МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
высшего профессионального образования
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ имени академика С.П.КОРОЛЕВА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)»

Е. В. Мясников

Основы трансляции языков программирования
Лабораторный практикум

Электронное учебное пособие

Самара
2011

Автор: МЯСНИКОВ Евгений Валерьевич

Пособие представляет собой сборник методических указаний к лабораторным работам. В сборник включены краткие теоретическими сведения, необходимые для выполнения лабораторных работ, задания на лабораторные работы, примеры выполнения таких заданий и необходимая справочная информация. Задания на лабораторные работы способствуют усвоению теоретического материала и выработке практических навыков по дисциплине. В процессе выполнения лабораторных работ студенты осваивают три важнейших этапа трансляции: лексический, синтаксический и семантический анализ. В завершение курса лабораторных работ студентам предлагается реализовать интерпретатор.

Пособие предназначено для студентов факультета информатики, направление 010400 – Прикладная математика и информатика, бакалавриат (010400.62)/магистратура (010400.68, магистерская программа – Технологии параллельного программирования и суперкомпьютинг).

Оглавление

1. Лабораторная работа №1

Тема лабораторной работы: Построение лексического анализатора.

1.1 Теоретические основы лабораторной работы

Лексическим анализом будем называть процесс перевода исходного текста программы в последовательность лексем. За выполнение лексического анализа отвечает лексический анализатор (сканер).

Лексемами будем считать минимально значимые с точки зрения синтаксиса языка единицы. В зависимости от языка, можно выделить различные типы лексем. Тем не менее, для языков высокого уровня можно определить следующие основные элементы:

- ключевые слова (begin, while, for),
- идентификаторы (i, j, myfunction),
- константы (123, 1E-5, “name”),
- знаки операций (:=, ++, <<),
- разделители (, ; .)

В процессе лексического анализа комментарии и незначащие символы никак не учитываются, и на выходе лексического анализатора формируется последовательность лексем, которая может быть подана на вход синтаксического анализатора.

В самом анализаторе лексема описывается типом лексемы и некоторой информацией, которая может зависеть от типа лексемы. Для хранения информации по лексемам таких типов, как идентификаторы и константы, организуются отдельные таблицы. Данные этих таблиц будут использоваться на следующих этапах трансляции.

Сам алгоритм лексического анализа в большинстве случаев может быть построен как анализатор автоматного языка. В простых случаях границы лексем определяются по наличию незначащих символов или разделителей, после чего распознается сама лексема. Другим способом может быть определение типа лексемы по первому значащему символу, непосредственно после чего, может быть начато распознавание самой лексемы, которое продолжается, пока не встретится символ, противоречащий правилам построения лексем данного типа. В более сложных случаях лексический анализатор может представлять собой анализатор с возвратами.

При распознавании лексем используется так называемая таблица терминальных символов, к которым относят ключевые слова, знаки операций и другие символы. В этой таблице содержатся такие сведения, как код лексемы, символьное представление лексемы,

признак того, является ли данная лексема разделителем (например, знаки операций, как правило, являются разделителями) и т.д.

После распознавания типа лексемы, для идентификаторов и констант проверяется, есть ли уже такой идентификатор или константа в таблице идентификаторов или констант. Если такого идентификатора или константы еще нет, то в соответствующую таблицу помещается необходимая информация. После этого производится формирование выходной лексемы с сопутствующей информацией (тип лексемы, индекс в таблице идентификаторов или констант, положение лексемы в исходном тексе программы и т.д.).

Как было сказано выше, построение лексического анализатора производится , как построение анализатора автоматного языка. Построение такого анализатора производится по следующей схеме:

1) составление автоматной грамматики, описывающей цепочки входного языка (или эквивалентного конечного автомата),
2) в том случае, если грамматика является недетерминированной, приведение ее к детерминированной форме (или построение эквивалентного детерминированного конечного автомата),
3) приведение грамматики ко вполне детерминированной форме,
4) написание программы

Рассмотрим перечисленные этапы более подробно, но вначале дадим необходимые определения.

Грамматикой называется четверка

(S,V_N,V_T,R)

где S – начальный символ грамматики,

V_N - множество нетерминальных символов грамматики,

V_T – множество терминальных символов грамматики, определяемое алфавитом языка

R – множество правил грамматики.

Каждое правило грамматики в общем случае записывается в виде $\alpha\rightarrow\beta$, где α, β – произвольные цепочки, составленные из терминальных и нетерминальных символов грамматики. Нас, тем не менее, здесь будет интересовать очень узкий класс грамматик, называемых автоматными грамматиками.

Автоматной называется грамматика, все правила которой имеют вид:

$A\rightarrow aB$, $A\rightarrow a$,

где A, B – нетерминальные символы, a – терминальный символ. Правила вида $A\rightarrow a$ называют заключительными (терминальными) правилами.

Так, например, автоматная грамматика для описания целочисленных констант имеет вид:

S→0S|1S|…|9S|0|1|…|9

Автоматная грамматика для описания идентификаторов:

S→aA|bA|…|zA

A→0A|1A|…|9A|0|1|…|9|aA|bA|…|zA|a|b|…|z

Альтернативой формой задания языка по отношению к автоматной грамматике является описание языка с использованием конечного автомата. Конечный автомат представляет собой пятерку

$(\Sigma, Q, q_0, \delta, F)$,

где Σ - конечное множество допустимых входных символов (алфавит),

Q – множество состояний автомата,

q_0 – множество начальных состояний автомата,

δ - функция переходов,

F - множество допускающих состояний автомата.

По автоматной грамматике конечный автомат может быть построен элементарно. Для этого можно руководствоваться следующими правилами:

1) производят замену терминальных правил A→a на правила вида A→aF, где F – новый нетерминальный символ;

2) множеству терминальных символов грамматики V_T ставят в соответствие алфавит конечного автомата Σ, множеству нетерминальных символов грамматики V_N - множество состояний автомата Q, начальному символу грамматики S - начальное состояние автомата q_0, нетерминальному символу F - допускающее состояние автомата F;

3) Функцию перехода δ определяют следующим образом: δ(A,a) := {B|A→aB}.

Для приведенного выше примера описания идентификаторов функция перехода, представленная в виде таблицы переходов, имеет вид (начальное состояние помечено стрелкой сверху, допускающее состояние – единицей снизу):

↓

	S	A	F
a,b…z	A	A, F	
0,1…9		A, F	

1

В контексте построения программы анализа большое значение имеет то, в какой форме представлена автоматная грамматика. Дело в том, что программа может быть непосредственно написана лишь по грамматике, представленной во вполне

детерминированной форме, в то время как исходная грамматика часто представлена в недетерминированной форме. Дадим определения формам грамматики.

Говорят, что автоматная грамматика представлена в детерминированной форме, если для любого заданного нетерминального символа A и любого заданного терминального символа a существует не более одного правила $A \to aX$ (X – нетерминальный символ). В противном случае автоматная грамматика представлена в недетерминированной форме.

Говорят, что автоматная грамматика представлена во вполне детерминированной форме, если для любого заданного нетерминального символа A и любого заданного терминального символа a существует единственное правило $A \to aX$ (X – нетерминальный символ).

Аналогичные определения можно дать и для конечных автоматов. Конечный автомат является детерминированным (ДКА), если множество начальных q_0 состояний состоит из единственного начального состояния, а функция переходов $\delta(q,a)$ возвращает не более одного состояния для любых q, a. В противном случае мы имеем дело с недетерминированным конечным автоматом (НКА).

Тот факт, что исходная автоматная грамматика представлена в недетерминированной форме, некритичен, так как для такой грамматики можно найти эквивалентную автоматную грамматику в детерминированной форме. Часто такое преобразование грамматик можно выполнить, исходя из общих соображений, но существует формализованный алгоритм такого преобразования.

Учитывая, что для каждой автоматной грамматики существует эквивалентный ей конечный автомат, сформулируем этот алгоритм, как алгоритм перехода от НКА к ДКА.

Пусть $(\Sigma, Q, q_0, \delta_N, F_N)$ – исходный НКА, $(\Sigma, P, p_0, \delta_D, F_D)$ – искомый ДКА, эквивалентный исходному. Тогда алгоритм перехода может быть записан следующим образом:

1. Создадим заготовку таблицы переходов искомого автомата и добавим в нее столбец, соответствующий начальному состоянию p_0 ДКА, помечая его в шапке таблицы множеством начальных состояний НКА: $p_0 := \{q_i | q_i \in q_0\}$.

2. В том случае, если для некоторого состояния ДКА p_x, помеченного множеством состояний НКА Q_x функция переходов еще не вычислена, вычисляем ее следующим образом:

$\delta_D(p_x,a) := \{\delta_N(q_i,a) | q_i \in Q_x\}$, $a \in \Sigma$.

Другими словами для каждого символа *a* алфавита Σ, помечаем соответствующую ячейку $\delta_D(p_x,a)$ ДА множеством всех состояний $\delta_N(q_i,a)$ НКА, достижимых по символу a из тех состояний q_i НКА, которыми помечен текущий рассматриваемый столбец p_x ДА.

3. Для каждого вычисленного на шаге 2 множества $\delta_D(p_x,a)$ проверяем, существует ли в таблице ДА столбец, обозначенный этим множеством. Если такого столбца нет, то создаем его, включая в множество состояний ДКА новое состояние, помеченное как $\delta_D(p_x,a)$.

4. Повторяем шаги 2 и 3 до тех пор, пока все переходы $\delta_D(p_x,a)$ не будут вычислены и порождение новых столбцов (состояний) ДКА не будет прекращено.

5. Определяем заключительные состояния ДКА. Для этого помечаем столбец p_x ДКА как допускающее состояние, если он содержит хотя бы одно допускающее состояние НКА, т.е.

$F_D=\{p_i|q_j\in p_i\cup q_j\in F_N\}$.

С использованием приведенного выше алгоритма, например, НКА следующего вида:

↓ ↓

	A	B	C	D
a	A,B		A	
b		B,D		
c			C,B	

1

будет преобразован к следующему ДКА

↓

	{A,C}	{A,B}	{C,B}	{A}	{B,D}
a	{A,B}	{A,B}	{A}	{A,B}	
b		{B,D}	{B,D}		{B,D}
c	{C,B}		{C,B}		

1

Переход от детерминированной формы ко вполне детерминированной форме осуществляется достаточно просто. Для этого в грамматику вводят так называемый ошибочный нетерминальный символ E и для каждого нетерминального символа A ($A\neq E \cup A\neq F$) добавляют правила $A\rightarrow aE$ для тех терминальных символов a, которые еще не участвуют в правилах для A:

$R=R\cup\{ A\rightarrow aE \mid (A\neq E) \cup (A\neq F) \cup (A\rightarrow aB \notin R) \}$, $A,B\in V_N$, $a\in V_N$.

В таблице переходов ДКА это аналогично введению дополнительного ошибочного состояния и заполнению пустых клеток переходами в новое ошибочное состояние.

Для написания программы удобно использовать граф переходов, построенный по грамматике или ДКА.

1.2 Задание на лабораторную работу

Составить автоматную грамматику и на ее основе реализовать лексический анализатор языка, цепочки которого имеют вид, указанный в задании. Лексический анализатор должен преобразовывать исходный текст в последовательность лексем. По результатам работы анализатора должны формироваться таблицы идентификаторов и констант.

1.3 Пример выполнения лабораторной работы

Пусть необходимо построить лексический анализатор для цепочек следующего вида:

while <условие> **do** <оператор> **end**

<условие> → <сравнение>|<условие><логическая операция><сравнение>

<сравнение> → <операнд>|<операнд><операция сравнения><операнд>

<операция сравнения> → <|<=|<>

<операнд> → <идентификатор>|<константа>

<логическая операция> → **and**|**or**

<оператор> → <идентификатор> = <арифметическое выражение>

<арифметическое выражение> → <операнд>|

<арифметическое выражение><арифметическая операция><операнд>

<арифметическая операция> → +|-

Пример цепочки

while a < b and b <= c do b=b+c-20 end

Решение

Очевидно, в задании описан цикл с предусловием некоторого языка. В качестве алфавита выступают буквы латинского алфавита, цифры, другие значащие символы (+,-,=,<,>) и незначащие символы (пробел, табуляция, возврат коретки, перевод строки).

Исходя из задания, можно выделить следующие базовые синтаксические элементы языка:

- ключевые (зарезервированные) слова (while, do, end, and, or)
- идентификаторы,
- константы,
- специальные символы (+,-,<,<=,<>,=).

При этом идентификаторы состоят из букв и цифр и обязательно начинаются с буквы, константы - только из цифр, специальные символы - могут состоять из одного или двух символов алфавита.

При разработке лексического анализатора примем, что перечисленные базовые элементы языка разделяются в программе произвольным количеством незначащих символов, при этом роль разделителя могут играть специальные символы. Другими словами специальные символы могут следовать непосредственно за другим элементом языка, ровно, как и любой элемент языка может следовать за специальным символом, не будучи отделенным от него незначащими символами.

На основе выделенных базовых элементов можно составить таблицу терминальных символов языка.

Таблица 1.1 - Таблица терминальных символов:

Индекс	**Символ**	**Категория**	**Тип**	**комментарий**
0	while	ключевое слово	while	начало заголовка цикла
1	do	ключевое слово	do	начало тела цикла
2	end	ключевое слово	end	конец тела цикла
3	and	ключевое слово	and	логическое "И"
4	or	ключевое слово	or	логическое "ИЛИ"
5	<	специальные символ	rel	операция сравнения "строго меньше"
6	<=	специальные символ	rel	операция сравнения "меньше или равно"
7	<>	специальные символ	rel	операция сравнения "неравно"
8	==	специальные символ	rel	операция сравнения "равно"
9	=	специальные символ	as	операция присваивания
10	+	специальные символ	ao	операция сложения
11	-	специальные символ	ao	операция вычитания

При составлении таблицы мы сопоставили каждому базовому элементу языка тип соответствующей лексемы. При этом некоторые базовые элементы объединены в один тип. В частности, мы объединили в тип **rel** операции сравнения <,<=<> и ==, в тип **ao** арифметические операции + и -. Причина, по которой не объединены в один тип логические операции **and** и **or** заключается в том, что эти операции имеют различный приоритет. В случае их объединения учесть приоритет на этапе синтаксического анализа программы станет затруднительно.

Полученная таблица терминальных символов языка будет использоваться лексическим анализатором при трансформации исходной программы в цепочку лексем. Сам лексический анализатор будем реализовывать как анализатор автоматного языка. Для начала составим автоматные грамматики для различных типов лексем нашего языка. Начнем с идентификаторов и констант языка (здесь и далее символом ⊥ обозначен символ конца строки):

S→aA|bA|…|zA

A→aA|bA|…|zA

A→ 0A|1A|…|9A

A→⊥

S→ 0A|1A|…|9A

A→ 0A|1A|…|9A

A→⊥

Обычно в языках программирования на идентификаторы накладывается ограничение, чтобы идентификаторы не совпадали с ключевыми словами. Тем не менее, согласно описанной грамматике идентификаторов можно породить и ключевые слова языка. Выйти из ситуации можно, видоизменив грамматику таким образом, чтобы она не могла породить ключевые слова. Например, следующая грамматика не может породить ключевое слово do:

S→aA|bA|cA|dB|eA|…|zA

A→aA|bA|…|zA|0A|1A|…|9A|⊥

B→aA|bA|…|nA|oC|pA|…|zA|0A|1A|…|9A|⊥

C→aA|bA|…|zA|0A|1A|…|9A

Конечно, даже в случае очень простого языка, описанного в задании, нам придется потрудиться, чтобы указанным образом обойти все ключевые слова. Грамматика в этом случае сильно разрастется и писать по такой грамматике анализатор будет гораздо сложнее.

В случае практически значимых языков так, конечно не поступают, а просто сравнивают выделенный идентификатор с ключевыми словами из таблицы терминальных символов и в случае совпадения считают выделенную лексему ключевым словом, иначе – идентификатором.

Для специальных символов грамматики строятся довольно просто:

S→<A	S→=B	S→+C|-C
A→=D|>D|⊥	B→=G|⊥	C→⊥
D→⊥	G→⊥	

Еще проще строятся грамматики для последовательностей незначащих символов (знак # с числом означает код символа):

S→#32S|…|#10S|#13S|⊥

Как видно, все описанные грамматики уже находятся в детерминированной форме. Значит, мы можем реализовать несколько распознавателей, отвечающих каждый за свой тип лексем. В этом случае, когда один анализатор дает отрицательный ответ, мы можем возвращаться к началу подцепочки и пытаться распознать ее другим анализатором и т.д. Такой подход действительно имеет право на существование, но нам хотелось бы построить однопроходный лексический анализатор без возвратов.

Для этого выполним объединение языков идентификаторов, констант, специальных символов, и незначащих символов, совмещая начальные и конечные вершины графов, описывающих соответствующие грамматики. Полученная грамматика представлена на рис.1 в виде графа.

Без сомнения, представленная грамматика будет допускать любой из базовых элементов языка, однако не сможет работать с последовательностями таких элементов. Другими словами, искомая грамматика должна определять не язык базовых элементов, а итерацию такого языка. С учетом того, что базовые элементы языка (за исключением специальных символов) отделяются друг от друга незначащими символами, и что специальные символы могут не отделяться от других элементов языка, искомая грамматика примет вид, представленный на рис. 2.

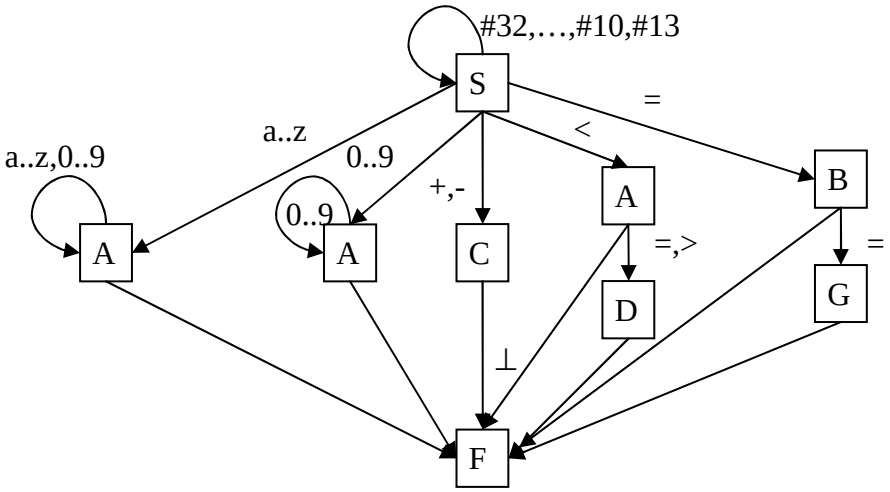

Рис.1.1 – Представление грамматики базовых элементов языка в виде графа

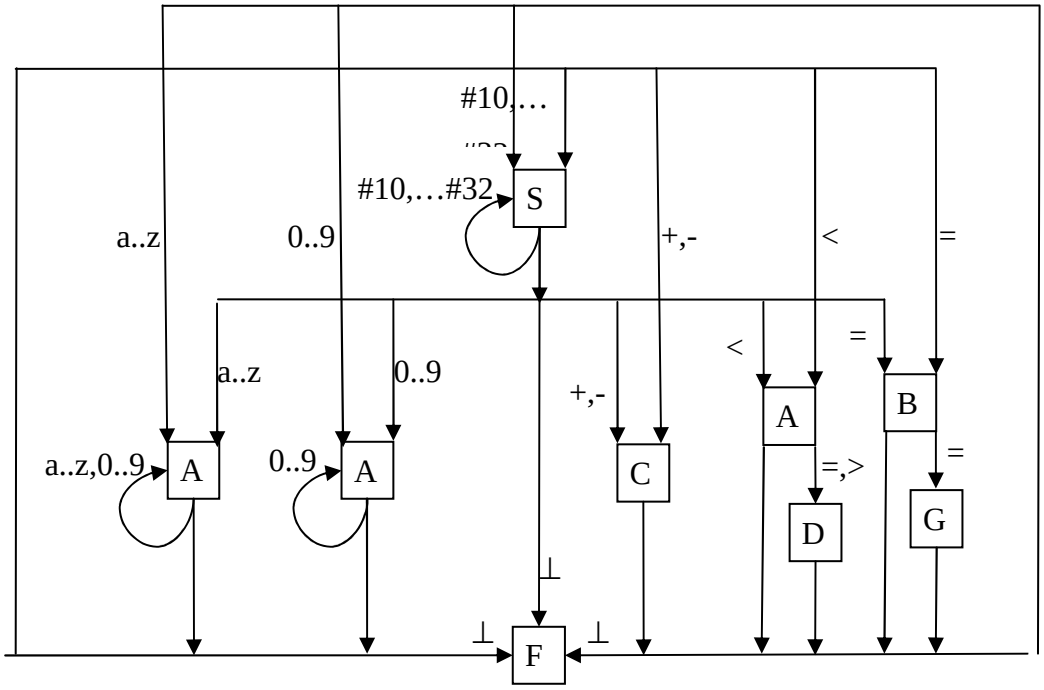

Рис.1.2 – Представление грамматики базовых элементов языка в виде графа

Очевидно, приведенный на рис. 2 граф представляет автоматную грамматику в детерминированной форме. Для приведения грамматики ко вполне детерминированной форме вводят дополнительную (ошибочную) вершину графа и проводят в нее ребра из всех вершин (за исключением заключительной), помечая ребра теми символами алфавита, по которым переходов из рассматриваемых вершин нет. Мы, однако, для краткости не будем приводить здесь этот граф,

Процедура анализа по полученному графу составляется достаточно легко.

```
enum EState {S, Ai, Ac, As, Bs, Cs, Ds, Gs, E, F};
enum ELexType{ lWhile, lDo, lEnd, lAnd, lOr, lRel, lAs, lAo,
               lVar, lConst };

struct Lex{
    ELexType type;
    int index;
    int pos;
    Lex* next;
} *pFirst = NULL, *pLast = NULL;

bool LexAnalysis( const char* text )
{
    const char *str = text, *lexstart;
    EState state = S, prevState;
    int add;

    while ((state != E) && (state != F))
    {
        prevState = state;
        add = true;
        switch (state) {
            case S:  {
                if (isspace(*str)) ;
                else if (isalpha(*str))  state = Ai;
                else if (isdigit(*str))  state = Ac;
                else if (*str=='<')      state = As;
                else if (*str=='=')      state = Bs;
                else if ((*str=='+')||(*str == '-'))   state = Cs;
                else if (*str==0)        state = F;
                else state = E;
                add = false;
                break;
            }
            case Ai:  {
                if (isspace(*str))       state = S;
```

```
            else if (isalnum(*str))  add = false;
            else if (*str=='<')      state = As;
            else if (*str=='=')      state = Bs;
            else if ((*str=='+')||(*str == '-'))  state = Cs;
            else if (*str==0)        state = F;
            else { state = E;  add = false; }
            break;
        }
        case Ac:  {
            if (isspace(*str))       state = S;
            else if (isdigit(*str))  add = false;
            else if (*str=='<')      state = As;
            else if (*str=='=')      state = Bs;
            else if ((*str=='+')||(*str == '-'))  state = Cs;
            else if (*str==0)        state = F;
            else { state = E; add = false; }
            break;
        }
        case As:  {
            if (isspace(*str))       state = S;
            else if (isalpha(*str))  state = Ai;
            else if (isdigit(*str))  state = Ac;
            else if ((*str=='=')||(*str == '>'))
                {  state = Ds;  add = false;  }
            else if (*str==0)        state = F;
            else { state = E; add = false; }
            break;
        }
        case Bs:  {
            if (isspace(*str))       state = S;
            else if (isalpha(*str))  state = Ai;
            else if (isdigit(*str))  state = Ac;
            else if (*str=='=') {  state = Gs;  add = false;
}
            else if (*str==0)        state = F;
            else { state = E; add = false; }
```

```
                break;
            }
            case Cs: case Ds: case Gs:  {
                if (isspace(*str))       state = S;
                else if (isalpha(*str))  state = Ai;
                else if (isdigit(*str))  state = Ac;
                else if (*str==0)        state = F;
                else { state = E; add = false; }
                break;
            }
        }
        if ( add )  AddLex(prevState, text, lexstart, str);
        if ( (state != prevState) &&
             (state==Ai || state==Ac || state==As ||
              state==Bs || state==Cs) )
            lexstart = str;

        if ((state!=E)&&(state !=F)) str++;
    }

    return (state == F);
}


void AddLex( EState state, const char* txt,
    const char* lexPtr, const char* curPtr);

int AddVar(const char* var);

int AddConst(const char* cons);
```

1.4 Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Титульный лист.
2. Задание на лабораторную работу.

3. Описание цепочек анализируемого языка.

4. Таблица терминальных символов с подробным описанием.

5. Описание основных типов лексем с использованием грамматик.

6. Описание грамматики автоматного языка в виде графа.

7. Описание основных алгоритмов и структур данных, используемых в программе:

а) описание структур данных, используемых для представления последовательности лексем, таблиц идентификаторов и констант;

б) описание процедур и функций, отвечающих за работу с введенными структурами данных;

в) описание алгоритма анализа автоматного языка.

5. Описание интерфейса пользователя программы.

6. Контрольный пример и результаты тестирования.

7. Листинг программы.

1.5 Контрольные вопросы

1. Назовите основные этапы компиляции.

2. Дайте краткую характеристику этапу лексического анализа.

3. Что такое лексема? Приведите примеры.

4. С какими таблицами осуществляется работа на этапе лексического анализа?

5. Дайте определение грамматики

6. Дайте определение автоматной грамматики

7. Назовите основные этапы создания анализатора по автоматной грамматике.

8. Какие способы задания автоматного языка Вы знаете?

9. Дайте определение конечного автомата.

10. Что такое детерминированная, недетерминированная и вполнедетерминированная формы автоматных грамматик? Приведите примеры.

11. Дайте определение детерминированному (ДКА) и недетерминированному конечному автомату (НКА).

12. В чем заключается алгоритм перехода от НКА к ДКА. Приведите пример.

13. Как перейти от грамматики в детерминированной форме к грамматике во вполне детерминированной форме?

14. Как по вполне детерминированной автоматной грамматике составить программу анализа?

2 Лабораторная работа №2

Тема лабораторной работы: Построение синтаксического анализатора методом рекурсивного спуска

2.1 Теоретические основы лабораторной работы

Метод рекурсивного спуска относится к нисходящим методам анализа КС-языков. Класс КС-грамматик, для которых может быть применен метод рекурсивного спуска, довольно узок, однако достаточно часто с использованием эквивалентных преобразований грамматику можно привести к классу, допускающему анализ методом рекурсивного спуска. Основным ограничением, накладываемым на грамматику $G=(S,V_N,V_T,R)$ является то, что для правил $A\rightarrow\alpha_1|\alpha_2|\ldots|\alpha_N$, в которых нетерминальный символа A стоит в левой части, множества $First_1(\alpha_1)$, $First_1(\alpha_2)$, …, $First_1(\alpha_N)$ не должны пересекаться, то есть:

$First_1(\alpha_i) \cap First_1(\alpha_j) = \varnothing, \forall\ \alpha_i, \alpha_j: \{A\rightarrow\alpha_i, A\rightarrow\alpha_j\}\subseteq R$

Собственно, построение анализатора методом рекурсивного спуска выполняется следующим образом:

1. Для каждого нетерминального символа A грамматики определяется своя процедура, которая отвечает за разбор цепочек, выводимых из нетерминального символа A.

2. Распознающая процедура для нетерминального символа A, принимает на вход текущую позицию просмотра входной строки и сопоставляет терминальный символ a, стоящий в позиции просмотра с символами множеств $First_1(\alpha_1)$, $First_1(\alpha_2)$, …, $First_1(\alpha_N)$, составленных для правых частей правил грамматики $A\rightarrow\alpha_1|\alpha_2|\ldots|\alpha_N$.

3. В том случае, если символ a входит во множество $First_1(\alpha_i)$ для правила $A\rightarrow\alpha_i$ ($a\in First_1(\alpha_i)$), то управление передается той ветви анализа процедуры, которая отвечает за распознавание соответствующей альтернативы $A\rightarrow\alpha_i$.

4. Ветвь анализа, соответствующая альтернативе $A\rightarrow\alpha_i$, строится как последовательность действий по распознаванию цепочки

$\alpha_i=v_1v_2\ldots v_N$, где $v_1,v_2,\ldots v_N\in(V_N\cup V_T)$.

При этом, если v_i является терминальным символом, то он сопоставляется с текущим анализируемым символом a. Если символы совпадают ($a=v_i$), то происходит переход к следующей позиции просмотра входной строки, в противном случае, фиксируется синтаксическая ошибка во входной строке.

Если v_i является нетерминальным символом, то вызывается процедура, отвечающая за разбор цепочек, выводимых из нетерминального символа v_i. Таким образом, реализации

функции может содержать как обращение к другим процедурам, так и рекурсивное обращение к самой себе в том случае, когда правила для нетерминального символа A содержат рекурсию.

5. В том случае, если символ a не входит ни в одно из множеств $First_1(\alpha_i)$ ($a \notin First_1(\alpha_i)$, i=1..N), и среди правил для A есть аннулирующее правило $A \rightarrow \varepsilon$, то распознающая процедура для нетерминального символа A считается успешно завершенной. В противном случае фиксируется синтаксическая ошибка во входной строке.

На практике разработку анализатора методом рекурсивного спуска удобно выполнять по расширенной бэкусовой нормальной форме, в которой помимо обозначения альтернативы "|" при записи правил грамматики используется обозначение необязательных (факультативных) элементов "[…]" и повторяющихся элементов (итерация) "{…}". В этом случае альтернативным частям правил будут соответствовать разные ветви условного оператора (или оператора множественного выбора) соответствующей распознающей процедуры. Факультативным элементам будут соответствовать отдельные условные операторы в ветвях соответствующих альтернатив, а повторяющимся элемментам – циклы в ветвях соответствующих альтернатив.

В качестве примера применения метода рекурсивного спуска рассмотрим следующую грамматику:

<Assignment>→**id** = <Expression>;

<Expression>→<Term> | <Expression> + <term>

<Term>→<Factor> | <Term> * <factor>

<Factor>→**id** | **id** (<Parameters>) | **const** | (<Expression>)

<Parameters>→<Expression>|<Parameters>**,** <Expression>

Переходя к расширенной бэкусовой нормальной форме грамматика примет вид:

<Assignment>→**id** = <Expression>;

<Expression>→<Term> { + <Term> }

<Term>→<Factor> { * <Factor> }

<Factor>→**id** [(<Parameters>)]| **const** | (<Expression>)

<Parameters>→<Expression> {**,** <Expression>}

Положим теперь, что текущий анализируемый символ всегда доступен в виде переменной symbol, принимающей значения некоторого перечислимого типа. Пусть алфавит грамматики задается константами того же перечислимого типа:

Положим также, что функция Scan() осуществляет переход к следующей позиции входной строки, а Error() – выводит сообщение об ошибке и прекращает анализ. В этом

случае процедуры, реализующие метод рекурсивного спуска приведенной выше грамматики имеют вид:

Таблица 2.1 – Задание констант перечислимого типа

Константа	**Символ грамматики**
ID	**id**
CONST	**const**
ASSIGN	=
SEMICOLON	;
COLON	,
PLUS	+
MULT	*
OPENPAREN	(
CLOSEPAREN	)

```
void Assignment()
{
    if (symbol==ID)  {
        Scan();
        if (symbol != ASSIGN) Error();
        Scan();
        Expression();
        if (symbol != SEMICOLON) Error();
        Scan();
    }
    else Error();
}

void Expression()
{
    Term();
    while (symbol == PLUS)  {
        Scan();
        Term();
    }
}
```

```
void Term()
{
    Factor();
    while (symbol == MULT) {
        Scan();
        Factor();
    }
}

void Factor()
{
    if (symbol == ID)  {
        Scan();
        if (symbol == OPENPAREN)  {
            Scan();
            Parameters();
            if (symbol != CLOSEPAREN)  Error();
            Scan();
        }
    }
    else if (symbol == CONST)
        Scan();
    else if ( symbol == OPENPAREN)  {
        Scan();
        Expression();
        if (symbol != CLOSEPAREN)  Error();
        Scan();
    }
    else Error();
}

void Parameters()
{
    Expression();
    while (symbol == COLON)  {
        Scan();
```

```
        Expression();
    }
}
```

2.2 Задание на лабораторную работу

С использованием метода рекурсивного спуска реализовать синтаксический анализатор языка, цепочки которого имеют вид, указанный в задании на первую лабораторную работу. Синтаксический анализатор должен принимать на вход последовательность лексем, сформированную лексическим анализатором и восстанавливать дерево разбора цепочки. В случае ошибок во входной цепочке анализатор должен дать пользователю информативное сообщение с указанием причины и места возникновения ошибки.

2.3 Пример выполнения лабораторной работы

Начнем выполнение лабораторной работы с составления контекстно-свободной грамматики языка с использованием тех типов лексем, которые были введены при выполнении лабораторной работы №1.

Напомним, что грамматика – это четверка объектов <S,V_T,V_N,R>, где V_T – множество терминальных символов грамматики, V_N - множество нетерминальных символов грамматики, S – начальный символ грамматики,R- множество правил грамматики. Очевидно, терминальными символами грамматики в нашем случае будут являться типы лексем, введенные при выполнении лабораторной работы №1, то есть: **while, do, end, and, or, rel, as, ao, var, const**.

Для нетерминальных символов грамматики введем следующие обозначения:

WhileStatement – оператор цикла while, начальный символ грамматики,

Condition – условие цикла

RelExpr – выражение сравнения

Operand – операнд выражения

LogicalOp - логическая операция

Statement – оператор языка

ArithExpr - арифметическое выражение

С использованием введенных символов грамматики правила грамматики примут следующий вид:

<WhileStatement> → **while** <Condition> **do** <Statement> **end**

<Condition> → <RelExpr> | <Condition><LogicalOp><RelExpr >

<RelExpr> → <Operand>|<Operand> **rel** <Operand>

<Operand> → **var | const**

<LogicalOp> →**and** | **or**

<Statement>→**var as** <ArithExpr>

<ArithExpr>→ <Operand> | <ArithExpr> **ao** <Operand>

Нетрудно видеть, что в рассматриваемой грамматике отсутствует учет приоритета логических операций в правилах

<Condition> → <RelExpr>|< Condition><LogicalOp><RelExpr >

<LogicalOp> →**and** | **or**

Для учета приоритета операций видоизменим указанные правила следующим образом:

<Condition> → <LogExpr>|<Condition><LogicalOp1><LogExpr>

<LogExpr> → <RelExpr>|< LogExpr ><LogicalOp2><RelExpr >

Здесь <LogicalOp1> - логические операции низкого приоритета, <LogicalOp2> - логические операции высокого приоритета:

<LogicalOp1> → **or**

<LogicalOp2> → **and**

Упростим грамматику, устранив нетерминалы <LogicalOp1> и <LogicalOp2>, содержащие по единственному терминальному символу. Полученная в результате грамматика примет следующий вид.

<WhileStatement> → **while** <Condition> **do** <Statement> **end**

<Condition> → <LogExpr>|<Condition>**or**<LogExpr>

<LogExpr> → <RelExpr>|< LogExpr >**and**<RelExpr>

<RelExpr> → <Operand>|<Operand> **rel** <Operand>

<Operand> → **var | const**

<Statement>→**var as** <ArithExpr>

<ArithExpr>→ <Operand> | <ArithExpr> **ao** <Operand>

Нетрудно видеть, что представленная грамматика не пригодна для анализа методом рекурсивного спуска. В частности, в ней содержатся леворекурсивные правила для нетерминалов <Condition>, <LogExpr>, <ArithExpr>. Устраним левую рекурсию, заменив правила для указанных нетерминалов на следующую группу правил:

<Condition> → <LogExpr><Condition1>

<Condition1> → **or** <LogExpr><Condition1> | ε

<LogExpr> → <RelExpr><LogExpr1>

<LogExpr1> → **and** <RelExpr><LogExpr1> **|** ε

<ArithExpr>→ <Operand><ArithExpr1>

<ArithExpr1>→ **ao** <Operand><ArithExpr1> **|** ε

Кроме того, для нетерминала <RelExpr> в грамматике имеются две альтернативы, начинающиеся с одного и того же нетерминального символа (<Operand>), что неизбежно влечет за собой совпадение множеств $First_1$ для этих альтернатив. Чтобы устранить проблему, выполним левую факторизацию:

<RelExpr> → <Operand><RelExpr1>

<RelExpr1> → **rel** <Operand> **|** ε

Окончательно грамматика принимает следующий вид:

<WhileStatement> → **while** <Condition> **do** <Statement> **end**

<Condition> → <LogExpr><Condition1>

<Condition1> → **or** <LogExpr><Condition1> **|** ε

<LogExpr> → <RelExpr><LogExpr1>

<LogExpr1> → **and** <RelExpr><LogExpr1> **|** ε

<RelExpr> → <Operand><RelExpr1>

<RelExpr1> → **rel** <Operand> **|** ε

<Operand> → **var | const**

<Statement>→**var as** <ArithExpr>

<ArithExpr>→ <Operand><ArithExpr1>

<ArithExpr1>→ **ao** <Operand><ArithExpr1> **|** ε

Построение анализаторов методом рекурсивного спуска удобно производить по грамматике, представленной в расширенной нотации Бэкуса-Наура. Представим нашу грамматику в такой нотации:

<WhileStatement> → **while** <Condition> **do** <Statement> **end**

<Condition> → <LogExpr>{**or** <LogExpr>}

<LogExpr> → <RelExpr>{**and** <RelExpr>}

<RelExpr> → <Operand>[**rel** <Operand>]

<Operand> → **var | const**

<Statement> → **var as** <ArithExpr>

<ArithExpr> → <Operand> {**ao** <Operand>}

На основе полученного представления реализуем анализатор методом рекурсивного спуска.

Ниже представлен набор функций, анализирующий входные цепочки в соответствии с рассматриваемой грамматикой. Представленные функции не имеют параметров и возвращают булево значение, показывающее, успешно ли проведен анализ подцепочки для рассматриваемого нетерминала. Имена функций совпадают с именами нетерминалов в рассмотренной грамматике.

Предполагается, что в каждой функции доступен указатель p на текущую рассматриваемую лексему (представляющую собой структуру, определенную в лабораторной работе №1), а также перечислимый тип для лексем (lWhile, lDo и т.д.). Кроме того, предполагается, что реализована функция Error, осуществляющая вывод пользователю сообщения об ошибке со следующей сигнатурой:

```
void Error( const char* msg, int pos )
```

Здесь msg – текст сообщения об ошибке, pos – знакопозиция места возникновения ошибки в исходном тексте программы.

```
bool WhileStatement()
{
     if (!p || p->type != lWhile) { Error("Ожидается while", p-
>pos); return false; }
     p=p->next;
     if (! Condition()) return false;
     if (!p || p->type != lDo) { Error("Ожидается do", p->pos);
return false; }
     p=p->next;
     if (! Statement()) return false;
     if (!p || p->type != lEnd) { Error("Ожидается end", p->pos);
return false; }
     p=p->next;
     if (p) { Error("Лишние символы", p->pos); return false; }
     return true;
}

bool Condition()
{
```

```
     if (! LogExpr()) return false;
     while (p && p->type==lOr)  {
         p=p->next;
         if (! LogExpr()) return false;
     }
     return true;
}

bool LogExpr()
{
     if (! RelExpr()) return false;
     while (p && p->type==lAnd)  {
         p=p->next;
         if (! RelExpr()) return false;
     }
     return true;
}

bool RelExpr()
{
     if (! Operand()) return false;
     if (p && p->type==lRel )  {
         p=p->next;
         if (! Operand()) return false;
     }
     return true;
}

bool Operand()
{
    if ( !p || (p->type != lVar && p->type != lConst) )
        { Error("Ожидается переменная или константа", p->pos);
return false; }
    p=p->next;
    return true;
}
```

```
bool LogicalOp()
{
    if ( !p || (p->type != lAnd && p->type != lOr) )
        { Error("Ожидается логическая операция", p->pos); return
false; }
    p=p->next;
    return true;
}

bool Statement()
{
    if (!p || p->type != lVar)  { Error("Ожидается переменная", p-
>pos); return false; }
    p=p->next;
    if (!p || p->type != lAs)   { Error("Ожидается присваивание",
p->pos); return false; }
    p=p->next;
    if (! ArithExpr()) return false;
    return true;
}

bool ArithExpr()
{
    if (! Operand()) return false;
    while (p && p->type == lAo)  {
        p=p->next;
        if (! Operand()) return false;
    }
    return true;
}
```

2.4 Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Титульный лист.

2. Задание на лабораторную работу.

3. Описание грамматики исходного языка.

4. Описание всех преобразований исходной грамматики и грамматику в окончательном виде, представленную в расширенной бэкусовой нормальной форме.

5. Описание основных алгоритмов и структур данных, используемых в программе:

а) описание процедур и функций рекурсивного спуска;

б) описание типов данных, используемых для представления последовательности лексем;

в) описание процедур и функций, отвечающих за работу с последовательностью лексем и вывод синтаксических ошибок.

5. Описание интерфейса пользователя программы.

6. Контрольный пример и результаты тестирования.

7. Листинг программы.

2.5 Контрольные вопросы

1. Дайте определение контекстно-свободной грамматики

2. Скажите, какой класс грамматик допускает построение анализаторов методом рекурсивного спуска.

3. Дайте определение функции $First_1(\ldots)$. Приведите примеры.

4. Как строится анализатор методом рекурсивного спуска?

5. Какие элементы используются в расширенной бэкусовой нормальной форме?

3 Лабораторная работа №3

Тема лабораторной работы: Включение семантики в анализатор. Создание внутренней формы представления программы.

3.1 Теоретические основы лабораторной работы

В большинстве случаев при трансляции целевой код не создается транслятором непосредственно по исходному коду. Вместо этого исходная программа вначале переводится в некую внутреннюю форму. При этом используемая форма внутреннего представления исходной программы выбирается таким образом, чтобы было возможно проводить эффективную машинно-независимую оптимизацию. Следует отметить, что такое разделение транслятора на части и использование внутренней формы представления позволяет упростить построение трансляторов с различных языков и для разных целевых платформ. Так, в случае, когда требуется построить транслятор с нового языка, достаточно разработать ту часть транслятора, которая отвечает за генерацию внутренней формы представления программы. В том случае, когда необходимо построить компилятор для новой целевой платформы или транслятор для нового целевого языка, требуется лишь разработать часть, отвечающую за генерацию целевого кода по внутренней форме представления программы. При этом часть транслятора, отвечающая за оптимизацию внутреннего представления программы, в обоих случаях остается прежней.

Любое внутреннее представление содержит элементы двух типов – операторы и операнды. В зависимости от того, какая модель вычислительной машины берется за основу при разработке внутренней формы представления, можно выделить два основных направления. В первом из них за основу берется модель ЭВМ с регистрами (в том числе ЭВМ с произвольным доступом к памяти), во втором - абстрактная стековая машина. Если в первом случае предполагается, что операции производятся с регистрами ЭВМ или непосредственно с памятью с произвольным доступом, то во втором – все операции производятся с содержимым стека, то есть выполнении операции операнды извлекаются из вершины стека, и на их место помещается результат.

Еще одной формой внутреннего представления программы является атрибутные деревья разбора. В такой форме представления вся программа представляется в виде дерева, а отдельные конструкция языка – в виде поддеревьев этого дерева. При этом каждый узел дерева может иметь дополнительные атрибуты: ссылки на таблицы терминальных символов, идентификаторов, констант и т.п.

В том случае, когда при разработке внутреннего представления программы за основу берется регистровая машина или модель ЭВМ с произвольным доступом к памяти, для формирования внутреннего представления часто используется так называемый трехадресный код (тетрады). В нем каждая инструкция описывается как четверка

(<оператор>,<операнд1>,<операнд2>,<результат>)

Операндами и результатом здесь могут являться переменные и константы, описанные в исходной программе или временные переменные, автоматически сгенерированные компилятором. Сам оператор во временной форме представления должен входить в набор команд, поддерживаемый выбранной моделью ЭВМ. Главное здесь, что каждая команда выполняет ровно одну базовую операцию над аргументами, находящимися в регистрах (памяти) и помещает результат также в регистр (память).

С использованием тетрад, например, следующий фрагмент кода на C++, вычисляющий сумму элементов массива:

```
s = 0;
for (i = 0; i < 10; ++i) {
    s += ar[i];
}
```

может быть представлен следующим образом:

```
      (:=, 0, , s)       // s = 0
      (:=, 0, , i)       // i = 0
L1:   (<, i, 10, t0)     // t0 = i<10
      (JZ, t0, L2,)      // if (t0 == 0) goto L2
      (+, ar, i, t1)     // t1 = ar+i
      (**, t1, , t2)     // t2=*t1, разыменование
      (+, s, t2, s)      // s=s+t2
      (+, i, 1, i)       // i=i+1
      (JMP, L1, ,)       // goto L1
L2:
```

Основным недостатком тетрад является большое количество временных переменных, порождаемых при генерации внутренней формы представления программы. Частично разрешить эту проблему помогают так называемые триады, имеющие следующий вид:

(<оператор>,<операнд1>,<операнд2>)

В триадах отсутствует поле результата и если какая-либо операция должна будет воспользоваться результатом сгенерированной ранее тетрады, в качестве аргумента будет

использоваться ссылка на сгенерированную ранее триаду. В качестве примера приведем пример для выражения a*b+c*d:

```
1:    (*, a, b)
2:    (*, c, d)
3:    (+,(1),(2))
```

Таким образом, тетрады и триады представляют собой довольно близкую по своей сути форму к объектному коду программы. Достоинствами тетрад и триад является относительная простота оптимизации внутренней формы представления программы, так как их можно переставлять и удалять.

Внутреннее представление программы, при разработке которого за основу взята абстрактная стековая машина, в большинстве случаев основано на выполнении операций, над аргументами, располагающимися в двух верхних позициях стека, а не в регистрах или памяти с произвольным доступом.

Наиболее известной из таких форм внутреннего представления является польская инверсная запись (ПОЛИЗ), известная также как постфиксная нотация. В отличие от обычной инфиксной формы записи в постфиксной нотации операции записываются после аргументов, а не между ними. Например, операции a + b и a := b, представленные в традиционной инфиксной форме, в постфиксной нотации примут вид: a b + и a b :=.

Операции в выражениях, представленных в постфиксной форме записи, выполняются в порядке записи слева-направо, приоритеты и ассоциативность операций не принимаются во внимание. Не используются в постфиксной форме записи и скобки.

Для вычисления вручную выражения, представленного в постфиксной форме записи, оно читается слева-направо. Как только в выражении встречается операция, она выполняется над двумя стоящими непосредственно перед ней операндами, после чего операнды и операция вычеркиваются из выражения и на их место вписывается результат выполнения операции. После этого выражение вычисляется дальше по тому же правилу, пока не будет получено единственное значение - результат вычисления выражения.

В качестве примера рассмотрим вычисление выражения (2+3)*7+4/2. Соответствующая ему польская инверсная запись имеет вид: 2 3 + 7 * 4 2 / +.

```
2 3 + 7 * 4 2 / +
5 7 * 4 2 / +
35 4 2 / +
35 2 +
37
```

Составить вручную ПОЛИЗ по инфиксной записи для небольших выражений не составляет большого труда. Для этого необходимо помнить, что операции в ПОЛИЗе всегда следуют в том порядке, в котором они должны выполняться, а операнды – в том же порядке, в каком они следовали в инфиксной записи. При этом операторы должны располагаться в ПОЛИЗе сразу же за своими операндами.

Следует отметить, что польская инверсная запись короче инфиксной формы записи. Это дает возможность уменьшить объём программ по сравнению с традиционной формой записи, что немаловажно для тех вычислительных устройств, где имеются жёсткие требования по использованию памяти.

Следует отметить, что с использованием постфиксной формы записи возможно описание не только арифметических операций, но и любых других операций, в частности операций управления потоком вычислений в программе (см. таблицу 3.1).

Таблица 3.1 – Операции в постфиксной форме записи

Операция	**Описание**	**Формат**
JMP	Команда безусловного перехода	унарная
JZ	Команда условного перехода по лжи (если значение первого аргумента ложно - имеет нулевое значение)	бинарная
:=	Операция присваивания	бинарная
+	Операция сложения	бинарная
-	Операция вычитания	бинарная
*	Операция умножения	бинарная
/	Операция деления	бинарная
NOT	Операция логического «не»	унарная
AND	Операция логического «и»	бинарная
OR	Операция логического «или»	бинарная
==	Операция сравнения на равенство	бинарная
!=	Операция сравнения на неравенство	бинарная
<	Операция сравнения «строго меньше»	бинарная
<=	Операция сравнения «меньше или равно»	бинарная
>	Операция сравнения «строго больше»	бинарная
>=	Операция сравнения «больше или равно»	бинарная
<<	Ввод значения из входного потока	унарная
>>	Вывод значения во входной поток	унарная

Этот факт позволяет использовать постфиксную форму записи в качестве внутренней формы представления программы. Рассмотрим, как в ПОЛИЗе будут выглядеть традиционные управляющие конструкции языков высокого уровня.

Условный оператор, имеющий вид

```
if (<условие>) <оператор1> else <оператор2>;
```

в ПОЛИЗе может быть представлен выражением:

```
<условие> <адр7> JZ   <оператор1> <адр2> JMP  <оператор2> …
адр1      адр2   адр3 адр4        адр5   адр6 адр7        адр8
```

Оператор цикла с предусловием, имеющий вид

```
while (<условие>) <оператор1>;
```

в ПОЛИЗе выглядит следующим образом:

```
<условие> <адр7> JZ   <оператор1> <адр1> JMP  …
адр1      адр2   адр3 адр4        адр5   адр6 адр7
```

Оператор цикла с постусловием, имеющий вид

```
do <оператор1> while (<условие>);
```

в ПОЛИЗе выглядит следующим образом:

```
<оператор1> <условие> NOT  <адр1> JZ   …
адр1        адр2      адр3 адр4   адр5 адр6
```

Для представления цикла

```
for (<выражение1>; <выражение2>; <выражение3>) <оператор>
```

достаточно вспомнить, что этот цикл эквивалентен циклу с предусловием:

```
<выражение1>;
while (<выражение2>) {
    <оператор>;
    <выражение3>;
}
```

Особое внимание следует уделить операции обращения по индексу. В том случае, когда допускается использование только одномерных массивов, такая операция в ПОЛИЗе может иметь вид

```
<имя_массива> <индекс> [
```

Если допустимо использование многомерных массивов операция принимает вид:

```
<имя_массива> <индекс1><индекс2>…<индексN> N [
```

В качестве примера приведем следующий фрагмент программы на языке С:

```
for (i=0; i<10; i++) {
    if (i<5)&&(j>0)  a[i] = j;
```

```
    else  a[i]=0;
}
```

Соответствующая приведенному выше фрагменту запись в ПОЛИЗе имеет вид:

```
i 0 := i 10 < 36 JZ i 5 <  j  0  >  AND 24 JZ a  i  [  j  :=
0 1 2  3 4  5 6  7  8 9 10 11 12 13 14  15 16 17 18 19 20 21

29 JMP a  i  [  0  := i  i  1  +  := 3  JMP
22 23  24 25 26 27 28 29 30 31 32 33 34 35  36
```

В заключение отметим, что в постфиксной записи недопустимо использование одних и тех же обозначений операций для унарных и бинарных операций, так как из контекста записи невозможно определить унарность или бинарность операции. По этой причине для унарных операций следует вводить новые обозначения, либо обеспечивать эмуляцию таких операций через соответствующие бинарные операции (например, унарную операция «-a» трактовать, как бинарную «0-a»).

3.2 Задание на лабораторную работу

Дополнить анализатор, разработанный в рамках лабораторных работ №1 и 2, этапом формирования внутренней формы представления программы.

3.3 Пример выполнения лабораторной работы

В качестве внутренней формы представления программы выберем польскую инверсную запись (ПОЛИЗ). Эта форма представления наглядна и достаточно проста для последующей интерпретации, которая может быть выполнена с использованием стека. Для выбранной формы представления примем следующий набор операций:

Таблица 3.2 – Набор операций внутренней формы представления программы

Операция	**Формат**	**Описание**
JMP	Операция безусловного перехода adr JMP	При выполнении этой операции из стека извлекается единственный операнд adr и осуществляется переход по указанному адресу
JZ	Операция условного перехода по лжи (если значение первого аргумента ложно) val, adr JZ	При выполнении этой операции из стека извлекаются два операнда: булево значение val и адрес перехода adr. В случае если значение val ложно, осуществляется переход по адресу adr. В противном случае управление переходит по следующему за JZ адресу

SET	Операция присваивания var, val SET	При выполнении этой операции из стека извлекаются два операнда: переменная var и значение val. Команда записывает значение val в переменную var
ADD	Операция сложения val1, val2 ADD	При выполнении этой операции из стека извлекаются два операнда: val1 и val2. В стек заносится сумма операндов
SUB	Операция вычитания val1, val2 SUB	При выполнении этой операции из стека извлекаются два операнда: val1 и val2. В стек заносится разность операндов
AND	Операция логического «и» val1, val2 AND	При выполнении этой операции из стека извлекаются два операнда: val1 и val2. В стек заносится результат логического «и»
OR	Операция логического «или» val1, val2 OR	При выполнении этой операции из стека извлекаются два операнда: val1 и val2. В стек заносится результат логического «или»
CMPE	Операция сравнения на равенство val1, val2 CMPE	При выполнении этой операции из стека извлекаются два операнда: val1 и val2. В стек заносится результат выполнения операции сравнения val1=val2
CMPNE	Операция сравнения на неравенство val1, val2 CMPNE	При выполнении этой операции из стека извлекаются два операнда: val1 и val2. В стек заносится результат выполнения операции сравнения val1<>val2
CMPL	Операция сравнения «строго меньше» val1, val2 CMPL	При выполнении этой операции из стека извлекаются два операнда: val1 и val2. В стек заносится результат выполнения операции сравнения val1<val2
CMPLE	Операция сравнения «меньше или равно» val1, val2 CMPLE	При выполнении этой операции из стека извлекаются два операнда: val1 и val2. В стек заносится результат выполнения операции сравнения val1<=val2

Формирование внутренней формы представления будем осуществлять при выполнении синтаксического анализа. Включать действия по формированию ПОЛИЗа будем непосредственно в функции, реализующие метод рекурсивного спуска.

Собственно внутреннюю форму представления программы реализуем как последовательность (одномерный массив или список) элементов, каждый из которых может в свою очередь быть константой, переменной, командой или адресом. Хранение одного элемента в этом случае удобно организовать в виде структуры (записи) следующим образом:

```
enum EEntryType {etCmd, etVar, etConst, etCmdPtr };
enum ECmd {JMP, JZ, SET, ADD, SUB, AND, OR, CMPE, CMPNE, CMPL,
CMPLE };

struct PostfixEntry {
    EEntryType type;
    int index;
};
```

В этом случае в зависимости от типа содержимого, задаваемого полем type перечислимого типа EEntryType, поле index будет интерпретироваться как индекс в таблице переменных или констант, как код команды, или, как адрес другого элемента в ПОЛИЗе.

Для формирования ПОЛИЗа введем следующие функции

Таблица 3.3 – Набор функций для формирования ПОЛИЗа

int WriteCmd(ECmd cmd);	Поместить команду cmd в ПОЛИЗ и возвратить адрес команды в ПОЛИЗЕ
int WriteVar(int ind);	Поместить переменную с индексом ind в ПОЛИЗ и возвратить адрес переменной в ПОЛИЗЕ
int WriteConst(int ind);	Поместить константу с индексом ind в ПОЛИЗ и возвратить адрес константы в ПОЛИЗЕ
int WriteCmdPtr(int ptr);	Поместить адрес ptr в ПОЛИЗ и возвратить адрес помещенного элемента в ПОЛИЗЕ
void SetCmdPtr(int ind, int ptr);	Установить в ПОЛИЗЕ для элемента с адресом ind новое значение ptr

Вызов функций по формированию ПОЛИЗа будем вставлять в те места функций анализа, где анализатор уже завершил распознавание конструкции языка в соответствии с некоторым правилом грамматики.

Рассмотрим правило для начального символа рассматриваемой грамматики

<WhileStatement> → **while** <Condition> **do** <Statement> **end**

Реализация управляющей конструкции цикла while с использованием принятых нами команд ПОЛИЗа, может выглядеть следующим образом

<вычисление условия><следующий за циклом адрес >JZ<тело цикла><адрес начала цикла>JMP

Очевидно, включение процедур формирования ПОЛИЗа будет производиться в двух местах: после формирования части ПОЛИЗа, связанной с вычислением условия цикла и после формирования части, связанной с телом цикла. Обратим внимание на то, что при формировании команды условного перехода JZ нам еще не будет известен адрес конца цикла, так как часть ПОЛИЗа для тела цикла еще не сформирована. Для решения этой проблемы мы будем изменять первоначально заданный адрес перехода после формирования ПОЛИЗа для всей конструкции. Ниже показано включение семантики с комментариями (фрагменты программного кода, относящиеся к формированию внутренней формы представления, выделены жирным шрифтом).

```
bool WhileStatement()
{
     int indFirst = postfixSize;  // сохраняем адрес начала цикла
     if (!p || p->type != lWhile)
     { Error("Ожидается while", p->pos); return false; }
     p=p->next;
     if (! Condition()) return false;
     if (!p || p->type != lDo)
     { Error("Ожидается do", p->pos); return false; }
     p=p->next;
     // сформирована часть ПОЛИЗа, вычисляющая условие цикла
     int indJmp = WriteCmdPtr(-1); //заносим фиктивное значение
                                   //адреса условного перехода
     WriteCmd(JZ);           //заносим команду условного перехода
     if (! Statement()) return false;
     // сформирована часть ПОЛИЗа для тела цикла
     if (!p || p->type != lEnd)
     { Error("Ожидается end", p->pos); return false; }
     p=p->next;
     WriteCmdPtr(indFirst);  //заносим адрес начала цикла
```

```
    int indLast = WriteCmd(JMP);  //заносим команду безусловного
                                  //перехода и сохраняем ее адрес
    SetCmdPtr(indJmp, indLast+1);  //изменяем фиктивное значение
                                   //адреса условного перехода
    if (p) { Error("Лишние символы", p->pos); return false; }
    return true;
}
```

Изменения, связанные с генерацией ПОЛИЗа для функции анализа, соответствующей правилу

<Condition> → <LogExpr>{**or** <LogExpr>}

минимальны. Так как ПОЛИЗ для <LogExpr> , будет формироваться в соответствующей функции, то все, что необходимо сделать – занести операцию OR в ПОЛИЗ по окончании распознавания фрагмента «**or** <LogExpr>». При этом аргументами операции будут являться:

- либо два рассчитанных значения <LogExpr>

(<значение LogExpr><значение LogExpr>OR),

- либо результат выполненной ранее операции OR и рассчитанное значение <LogExpr>

(<результат OR><значение LogExpr>OR).

```
bool Condition()
{
    if (! LogExpr()) return false;
    // сформирована часть ПОЛИЗа для вычисления
    // логического подвыражения
    while (p && p->type==lOr)  {
        p=p->next;
        if (! LogExpr()) return false;
        // сформирована часть ПОЛИЗа для вычисления
        // логического подвыражения
        WriteCmd(OR);   //заносим операцию OR в ПОЛИЗ
    }
    return true;
}
```

Полностью аналогично выполняется включение семантики для правила

<LogExpr> → <RelExpr>{**and** <RelExpr>}

```
bool LogExpr()
{
     if (! RelExpr()) return false;
     // сформирована часть ПОЛИЗа для вычисления
     // подвыражения сравнения
     while (p && p->type==lAnd)  {
         p=p->next;
         if (! RelExpr()) return false;
         // сформирована часть ПОЛИЗа для вычисления
         // подвыражения сравнения
         WriteCmd(AND);  //заносим операцию AND в ПОЛИЗ
     }
     return true;
}
```

Формирование ПОЛИЗа для правила

<RelExpr> → <Operand>[**rel** <Operand>]

выполняется похожим образом. Однако, сформированная по результатам лексического анализа лексема **rel,** может представлять различные операции сравнения. И хотя эти операции и имеют одинаковый приоритет при генерации ПОЛИЗа необходимо выяснить, какая именно команда сравнения должна быть сгенерирована. Сделать это можно путем анализа индекса в таблице терминальных символов. Приведем часть таблицы терминальных символов, относящуюся к операциям сравнения.

Таблица 3.4 – Таблица терминальных символов: операции сравнения

Индекс	**Символ**	**Тип**
5	<	rel
6	<=	rel
7	<>	rel
8	==	rel

Код соответствующей функции представлен ниже.

```
bool RelExpr()
{
     if (! Operand()) return false;
     if (p && p->type==lRel )  {
         ECmd cmd;                          // по индексу в
таблице
         if      (p->index == 5) cmd=CMPL;  // терминальных
символов
         else if (p->index == 6) cmd=CMPLE; // определяем код
         else if (p->index == 7) cmd=CMPNE; // операции
         else if (p->index == 8) cmd=CMPE;

         p=p->next;
         if (! Operand()) return false;
         WriteCmd( cmd );          // заносим операцию в ПОЛИЗ
     }
     return true;
}
```

Формирование ПОЛИЗа в функции, соответствующей правилу

<Operand> → **var** | **const**

выполняется достаточно просто. В зависимости от того, является ли операнд переменной или константой вызывается соответствующая функция.

```
bool Operand()
{
    if ( !p || (p->type != lVar && p->type != lConst) )  {
        Error("Ожидается переменная или константа", p->pos);
        return false;
    }
    if (p->type == lVar) WriteVar(p->index); // тип лексемы –
                                             // переменная
    else WriteConst(p->index);     // тип лексемы - константа
    p=p->next;
    return true;
```

```
}
```

При включении семантики в функцию анализа оператора присваивания в соответствии с правилом

<Statement> → **var as** <ArithExpr>

необходимо учесть, что вначале в ПОЛИЗ заносится переменная, которой присваивается значение, а уже затем, после формирования ПОЛИЗа для выражения <ArithExpr> в ПОЛИЗ заносится команда SET.

```
bool Statement()
{
    if (!p || p->type != lVar)
    { Error("Ожидается переменная", p->pos); return false; }
    WriteVar(p->index);       // заносим в ПОЛИЗ переменную
    p=p->next;
    if (!p || p->type != lAs)
    { Error("Ожидается присваивание", p->pos); return false; }
    p=p->next;
    if (! ArithExpr()) return false;
    // ПОЛИЗ для выражения уже сформирован
    WriteCmd(SET);       // заносим в ПОЛИЗ команду присваивания
    return true;
}
```

Формирование ПОЛИЗа для правила

<ArithExpr> → <Operand> {**ao** <Operand>}

выполняется аналогично рассмотренным выше правилам. Так же как и для <RelExpr> здесь используется определение команды по индексу в таблице терминальных символов.

Таблица 3.5 – Таблица терминальных символов: арифметические операции

Индекс	Символ	Тип
10	+	ao
11	-	ao

bool ArithExpr()

```
{
   if (! Operand()) return false;
    // сформирована часть ПОЛИЗа для операнда
   while (p && p->type == lAo)  {
      ECmd cmd;                          // по индексу в таблице
      if     (p->index == 10) cmd = ADD;          //  терминальных  символов
определяем
      else if (p->index == 11) cmd = SUB;          // код операции
      p=p->next;
      if (! Operand()) return false;
      // сформирована часть ПОЛИЗа для операнда
      WriteCmd(cmd);                          // заносим операцию в ПОЛИЗ
   }
   return true;
}
```

3.4 Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Титульный лист.

2. Задание на лабораторную работу.

3. Описание внутренней формы представления программы и набора операций

4. Описание типов данных, используемых для хранения внутреннего представления программы.

5. Функции и процедуры, в которые осуществлялось включение семантики с подробным описанием.

6. Контрольный пример и результаты тестирования.

7. Листинг программы.

3.5 Контрольные вопросы

1. Что такое внутренняя форма представления программы? Зачем она используется?

2. Какие формы внутреннего представления программ Вы знаете?

3. Что такое трехадресный код? Приведите пример.

4. В чем отличие триад и тетрад? Приведите пример.

5. В чем особенности постфиксной формы записи?

6. Как вычислить выражение в ПОЛИЗе вручную?
7. Как перевести выражение в ПОЛИЗ вручную?
8. Как представить в ПОЛИЗе операции присваивания и обращения по индексу?
9. Как представить в ПОЛИЗе условный оператор?
10. Как представить в ПОЛИЗе операторы цикла?
11. Существуют ли преимущества у постфиксной формы записи перед традиционной?

4 Лабораторная работа №4

Тема лабораторной работы: Создание интерпретатора.

4.1 Теоретические основы лабораторной работы

Транслятором называется программа перевода текста программы с некоторого исходного языка на целевой язык. В случае, если исходным языком является язык высокого уровня, а целевым языком – язык машинных кодов, то такой транслятор называется компилятором.

В отличие от компилятора интерпретатор не переводит текст программы на целевой язык, а непосредственно исполняет программу. При этом чистый интерпретатор анализирует и выполняет инструкции программы последовательно. Эффективность такой схемы, конечно же, невелика, так как повторно выполняемую инструкцию интерпретатор анализирует каждый раз, когда необходимо ее выполнить. По этой причине при построении интерпретаторов часто применяется схема, при которой интерпретаторы производят анализ исходной программы и транслируют ее в некоторую форму внутреннего представления, которая допускает последующее эффективное выполнение. Далее в интерпретаторе внутренняя форма представления программы непосредственно выполняется, в отличие от компилятора, где она используется для генерации объектного кода. Таким образом, можно говорить о том, что начальные фазы компилятора и интерпретатора совпадают.

Конечно, тот факт, что исходная программа анализируется всякий раз, когда должна выполняться, говорит о том, что такой подход менее эффективен, по сравнению с компиляцией. При интерпретации также существуют определенные проблемы, связанные с оптимизацией кода. Кроме того, для выполнения интерпретируемого программного кода у каждого пользователя должен быть установлен соответствующий интерпретатор. Тем не менее, интерпретация обладает по сравнению с компиляцией и рядом преимуществ. Основным ее достоинством является исключительная переносимость программ (или интерпретируемой формы внутреннего представления). Кроме того, исполнение интерпретируемой программы может быть начато даже в том случае, когда весь ее исходный код недоступен, что особенно актуально, когда получение команд происходит в режиме диалога или исходный код поступает по удаленным каналам связи. Следует отметить, что для интерпретируемых языков легко вводятся ограничения по использованию системных программных и аппаратных средств.

В качестве формы представления программы для последующей интерпретации часто выбирается код некоторой условной машины, например виртуальной стековой машины или виртуальной регистровой машины. Могут использоваться и другие формы, например абстрактные синтаксические деревья или машинный код заданной аппаратной платформы (в последнем случае предполагается, что программа исполняется на аппаратной платформе, отличной от заданной). В качестве примеров интерпретируемых языков можно привести:

- Java, Python, PHP, Forth, Tcl, MATLAB (в качестве промежуточной формы представления используется байт-код или p-код),

- Perl, Ruby (используются абстрактные синтаксические деревья).

Отметим также, что с использованием средств так называемой динамической компиляции (just-in-time compilation - JIT) внутренняя форма представления программы может быть оттранслирована в исполняемый код для целевой машины непосредственно во время ее исполнения. Промежуточный код в этом случае компилируется в машинный код во время исполнения программы частями. Такой подход устраняет ряд недостатков классических интерпретаторов за счет кэширования оттранслированного кода. Более того, за счет того, что компиляция производится на той машине, на которой программа будет выполняться, существует возможность выполнить машинно-зависимую оптимизацию в расчете на конкретную ЭВМ. Подход с использованием динамической компиляции применен в платформе .NET Framework (в качестве промежуточной формы представления используется байт-код Common Intermediate Language - CIL) и виртуальной машиной Java (Java Virtual Machine - JVM). Впервые элементы такого подхода были применены при создании языков LISP и Smalltalk.

Отметим, что код виртуальной стековой машины используется в качестве внутренней формы представления достаточно часто (например, байт-коды JVM и CIL). Одним из наиболее известных способов внутреннего представления для выполнения стековой машиной является польская инверсная запись (ПОЛИЗ), описание которой было дано ранее.

Пусть набор команд стековой машины включает в себя команды, представленные в таблице 4.1. В этом случае алгоритм интерпретации ПОЛИЗа достаточно прост и состоит в следующем:

1. Читаем очередной символ ПОЛИЗа

2. Если считанный символ является операндом (идентификатором или константой), то заносим соответствующее им значение в стек.

3. Если считанный символ является командой, то выполняем действия в соответствие с приведенной таблицей 4.1.

4. Осуществляем переход к следующему символу ПОЛИЗа.

В том случае, когда реализация интерпретатора производится для конкретного языка внутренней формы представления программы, наиболее простым решением будет выбрать такой набор команд интерпретатора, чтобы он совпадал с набором операций языка. Реализовать алгоритм интерпретации в этом случае достаточно просто.

Таблица 4.1 – Команды стековой машины

Команда	**Описание**	**Формат**	**Действие**
Команды управления потоком вычислений			
JMP	Команда безусловного перехода	adr JMP	По этой команде из стека извлекается единственный операнд adr и осуществляется переход по указанному адресу
JZ	Команда условного перехода по лжи (если значение первого аргумента ложно)	val, adr JZ	По этой команде из стека извлекаются два операнда: булево значение val и адрес перехода adr. В случае если значение val ложно, осуществляется переход по адресу adr. В противном случае управление переходит по следующему за JZ адресу
Арифметическо-логические команды			
SET	Операция присваивания	var, val SET	По этой команде из стека извлекаются два операнда: переменная var и значение val. Команда записывает значение val в переменную var
ADD	Операция сложения	val1, val2 ADD	По этой команде из стека извлекаются два операнда: val1 и val2. В стек заносится сумма операндов
SUB	Операция вычитания	val1, val2 SUB	По этой команде из стека извлекаются два операнда: val1 и val2. В стек заносится разность операндов
MUL	Операция умножения	val1, val2 MUL	По этой команде из стека извлекаются два операнда: val1 и val2. В стек заносится произведение операндов
DIV	Операция деления	val1, val2 DIV	По этой команде из стека извлекаются

			два операнда: val1 и val2. В стек заносится частное операндов
NOT	Операция логического «не»	val1, val2 NOT	По этой команде из стека извлекаются два операнда: val1 и val2. В стек заносится результат логического отрицания
AND	Операция логического «и»	val1, val2 AND	По этой команде из стека извлекаются два операнда: val1 и val2. В стек заносится результат логического «и»
OR	Операция логического «или»	val1, val2 OR	По этой команде из стека извлекаются два операнда: val1 и val2. В стек заносится результат логического «или»
CMPE	Операция сравнения на равенство	val1, val2 CMPE	По этой команде из стека извлекаются два операнда: val1 и val2. В стек заносится результат выполнения операции сравнения val1=val2
CMPNE	Операция сравнения на неравенство	val1, val2 CMPNE	По этой команде из стека извлекаются два операнда: val1 и val2. В стек заносится результат выполнения операции сравнения val1<>val2
CMPL	Операция сравнения «строго меньше»	val1, val2 CMPL	По этой команде из стека извлекаются два операнда: val1 и val2. В стек заносится результат выполнения операции сравнения val1<val2
CMPLE	Операция сравнения «меньше или равно»	val1, val2 CMPLE	По этой команде из стека извлекаются два операнда: val1 и val2. В стек заносится результат выполнения операции сравнения val1<=val2
CMPG	Операция сравнения «строго больше»	val1, val2 CMPG	По этой команде из стека извлекаются два операнда: val1 и val2. В стек заносится результат выполнения операции сравнения val1>val2
CMPGE	Операция сравнения	val1, val2 CMPGE	По этой команде из стека извлекаются два операнда: val1 и val2. В стек

	«больше или равно»		заносится результат выполнения операции сравнения val1>=val2
Команды ввода-вывода			
INP	Ввод значения из входного потока	var INP	По этой команде из стека извлекается переменная var. Команда записывает считанное из входного потока значение в переменную var
OUT	Вывод значения во входной поток	val OUT	По этой команде из стека извлекается операнд val. Команда записывает в выходной поток значение операнда val

4.2 Задание на лабораторную работу

Реализовать интерпретатор польской инверсной записи. Дополнить анализатор, разработанный в рамках лабораторных работ №1-3 реализованным интерпретатором.

4.3 Пример выполнения лабораторной работы

Как было сказано выше, интерпретация польской инверсной записи может быть выполнена с использованием стека. При этом стек может быть организован как в виде массива, так и в виде динамической структуры. Предположим, что за работу со стеком отвечают следующие функции.

Таблица 4.2 – Функции работы со стеком

Функция	**Назначение**
int PopVal();	Извлечь значение константы или переменной из вершины стека
void PushVal(int val);	Поместить значение в вершину стека
void PushElm(PostfixEntry &elm);	Поместить элемент ПОЛИЗа (переменную, константу или адрес) в вершину стека
void SetVarAndPop(int val);	Установить значение переменной, лежащей в вершине стека и извлечь ее

С использованием представленных выше функций процедура интерпретации может выглядеть следующим образом.

```
void Interpret()
{
    int tmp;
```

```
while (pos < postfixPos) {
    if ( postfix[pos].type == etCmd ) {
        ECmd cmd = postfix[pos].index;
        switch (cmd)   {
            case JMP:   pos = PopVal();
                        break;
            case JZ:    tmp = PopVal();
                        if (PopVal()) pos++; else pos = tmp;
                        break;
            case SET:   SetVarAndPop( PopVal() );
                        pos++;
                        break;
            case ADD:   PushVal( PopVal() + PopVal());
                        pos++;
                        break;
            case SUB:   PushVal( -PopVal() + PopVal() );
                        pos++;
                        break;
            case AND:   PushVal( PopVal() && PopVal() );
                        pos++;
                        break;
            case OR:    PushVal( PopVal() || PopVal() );
                        pos++;
                        break;
            case CMPE:  PushVal( PopVal() == PopVal() );
                        pos++;
                        break;
            case CMPNE: PushVal( PopVal() != PopVal());
                        pos++;
                        break;
            case CMPL:  PushVal( PopVal() > PopVal());
                        pos++;
                        break;
            case CMPLE: PushVal( PopVal() >= PopVal() );
                        pos++;
                        break;
```

```
            }
        }
        else PushElm(postfix[pos++]);
    }
}
```

Естественно, что для работы интерпретатора будут необходимы значения переменных, фигурирующих в строке ПОЛИЗа. Для хранения этих значений можно как выделить отдельный массив, так и хранить эти значения в полях таблицы переменных. Следует также предусмотреть механизм инициализации этих переменных. В рамках лабораторной работы такая инициализация может производиться прямо во время интерпретации при обнаружении неинициализированной переменной (значение переменной запрашивается у пользователя), либо непосредственно перед запуском интерпретатора (значения переменных могут считываться из файла или также запрашиваться у пользователя).

Приведем пример работы интерпретатора для следующего простого примера:

```
while a<3 do a=a+2 end
```

Таблица 4.3 – Пример работы интерпретатора

Шаг	**Инструкция**	**Стек**	**Значения переменных**
0	**a** 3 CMPL 12 JZ a a 2 ADD SET 0 JMP		a=0
1	a **3** CMPL 12 JZ a a 2 ADD SET 0 JMP	a	a=0
2	a 3 **CMPL** 12 JZ a a 2 ADD SET 0 JMP	a 3	a=0
3	a 3 CMPL **12** JZ a a 2 ADD SET 0 JMP	1	a=0
4	a 3 CMPL 12 **JZ** a a 2 ADD SET 0 JMP	1 12	a=0
5	a 3 CMPL 12 JZ **a** a 2 ADD SET 0 JMP		a=0
6	a 3 CMPL 12 JZ a **a** 2 ADD SET 0 JMP	a	a=0
7	a 3 CMPL 12 JZ a a **2** ADD SET 0 JMP	a a	a=0
8	a 3 CMPL 12 JZ a a 2 **ADD** SET 0 JMP	a a 2	a=0
9	a 3 CMPL 12 JZ a a 2 ADD **SET** 0 JMP	a 2	a=0
10	a 3 CMPL 12 JZ a a 2 ADD SET **0** JMP		a=2
11	a 3 CMPL 12 JZ a a 2 ADD SET 0 **JMP**	0	a=2

12	**a** 3 CMPL 12 JZ a a 2 ADD SET 0 JMP		a=2
13	a **3** CMPL 12 JZ a a 2 ADD SET 0 JMP	a	a=2
14	a 3 **CMPL** 12 JZ a a 2 ADD SET 0 JMP	a 3	a=2
15	a 3 CMPL **12** JZ a a 2 ADD SET 0 JMP	1	a=2
16	a 3 CMPL 12 **JZ** a a 2 ADD SET 0 JMP	1 12	a=2
17	a 3 CMPL 12 JZ **a** a 2 ADD SET 0 JMP		a=2
18	a 3 CMPL 12 JZ a **a** 2 ADD SET 0 JMP	a	a=2
19	a 3 CMPL 12 JZ a a **2** ADD SET 0 JMP	a a	a=2
20	a 3 CMPL 12 JZ a a 2 **ADD** SET 0 JMP	a a 2	a=2
21	a 3 CMPL 12 JZ a a 2 ADD **SET** 0 JMP	a 2	a=2
22	a 3 CMPL 12 JZ a a 2 ADD SET **0** JMP		a=4
23	a 3 CMPL 12 JZ a a 2 ADD SET 0 **JMP**	0	a=4
24	**a** 3 CMPL 12 JZ a a 2 ADD SET 0 JMP		a=4
25	a **3** CMPL 12 JZ a a 2 ADD SET 0 JMP	a	a=4
26	a 3 **CMPL** 12 JZ a a 2 ADD SET 0 JMP	a 3	a=4
27	a 3 CMPL **12** JZ a a 2 ADD SET 0 JMP	0	a=4
28	a 3 CMPL 12 **JZ** a a 2 ADD SET 0 JMP	0 12	a=4
цикл завершен			a=4

4.4 Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Титульный лист.

2. Задание на лабораторную работу.

3. Описание внутренней формы представления программы.

4. Описание основных алгоритмов и структур данных, используемых в программе:

а) описание алгоритма интерпретации;

б) описание структур данных, используемых при интерпретации;

в) описание процедур и функций, отвечающих за выполнение команд интерпретатора и работу со структурами данных интерпретатора.

5. Описание интерфейса пользователя.

6. Контрольный пример и результаты тестирования.

7. Листинг программы.

4.5 Контрольные вопросы

1. Дайте определения понятиям транслятор, компилятор, интерпретатор.

2. В чем преимущества и недостатки интерпретаторов?

3. Какие внутренние формы представления могут использоваться при интерпретации?

4. Что такое динамическая компиляция?

5. В чем заключается алгоритм интерпретации польской инверсной записи с использованием стека?

6. Расскажите об особенностях реализации команд при создании интерпретатора.

5 Варианты заданий

Номер задания на лабораторную работу выдается преподавателем. Содержание варианта определяется из таблицы 5.1.

Таблица 5.1 - Варианты заданий

Номер варианта	Номер конструкции	Уровень сложности	Чувствительность к регистру для ключевых слов и идентификаторов
1	1	1	Да
2	2	1	Нет
3	3	1	Да
4	4	1	Нет
5	5	1	Да
6	6	1	Нет
7	7	1	Да
8	8	1	Нет
9	1	2	Да
10	2	2	Нет
11	3	2	Да
12	4	2	Нет
13	5	2	Да
14	6	2	Нет
15	7	2	Да
16	8	2	Нет
17	1	3	Да
18	2	3	Нет
19	3	3	Да
20	4	3	Нет
21	5	3	Да
22	6	3	Нет
23	7	3	Да
24	8	3	Нет
25	1	4	Да
26	2	4	Нет
27	3	4	Да
28	4	4	Нет
29	5	4	Да
30	6	4	Нет
31	7	4	Да
32	8	4	Нет

После получения варианта примеры тестовых цепочек разрабатываются студентом самостоятельно.

Во всех приведенных вариантах идентификатор начинается с буквы, за которой может следовать произвольное количество букв или цифр (до 255 символов), константа - целочисленная константа, возможно со знаком, в диапазоне от -32768 до +32767.

Ключевое слово output представляет собой инструкцию, по которой содержимое следующей за ней переменной выводится пользователю на экран. Ключевое слово input представляет собой инструкцию, по которой в следующую за ней переменную загружается введенное пользователем с клавиатуры значение.

Перечень конструкций

Конструкция 1 - оператор цикла без счетчика с предусловием.

```
do while <логическое выражение>
    <операторы>
loop
```

Оператор цикла повторяет <операторы> пока условие <логическое выражение> истинно.

Конструкция 2 - оператор цикла без счетчика с предусловием.

```
do until <логическое выражение>
    <операторы>
loop
```

Оператор цикла повторяет <операторы>, до тех пор, пока условие <логическое выражение> не станет истинным.

Конструкция 3 - оператор цикла без счетчика с постусловием.

```
do
    <операторы>
loop while <логическое выражение>
```

Оператор цикла повторяет <операторы> пока условие <логическое выражение> истинно. Цикл выполняется хотя бы один раз.

Конструкция 4 - оператор цикла без счетчика с постусловием.

```
do
    <операторы>
loop until <логическое выражение>
```

Оператор цикла повторяет <операторы>, до тех пор, пока условие <логическое выражение> не станет истинным. Цикл выполняется хотя бы один раз.

Конструкция 5 - оператор цикла со счетчиком.

```
for <идентификатор> = <арифметическое выражение> to
                      <арифметическое выражение>
    <операторы>
next
```

Выполняет <операторы> заданное число раз.

Конструкция 6 - условный оператор.

```
if <логическое выражение> then
    <операторы>
[ else
    <операторы> ]
end
```

Конструкция 7 - условный оператор.

```
if <логическое выражение> then
    <операторы>
[ elseif <логическое выражение> then
    <операторы> ]
…
[ else
    <операторы> ]
end
```

Конструкция 8 - оператор ветвления.

```
select <арифметическое выражение>
    case <константа> <операторы>
    [case <константа> <операторы>]
    …
    [default <операторы>]
end
```

Уровни сложности заданий

Уровень 1

<логическое выражение> → <операнд> | <операнд><операция сравнения><операнд>

<операция сравнения> → < | > | <= | >= | = | <>

<операнд> → <идентификатор>|<константа>

<операторы> → <идентификатор> = <арифметическое выражение>

<арифметическое выражение> → <операнд> |

<операнд><арифметическая операция><арифметическое выражение>

<арифметическая операция> → +|-

Уровень 2

<логическое выражение> → <выражение сравнения> |

<логическое выражение><логическая операция><выражение сравнения>

<выражение сравнения> → <операнд> | <операнд><операция сравнения><операнд>

<операция сравнения> → <|>|=|<>

<логическая операция> → **and**|**or**

<операнд> → <идентификатор>|<константа>

<операторы> → <идентификатор> = <арифметическое выражение>

<арифметическое выражение> → <операнд> |

<арифметическое выражение><арифметическая операция><операнд>

<арифметическая операция> → + | - | / | *

Уровень 3

<логическое выражение> → <выражение сравнения> **|**

<унарная логическая операция><выражение сравнения> |

<логическое выражение><бинарная логическая операция><выражение сравнения>

<выражение сравнения> → <операнд> | <операнд><операция сравнения><операнд>

<операция сравнения> → < | > | = | <>

<унарная логическая операция> → **not**

<бинарная логическая операция> → **and**|**or**

<операнд> → <идентификатор>|<константа>

<операторы> → <операторы>; <оператор> | <оператор>

<оператор> → <идентификатор> = <арифметическое выражение> |

output <операнд>

<арифметическое выражение> → <операнд> |

<операнд ><арифметическая операция><арифметическое выражение>

<арифметическая операция> → + | - | / | *

Уровень 4

<логическое выражение> → <выражение сравнения> **|**

<логическое выражение><бинарная логическая операция><выражение сравнения>

<выражение сравнения> → <операнд> | <операнд><операция сравнения><операнд>

<операция сравнения> → < | > | = | <>

<бинарная логическая операция> → **and**|**or**

<операнд> → <идентификатор>|<константа>

<операторы> → <операторы>; <оператор> | <оператор>

<оператор> → <инструкция> |

<идентификатор> = <арифметическое выражение> |

input <идентификатор> |

output <операнд>

<арифметическое выражение> → <операнд> |

<арифметическое выражение><арифметическая операция><операнд> |

(<арифметическое выражение>)

<арифметическая операция> → + | - | / | *

6 Библиографический список

1. Компиляторы: принципы, технология / А. Ахо, Р. Сети, Дж. Ульман. - М.: Вильямс, 2003. - 767 с.

2. Основные структуры данных и алгоритмы компиляции / М. А. Шамашов. - Самара : СМС, 1999. - 115 с.

3. Теория конечных автоматов и формальных языков / Е. И. Чигарина, М. А. Шамашов. Федер. агентство по образованию, Самар. гос. аэрокосм. ун-т им. С. П. Королева. - Самара : Изд-во СГАУ, 2007. - 95 с.

4. Теория и технология программирования. Основы построения трансляторов / Ю. Г. Карпов. - СПб. : БХВ-Петербург, 2005. - 270 с.

7 Справочная информация

7.1 Стандартный заголовочный файл <ctype.h>

Стандартный заголовочный файл <ctype.h> содержит функции преобразования и классификации символов. Все функции приниают на вход аргумент типа int (аргумент должен быть представим в формате unsigned char или быть макросом EOF). Поведение ряда функций зависит от конкретной культурной среды. Ниже следует краткое описание функций.

Таблица 7.1 - Функции преобразования и классификации символов

Функция	**Описание**
int isalnum(int c);	является ли символ алфавитно-цифровым;
int isalpha(int c);	является ли символ алфавитным (буквой);
int ischtrl (int c);	является ли символ управляющим;
int isdigit(int c);	является ли символ цифрой;
int isgraph(int c);	является ли символ графическим;
int islower(int c);	является ли символ строчным (в нижнем регистре);
int isprint(int c);	является ли символ печатным;
int ispunct(int c);	является ли символ знаком пунктуации;
int isspace(int c);	является ли символ пробелом;
int isupper(int c);	является ли символ прописным/в верхнем регистре;
int isxdigit(int c);	является ли символ цифрой 16-ти разрядной системы счисления;
int tolower(int c);	перевод в строчный;
int toupper (int c);	перевод в прописной.