот случай имеет место для стандартных функций ПАСКАЛЯ. Выражение на ПАСКАЛЕ:

y – f (x , y , + 1 , z) **(6)**

содержит указатель функции с идентификатором f. Внешне указатель функции отличается от переменной с индексами лишь тем, что после идентификатора функции записана строка, заключенная в круглые скобки, а не в квадратные, как у элемента массива. Поэтому дерево для указателя функции и алгоритмы перевода указателя функции в обратную польскую запись практически те же, что для переменных с индексом.

Введем операцию ФУНКЦИЯ, операнды которой - идентификатор функции и значения (или идентификаторы) ее аргументов, а результат - значение функции (точнее, адрес значения функции). Тогда выражение (6) можно представить в виде дерева, изображенного на рис.2. Обход этого дерева дает обратную польскую запись выражения (6).

Очевидно, как и в случае переменных с индексами, в обратной польской записи целесообразно вместе со знаком операции ФУНКЦИЯ указывать количество операндов. Это облегчает последующую трансляцию указателя функции в машинные команды и позволяет контролировать правильность обращения к функции (соответствие числа фактических и формальных параметров).

Будем обозначать операцию ФУНКЦИЯ парой символов

КФ,

где К - количество операндов, а Ф - знак операции ФУНКЦИЯ. Тогда обратная польская запись выражения (6) примет следующий вид

y f x y 1 + z 4 Ф -.

Алгоритм перевода в обратную польскую запись функции, имеющий не менее одного параметра, тот же, что для переменной с индексом. Различие состоит лишь в том, что в момент прихода закрывающей круглой скобки в выходную строку записывается символ Ф. Чтобы отличить открывающую круглую скобку в начале списка фактических параметров от открывающей круглой скобки в начале выражения, можно использовать переменную состояния F (признак функции). Эта переменная обычно имеет значение ноль. В момент появления идентификатора функции она принимает значение 1, а после занесения в стек круглой скобки и начального значения счетчика операндов, равно 2 (см 1.2), вновь принимает значение 0. Закрывающая скобка, встретив в стеке открывающую круглую, записанную вместе со значением счетчика операндов, занесет это значение в выходную строку, запишет туда знак Ф и уничтожит в стеке круглую скобку и значение счетчика операндов.

Пример. Перевести в обратную польскую запись выражение (6).

Решение приведено в табл.7.4.

Таблица 7.4

Перевод в обратную польскую запись выражения, Содержащего указатель функции

Вы	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**
х	**y**	**y**									
од			**f**	**f**	**f**	**f**	**f**	**f**	**f**	**f**	**f**
с	**f**	**f**									
					x	**x**	**x**	**x**	**x**	**x**	**x**

нтрая строка	**x x y y 1 1 + + z z ф ф -** **y y y y y 1 1 1 + + z 4 4**
F	**0 0 1 0 0 0 0 0 0 0 0 0 0 0**
Стек	**- - (2 - (2 - (3 - (3 - (3 + - (3 + - (4 - (4 -**
	y - f (x , y + 1 , z) -
	Входная строка

Как видно из таблицы, окончательная выходная строка совпадает с обратной польской записью, полученной обходом дерева, показанного на рис.7.2.

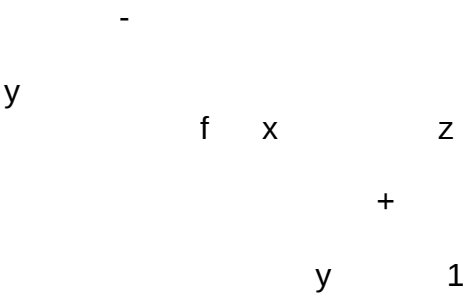


Рис.7.2. Дерево выражения, содержащего указатель функции

Особый случай, когда указатель функций употребляется без параметров, при переводе в обратную польскую запись можно не выделять, поскольку в этом случае указатель функций, как и все идентификаторы, будет перенесен непосредственно в выходную строку. Одновременно переменная состояния F примет значение 1. Если следующий символ не круглая скобка, то признаку присваивается значение 0, а в выходную строку заносится запись 1Ф в знак того, что операция ФУНКЦИЯ имеет всего один операнд - идентификатор функции

7.2. Задание на лабораторную работу

Разработать программу для трансляции выражений в ПОЛИЗ и исследовать ее пространственные и временные характеристики в соответствии с табл. 7.5.

Таблица 7.5

№ вар.	*Исходное выражение*	*Тип выражения*	*Система программирования*			
			Турбо Си	*Турбо Паскаль*	*C++ Builder*	*Delphi*
1	СИ	Простое АВ	+			
2	Паскаль	Простое АВ	+			
3	СИ	Простое АВ		+		
4	Паскаль	Простое АВ		+		
5	СИ	Простое АВ			+	
6	Паскаль	Простое АВ			+	
7	СИ	Простое АВ				+
8	Паскаль	Простое АВ				+
9	СИ	Логическое	+			
10	СИ	выражение		+		
11	СИ	Логическое			+	
12	СИ	выражение				+
13	Паскаль	Логическое	+			
14	Паскаль	выражение		+		
15	Паскаль	Логическое			+	
16	Паскаль	выражение				+
17	СИ	Логическое	+			
18	СИ	выражение		+		
19	СИ	Логическое			+	
20	СИ	выражение				+
21	Паскаль	Логическое	+			
22	Паскаль	выражение		+		
23	Паскаль	Логическое			+	
24	Паскаль	выражение				+
25	СИ	Указатель функции	+			
26	СИ	Указатель функции		+		
27	СИ	Указатель функции			+	
28	СИ	Указатель функции				+
29	Паскаль	Указатель функции	+			
30	Паскаль	Указатель функции		+		
31	Паскаль	Указатель функции			+	
32	Паскаль	Указатель функции				+
		Перемен. с индексами				
		Перемен. с индексами				
		Перемен. с индексами				
		Перемен. с индексами				
		Перемен. с индексами				
		Перемен. с индексами				
		Перемен. с индексами				
		Перемен. с индексами				

7.3. Содержание отчета

7.3.1. Постановка задачи

7.3.2. Описание программы
7.3.3. Текст программы
7.3.4. Описание тестовых примеров
7.3.5. Распечатка результатов

Библиографический список

1. Ренделл Б., Рассел Л. Реализация Алгола-60, - М.: Мир, 1967.
2. Лебедев В.Н. Введение в системы программирования, -М.: Статистика, 1975.

Приложение 1

Грамматика языка PL0

<программа>::=<блок>.
<блок>::=<декларации><список операторов>
<декларации>::=<декларация констант><декларация переменных><декларация процедур>
<декларация констант>::=CONST<список определений констант>;
<декларация переменных>::=VAR<список переменных>;|$
<декларация процедур>::=<список описаний процедур>
<список описаний процедур>::=<список описаний процедур><описание процедуры>;|$
<список определений констант>::=<список определений констант>,
<определение константы>|<определение константы>
<определение константы>::=<имя константы>=<число>
<имя константы>::=<переменная>
<список переменных>::=<список переменных>,<переменная>|<переменная>
<переменная>::=<идентификатор>
<описание процедуры>::=PROCEDURE<имя процедуры>;<блок>

<оператор>::=<переменная>:=<выражение>|
CALL<имя процедуры>|
BEGIN<список операторов>END|
IF<условие>THEN<оператор>|
WHILE<условие>DO<оператор>|$
<имя процедуры>::=<идентификатор>
<список операторов>::=<список операторов>;<оператор>|<оператор>
<условие>::=<выражение>|<выражение>==<выражение>
<выражение><><выражение>
<выражение><<<выражение>
<выражение>>><выражение>
<выражение>>=<выражение>
<выражение><=<выражение>
<выражение>::=<терм>|+<терм>|-<терм>
<терм>::=<слагаемое>|<терм>+<слагаемое>|<терм>-<слагаемое>
<слагаемое>::=<множитель>|<слагаемое>*<множитель>|<слагаемое>/<множитель>
<множитель>::=<переменная>|<число>|(<выражение>)
<идентификатор>::=<буква>|<идентификатор><буква>|<идентификатор><цифра>
<число>::=<натуральное число>|+<натуральное число>|-<натуральное число>
<натуральное число>::=<цифра>|<натуральное число><цифра>
<буква>::=A|B|C|D|E|F|G|N|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<цифра>::=0|1|2|3|4|5|6|7|8|9

Приложение 2

Грамматика языка ASPLE

<программа>::=BEGIN<последовательность описаний>;<последовательность операторов>END
<последовательность описаний>::=<описание>|<описание>;<последовательность описаний>
<последовательность операторов>::=<оператор>|<оператор>;<последовательность операторов>
<описание>::=<вид><список идентификаторов>

<вид>::=BOOL|INT|REF<вид>
<список идентификаторов>::=<идентификатор>|<идентификатор>,<список идентификаторов>
<оператор>::=<оператор присваивания>|<условный оператор>|<оператор цикла>|<оператор обмена>
<оператор присваивания>::= <идентификатор>:=<выражение>
<условный оператор>::=IF<выражение>THEN<последовательность операторов>
ELSE <посл. операторов>FI
<оператор цикла>::=WHILE<выражение>DO<последовательность операторов>END
<оператор обмена>::=INPUT<идентификатор>|OUTPUT<выражение>
<выражение>::=<фактор>|<выражение>+<фактор>|<выражение>-<фактор>
<фактор>::=<первичное>|<фактор>*<первичное>|<фактор>DIV<первичное>|
<фактор>MOD<первичное>
<первичное>::=<идентификатор>|<константа>|(<выражение>)|(<сравнение>)
<сравнение>::=<выражение>==<выражение>|<выражение><><выражение>
<константа>::=<логическая константа>|<целая константа>
<логическая константа>::=TRUE|FALSE
<целая константа>::=<число>|(<число>|-<число>
<число>::=<цифра>|<число><цифра>
<цифра>::=0|1|2|3|4|5|6|7|8|9
<идентификатор>::=<буква>|<идентификатор><буква>
<буква>::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

Приложение 3

Варианты подмножеств языка PL0

N вар.	**Начальный символ**	**Типы лексем**
1	<декларации>	<переменная> <число> <описание процедуры> CONST VAR = , ;
2	<оператор>	<идентификатор> <выражение> <условие> CALL BEGIN END IF THEN ;
3	<выражение>	<переменная> <число> + - * / ( )
4	<условие>	<переменная> <число> <слагаемое> == >> < >= <=

Приложение 4

Варианты подмножеств языка ASPLE

N вар.	Начальный символ	Типы лексем
1	<последовательность описаний>	<идентификатор>
		BOOL
		INT
		REF
		,
		;
2	<условный оператор>	<выражение>
		<оператор> (не условный)
		IF
		THEN
		ELSE
		FI
		;
3	<оператор цикла>	<выражение>
		<оператор> (не цикла)
		WHILE
		DO
		END
		;
4	<выражение>	<идентификатор>
		<число>
		<логическая константа>
		+
		-
		*
		DIV
		MOD