Министерство образования
Республики Беларусь

Учреждение образования
«Гомельский государственный университет
имени Франциска Скорины»

В. А. КОРОТКЕВИЧ, Л. И. КОРОТКЕВИЧ

ЭВМ И ПРОГРАММИРОВАНИЕ

ПРАКТИЧЕСКОЕ РУКОВОДСТВО
по изучению темы «Методы разработки компиляторов для языков программирования» для студентов специальности 1-31 03 03-01 «Прикладная математика (научно-производственная деятельность»

Гомель
УО «ГГУ им. Ф. Скорины»
2009

УДК 004.42 (075.8)
ББК 32.973.26 – 018я73
К687

Рецензент:
кафедра математических проблем управления
учреждения образования «Гомельский государственный
университет имени Франциска Скорины».

Рекомендовано к изданию научно-методическим советом
учреждения образования «Гомельский государственный
университет имени Франциска Скорины»

Короткевич, В.А.

К687 ЭВМ и программирование: практическое руководство по изучению темы «Методы разработки компиляторов для языков программирования» для студентов специальности 1-31 03 03-01 «Прикладная математика (научно-производственная деятельность)» / В. А. Короткевич, Л. И. Короткевич; М-во образ. РБ, Гомельский государственный университет им. Ф. Скорины. - Гомель: ГГУ им. Ф. Скорины, 2009. – 83 с.

Целью практического руководства по изучению темы «Методы разработки компиляторов для языков программирования» дисциплины «ЭВМ и программирование» является оказание помощи студентам в усвоении учебного материала. Практическое руководство содержит теоретическую и практическую часть – лабораторные работы и адресовано студентам специальности 1-31 03 03-01 «Прикладная математика (научно-производственная деятельность)».

УДК 004.42 (075.8)
ББК 32.973.26 – 018я73

СОДЕРЖАНИЕ

Введение

Компиляторы с языков программирования являются неотъемлемой частью инструментальных систем разработки программного обеспечения. Знания о принципах работы компиляторов позволяют ускорить и упростить процесс отладки программ. Методы и алгоритмы синтаксического анализа программ могут быть использованы для построения эффективных алгоритмов анализа текстовой информации в самых различных приложениях.

Целью практического руководства по изучению темы «Методы разработки компиляторов для языков программирования» дисциплины «ЭВМ и программирование» является оказание помощи студентам в усвоении учебного материала. Практическое руководство содержит теоретические разделы и практическую часть – лабораторные работы по темам. Практическое руководство адресовано студентам специальности 1-31 03 03-01 «Прикладная математика (научно-производственная деятельность)».

1 Построение формальных грамматик

1.1 Понятие формальной грамматики

Любой язык можно неформально определить как подмножество множества всех предложений из слов или символов некоторого основного словаря. Слова объединяются в предложения, но только очень малая часть из всех возможных сочетаний слов является допустимой с точки зрения синтаксиса языка.

Язык программирования состоит из программ, которые являются последовательностями операторов, состоящих из служебных слов (if, else и т.п.), знаков пунктуации, идентификаторов и констант. Одной из основных задач компилятора является синтаксический анализ исходной программы, написанной на каком-либо языке программирования. Реализация этой функции в компиляторах основана на теории формальных грамматик и языков.

Перейдем к формальному описанию языка.

Алфавит – это непустое конечное множество элементов, называемых символами.

Всякая последовательность символов алфавита называется цепочкой или строкой. Если A = {a, b, c} – алфавит языка, то цепочками являются a, abc, bbaa. Пустую цепочку, не содержащую ни одного символа, будем обозначать λ.

Правилом подстановки называется упорядоченная пара (U, x), где U – символ, x – конечная цепочка символов. U называется левой частью, x – правой частью правила подстановки.

Грамматикой G[Z] называется конечное, непустое множество правил. Z – это символ, который должен встретиться в левой части хотя бы одного правила. Он называется начальным символом грамматики.

Все символы, которые встречаются в левых и правых частях правил образуют словарь грамматики – V.

Множество всех конечных цепочек из V, включая λ, обозначается V^*.

При написании грамматик будет использоваться некоторая нотация, называемая бэкусовой нормальной формой, в которой правила подстановки имеют вид U → x, а правила с одинаковыми левыми частями собираются в одно, с использованием в качестве разделителя правых частей знака | (или). Например, правила U → x, U → y записываются в виде U → x | y.

В заданной грамматике G символы, которые встречаются в левой части правил, называются нетерминальными. Они образуют множество нетерминальных символов VN. Остальные символы называются терминальными и образуют множество VT (V=VN U VT). Нетерминальные символы в дальнейшем будут обозначаться большими латинскими буквами или заключаться в скобки <>.

Пример грамматики:

<предложение> → <подлежащее> <сказуемое> <обстоятельство>

<подлежащее> → ОН | ОНА

<сказуемое> → ИДЕТ

<обстоятельство> → В_КИНО | НА_ЛЕКЦИЮ

Здесь

VN={<предложение>,<подлежащее>,<сказуемое>,<обстоятельство>}
VT={ОН,ОНА,ИДЕТ,В_КИНО,НА_ЛЕКЦИЮ},

V=VN U VT,

Z=<предложение>.

1.2 Понятие формального языка

Формальная грамматика порождает формальный язык, состоящий из предложений, удовлетворяющий правилам грамматики. Дадим определение такого языка.

Цепочка x прямо порождает цепочку y ($x \Rightarrow y$), если x=pUq, y=pwq, где p, w, q принадлежат V^*, и существует правило U → w. Т.е. y –

прямое порождение x, если y можно получить из x заменой в x нетерминального символа U на строку w по правилу U → w.

Для примера:

ИДЕТ <обстоятельство> $\Rightarrow$ ИДЕТ В_КИНО

по правилу:

<обстоятельство> → В_КИНО.

Строка x порождает строку y ($x \Rightarrow^{+} y$), если существует последовательность строк $x = x_0, x_1, \ldots, x_n = y$, таких, что $x_i \Rightarrow x_{i+1}$ для i = 0, …, n-1.

Любое порождение начального символа Z грамматики G называется сентенциальной формой. Язык L(G), порождаемый грамматикой G, есть множество всех сентенциальных форм, состоящих только из терминальных символов.

Для заданного примера:

<предложение> $\Rightarrow$ <подлежащее> <сказуемое> <обстоятельство>
$\Rightarrow$ ОНА <сказуемое> <обстоятельство> $\Rightarrow$
ОНА ИДЕТ <обстоятельство> $\Rightarrow$ ОНА ИДЕТ В_КИНО

Язык, порождаемый этой грамматикой, состоит из 4-х предложений:

ОН ИДЕТ В_КИНО, ОН ИДЕТ НА_ЛЕКЦИЮ,
ОНА ИДЕТ В_КИНО, ОНА ИДЕТ НА_ЛЕКЦИЮ

Другой пример грамматики:

<целое> → <цифра> | <целое> <цифра>
<цифра> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Язык, порождаемый этой грамматикой, состоит из бесконечного количества предложений – целых неотрицательных чисел. Пример вывода предложения:

<целое> $\Rightarrow$ <целое><цифра> $\Rightarrow$ <целое>1 $\Rightarrow$
<целое><цифра>1 $\Rightarrow$ <целое>41 $\Rightarrow$ <цифра>41 $\Rightarrow$ 841

Лабораторная работа

Цель: получение практических навыков разработки формальных грамматик.

Материалы и оборудование: персональный компьютер.

Задание на лабораторную работу состоит из 3-х задач:

1) для грамматики, порождающей язык из конечного множества предложений, указать множества терминальных и нетерминальных символов, выполнить вывод всех предложений (построить цепочки прямого порождения из начального символа грамматики для каждого предложения языка);

2) для грамматики, порождающей язык из бесконечного множества предложений, указать множества терминальных и нетерминальных символов, описать формат предложений языка;

3) для заданного формата предложений языка построить грамматику.

При описании грамматик и формата предложений языка использованы следующие обозначения и конструкции:

`ц` – терминальный символ «цифра»;

`ид` – терминальный символ «идентификатор»;

`{a|b}` – в предложении обязательно присутствует символ `a` или символ `b`;

`[x]` – в предложении может присутствовать цепочка символов `x`;

`x...` или `x...x` – цепочка символов `x` повторяется произвольное не нулевое количество раз.

Вариант 1

1. Для грамматики

$S \rightarrow ABC$

$A \rightarrow aa \mid C$

$B \rightarrow b \mid bb$

$C \rightarrow cc$

указать множества терминальных и нетерминальных символов, выполнить вывод всех предложений порождаемого языка.

2. Для грамматики

S → +T | -T

T → (T) | a

указать множества терминальных и нетерминальных символов, описать формат предложений языка.

3. Язык содержит предложения следующего формата:

ц | b[a...a]d

Примеры корректных предложений: ц, bd, bad, baaad.
Для заданного языка построить формальную грамматику.

Вариант 2

1. Для грамматики

S → aX | Yd

X → bcb

Y → cda

указать множества терминальных и нетерминальных символов, выполнить вывод всех предложений порождаемого языка.

2. Для грамматики

S → (P+ид)

P → ид | (P+ид)

указать множества терминальных и нетерминальных символов, описать формат предложений языка.

3. Язык содержит предложения следующего формата:

kk[d...d]ac

Примеры корректных предложений: kkac, kkdac, kkddddac.
Для заданного языка построить формальную грамматику.

Вариант 3

1. Для грамматики

S → XaaY

X → (a+a) | (a-a)

Y → cc | d

указать множества терминальных и нетерминальных символов, выполнить вывод всех предложений порождаемого языка.

2. Для грамматики

X → abY

Y → cY | de

указать множества терминальных и нетерминальных символов, описать формат предложений языка.

3. Язык содержит предложения следующего формата:

(...(ц)...)[(...(ц)...)]

Примеры корректных предложений: (ц), ((ц)), (ц)(((ц))).
Для заданного языка построить формальную грамматику.

Вариант 4

1. Для грамматики

S → A | AB

A → aa | a

B → Ad

указать множества терминальных и нетерминальных символов, выполнить вывод всех предложений порождаемого языка.

2. Для грамматики

X → AbC

A → nn | nnA

C → Cq | q

указать множества терминальных и нетерминальных символов, описать формат предложений языка.

3. Язык содержит предложения следующего формата:

a[{+|*}a ...]

Примеры корректных предложений: a, a+a, a*a*a+a.
Для заданного языка построить формальную грамматику.

Вариант 5

1. Для грамматики

A → Xabc | Z

X → (Z) | Z+

Z → abc

указать множества терминальных и нетерминальных символов, выполнить вывод всех предложений порождаемого языка.

2. Для грамматики

```
S → aX
X → b | cX
```

указать множества терминальных и нетерминальных символов, описать формат предложений языка.

3. Язык содержит предложения следующего формата:

```
{*|+}[(...(]ц[)...)]
```

Примеры корректных предложений: `+ц, *(ц), +(((ц)))`.
Для заданного языка построить формальную грамматику.

Вариант 6

1. Для грамматики

```
S → xBx
A → (c) | c
B → A | (A)
```

указать множества терминальных и нетерминальных символов, выполнить вывод всех предложений порождаемого языка.

2. Для грамматики

```
S → a | aBa
B → q | qB
```

указать множества терминальных и нетерминальных символов, описать формат предложений языка.

3. Язык содержит предложения следующего формата:

```
(ccd[,ccd...])
```

Примеры корректных предложений: `(ccd), (ccd,ccd,ccd)`.
Для заданного языка построить формальную грамматику.

Вариант 7

1. Для грамматики

```
Z → Ац | цB
A → (ид) | ид
B → ид
```

указать множества терминальных и нетерминальных символов, выполнить вывод всех предложений порождаемого языка.

2. Для грамматики

S → a | Tb
T → c | Td

указать множества терминальных и нетерминальных символов, описать формат предложений языка.

3. Язык содержит предложения следующего формата:

(...(ab+ab)+ab)+...ab)

Примеры корректных предложений: (ab+ab), ((ab+ab)+ab).
Для заданного языка построить формальную грамматику.

Вариант 8

1. Для грамматики

S → D:E | E
D → a
E → D | b

указать множества терминальных и нетерминальных символов, выполнить вывод всех предложений порождаемого языка.

2. Для грамматики

S → C | S+C
C → ц | Сц

указать множества терминальных и нетерминальных символов, описать формат предложений языка.

3. Язык содержит предложения следующего формата:

{?|!}({d|a})d...d

Примеры корректных предложений: !(a)d, ?(a)dd, ?(d)ddd.
Для заданного языка построить формальную грамматику.

Вариант 9

1. Для грамматики

A → X+Y
X → ц | (ц+ц)
Y → (ид) | (ид+ид)

указать множества терминальных и нетерминальных символов, выполнить вывод всех предложений порождаемого языка.

2. Для грамматики

```
S → T | T+S | T*S
T → ц
```

указать множества терминальных и нетерминальных символов, описать формат предложений языка.

3. Язык содержит предложения следующего формата:

```
!...!+...+[*!...!+...+[*...]]
```

Примеры корректных предложений: `!+, !!+*!+, !++*!+++*!+`. Для заданного языка построить формальную грамматику.

Вариант 10

1. Для грамматики

```
S → ABC*
A → a | B
B → x
C → c
```

указать множества терминальных и нетерминальных символов, выполнить вывод всех предложений порождаемого языка.

2. Для грамматики

```
S → T
T → ид | (S)
```

указать множества терминальных и нетерминальных символов, описать формат предложений языка.

3. Язык содержит предложения следующего формата:

```
b...b[+b...b[+...]]
```

Примеры корректных предложений: `b, bb, bbb+b, bb+b+bbb`. Для заданного языка построить формальную грамматику.

Вариант 11

1. Для грамматики

```
S → b | bTTd
T → bc | a
```

указать множества терминальных и нетерминальных символов, выполнить вывод всех предложений порождаемого языка.

2. Для грамматики

```
S → P | P,S
P → AB
A → * | *A
B → / | B/
```

указать множества терминальных и нетерминальных символов, описать формат предложений языка.

3. Язык содержит предложения следующего формата:

```
ц[a...a][+b...+b]
```

Примеры корректных предложений: `4, 3aa, 2+b+b, 6aaa+b`.
Для заданного языка построить формальную грамматику.

Вариант 12

1. Для грамматики

```
S → D | (D) | ADB
D → AB
A → +
B → -
```

указать множества терминальных и нетерминальных символов, выполнить вывод всех предложений порождаемого языка.

2. Для грамматики

```
Z → (A)
A → Ац | ц
```

указать множества терминальных и нетерминальных символов, описать формат предложений языка.

3. Язык содержит предложения следующего формата:

```
s | sb...bs
```

Примеры корректных предложений: `s, sbs, sbbbs`.
Для заданного языка построить формальную грамматику.

Вариант 13

1. Для грамматики

```
S → aB | Ab
A → c | Bc
B → a
```

указать множества терминальных и нетерминальных символов, выполнить вывод всех предложений порождаемого языка.

2. Для грамматики

```
S → A | SA
A → (a) | (A)
```

указать множества терминальных и нетерминальных символов, описать формат предложений языка.

3. Язык содержит предложения следующего формата:

```
n[q...q]m
```

Примеры корректных предложений: `nm`, `nqm`, `nqqqm`.
Для заданного языка построить формальную грамматику.

Вариант 14

1. Для грамматики

```
S → B*B | A
B → A | (A)
A → ц
```

указать множества терминальных и нетерминальных символов, выполнить вывод всех предложений порождаемого языка.

2. Для грамматики

```
S → (T)
T → abb | T,abb
```

указать множества терминальных и нетерминальных символов, описать формат предложений языка.

3. Язык содержит предложения следующего формата:

```
(n...n)
```

Примеры корректных предложений: `(n)`, `(nn)`, `(nnnn)`.
Для заданного языка построить формальную грамматику.

Вариант 15

1. Для грамматики

```
S → aABd
A → (B) | c
B → b
```

указать множества терминальных и нетерминальных символов, выполнить вывод всех предложений порождаемого языка.

2. Для грамматики

```
A → B | BC
B → a | Bb
C → +ц | C+ц
```

указать множества терминальных и нетерминальных символов, описать формат предложений языка.

3. Язык содержит предложения следующего формата:

```
a...acbb...bb
```

Примеры корректных предложений: `acbb, aacbbbb, acbbbbbb`. Для заданного языка построить формальную грамматику.

2 Синтаксический анализ предложений для регулярных грамматик

2.1 Понятие регулярной грамматики
2.2 Построение и использование диаграмм состояний
2.3 Перевод лексических единиц во внутримашинное представление

2.1 Понятие регулярной грамматики

Регулярные грамматики содержат правила вида U → t или U → Qt, где U, Q принадлежат VN – множеству нетерминальных символов, t принадлежит VT – множеству терминалов или t = λ – пустой цепочке.

Пример регулярной грамматики:

```
Z → P | F
P → Рц | F.          Грамматика описывает числовые
F → Fц | Хц          константы вида
X → + | - | λ        [+|-]целое[.[целое]]
```

здесь ц – терминальный символ «цифра».

В данном примере правила

`X → + | - | λ`

относятся к виду U → t,

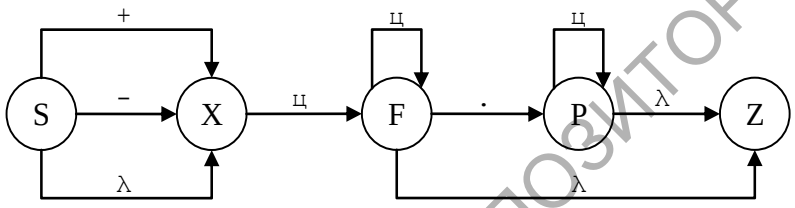
правила

```
Z → P | F
P → Рц | F.
F → Fц | Хц
```

к виду U → Qt.

2.2 Построение и использование диаграмм состояний

По регулярной грамматике может быть построена диаграмма состояний. Это ориентированный граф, в котором каждый нетерминал грамматики представлен узлом, или состоянием; кроме того добавляется узел для начального состояния – S. Каждому правилу Q → t сопоставляется дуга из S в Q, помеченная t. Каждому правилу Q → Rt сопоставляется дуга из R в Q, помеченная t. Диаграмма состояний для приведенного выше примера представлена на рисунке 2.1.



Z – заключительное состояние, соответствующее начальному символу грамматики

Рисунок 2.1 – Пример диаграммы состояний для регулярной грамматики

При наличии в грамматике правила вида Q → R (Q → λ) в диаграмме состояний присутствует ветвь, помеченная λ. Такую диаграмму нужно преобразовать по правилам, применение которых проиллюстрировано на рисунке 2.2:

1) если Q – заключительное состояние, то R также объявляется заключительным состоянием;

2) исключается дуга, помеченная λ, и в добавление к каждой дуге, помеченной t и ведущей из Q к некоторому состоянию P, добавляется дуга из R к P, также помеченная t.

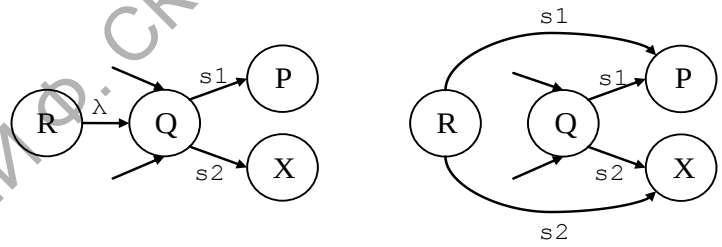


Рисунок 2.2 – Преобразование диаграммы состояний

Результат преобразования диаграммы состояний для грамматики числовых констант представлен на рисунке 2.3.

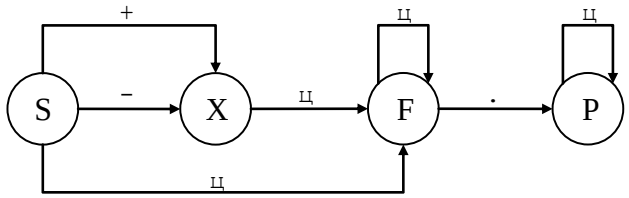


Диаграмма состояний после преобразования.
F, P – заключительные состояния. Состояние Z исключено из диаграммы, т.к. к нему не ведет ни одной дуги.

Рисунок 2.3 – Результат преобразования диаграммы состояний

По диаграмме состояний может быть выполнен грамматический разбор предложения, в ходе которого проверяется его соответствие

правилам грамматики. Грамматический разбор некоторой строки X начинается с состояния S. Для очередного символа строки отыскивается дуга, помеченная этим символом, и выполняется переход в следующее состояние. При отсутствии дуги, X не является правильным предложением языка. По концу строки текущее состояние должно быть заключительным. Примеры грамматического разбора числовых констант представлены в таблице 2.1.

Таблица 2.1 – Примеры грамматического разбора

Состояние	Строка	Состояние	Строка	Состояние	Строка
S	-23.7	S	23..7	S	+
X	23.7	F	3..7	X	λ
F	3.7	F	..7	не заклю-	
F	.7	P	.7	чительное	
P	7	нет дуги		состояние	
P	λ	- ошибка		- ошибка	

В ходе реализации алгоритма грамматического разбора диаграмма представляется в виде матрицы состояний. Элемент матрицы:

a[i][j]=k – номер состояния, в которое осуществляется переход из состояния i по терминалу j;
a[i][j]=0, если дуги из состояния i по терминалу j нет.

Матрица переходов из состояния в состояние для грамматики, заданной в примере. Нумерация состояний: S – 1, X – 2, F – 3, P – 4. Нумерация терминалов: '+' – 1, '-' – 2, 'ц' – 3, '.' – 4.

	+	–	ц	.
S	2	2	3	0
X	0	0	3	0
F	0	0	3	4
P	0	0	4	0

Алгоритм разбора предложений в регулярных грамматиках:

1) i = 1 – номер начального состояния;
2) t = очередной_символ_предложения. Если предложение закончилось, то переход к пункту 5;
3) j = номер_столбца_матрицы_A, соответствующего символу t. Если t не принадлежит VT, то фиксируется ошибка;

4) i = a[i][j] – номер очередного состояния. Если i=0, то фиксируется ошибка. Иначе переход к пункту 2;
5) если i не соответствует заключительному состоянию, то фиксируется ошибка. Иначе, конец разбора, предложение корректно.

2.3 Перевод лексических единиц во внутримашинное представление

Одновременно с грамматическим разбором предложений может быть выполнен перевод содержащихся в них лексических единиц во внутримашинное представление. С этой целью должен быть разработан набор подпрограмм, вызываемых после распознавания очередного символа предложения.

Например, перевод во внутримашинное представление числовых констант, определенных грамматикой, заданной в примере, может быть выполнен путем использования следующих подпрограмм:

```
1. s=-1          s – знак числа – {+|-}1, начальное значение – +1;
2. m=m*10+d      m – текущее значение мантиссы числа (начальное
                     значение – 0);
3. m=m*10+d;     d – значение очередной цифры числа;
   n=n+1         n – количество цифр после точки (начальное
                     значение – 0).
```

Номера вызываемых подпрограмм задаются матрицей B:

	+	–	ц	.
S	0	1	2	0
X	0	0	2	0
F	0	0	2	0
P	0	0	3	0

Элемент матрицы b[i][j] определяет номер подпрограммы, которую нужно вызвать в состоянии i при распознавании очередного терминала j. Вызов выполняется перед изменением состояния в п.4 алгоритма грамматического разбора.

По концу разбора значение константы определяется как
`r = s * m * (10**(-n)).`

Пример разбора константы `-23.75` представлен в таблице 2.2.

Таблица 2.2 – Пример разбора числовой константы

Состояние	Символ	Подпрограмма	Результат	Новое состояние
S (1)	- (2)	1	s=-1	X (2)
X (2)	2 (3)	2	m=2	F (3)
F (3)	3 (3)	2	m=23	F (3)
F (3)	. (4)	-		P (4)
P (4)	7 (3)	3	m=237; n=1	P (4)
P (4)	5 (3)	3	m=2375; n=2	P (4)

Значение константы `r=(-1)*2375*(10**(-2))=-23.75`.

Лабораторная работа

Цель: получение практических навыков в использовании методов синтаксического анализа предложений для регулярных грамматик.

Материалы и оборудование: персональный компьютер.

Для заданной регулярной грамматики написать и отладить программу синтаксического анализа предложений порождаемого языка. В ходе выполнения задания:

1) построить диаграмму состояний;

2) разработать подпрограммы, обеспечивающие выделение лексических единиц из входного предложения;

3) построить матрицы смен состояний и вызова подпрограмм;

4) в ходе тестирования программы использовать как синтаксически корректные, так и некорректные тестовые примеры;

5) обеспечить выдачу сообщений о различных видах синтаксических ошибок во входных предложениях.

Каждый приведенный далее вариант условия задачи содержит описание грамматики, описание формата предложений порождаемого языка и задание на обработку лексических единиц во входных предложениях.

Обозначения терминальных символов при написании грамматик:

ц – терминальный символ «цифра»;
б – терминальный символ «буква»;
с – терминальный символ «символ» (буква или цифра).

Вариант 1

```
Z → Тц | Zц      Грамматика описывает предложения вида:
T → P+ | P-      <идентификатор>=<идентификатор>{+|-}<целое>
P → Rб | Рб | Рц
R → Q=
Q → б | Qб | Qц
```

В ходе разбора предложения должны быть получены две переменные с идентификаторами и значение константы (возможно отрицательное). Например, для предложения `a11=b-5` по концу разбора должно быть получено `p1=a11`, `p2=b`, `k=-5`.

Вариант 2

```
Z → P)                Грамматика описывает предложения вида:
P → Qц | Рц           (<целое>,...,<целое>)
Q → ( | R
R → P,
```

В ходе разбора предложения заданными целыми числами должен быть заполнен массив `M`. Например, для предложения `(1,10,15)` по концу разбора должно быть получено `M(1)=1`, `M(2)=10`, `M(3)=15`.

Вариант 3

```
Z → Q;                Грамматика описывает предложения вида:
P → Q,                <идентификатор>,...,<идентификатор>;
Q → б | Рб | Qб | Qц
```

В ходе разбора предложения заданными идентификаторами должен быть заполнен массив `S`. Например, для предложения `aaa,bb,d7;` по концу разбора должно быть получено `S(1)=aaa`, `S(2)=bb`, `S(3)=d7`.

Вариант 4

```
Z → Zц | Pц          Грамматика описывает предложения вида:
P → ц | Qц           <число>[-<число>...]
Q → Z-               для чисел, содержащих не менее двух цифр
```

В ходе разбора предложения должны быть подсчитаны количество цифр в каждом числе и сумма этих цифр. Например, для предложения `14-0999-532` должно быть получено `n(1)=2, n(2)=4, n(3)=3` и `m(1)=5, m(2)=27, m(3) = 10`.

Вариант 5

```
Z → Q | P | B        Грамматика описывает предложения вида:
Q → ц | Qц           <часы>[.<минуты>[.[<секунды>]]]
T → Q.
P → Tц | Pц
B → P. | Bц
```

В ходе разбора предложения необходимо вычислить значение времени дня в секундах. Например, для предложения `12.55` по концу разбора должно быть получено `t=46500`.

Вариант 6

```
Z → P | A            Грамматика описывает предложения вида:
T → ц | Tц           <число>{+|-}<число>[.<число>]
B → T+ | T-          причем число после точки должно
P → Bц | Pц          содержать не менее двух цифр
C → P.
D → Cц
A → Dц | Aц
```

В ходе разбора предложения должны быть получены значение всего выражения и число знаков после точки. Например, для предложения `10-5.50` по концу разбора должно быть получено `m=4.50` и `n=2`.

Вариант 7

```
Z → Zц | Pц | ц      Грамматика описывает предложения вида:
P → Z- | Z+                    <число>[{+|-}<число>...]
```

В ходе разбора предложения должно быть выполнено заполнение массива M заданными числами и подсчитано значение выражения. Например, для предложения 6+10-12 по концу разбора должно быть получено M(1)=6, M(2)=10, M(3)=12, v=4.

Вариант 8

```
Z → T'                    Грамматика описывает предложения вида:
T → ' | T, | Tc | Z' | P'          '<слово>',...,'<слово>'
P → Z,
```

Здесь c – любой символ, кроме апострофа и запятой. <Слово> может содержать запятую и апостроф (в этом случае апостроф удваивается).

В ходе разбора предложения должно быть выполнено заполнение массива S заданными символьными константами (без ограничивающих апострофов). Например, для предложения 'кот','об''явление','ку,ку' по концу разбора должно быть получено S(1)=кот, S(2)=об'явление, S(3)=ку,ку.

Вариант 9

```
Z → Zц | Zб | Tб     Грамматика описывает предложения вида:
T → Z\ | Q\          <диск>:\<каталог1>[\<каталог2>...]
Q → B:
B → б
```

В ходе разбора предложения необходимо определить имя диска и имена всех каталогов. Например, для предложения c:\ncc\kt по концу разбора должно быть получено D=c, S(1)=ncc, S(2)=kt.

Вариант 10

```
Z → D}              Грамматика описывает предложения вида:
D → Dц | Aц         {<целое>..<целое>[,<целое>..<целое>...]}
A → C.
C → Q.
Q → Qц | Bц | Tц
B → D,
T → {
```

В ходе разбора предложения должно быть выполнено заполнение массива M заданными целыми числами и подсчитано количество пар чисел. Например, для предложения {1..10,5..777,0..100} по концу разбора должно быть получено M(1)=1, M(2)=10, M(3)=5, M(4)=777, M(5)=0, M(6)=100 и n=3.

Вариант 11

```
Z → Dц | Bц          Грамматика описывает предложения вида:
B → Z,               <идентификатор>:<цифра>[,<цифра>...]
D → K:
K → б | Kц | Kб
```

В ходе разбора предложения необходимо получить имя идентификатора S и заполнить цифрами массив I. Например, для предложения a12:5,6,9 по концу разбора должно быть получено S=a12, I(1)=5, I(2)=6 и I(3)=9.

Вариант 12

```
Z → Qc | Lc | Zc     Грамматика описывает предложения вида:
L → Z.               <строка>@<строка>[.<строка>...]
Q → N@
N → c | Nc
```

В ходе разбора предложения строками должен быть заполнен массив S. Например, для предложения pupkin@gsu.unibel.by по концу разбора должно быть получено S(1)=pupkin, S(2)=gsu, S(3)=unibel и S(4)=by.

Вариант 13

```
Z → D]          Грамматика описывает предложения вида:
D → Аб          [<буква>..<буква>[,<буква>..<буква>[,...]]]
A → C.
C → Q.
Q → Вб | Тб
B → D,
T → [
```

В ходе разбора предложения должно быть выполнено заполнение массива L числами, равными количествам букв алфавита, попадающих в указанный интервал. Если вторая буква расположена раньше первой в алфавите, то число должно быть отрицательным. Например, для предложения [a..a,a..z,z..a,a..b,l..k] по концу разбора должно быть получено L(1)=0, L(2)=25, L(3)=-25, L(4)=1 и L(5)=-1.

Вариант 14

```
Z → ц | Zц | Dц
D → Z* | Z/
```

Грамматика описывает предложения вида: <число>[*|/<число>...]

В ходе разбора предложения необходимо вычислить значение R данного выражения. Например, для предложения 5*6/2 по концу разбора должно быть получено R=15.

Вариант 15

```
Z → P ) | M )
P → Qц | Pц
M → Hц | Mц
H → P.
Q → ( | R
R → M, | P,
```

Грамматика описывает предложения вида: (<число>,...,<число>)

В ходе разбора предложения заданными числами (возможно, дробными) должен быть заполнен массив M. Например, для предложения (1.5,10,15.1239) по концу разбора должно быть получено M(1)=1.5, M(2)=10 и M(3)=15.1239.

3 Нисходящий грамматический разбор

3.1 Алгоритм нисходящего разбора

Цель нисходящего грамматического разбора – найти вывод $Z \Rightarrow + a_1...a_N$, где Z – начальный символ грамматики, $a_1...a_N$ – входная строка. Алгоритм разбора:

1) начальный символ грамматики заменяется по какому-либо правилу: $Z \rightarrow U_1...U_N$;

2) в дальнейшем, если U_1 – терминал, то он сравнивается с первым символом входной строки и при совпадении исключается из входной строки и из строки элементов разложения Z. Если U_1 – нетерминал, то он заменяется по какому-либо правилу, и процесс повторяется.

Пример грамматики для выражений, составленных из идентификаторов (ид – терминальный символ) и знаков +, *:

```
S → T | T+S
T → ид | ид*T
```

Разбор предложения a+b*c представлен на рисунке 3.1.

a+b*c	S	S → T+S	Указаны остатки входной строки
a+b*c	T+S	T → ид	и строки разложения начального
a+b*c	ид+S		символа грамматики - S, а также
+b*c	+S		правила, применяемые для разло-
b*c	S	S → T	жения левого нетерминального
b*c	T	T → ид*T	символа. На последнем шаге оба
b*c	ид*T		остатка пусты, что соответ-
*c	*T		ствует правильному разбору.
c	T	T → ид	
c	ид		
λ	λ		

Рисунок 3.1 – Нисходящий разбор предложения

Так как разложение нетерминалов грамматики обычно может быть выполнено по нескольким правилам, необходимо осуществить полный перебор вариантов разложения, что показано на рисунке 3.2.

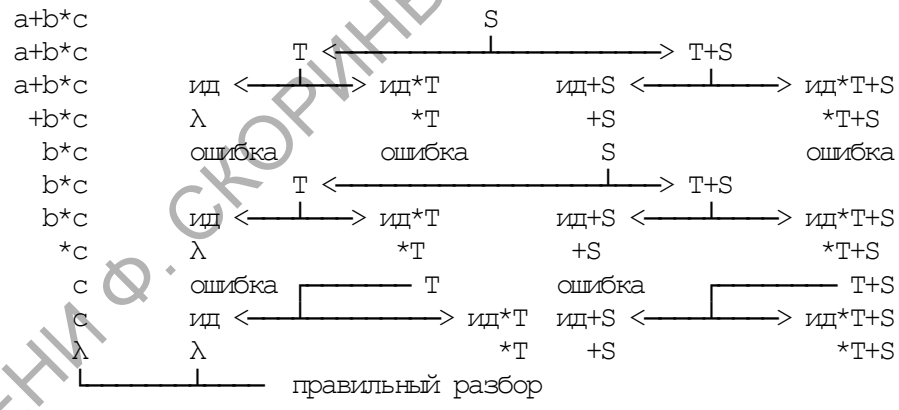


Рисунок 3.2 – Разбор при полном переборе правил грамматики

3.2 Преобразование грамматики для ускорения нисходящего разбора

Сократить перебор вариантов можно путем преобразования грамматики таким образом, чтобы правые части правил подстановки начинались с различных символов. Для этого правила вида
`X → ap|aq` (a – символ, p и q – строки)
заменяются на правила
`X → aY, Y → p|q.`

```
S → TP          Преобразованная таким способом
P → λ | +S      грамматика для выражений из
T → идR         идентификаторов и знаков +, *.
R → λ | *T
```

Нисходящий разбор предложения a+b*c для преобразованной грамматики представлен на рисунке 3.3.

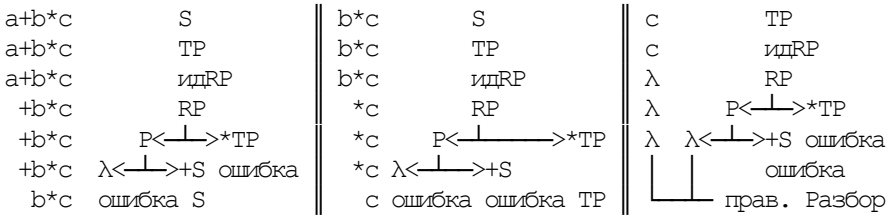


Рисунок 3.3 – Нисходящий разбор для преобразованной грамматики

Лабораторная работа

Цель: получение практических навыков в использовании методов нисходящего грамматического разбора.

Материалы и оборудование: персональный компьютер.

Следующая грамматика определяет произвольные арифметические выражения, содержащие идентификаторы, константы, знаки арифметических операций и скобки:

```
В → Т | -Т | Т+В | Т-В
Т → Э | Э*Т | Э/Т
Э → ид | кн | (В)
```

Здесь ид – терминальный символ «идентификатор», кн – терминальный символ «константа».

Требуется преобразовать заданную грамматику таким образом, чтобы правые части правил вывода не начинались с одинаковых символов. Для заданных предложений провести синтаксический анализ, используя метод нисходящего грамматического разбора с полным перебором вариантов. Каждый приведенный в таблице 3.1 вариант условия задачи содержит синтаксически корректное и синтаксически некорректное предложение.

Таблица 3.1 – Варианты заданий лабораторной работы

Вариант	Предложения языка	Вариант	Предложения языка
1	((a+b)-(b+c))*5 -a/5*(7+d))	6	((7*5)+(a/b))/d (-a/8)*d/(5))
2	(3/a)**7 (8+3)*(a-b)/c	7	(-2)-a+(c+d)*5 (a/b**2)/a
3	(-a)+((a/b))*(d-3) (2)+a/b++c	8	(a+5)/(b**c 2*(-5)+(a/b)
4	5/a-7+(3*d) ((a+b)/(7/-2))*3	9	((a+b)/2-5)+d -a+(-b)/(2+c)
5	a*(b-1)+5*f/(3+2) 5-(-3)/a++b	10	a/b-(-7)*(--5) 3-a/b+(a*b)/5
11	(2+(b-1))/5*f/3 b+(-3)/(5+*n)	12	(a*7)/c+7*(d-3) (2)+a/(b*(c-1)+d
13	(a*(7+d))/(5-t) (a/b**2)/a	14	5-(a-c)/3*d/(a+b) a*((d/-2)*(3-f))
15	((g*5)+a/9)*(5-y) (-5*f)/7/(5+7))		

4 Синтаксический анализ предложений для грамматик с простым предшествованием

4.1 Восходящий грамматический разбор

При восходящем грамматическом разборе правила подстановки применяются для свертки (приведения) всего исходного предложения к начальному символу грамматики. С этой целью в приводимой строке последовательно отыскивается основа – самая левая часть строки, которую по одному из правил подстановки следует заменить на нетерминал, и выполняется эта замена.

Например, рассмотрим грамматику арифметических выражений с использованием идентификаторов и операций + и *:

S → T | T+S
T → ид | ид*T

a***b**+c ⇒ **a*T**+c ⇒ T+**c** ⇒ T+**T** ⇒ **T+S** ⇒ S

Основная проблема – нахождение основы. Так, если первой выполнить свертку a*b+c ⇒ T*b+c, т.е. за основу принять символ ‘a’, то в дальнейшем полностью свернуть предложение к начальному символу грамматики уже не удастся. Нахождение основы может быть выполнено различными методами, в частности, путем определения отношений предшествования между символами.

4.2 Отношения предшествования в грамматиках с простым предшествованием

Между любыми двумя последовательными символами приводимой строки p и q могут существовать три отношения:

p<q, если q – самый левый символ некоторой основы: pq...
└ основа →

p>q, если p – самый правый символ некоторой основы: ...pq
← основа ┘

p=q, если символы p и q находятся внутри основы: ...pq...
└ основа ┘

Отношения <, >, = называются отношениями предшествования.

Пусть p, q – любые символы (терминальные или нетерминальные), U, C, D – нетерминальные символы, x, y, z, w – любые строки, возможно пустые. Грамматика называется грамматикой с простым предшествованием, если правые части всех правил грамматики различны и для каждой упорядоченной пары символов существует не более чем одно из трех отношений предшествования:

p=q, если существует правило U → xpqy;

p<q, если существует правило U → xpDy и вывод $D \Rightarrow + qz$;

p>q, если существует правило U → xCqy и вывод $C \Rightarrow + zp$ или правило U → xCDy и выводы $C \Rightarrow + zp$, $D \Rightarrow + qw$.

Для определения отношений предшествования предварительно для каждого нетерминального символа U находятся два множества:

L(U) – множество самых левых символов в разложениях U, т.е. множество таких символов q, что существует вывод $U \Rightarrow + qz$;

R(U) – множество самых правых символов в разложениях U, т.е. множество таких символов p, что существует вывод $U \Rightarrow + zp$.

Построение множеств L(U), R(U) состоит из следующих шагов:

1) для каждого правила вида U → z в L(U) заносится самый левый символ правой части, в R(U) – самый правый символ;
2) если L(U) содержит нетерминальные символы U_1, U_2, ..., то L(U) дополняется символами, входящими в $L(U_1)$, $L(U_2)$ ... и не входящими в L(U). Аналогично дополняется R(U);
3) шаг 2 повторяется, пока L(U), R(U) не перестанут изменяться.

Пример:

```
Z → bMb          Грамматика определяет цепочки символов
M → (N | a       bab, b(aa)b, b((aa)a)b, b(((aa)a)a)b и т.д.
N → Ma)
```

Построение множеств левых и правых символов для нетерминалов показано на рисунке 4.1.

L(U), R(U) после первого шага

U	L(U)	R(U)
Z	b	b
M	(, a	N, a
N	M	)

окончательно

U	L(U)	R(U)
Z	b	b
M	(, a	N, a,)
N	M, (, a	)

Рисунок 4.1 – Построение множеств L(U), R(U)

Для вывода отношений между символами просматриваются правые части правил:

1) для каждой последовательной пары символов pq принимается p=q;
2) для каждого сочетания pU принимается $p<q$, $\forall q \in L(U)$;
3) для каждого сочетания Uq принимается $p>q$, $\forall p \in R(U)$;
4) для каждого сочетания CD принимается $p>q$, $\forall p \in R(U)$ и $\forall q \in L(U)$.

Отношения между символами для грамматики из примера представлены в таблице 4.1.

Таблица 4.1 – Отношения между символами грамматики

	Z	b	M	N	a	(	)
Z							
b			=		<	<	
M		=			=		
N		>			>		
a		>			>		=
(			<	=	<	<	
)		>			>		

Например:

1) b = M, т.к. существует правило Z → bMb;

2) (< a, т.к. существует правило M → (N и $a \in L(N)$;

3) N > b, т.к. существует правило Z → bMb и $N \in R(M)$.

4.3 Разбор предложений в грамматиках с простым предшествованием

Разбор предложений в грамматиках с простым предшествованием ведется с использованием стека терминальных и нетерминальных символов.

Пусть необходимо выполнить грамматический разбор для некоторой входной строки терминальных символов $t_1...t_n$. Строка ограничивается с двух сторон терминальным символом ⊥, при этом считается, что для любого символа p: ⊥ < p и p > ⊥.

Формируется стек S, элементы которого s_i – терминальные и нетерминальные символы. В стек заносится символ левый ограничитель строки – ⊥ (s_1=⊥). Далее, на каждом промежуточном этапе действия определяются состоянием стека – $s_1...s_k$ и остатком входного предложения – $t_j...t_n$⊥. Проверяется отношение между верхним элементом стека s_k и очередным символом строки t_j. Если отношения нет, то входная строка некорректна. Если $s_k<t_j$ или $s_k=t_j$, то t_j заносится в стек. Если $s_k>t_j$, то отыскивается множество символов в вершине стека, удовлетворяющих отношению: $s_{i-1}<s_i=s_{i+1}=...=s_k>t_j$. Элементы $s_i...s_k$ принадлежат одной основе и могут быть свернуты по некоторому правилу грамматики: U → $s_i...s_k$. Выполняется свертка: отыскивается это правило, символы $s_i...s_k$ удаляются из стека, а U – заносится. Процесс повторяется: символ U, как верхний элемент стека, сравнивается с t_j и т.д.

По концу разбора правильного предложения в стеке должны остаться символы ⊥Z, где Z – начальный символ грамматики.

Пример разбора предложения ⊥b((aa)a)b⊥ представлен на рисунке 4.2.

Стек	Остаток	Отношение	Основа	Правило
⊥	b((aa)a)b⊥	⊥ < b		
⊥b	((aa)a)b⊥	b < (		
⊥b(	(aa)a)b⊥	(< (		
⊥b((	aa)a)b⊥	(< a		
⊥b((a	a)a)b⊥	a > a	(< a > a	M → a
⊥b((M	a)a)b⊥	M = a		
⊥b((Ma	)a)b⊥	a =)		
⊥b((Ma)	a)b⊥	) > a	(< M=a=) > a	N → Ma)
⊥b((N	a)b⊥	N > a	(< (=N > a	M → (N
⊥b(M	a)b⊥	M = a		
⊥b(Ma	)b⊥	a =)		
⊥b(Ma)	b⊥	) > b	(< M=a=) > b	N → Ma)
⊥b(N	b⊥	N > b	b < (=N > b	M → (N
⊥bM	b⊥	M = b		
⊥bMb	⊥	b > ⊥	⊥ < b=M=b > ⊥	Z → bMb
⊥Z	⊥	конец		

Рисунок 4.2 – Разбор предложения в грамматике с простым предшествованием

Одной из проблем, возникающих при разработке трансляторов с реальных языков программирования, является неоднозначность предшествований.

Рассмотрим пример:

S → T | S+T
T → ид | T*ид

Так как существует правило S→S+T, то + = T. Но, исходя из того же правила и вывода T ⇒ T*ид, имеем + < T.

Как правило, неоднозначные предшествования можно ликвидировать путем модификации грамматики:

S → T | S+T Здесь + = T, + < U
T → U
U → ид | U*ид

Однако здесь нет строго формальных методов преобразования, и, как правило, такое преобразование сильно увеличивает размер грамматики.

Другой способ – использование контекста при разборе. Например, пусть между S_i, S_{i+1} существуют отношения $S_i < S_{i+1}$, $S_i = S_{i+1}$ и нужно определить, какое именно отношение нужно взять в контексте $S_i\ S_{i+1}\ S_{i+2}$. Если существует правило, правая часть которого начинается с $S_{i+1}S_{i+2}$, значит надо взять отношение $S_i < S_{i+1}$. Иначе $S_i = S_{i+1}$.

Для приведенного примера:

```
+T*  -  здесь + < T, есть правило T → T*ид
+T+  -  здесь + = T, нет правила    → T+
```

Другой проблемой в грамматиках реальных языков программирования является совпадение правых частей правил подстановки. Это не позволяет однозначно выбрать нетерминал, на который следует заменить основу, выделенную в ходе разбора.

Например:

```
S → T | S+T              Имеются два правила с
T → ид | T*P             одинаковыми правыми частями
P → ид
```

Тогда для входного предложения a*b+c идентификаторы a, c могут быть заменены только на T, b – только на P:

a*b+c ⇒ T*b+c ⇒ T*P+c ⇒ T+c ⇒ S+c ⇒ S+T ⇒ S

Задача выбора правила также решается путем дополнительного анализа контекста. Для заданного примера замена ид ⇒ P должна проводиться только в контексте *ид ⇒ *P.

Лабораторная работа

Цель: получение практических навыков в использовании методов синтаксического анализа предложений для грамматик с простым предшествованием.

Материалы и оборудование: персональный компьютер.

1. Задана грамматика с простым предшествованием:

```
Z → P | ZOP
P → ид | P.ид
O → + | -
```

Грамматика определяет выражения вида:

```
ид[.ид...][{+|-}ид[.ид...]...],
```

где ид – терминальный символ «идентификатор» (например: a.b.c+a.b-c).

Требуется вывести отношения между символами грамматики и провести разбор заданных предложений. Каждый приведенный в таблице 4.2 вариант условия задачи содержит синтаксически корректное и синтаксически некорректное предложение.

Таблица 4.2 – Варианты заданий лабораторной работы

Вариант	Предложения языка	Вариант	Предложения языка
1	a.b+c-a.b.c.d c+h.e+a.b.c.-h.h	6	g+t.d.h.j-j.h.-n f.f.f+a-v-h.g
2	c.a.a-a-v.b+ f.s+h+s-d.t.y.r	7	g+t.i.t+d.f-s f.t.d+y.t-s-f.g.
3	h-w-t.y.f+s.a g.t-f+d.t.e++a.a	8	d.t-h.w.r+h-r.e f-r.y+s.d.f-
4	f+t.f-a-t.y.v. g.y+f.i.w-f-r.w	9	w.e.f+d.t-g.+ s.f-t.h-e+d.r.s.c
5	f.w-h.f-s.t+f d.g.h.j+f.s-f.+d	10	f.h+d+s+s.v-f.r.t f.t.h.g-t+f.f.-d
11	g.t.q.j+m.e+k+s l+m.f.t.h+f.m.-d	12	k+b+s-s.v-f.r.a q.t.g.g-t.k+f-
13	o.p.e.r+d-s+s.f-z b+t.g-t+s..f.-x	14	d.f.y+d.t+s.r+f d.a.h.g-t+-d
15	g.s+h.y.t-s.v-w.t k.o.t.h.g-t+		

2. Для заданной грамматики с простым предшествованием найти отношения между символами грамматики. Реализовать на компьютере программу синтаксического анализа для предложений языка. Обеспечить в программе вывод необходимых диагностических сообщений при обнаружении синтаксических ошибок во входных предложениях.

Обозначения терминальных символов при описании грамматик:

ид – терминальный символ «идентификатор»,
кн – терминальный символ «константа».

Вариант 1

```
Z → N | GN
G → Z,
N → кн–N
```

Вариант 2

```
Z → (C)
C → D | C,D
D → ид | Z
```

Вариант 3

```
Z → (C) | (Z)
C → кн | C,кн
```

Вариант 4

```
Z → T | Z!T
T → P | T&P
P → ид| ~P | (Z)
```

Вариант 5

```
Z → C*D | C/D
D → C | C.кн
C → кн
```

Вариант 6

```
Z → C+D | C–D
D → C | C.кн
C → кн
```

Вариант 7

```
Z → (G)
G → FL | FZL
F → a
L → b
```

Вариант 8

```
Z → идG
G → (C)
C → T | C,T
T → кн | кн:кн
```

Вариант 9

```
Z → (Q)
Q → T | Q,T
T → кн | Z
```

Вариант 10

```
Z → D:C
D → ид
C → F | E\F
F → ид | ид.ид
E → \ид | E\ид
```

Вариант 11

```
Z → G | ZG
G → CD | CGD
C → *
D → -
```

Вариант 12

```
Z → L=R
L → ид
R → ид+T | ид–T
T → (C) | C
C → кн
```

Вариант 13

```
Z → (Q) | (Z)
Q → ид | Qк/н | Q/ид
```

Вариант 14

```
Z → (C)
C → F | C,F
F → ид | ид–ид
```

Вариант 15

```
Z → 'S'
S → ид | S/ид
```

5 Синтаксический анализ предложений для грамматик с операторным предшествованием

5.1 Отношения предшествования в грамматиках с операторным предшествованием

5.2 Разбор предложений в грамматиках с операторным предшествованием

5.1 Отношения предшествования в грамматиках с операторным предшествованием

Операторной грамматикой называется грамматика, не содержащая правил вида A → xBCy, где x, y – произвольные строки, B, C – нетерминалы.

Пусть U, C, D – нетерминалы. Грамматикой с операторным предшествованием называется операторная грамматика, в которой все правые части правил различны и между двумя любыми терминальными символами p, q выполняется не более чем одно из следующих отношений предшествования:

p=q, если существует правило U → xpqy или U→xpCqy;

p<q, если существует правило U → xpCy и вывод $C \Rightarrow + qz$ или $C \Rightarrow + Dqz$;

p>q, если существует правило U → xCqy и вывод $C \Rightarrow + zp$ или $C \Rightarrow + zpD$.

Для определения отношений строятся множества:

LT(U) – множество левых терминалов в разложениях U, т.е. множество таких терминалов q, что существует вывод $U \Rightarrow + qz$ или $U \Rightarrow + Dqz$;

RT(U) – множество правых терминалов в разложениях U, т.е. множество таких терминалов p, что существует вывод $U \Rightarrow + zp$ или $U \Rightarrow + zpD$.

С этой целью:

1) строятся множества левых и правых символов – L(U), R(U) (см. п.4.2);
2) для правил вида U → qz, U → Cqz терминал q заносится в LT(U); для правил вида U → zp, U → zpC терминал p заносится в RT(U);
3) для каждого нетерминала U', принадлежащего L(U), LT(U) дополняется терминалами из LT(U'). Аналогично дополняется RT(U).

Пример:

```
S → T | S+T          Грамматика для выражений, состоящих из
T → ид | T*ид        идентификаторов и знаков +, *.
```

Построение множеств левых и правых терминалов для нетерминальных символов показано на рисунках 5.1, 5.2.

L(U), R(U) после первого шага

U	L(U)	R(U)
S	T, S	T
T	Ид, T	ид

окончательно

U	L(U)	R(U)
S	T, S, ид	T, ид
T	ид, T	ид

Рисунок 5.1 – Построение множеств L(U), R(U)

LT(U), RT(U) после первого шага

U	LT(U)	RT(U)
S	+	+
T	ид, *	ид

окончательно

U	LT(U)	RT(U)
S	+, ид, *	+, ид
T	ид, *	ид

Рисунок 5.2 – Построение множеств LT(U), RT(U)

Для вывода отношений между терминальными символами просматриваются правые части правил:

1) для каждой последовательной пары терминальных символов pq или сочетания pUq принимается p=q;
2) для каждого сочетания pU принимается $p<q, \forall q \in LT(U)$;
3) для каждого сочетания Uq принимается $p>q, \forall p \in RT(U)$.

Для примера

	+	*	ид
+	>	<	<
*			=
ид	>	>	

Например:
1) *=ид, т.к. существует правило T → T*ид;
2) + < *, т.к. существует правило S → S+T и $* \in LT(T)$;
3) + > +, т.к. существует правило S → S+T и $+ \in RT(S)$.

5.2 Разбор предложений в грамматиках с операторным предшествованием

Разбор в грамматиках с операторным предшествованием выполняется так же, как в грамматиках с простым предшествованием, но отношения проверяются только между терминалами.

Пусть стек содержит символы $s_1...s_k$, $t_1...t_n\perp$ – остаток входной строки, $p=s_m$ – верхний терминал стека ($m<=k$). Если $p=t_j$ или $p<t_j$, то t_j заносится в стек. Если $p>t_j$, то отыскивается множество терминалов в вершине стека, удовлетворяющих отношениям $s_a<s_b=s_c=...=s_m>t_j$. Тогда все элементы стека от s_{a+1} до s_k составляют основу и могут быть свернуты в некоторый нетерминал U.

Пример разбора предложения ⊥a+b*c⊥ представлен на рисунке 5.3.

Стек	Остаток	Отношение	Выбор основы	Основа	Правило
⊥	a+b*c⊥	⊥ < a			
⊥a	+b*c⊥	a > +	⊥ < a > +	a	T → ид
⊥T	+b*c⊥	⊥ < +			
⊥T+	b*c⊥	+ < b			
⊥T+b	*c⊥	b > *	+ < b > *	b	T → ид
⊥T+T	*c⊥	+ < *			
⊥T+T*	c⊥	* = c			
⊥T+T*c	⊥	c > ⊥	+ < *=c > ⊥	T*c	T → T*ид
⊥T+T	⊥	+ > ⊥	⊥ < + > ⊥	T+T	S → T, S → S+T
⊥S	⊥	конец			

Рисунок 5.3 – Разбор предложения в грамматике с операторным предшествованием

Лабораторная работа

Цель: получение практических навыков в использовании методов синтаксического анализа предложений для грамматик с операторным предшествованием.

Материалы и оборудование: персональный компьютер.

Задана грамматика с операторным предшествованием, которая определяет произвольные арифметические выражения, содержащие идентификаторы, константы, знаки арифметических операций и скобки:

```
В → Т | -Т  | В+Т | В-Т
Т → Э | Т*Э | Т/Э
Э → ид | кн | (В)
```

Здесь ид – терминальный символ «идентификатор», кн – терминальный символ «константа».

Требуется вывести все необходимые отношения между терминальными символами и провести восходящий разбор заданных предложений. Каждый приведенный в таблице 5.1 вариант условия задачи

содержит синтаксически корректное и синтаксически некорректное предложение.

Таблица 5.1 – Варианты заданий лабораторной работы

Вариант	Предложения языка	Вариант	Предложения языка
1	a*(b-1)+5*f/(3+2) 5-(-3)/a++b	6	a/b-(-7)*(--5) 3-a/b+(a*b)/5
2	(-a)+((a/b)/c)*(d-3) (2)+a/b++c	7	(a+5)/(b**c 2*(-5)+(a/b)
3	((a+b)-(b+c))*5 (-a/5*(7+d)	8	((7*5)+(a/b))/d (-a/8)*d/(5))
4	(3/a)**7 (8+3)*(a-b)/c	9	(-2)-a+(c+d)*5 (a/b**2)/a
5	5/a-7+(3*d) ((a+b)//(7-2))*3	10	((-a+b)/2-5 -a+(-b)/(2+c)
11	a/(p-5)+5*f*h+6 s+(-8)+b-	12	-h/(s-d)-5*2+(k+3) w/a+b-d+8)
13	5*((b-1)+f)+s/(1+2) -3/a+z+(--b)	14	w+q+e+g-(b-9)+5*f h+(s-a)*3+x/u*
15	(d+r)*a*(w+9)+5-f/2 9-(f)/a+ab		

6 Синтаксический анализ предложений для грамматик с ограниченным контекстом

6.1 Понятие грамматик с ограниченным контекстом
6.2 Структура правил подстановки Флойда
6.3 Построение правил подстановки Флойда
6.4 Грамматический разбор с использованием правил подстановки Флойда

6.1 Понятие грамматик с ограниченным контекстом

Грамматики, для которых основа в предложении определяется не более чем m символами, находящимися слева от основы, и не более чем n символами справа от основы называется грамматикой с (m,n) ограниченным контекстом. В частности, грамматики с простым предшествованием представляют собой грамматики с (1,1) ограниченным контекстом (в ходе проверки отношений рассматривается один символ слева от основы и один символ справа от неё).

Грамматики, для которых основа определяется всей находящейся слева от нее частью приводимой строки и не более, чем n расположенными справа терминальными символами, называется LR(n)-грамматикой. Для любой LR(n)-грамматики существует эквивалентная ей LR(1)-грамматика. Алгоритм грамматического разбора для LR(n)-грамматик основан на предварительном переводе грамматик на некоторый метаязык, называемый языком правил подстановки Флойда.

6.2 Структура правил подстановки Флойда

Правила подстановки Флойда для LR(1)-грамматик записываются в следующем формате:

M_1	$x_1 \ldots x_k$	$y_1 \ldots y_m$	СП	*	M_2

где

M_1 – метка правила (может отсутствовать);

$x_1...x_k$ – образец для сравнения с частью приводимой строки. Может содержать символ $\forall$ – любой символ;

$y_1...y_m$ – строка, которая заменяет часть приводимой строки $x_1...x_k$ (может отсутствовать);

СП – имя семантической подпрограммы (может отсутствовать). Это может быть фиксация обнаруженной ошибки, выход по концу разбора и т.п.;

* – знак, предписывающий прочитать следующий символ входной строки (может отсутствовать);

M_2 – метка правила, к которому нужно перейти после успешного применения данного правила (может отсутствовать).

Грамматический разбор предложений для LR(n)-грамматик ведется с использованием стека терминальных и нетерминальных символов. Возможность применения отдельных правил подстановки Флойда определяется совпадением образца $x_1...x_k$ из правила с k верхними символами стека.

Завершение грамматического разбора осуществляется путем вызова специальной семантической подпрограммы – «Выход». Для обеспечения автоматического вызова этой подпрограммы исходная грамматика дополняется правилом вида: $Z \rightarrow S\perp$, где S – начальный символ исходной грамматики, $\perp$ – ограничитель, добавляемый к входной строке.

6.3 Построение правил подстановки Флойда

Алгоритм построения правил подстановки Флойда для LR(1)-грамматик предполагает построение двух типов правил:

1) помеченных S_0, которые строятся для нетерминала S, если верхний терминальный символ стека определяет начало строки, которая может быть свернута в S;
2) помеченных S_1, если символ S (терминал или нетерминал) находится в следующей за вершиной позицией стека.

Обозначения:

L_0 – множество символов S, для которых нужно построить правила S_0;

L_0' – множество символов S, для которых уже построены правила S_0;

L_1 – множество символов S, для которых нужно построить правила S_1;

L_1' – множество символов S, для которых уже построены правила S_1.

Первоначально: L_0={Z} – начальный символ грамматики; L_0', L_1, L_1' – пустые множества.

Образование S_0-правил.

0.1 Если $L_0 \backslash L_0'$ – пусто, то перейти к пункту 1.1.

0.2 Для символа H, принадлежащего $L_0 \backslash L_0'$, построить LT'(H) – множество терминалов, с которых может начинаться разложение H. Для этого строится множество самых левых символов разложения H – L(H) и терминалы из L(H) включаются в LT'(H).

0.3 Для каждого t из LT'(H) найти множество правил грамматики: $\{K \mid K \rightarrow tz\}=\{k_1,...,k_r\}$, z – произвольная строка, и выполнить следующие действия:

0.3.1 Если r > 1, то сформировать правило подстановки Флойда | H_0 | t | | *| t_1 | и включить t в L_1.

0.3.2 Если r = 1, то в зависимости от вида правила:

0.3.2.1 K → t | H_0 | t | K | *| K_1 | и включить K в L_1;

0.3.2.2 K → tbw (b – терминал, w – произвольная строка)
| H_0 | t | | *| t_1 | и включить t в L_1;

0.3.2.3 K → tGw (G – нетерминал)
| H_0 | t | | *| G_0 | и включить G в L_0.

0.4 Уничтожить метку H_0 во всех правилах, кроме первого. Добавить правило | | ∀ | | Ошибка | | |. Включить H в L_0'. Перейти к пункту 0.1.

Образование S_1-правил.

1.1 Если $L_1\backslash L_1'$ – пусто, то конец работы.

1.2 Для каждого символа X из $L_1\backslash L_1'$ выделить все правила грамматики, содержащие в правых частях символ X.

1.3 Для каждого выделенного правила G → uXv образовать правило подстановки Флойда в зависимости от вида правила грамматики:

1.3.1 Z → S⊥ | S_1 | S⊥ | Z | Выход | | |
и включить S в L_1;

1.3.2 G → uX | X_1 | uX∀ | G∀ | | | G_1 |
и включить G в L_1;

1.3.3 G → uXt | X_1 | uXt | G | |*| G_1 |
и включить G в L_1;

1.3.4 G → uXUw, для каждого t из LT'(U)
| X_1 | uXt | | | | U_0 |
и включить U в L_0;

1.3.5 G → uXtbw (b – терминал)
| X_1 | uXt | | |*| t_1 |
и включить t в L_1;

1.3.6 G → uXtUw | X_1 | uXt | | |*| U_0 |
и включить U в L_0.

1.4 Правила с символом ∀ в конце образца поместить после остальных, упорядочив по убыванию длины образца.

1.5 Уничтожить метку X_1 во всех правилах, кроме первого. Добавить правило | ∀ | | Ошибка | | |. Включить X в L_1'. Перейти к пункту 0.1.

Рассмотрим пример грамматики для выражений из идентификаторов и знаков +, *. Символ ⊥ – ограничитель конца строки

Z → S⊥
S → T | S+T
T → ид | T*ид

Грамматика для выражений из идентификаторов и знаков +, *. Символ ⊥ – ограничитель конца строки.

U	L(U)	LT′(U)
Z	S, T, ид	ид
S	T, ид	ид
T	ид, T	ид

Построение правил подстановки Флойда показано на рисунке 6.1.

L_0	L_0'	L_1	L_1'	Правило грамматики	Пункт алгоритма	Правило подстановки Флойда					
Z				T → ид	0.3.2.1	(1) Z_0	ид	T		*	T_1
Z		T			0.4	(2)	∀		ош1		
Z	Z	T		S → T	1.3.2	(3) T_1	T∀	S∀			S_1
Z	Z	T,S		S → S+T	1.3.2	(4) T_1	S+T∀	S∀			S_1
Z	Z	T,S		T → T*ид	1.3.5	(5) T_1	T*			*	$*_1$
Z	Z	T,S,*			1.4	Упорядочение правил 5,4,3					
Z	Z	T,S,*			1.5	(6)	∀		ош2		
Z	Z	T,S,*	T	Z → S⊥	1.3.1	(7) S_1	S⊥	Z	вых		
Z	Z	T,S,*		S → S+T	1.3.6	(8) S_1	S+			*	T_0
Z,T	Z	T,S,*	T		1.5	(9)	∀		ош3		
Z,T	Z	T,S,*	T,S	T → ид	0.3.2.1	(10) T_0	ид	T		*	T_1
Z,T	Z	T,S,*	T,S		0.4	(11)	∀		ош4		
Z,T	Z,T	T,S,*	T,S	T → T*ид	1.3.3	(12) $*_1$	T*ид	T		*	T_1
Z,T	Z,T	T,S,*	T,S		1.5	(13)	∀		ош5		
Z,T	Z,T	T,S,*	T,S,*	конец							

Рисунок 6.1 – Построение правил подстановки Флойда

6.4 Грамматический разбор с использованием правил подстановки Флойда

Программа грамматического разбора использует таблицу правил подстановки и стек, содержащий уже рассмотренную и возможно ча-

стично свернутую часть строки. Исходная строка дополняется ограничителем ⊥, первый символ строки заносится в стек. Применение очередного правила начинается с сопоставления k верхних символов стека с образцом $x_1...x_k$. При несовпадении применяется следующее по порядку правило. При совпадении $x_1...x_k$ заменяется на $y_1...y_m$ (если $y_1...y_m$ пусто, то замена не производится). Далее, при необходимости, выполняется семантическая подпрограмма. Затем, если задан символ *, то в вершину стека добавляется очередной символ входной строки. Выполняется переход к правилу, помеченному M_2.

Пример грамматического разбора предложения a+b*c+d⊥ представлен на рисунке 6.2.

Правила подстановки Флойда (после упорядочения)					
(1) Z_0	ид	T		*	T_1
(2)	∀		ош1		
(3) T_1	T*			*	$*_1$
(4)	S+T∀	S∀			S_1
(5)	T∀	S∀			S_1
(6)	∀		ош2		
(7) S_1	S⊥	Z	вых		
(8)	S+			*	T_0
(9)	∀		ош3		
(10) T_0	ид	T		*	T_1
(11)	∀		ош4		
(12) $*_1$	T*ид	T		*	T_1
(13)	∀		ош5		

Метка	Стек	Остаток строки	Правило
Z_0	a	+b*c+d⊥	1
T_1	T+	b*c+d⊥	5
S_1	S+	b*c+d⊥	8
T_0	S+b	*c+d⊥	10
T_1	S+T*	C+d⊥	3
$*_1$	S+T*c	+d⊥	12
T_1	S+T+	d⊥	4
S_1	S+	d⊥	8
T_0	S+d	⊥	10
T_1	S+T⊥		4
S_1	S⊥		7
	Z	выход	

Рисунок 6.2 – Разбор предложения с использованием правил подстановки Флойда

Лабораторная работа

Цель: получение практических навыков в использовании методов синтаксического анализа предложений для грамматик с ограниченным контекстом.

Материалы и оборудование: персональный компьютер.

Для заданного варианта грамматики выполнить следующие действия:

1) определить вид предложений, составляющих язык, порождаемый грамматикой;
2) построить правила подстановки Флойда для грамматики;
3) разработать программу синтаксического анализа на основе правил подстановки Флойда.

Обозначения терминальных символов при написании грамматик:
ид – терминальный символ «идентификатор;
кн – терминальный символ «константа».

Вариант 1

Z → (C) | (Z)
C → кн | C,кн

Вариант 2

Z → T | Z!T
T → P | T&P
P → ид| ~P | (Z)

Вариант 3

Z → N | GN
N → кн | кн-N
G → Z,

Вариант 4

Z → (C)
C → D | C,D
D → ид | Z

Вариант 5

Z → (G)
G → FL | FZL
F → a
L → b

Вариант 6

Z → идG
G → (C)
C → T | C,T
T → кн | кн:кн

Вариант 7

Z → G | ZG
G → CD | CGD
C → *
D → -

Вариант 8

Z → C+D | C-D
D → C | C.кн
C → кн

Вариант 9

Z → C | ZC
C → D | C*D
D → ид | ид,C

Вариант 10

```
Z → (Z) | (P)
P → C,кн
C → кн | кн.кн
```

Вариант 11

```
Z → Z-T | T
T → P | T!P
P → ид
```

Вариант 12

```
Z → кнT
T → (T) | (P)
P → кн | кн+P
```

Вариант 13

```
Z → D:C
C → F | DF
D → ид | Z
F → ид.ид
```

Вариант 14

```
Z → L=R
L → ид
R → ид+T | ид-T
T → (C) | C
C → кн
```

Вариант 15

```
Z → (G)
G → C | C+D | C-D
C → кн*D
D → кн | кн,кн
```

7 Промежуточные языки трансляции

7.1 Назначение промежуточных языков трансляции

Перевод исходной программы на промежуточный язык трансляции осуществляется с целью обеспечить, в дальнейшем, возможность перевода программы в объектный вид на язык машинных команд за однократный просмотр программы на промежуточном языке.

Пусть, например, исходная программа содержит оператор:

```
a = (b + c / d) * (e - f);
```

который в процессе компиляции должен быть переведен на язык машинных команд. Этот оператор включает в себя следующие операции:

```
'=' (присваивание), '+', '/', '*', '-'.
```

Однако, с учетом приоритета операций и наличия скобок, эти операции должны выполняться в другой последовательности:

```
'/', '+', '-', '*', '='.
```

Требуемая последовательность выполнения операций может быть определена в ходе синтаксического анализа программы. И, чтобы исключить повторный анализ программы в процессе перевода отдельных операций на язык машинных команд, сама программа преобразуется в такой вид, который позволяет генерировать команды в процессе однократного просмотра программы строго в последовательности от начального к конечному символу.

Известны несколько промежуточных языков трансляции: тетрады, триады, косвенные триады и, наиболее распространенная при разработке компиляторов, – обратная польская запись (предложена польским математиком Лукашевичем).

7.2 Формат и основные операции обратной польской записи

Обратная польская запись представляет собой бесскобочную запись, которая строится по следующим правилам:

1) операнды в обратной польской записи следуют в том же порядке, как и в обычной (инфиксной) записи;
2) операторы следуют в том же порядке, как они должны выполняться;
3) операторы располагаются непосредственно за своими операндами, при этом операндом может являться как операнд инфиксной записи, так и результат какой-либо операции.

Основные операции обратной польской записи:

1) арифметические операции (+, –, ~ (унарный минус), *, /, ** и т.п.);
2) логические операции и операции сравнения;
3) операция присваивания;
4) вычисление адреса элемента массива, имеющая m+1 операнд, где m – размерность массива. Будет обозначаться `k]`, где k – количество операндов (k=m+1);
5) операция вызова функции, имеющая m+1 операнд, где m – количество параметров функции. Будет обозначаться `kФ`, где k – количество операндов (k=m+1);

6) операции перехода: <метка>БП – безусловный переход;
7) <логическая переменная><метка>УПЛ – переход по лжи (если логическая переменная или результат выражения имеет значение ложь). Место метки в программе отмечается операцией <метка>:.

Примеры перевода в обратную польскую запись выражений и операторов:

```
a*(-b+c/d)              ⇒    a b ~ c d / + *
(a<b || b>c) && d>e     ⇒    a b < b c > || d e > &&
y=-f(a/b,x[i][j])       ⇒    y f a b / x i j 3] 3Ф ~ =
```

Перевод в обратную польскую запись условного оператора:

```
if (L) A;               ⇒    L m1 УПЛ A m1:
if (L) A; else B;       ⇒    L m1 УПЛ A m2 БП m1: B m2:
```

Например:

```
if (a<b*x) x=y*a; else { x=y/a; a=b/a; }
a b x * < m1 УПЛ x y a * = m2 БП m1: x y a / = a b a / = m2:
```

Операторы цикла.

1. Оператор while для языка С

```
while (L) A;  ⇒  m1: L m2 УПЛ A m1 БП m2:
```

Например:

```
while (a<b) a=b+2;  ⇒  m1: a b < m2 УПЛ a b 2 + = m1 БП m2
```

2. Оператор for для языка С

```
for (A; L; B) C; ⇒ A  m1:L m2 УПЛ m3 БП  m4: B m1 БП  m3: C m4 БП m2:
                   └┘ └──────────────┘ └─────────┘ └──────────┘
                   A        L             B          C
```

В качестве примера рассмотрим программу определения значения и номера максимального элемента в массиве положительных чисел с использованием цикла for:

```
s = 0;
for (i = 0; i < n; i++)
  if (s < d[i]) { s = d[i]; j=i; }
```

В обратной польской записи программа имеет вид:

```
s 0 =
i 0 = m1: i n < m2 УПЛ m3 БП m4: i i 1 + = m1 БП
  m3: s d i 2] < m5 УПЛ s d i 2] = j i = m5: m4 БП m2:
```

Лабораторная работа

Цель: получение практических навыков в использовании методов перевода программ в обратную польскую запись.

Материалы и оборудование: персональный компьютер.

Перевести в обратную польскую запись выражения и операторы.

Вариант 1

```
1. n=(a+5)/(b*2)
2. d<=3 && a>5 || d>4
3. while (a[i]!=b[i]) {
     if (a[i]<b[j] || x+y<b[i]) {
       z[i][j]=f(a);
       c=ff(a[i],b[j]); }
     else {
       if (a[i]>5) c=ff(5,b[j]);
       z[i][j-1]=5*a[i]*b[j+1]; }
   }
```

Вариант 2

```
1. n=a*(b-1)+5*f/(3+2)
2. (3>a || a<=b) && c>b
3. if (a>b[i][j+1])
     if (a<b[i*2][j] && a>0) {
       c[i-1]=f(a,b[i][j]);
       while (x<n)
         x=2*a/b[i][j+1]; }
     else {
       c[i]=ff(b[i][j*a*2]);
       x=b[i][j-1]/8; }
```

Вариант 3

```
1. n=5/a-7+(3*d)
2. a>=b && c<=d || a && c
3. if (f(a,b)>5 && f(a,b)<z[j]) {
     if (z[i]<z[j]*5) {
       z[i]=x[i+1][j*2]/3;
       c=ff(x[i][j]); }
   }
   else {
     z[j]=f(a,b);
     while (z[j]<5*z[i*2])
       c=f(x[j*2][i*2],a/2); }
```

Вариант 4

```
1. n=(a+b)/c*(a-b)
2. a || c<=0 && a>b
3. c=y[f(d,x[j*i])][ff(i)];
   for (i=0; i<n+3; i++) {
     x[i]=x[i]/2;
     if (x[i]>1 && x[i]<9)
       a[i][i]=b/c+f(i,j);
     else {
       a[i][j+2]=b-ff(i*j);
       if (b>0) b=c/x[i*j];
       else c=f(b,c); } }
```

Вариант 5

```
1. n=3-a/b+(a*b)/5
2. 3<=a && (a>b || c<d)
3. while (i<n) {
     if (a[i]<b[j*2]) {
       a[i]=x[i][j]/ff(x[i][j]);
       if (z<0) b[j]=f(a[i],2);
       else b[i*2+1]=f(a[i],i); }
     else x[j][i]=5*ff(x[i][j]); }
```

Вариант 6

```
1. n=((7*5)+(a/b))/d
2. a || b || c>=3 && a>b
3. for (i=2; i>x[i]*8; i++) {
     if (a<x[i]) {
       c=5*f(a*3,x[i]/2);
       if (2-z[i][j-1]>5*a)
         c=5/ff(a*x[i]); }
     else c=x[i*2]/f(a+3,1); }
```

Вариант 7

```
1. n=((a+b)-(b+c))*5
2. a && (c<=d && (d || a<=5))
3. x[i]=ff(a*2,b[i-1]);
   for (x[i]=0; x[i]<x[i*i]; i++)
     if (a<0 && x[i]*2>f(b[j*2])) {
       a=b[ff(x[i*2],c/2)];
       if (a<0) c=c/2+f(a);
       else c=c*2+ff(1,b[j]); }
```

Вариант 8

```
1. n=(a/(b/c)-d)*3
2. (a>b || c<=5) && d
3. for (i=0; i<x[j][i]; i++) {
     if (a<x[i+3][j] && a>b) {
       c=ff(a+5,b*2,c/d);
       if (c>d) a=x[i][j]; }
     else a=x[f(a,b)][5];
   }
```

Вариант 9

```
1. n=-a+(-b)/(2+c)*3
2. (a>3 || a<=5) && (c>b || d)
3. while (a[i*2][j]>=5) {
     if (a[i][j*2]<b[j]) {
       x=a[f(d,2)][ff(b[j*2])];
       if (b[j]>0 && a[1][j]>0)
         x=z[i][j-1]; }
     else z[j][i]=a[5][ff(b[j])];
   }
```

Вариант 10

```
1. n=(8+3)*(a-b)/c
2. a && b<=c || c>d
3. if (a>x[i*2] && c[j]<0) {
     z[i][j+1]=f(b[i],a/2);
     a=b[ff(a,3,c[j]*3)]; }
   else
     for (a=10; a>-5; a--) {
       if (a<0) a=b[f(a,a)];
       else a=f(b[i*2],a); }
```

Вариант 11

```
1. n=5/(6-a)*(7+c)-b
2. (a>1 && b>2 || c>3) && d
3. if (a>0 || b<0 && y[i])
     while (a>0) {
       x=4-7*y[i]; a=a-2; }
   else {
```

Вариант 12

```
1. n=((-b-5)+3)+(a-b)/2
2. (a<b || c>0 && x<3) || x<0
3. if (b<7 || c>0 && x[i*2]<0) {
     x[j+1]=ff(y[i][j-4],a/2)+5;
     if (a*3>b/4) a=b+2*a; }
   else
```

```
    a=f(z[i+2][j],y[i-j]);
    if (y[i]<f(a,7))
      y[j]=ff(3)*f(2,y[i]); }
```

Вариант 13

```
1. n=a+b*c/(5-a/(n-3))
2. (x>y || y<z) && a<0 || d>5
3. if (f(a,5-b)<ff(x[i]))
     if (f(7,3)>a || y[2][j]<a/3)
       while (b[i+1][j]<a)
         if (a>x[j]) j=5;
         else a=-7+s/4*d-1;
     else y[j][i]=b[ff(x[j])][7];
   }
```

Вариант 14

```
1. n=(5+a/b*c)*(7-m)/(-b)
2. c>d && a<d || x>0 && y<z
3. if (a>4 && c<2 || d && k>a) {
     y[f(2)][j+1]=ff(x[i],d*2);
     if (b+a!=0) a=7/ff(x[i],0);
     else a=3*f(x[0]-7);
     for (i=7; i>0; i--)
       x[i]=f(2)+y[i+1][j-1]; }
   else a=f(x[ff(3,d-5)]);
```

Вариант 15

```
1. n=y+((-b)/2+c)*(3-x)
2. a || (x>y || x>z) && a>9
3. for (i=0; i<n; i++) {
     x[i]=f(a,x[i])*ff(y[i-1][i+1]);
     if (b>0 || x[i]>0 && x[i]>b) {
       if (a>b && c)
         a=x[i]/y[i+1][n]; }
     else y[2][i]=y[ff(9)][i*2];
   }
```

```
     for (n=a; n<b; n++)
       if (x[3]<0) x[n]=f(a,a);
   x[i]=x[i]*x[f(y[i][j],t*2)];
```

8 Генерация объектной программы

8.1 Алгоритм генерации машинных команд
8.2 Генерация команд перехода в объектной программе

8.1 Алгоритм генерации машинных команд

Для генерации машинных команд по обратной польской записи используется стек операндов – SO с указателем i (начальное значение

– i=1). Также используются рабочие переменные – r_j (начальное значение – j=1).

Алгоритм генерации команд:

1 Взять очередной символ s из обратной польской записи.

2 Если строка обратной польской записи закончилась, то завершить работу.

3 Если s – операнд, то занести s в стек операндов: SO_i=s, i=i+1 и перейти к пункту 1.

4 Иначе, s – знак операции с m операндами, которая может порождать результат или нет. Среди m верхних элементов стека: SO_{i-m}...SO_{i-1} определить рабочую переменную с минимальным номером n, которую можно использовать для размещения результата. Если среди этих элементов стека нет рабочих переменных, принимается n=j.

5 Генерируются машинные команды для реализации операции s. При этом, если операция порождает результат, то в этих командах результат должен заноситься в r_n.

6 Вычеркнуть из стека операнды: i=i-m, скорректировать j: j=n.

7 Если операция породила результат, то занести в стек r_n: SO_i=r_n, i=i+1 и скорректировать j: j=j+1.

8 Перейти к пункту 1.

8.2 Генерация команд перехода в объектной программе

Одной из проблем при генерации машинных команд является формирование адресов меток в командах перехода. Пусть, например, необходимо сгенерировать команды по оператору:

```
if (a<=b) a=c; else a=b+c;
```

Обратная польская запись имеет вид:

```
a b <= m1 УПЛ a c = m2 БП m1: a b c + = m2:
```

При генерации машинных команд по операциям УПЛ и БП необходимо в машинный код соответствующей команды перехода вставить адрес метки. Однако, это сделать невозможно, т. к. расположение этих меток в программе еще неизвестно. Поэтому генерация команд переходов выполняется в два этапа.

В ходе генерации команд формируются две таблицы: таблица адресов меток: | метка | адрес в программе | и таблица адресов недостроенных команд перехода: | адрес команды | метка |.

В первую таблицу элементы заносятся по операции «:» (заносится адрес очередной генерируемой команды). При генерации команды перехода просматривается таблица адресов меток. Если метка уже определена, ее адрес вставляется в команду. Иначе, генерируется команда перехода с нулевым адресом, адрес этой команды и нужная метка заносятся во вторую таблицу.

По концу генерации команд вторая таблица просматривается, адреса меток извлекаются из первой таблицы и заносятся в команды.

Например, пусть генерация объектной программы выполняется для компьютера, имеющего следующий состав машинных команд:

1) сложение:

```
<результат>=<операнд1>+<операнд2>
СЛ  <операнд1>,<операнд2>,<результат>
```

2) вычитание:

```
<результат>=<операнд1>-<операнд2>
ВЫЧ <операнд1>,<операнд2>,<результат>
```

3) пересылка:

```
<операнд2>=<операнд1>
ПЕР <операнд1>,<операнд2>
```

4) безусловный переход:

```
БП  <метка>
```

5) переход по минусу:

```
if (<операнд> < 0) goto <метка>
ПМ  <операнд>,<метка>
```

Пусть объектная программа располагается в памяти с адреса 1000, а каждая команда занимает 4 байта. Тогда процесс генерации программы будет иметь вид, представленный на рисунке 8.1.

s	SO	i	j	Команды	Таблица меток	Таблица команд перехода
	-	1	1			
a	a	2	1			
b	ab	3	1			
≤	r_1	2	2	1000 ВЫЧ b,a,r_1		
m_1	r_1m_1	3	2			
УПЛ	-	1	1	1004 ПМ r_1,0	→	1004 m_1
a	a	2	1			
c	ac	3	1			
=	-	1	1	1008 ПЕР b,a		
m_2	m_2	2	1			
БП	-	1	1	1012 БП 0	→	1012 m_2
m_1	m_1	2	1			
:	-	1	1		m_1 1016	
a	a	2	1			
b	ab	3	1			
c	abc	4	1			
+	ar_1	3	2	1016 СЛ b,c,r_1		
=	-	1	1	1020 ПЕР r_1,a		
m_2	m_2	2	1			
:	-	1	1		m_2 1024	

Рисунок 8.1 – Генерация объектной программы

По концу генерации команд просматривается таблица команд перехода, и адреса из таблицы меток заносятся в соответствующие команды.

Лабораторная работа

Цель: получение практических навыков в использовании методов генерации объектной программы.

Материалы и оборудование: персональный компьютер.

Перевести заданные операторы в обратную польскую запись и провести генерацию команд, реализующих эти операторы, для компьютера, имеющего следующий состав команд:

1) сложение: `<результат>=<операнд1>+<операнд2>`
   `СЛ <операнд1>,<операнд2>,<результат>`
2) вычитание: `<результат>=<операнд1>-<операнд2>`
   `ВЫЧ <операнд1>,<операнд2>,<результат>`
3) умножение: `<результат>=<операнд1>*<операнд2>`
   `УМН <операнд1>,<операнд2>,<результат>`
4) деление: `<результат>=<операнд1>/<операнд2>`
   `ДЕЛ <операнд1>,<операнд2>,<результат>`
5) пересылка: `<операнд2>=<операнд1>`
   `ПЕР <операнд1>,<операнд2>`
6) безусловный переход:
   `БП <метка>`
7) переход по плюсу: `if (<операнд> > 0) goto <метка>`
   `ПП <операнд>,<метка>`
8) переход по минусу: `if (<операнд> < 0) goto <метка>`
   `ПМ <операнд>,<метка>`
9) переход по нулю: `if (<операнд> == 0) goto <метка>`
   `ПН <операнд>,<метка>`

Предполагается, что объектная программа располагается в памяти с адреса 1000, и каждая команда занимает 4 байта.

Вариант 1

```
while (i < n) {
  if (a < b) {
    a=x/f*2;
    if (z < 0 || b > k) b=k+a*2;
    else b=k*2-a;
  }
  else x=5*f+(4-g)/3;
}
```

Вариант 2

```
while (a == b) {
  if (a < b*2 || b < x+y) {
    z=f*(a+b);
    c=k-a*i/b; }
  else {
    if (a > 5) c=k/5+b*2;
    z=5*a*b/(j+1); }
}
```

Вариант 3

```
x=y+f/d*(x-k);
for (i=0; i < n+3; i++)
  x=x/2;
if (x > 1 && x < 9)
  a=b/c+f;
else {
  a=5-b-k*j;
  if (a > 0) x=x/(x+2);
  else x=f*k-a; }
```

Вариант 4

```
if (a > b*(j+1)) {
  if (a < b && b < 0) {
    c=f*(a+b/(i-j));
    while (x < n)
      x=2*b+i/b*(i-1); }
  else {
    c=2*f/(b+a*2);
    x=b/8; }
}
```

Вариант 5

```
while (a >= 5) {
  if (a < b) {
    z=a+f/d*2;
    if (b > 0 && a > 0)
      x=z*8+j-1; }
  else z=a+5/k;
}
```

Вариант 6

```
x=f+a*(2-b);
for (x=0; x < 7; i++) {
  if (x < 0 && x*2 > f) {
    a=b+k/(x-c/2);
    if (a < 0) c=c/2+f;
    else c=c*2+k*8-x; }
}
```

Вариант 7

```
if (a > x*2 && c < 0) {
  z=f*(b+a/2);
  d=b-k*c*3; }
else {
  a=7*f/(a+b*j);
  for (a=a*2; a > z; a--) {
    if (a < 0) z=b-k+a/2;
```

Вариант 8

```
if (f*(a+b) > x && f < x) {
  if (z < z*5) {
    z=x*(i+1)-j*2/3;
    c=k-x/a; }
}
else {
  z=f/(a-b);
```

```
    else z=f*2;
} }
```

Вариант 9

```
for (i=2; i > z*c*8; i++) {
  if (a < x) {
    c=5*f+(a*3-x/2);
    if (z < 0 && z > 5*a)
      c=5/f*(a+x); }
  else z=(x+1)/f*(a+3/x); }
```

Вариант 11

```
while (x > a && c < 0) {
  a=(x-a*2)/d+2;
  c=c-7-a+d/3; }
if (a*2 > b-4) {
  a=7*f/(a+b*j);
  if (a <= b*2) z=a/b+4*d;
  if (d < 0) d=b*k*a/2;
  else d=a+b*2;
}
```

Вариант 13

```
if (b < d/3 && c != 0) {
  d=(f+3)/(a-c+5);
  if (x != b) x=b+d*3; }
else {
  n=f/(a-b*j)/(d+s);
  for (i=0; i < n; i=i+x) {
    if (a-i >= 0) x=k*d-2+f;
    else x=h+f/2;
} }
```

```
  while (z < 5*z/i*2)
    c=k*(x+a/2); }
```

Вариант 10

```
for (x=0; x == x*j*i; i++) {
  if (a < x/(i+3) && a > b) {
    c=f*(a+5/b*2);
    if (c > d) a=5*x/(j-1);
    else a=0; }
  else a=x+f*(a-b); }
```

Вариант 12

```
a=v+4*a/(k-1);
if (z <= 0 || k > 5*f) {
  while (z < 0)
    z=(k-x/a)/5;
}
else {
  k=d*(a-b)+3;
  if (z*(s-b/3) < 0) z=0;
  d=m-(n*2+a/7); }
```

Вариант 14

```
if (a+d < (7-f+3)/d) {
  if (a > b || c==0) x=a+b;
  else x=a-d*7-d; }
else
  z=k+h*7/(d-a-3);
if (z/b != n*2) {
  while (i <= n)
    i=i+2-d*h;
  c=(z+f)/3; }
```

Вариант 15

```
if (d*d > a/2 || f != a+b) {
  for (k=3; k <= a; k=k+2)
    if (k >= f-k/2) {
      a=f+(w*3-4)/b+t;
      b=(x+y)*(x-y); } }
else {
  x=(a-g)*2-r/(4+5*k);
  if (x == 7) c=a/2*(s/3-6);
  else d=0;
}
```

9 Статическое распределение памяти для данных

9.1 Блочная структура программ и область видимости переменных

В большинстве алгоритмических языков принята блочная структура программ, при которой область видимости переменных, описанных в каком-либо блоке, распространяется только на этот блок и блоки, вложенные в него.

Пример программы с блочной структурой на языке С представлен на рисунке 9.1.

```
main()
  { int a,b; m1:
     { int a1,b1; m2:

        { int d; m3: ... ; }

     m4:{ int d1; m5: ... ; }

     m6: ... ;
     }
  m7:{int f,e; m8: ... ; }

  m9: ... ;
  }
```

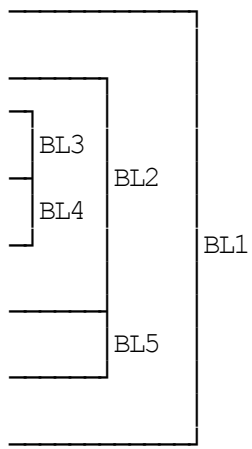


Рисунок 9.1 – Пример программы с блочной структурой

Допустим, память для данных распределяется с адреса k, и длина переменных типа int – 4 байта. Тогда состояние памяти при выполнении программы меняется в зависимости от видимости переменных, как это представлено в таблице 9.1.

Таблица 9.1 – Состояние памяти программы

адрес	**метка**	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9
k+0		a	a	a	a	a	a	a	a	a
k+4		b	b	b	b	b	b	b	b	b
k+8			a1	a1	a1	a1	a1		f	
k+12			b1	b1	B1	b1	b1		e	
k+16				d		d1				

Таким образом, одни и те же области памяти при работе программы могут быть использованы для размещения различных переменных.

Блочная структура программы может быть представлена в виде дерева, как это показано на рисунке 9.2.

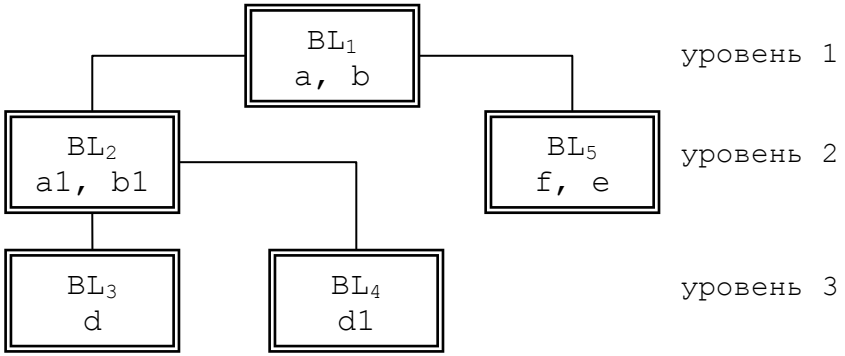


Рисунок 9.2 – Дерево блоков программы

Узлы дерева соответствуют блокам и расположены на разных уровнях. Внешнему блоку приписан уровень 1. Блоку, содержащемуся в блоке уровня r, приписывается уровень r+1.

9.2 Распределение памяти для блоков программы

Статическое распределение памяти для данных проводится в два этапа. На первом этапе выделяется секция памяти каждому блоку, на втором – назначаются адреса переменным внутри блока.

На первом этапе распределения памяти каждому блоку программы присваивается номер i в соответствии с последовательностью их описания в программе. Определяются величины n[i] – размер секции памяти i-го блока (n[1] = n[2] = n[5] = 8 байт, n[3] = n[4] = 4 байта) и строится таблица блоков.

Для приведенного примера программы таблица блоков имеет вид, представленный в таблица 9.2.

Таблица 9.2. – Таблица блоков программы

Номер блока	Уровень блока (УБ[i])	Объем памяти для блока (n[i])
1	1	8
2	2	8
3	3	4
4	3	4
5	2	8

Затем по таблице блоков определяются адреса начала секций памяти для блоков (A[i]). Для первого блока принимается A[1] = k – адрес начала области данных программы. Для каждого последующего блока i величина A[i] определяется как

A[i] = A[m] + n[m], где m = max(j | УБ[j]<УБ[i], j<i)

То есть, для определения m последовательно просматриваются строки таблицы блоков j = i-1,i-2,...,1 и ищется первая строка, для которой УБ[j] < УБ[i].

Результат реализации этого алгоритма представлен в таблице 9.3.

Таблица 9.3 – Адреса начала секций блоков

Адрес начала секции блока (A[i])
k
k + 8
k + 16
k + 16
k + 8

9.3 Распределение памяти для переменных и массивов

На втором этапе секция данных каждого блока распределяется по отдельным переменным и массивам. Для каждого блока выделяются счетчики T[i], указывающие текущий адрес в секции. Первоначально T[i] = A[i].

Просматривается таблица идентификаторов и для каждой переменной или массива извлекается номер блока – i и длина переменной – d. Величина T[i] принимается в качестве адреса переменной и корректируется T[i] = T[i] + d.

Результат распределения памяти для переменных приведен в таблице 9.4.

Таблица 9.4 – Распределение памяти для переменных программы

Переменная	Номер блока	Длина	Адрес	T[1]	T[2]	T[3]	T[4]	T[5]
				k	k+8	k+16	k+16	k+8
a	1	4	k	k+4				
b	1	4	k+4	k+8				
a1	2	4	k+8		k+12			
b1	2	4	k+12		k+16			
d	3	4	k+16			k+20		
d1	4	4	k+16				k+20	
f	5	4	k+8					k+12
e		4	k+12					k+16

Приведенное распределение памяти соответствует распределению, представленному в таблице 9.1 и.полученному в результате анализа хода выполнения программы.

Лабораторная работа

Цель: получение практических навыков в использовании методов статического распределения памяти для данных.

Материалы и оборудование: персональный компьютер.

Определить адреса переменных при статическом распределении памяти. Начальный адрес области размещения переменных – 2000. Длины переменных различных типов: char – 1, short – 2, int – 4, long – 4, float – 4, double – 8.

Вариант 1

```
main()
{ int a; double b[3];
   { char c; long a;
      { char b[6]; float x,y;
         { short c;          }
      }
      { int a; long b[2];
         { char x[4];        }
         { double x[2];      }
   }  }
   { int x,y,z; char c;
      { long a[2];
         { float c;          }
}  }  }
```

Вариант 2

```
main()
{ char e[4]; short d;
   { long h[4]; int n;        }
   { char g[6]; int z;
      { int s;
         { float q,w[3];      }
      }
      { int v[4];
         { char w[4]; int k;  }
         { short b;           }
   }  }
   { char m[4]; int a;        }
   { float s[2]; char i[8];   }
}
```

Вариант 3

```
main()
{ char r; double g;
   { float f[2];              }
   { int g[3]; short t[3];
      { long t;
         { char h[2];         }
         { int y,n;           }
   }  }
   { float e; int y;
      { double y;             }
```

Вариант 4

```
main()
{ int v,b; float n;
   { char f[2]; double t;
      { int b;
         { int g[3];          }
         { char r,n[4];       }
         { long g; short m;   }
   }  }
   { int g; short h;
      { float h[4]; int y;    }
```

```
      { char m;                          { int m; double n;     }
        { char m[4]; int n;}             { int f[5];            }
}  }  }                              }  }
```

Вариант 5

```
main()
{ double g; char t;
   { char f[3];
      { long t;
         { char y[4]; int n; }
      }
      { int r; short n;      }
      { long f[3]; char w;
         { int y,n;          }
         { float r[3];
            { double d;      }
         }
         { int x; short y,z; }
}  }  }
```

Вариант 6

```
main()
{ int n; double m;
{ int n[4];
   { char d,n; short k;
      { long f[2]; int y;
         { float e[3];     }
         { char f; int r;  }
         { double v[2];    }
      }
      { double f; int g;
         { long g; char e; }
      }
      { short f[5];        }
}  }  }
```

Вариант 7

```
main()
{ short f[2]; char w;
   { char g[4];
      { long f[2]; int r;   }
      { double f[2];        }
   }
   { char d; int r,y;
      { int g;              }
      { char y[2]; short f;
         { float v;         }
         { int v,b;         }
      }
      { int b[3];           }
}  }
```

Вариант 8

```
main()
{ int b[2]; short a[3];
   { char g[4]; int r,u;
      { double b;
         { char f,g;
            { long h; int b;  }
         }
         { int n[3];          }
         { char b; int j,i;   }
      }
      { long h[6]; float g;
         { char d[4];
            { short f; int n; }
}  }  }  }
```

Вариант 9

```
main()
{ int g[7]; char f,n;
   { int m[3]; float d
      { int y; short n;
         { char c;        }
      }
      { char f[5]; short k;
         { int b,n;       }
   }  }
   { float r[3]; int a;
      { char s[2];        }
      { double z[2];
         { long z,x;      }
}  }  }
```

Вариант 10

```
main()
{ int d[3]; char h;
   { long c; int a;
      { char d[3];
         { long s,v;      }
         { float x,c;     }
   }  }
   { int f[3]; char c,v;
      { int l;            }
      { short z;
         { long c[4];     }
         { int z,v;       }
         { double f[5];   }
}  }  }
```

Вариант 11

```
main()
{ int s; char b[5];
   { long k,b; char s;
      { double p[2];      }
      { float c;          }
   }
   { int m,n; short a;
      { char b[3]; double z;
         { short c;
            { int m[2];   }
         }
      }
      { float b; char t;
         { int x[4];      }
         { float x[3];    }
}  }  }
```

Вариант 12

```
main()
{ char e[4],f,t;
   { int a,b,c;                }
   { char c; short i[3];
      { double s; int t;       }
      { long f[2];
         { int d;              }
         { long k,h;           }
      }
   {  float e; char e[3];      }
   {  int j,i;
      { float z; double t;
         { int a;              }
      }
      { char s[5]; int d;      }
}  }
```

Вариант 13

```
main()
{ int r; char s[6];
   { float d; short t[3];
      { int h;              }
   }
   { short k; short j[3];
      { double u; int f;
         { char h,f,d;      }
         { long n;          }
   }  }
   { short x[3]; int y;
      { double y; char b;
         { int v[2];
            { char m;       }
}  }  }  }
```

Вариант 14

```
main()
{ float v[2]; double b;
   { char s[5]; int z;      }
   { short z,h;
      { char m[2],n;        }
      { long b,a;
         { short k;         }
         { char s[3]; int d; }
         { double m[3];     }
   }  }
   { float z[2],b;
      { short x; int y;
         { long m[5];       }
         { int b; char n[3]; }
}  }  }
```

Вариант 15

```
main()
{ short d; char b[7];
   { char c,d,t; int z;
      { float x,y;
         { short c; int f;  }
         { double x[2];     }
      }
      { int q; long b;      }
   }
   { short c[4];            }
   { int a,b; char c;
      { long a[2];
         { double c,h;      }
         { int c[3];        }
}  }  }
```

10 Динамическое распределение памяти для данных

10.1 Общие принципы определения адресов данных

При динамическом распределении памяти на этапе трансляции память распределяется для данных только внутри блоков, относительно начала области данных каждого блока. Это производится в ходе просмотра таблицы идентификаторов также, как и при статическом распределении, только начальные адреса областей данных для всех блоков принимаются равными T[i] = LS, где LS – длина служебной информации, размещаемой в начале области данных каждого блока. В процессе выполнения программы при входе в блок (в том числе и в процедуру) выделяется память для области данных блока.

Динамическое распределение памяти будет рассматриваться на примере программы, представленной на рисунке 10.1.

```
program PR;
var  a: real;
   procedure P1;
   var  x,y,z: real;
   begin ... end;
   procedure P2;
   var  g: real;
      procedure P3;
      var b,a: real;
      begin ... P1; ... end;
   begin ... P3; ... end;
begin ... P2 ... end.
```

Дерево блоков для данной программы определяет статические цепочки вложенности блоков:

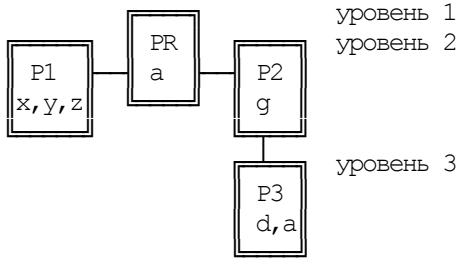


Рисунок 10.1 – Программа и дерево блоков

Последовательность работы блоков определяет динамическую цепочку вызова блоков, представленную на рисунке 10.2.

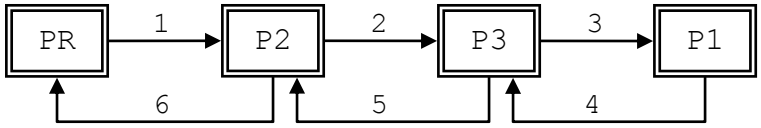


Рисунок 10.2 – Динамическая цепочка вызова блоков

При динамическом распределении памяти адрес любой переменной определяется как сумма базового адреса области данных блока и смещения переменной от начала этой области. Для хранения адресов областей данных используется массив указателей начала областей данных блоков (обычно базовых регистров) – УНБ[1], УНБ[2], ... В соответствии с правилами определения областей действия имен переменных при работе блока уровня i должны быть загружены указатели УНБ[1]...УНБ[i] в соответствии со статической цепочкой вложенности блоков.

Так, если обозначить [V] – адрес начала области данных блока V, то после входа в блок PR должно быть: УНБ[1]=[PR], после входа в блок P2: УНБ[1]=[PR], УНБ[2]=[P2], после входа в блок P3: УНБ[1]=[PR], УНБ[2]=[P2], УНБ[3]=[P3]; после входа в блок P1: УНБ[1]=[PR], УНБ[2]=[P1].

10.2 Реализация динамического распределения памяти в процессе выполнения программы

Средством реализации динамического распределения памяти является стек.

Обозначим:

УС – указатель стека;

k – номер блока;

`УБ[k]` – уровень блока k;

`УБТ` – уровень текущего блока;

`n[k]` – размер области данных блока k;

`УНБО[k]` – значение указателя начала области данных блока, в котором описана процедура или содержится блок k;

`УНБВ[k]` – значение указателя начала области данных блока, из которого произошел вызов процедуры или вход в блок k;

`УБВ[k]` – уровень блока, из которого произошел вызов процедуры или вход в блок k.

Связующей информацией блока k называется тройка значений (УНБО[k], УНБВ[k], УБВ[k]). Эта информация помещается в начало области данных каждого блока. Для самого внешнего блока связующей информацией считается (0,0,0).

При входе в самый внешний блок значения УС и УНБ[1] устанавливаются на начало стека, для уровня текущего блока принимается начальное значение – УБТ = 1. Далее, при входе в любой блок k выполняется подпрограмма:

```
ВХОД: СТЕК(УС) = (УНБ[УБ[k]-1], УНБ[УБТ], УБТ);
      УБТ = УБ[k];
      УНБ[УБ[k]] = УС;
      УС = УС + n[k]+ LS; // LS - длина связующей
                          // информации
```

Для приведенного примера после входов в процедуры P2, P3, P1 имеем состояние стека, представленное на рисунке 10.3.

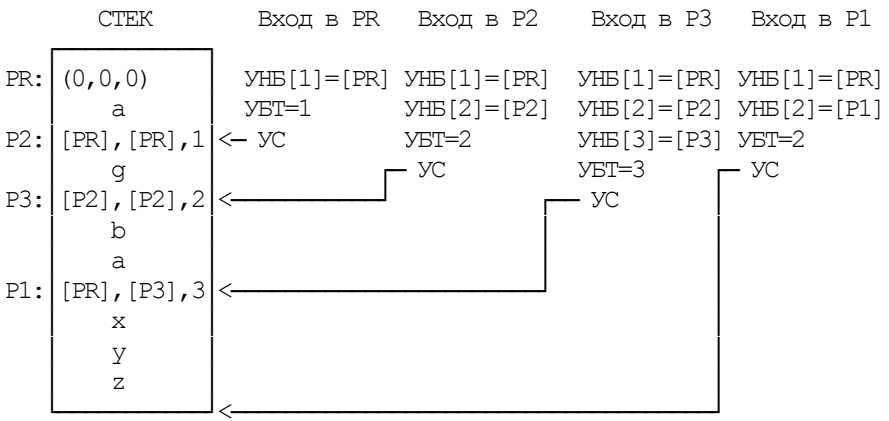


Рисунок 10.3 – Состояние стека после входа в процедуры P2, P3, P1

Первые элементы связующей информации определяют цепочку адресов областей данных блоков в соответствии со статической цепочкой вложенности блоков (P1→PR, P3→P2→PR). Вторые элементы определяют цепочку адресов в соответствии с динамической цепочкой вызова блоков (P1→P3→P2→PR).

При выходе из процедуры нужно восстановить УС, УБТ и цепочку УНБ. Алгоритм подпрограммы ВЫХОДПР:

1) восстанавливается УС = УНБ[УБТ] и из стека извлекаются УНБО[k], УНБВ[k], УБВ[k];
2) восстанавливается УБТ = УБВ[k];
3) если УБТ = 1, то конец, т.к. УНБ[1] никогда не модифицируется;
4) по динамической цепочке восстанавливается последний указатель УНБ[УБТ] = УНБВ[k];
5) для блоков на уровнях j = УБТ-1,УБТ-2,...,2 по статической цепочке восстанавливается УНБ, как это показано на рисунке 10.4.

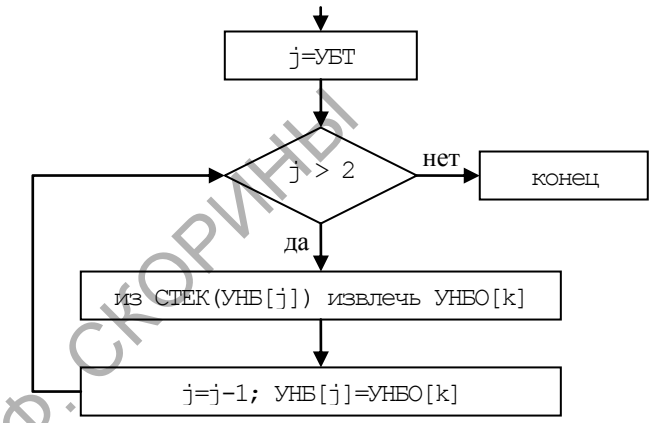


Рисунок 10.4 – Алгоритм восстановления УНБ

Таким образом, для примера восстанавливается:

```
УНБ[1]=[PR]      УБТ = 3
УНБ[2]=[P2]
УНБ[3]=[P3]
```

Для приведенного примера после выхода из процедур P1, P3, P2 имеем состояние стека, представленное на рисунке 10.5.

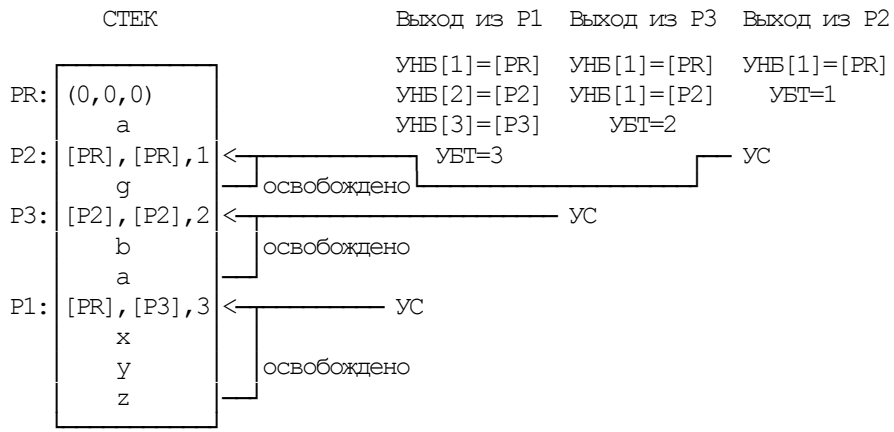


Рисунок 10.5 – Состояние стека после выхода из процедур P1, P3, P2

Выход из любого блока, не являющегося вызванной процедурой, по концу работы блока реализуется подпрограммой:

```
ВЫХОДБЛК:
          УС = УНБ[УБТ];
          УБТ = УБТ-1;
```

В частности, для приведенного примера с помощью подпрограммы ВЫХОДБЛК реализуется выход из блока PR по концу работы всей программы.

Лабораторная работа

Цель: получение практических навыков в использовании методов динамического распределения памяти для данных.

Материалы и оборудование: персональный компьютер.

Для заданной программы на языке Паскаль определить состояние стека данных, указателя стека и указателей начала областей данных блоков после входа в каждую процедуру и выходов из процедур. В процессе выполнения программы осуществляется динамическое распределение памяти для данных.

Вариант 1

```
program P;
var  a: real;
   procedure P1;
   begin ... end;
   procedure P2;
   var  a,b2: real;
      procedure P3;
      var  a: real;
      begin ... P1; ... end;
      procedure P4;
      var  d: real;
      begin ... P3; ... end;
   begin ... P4; ... end;
begin ... P1; P2; ... end.
```

Вариант 2

```
program P;
var  x,y: real;
   procedure P1;
      procedure P3;
      var  a,b,c: real;
      begin ... P2; ... end;
   begin ... P3; ... end;
   procedure P2;
   var  a: real;
      procedure P4;
      var  a1,a2: real;
      begin ... end;
   begin ... P4; ... end;
begin ... P1; P2; ... end.
```

Вариант 3

```
program P;
var  a,b,c: real;
   procedure P1;
   var  x: real;
   begin ... end;
   procedure P2;
   var  a,aa: real;
      procedure P4;
      begin ... P1; ... end;
   begin ...P3; P4; ... end;
   procedure P3;
   var  bb: real;
   begin ... end;
begin ... P2; P3; ... end.
```

Вариант 4

```
program P;
var  x,y,z: real;
   procedure P1;
   var  a: real;
      procedure P2;
      var  b,c: real;
         procedure P4;
         var  x: real;
         begin ... P3; ... end;
      begin ...P3; P4; ... end;
      procedure P3;
      begin ... end;
   begin ... P3; P2; ... end;
begin ... P1; ... end.
```

Вариант 5

```
program P;
var  x,y: real;
   procedure P1;
   var  a,b,c: real;
      procedure P2;
      begin ... end;
      procedure P3;
      var  x,c: real;
      begin ... P2; P4; ... end;
   begin ... P2; P3; ... end;
   procedure P4;
   var  a: real;
   begin ... end;
begin ... P1; ... end.
```

Вариант 6

```
program P;
var  a: real;
   procedure P1;
      procedure P4;
      var  x,y,z: real;
      begin ... P3; ... end;
   begin ... P4; ... end;
   procedure P2;
   var  d: real;
   begin ... P3; ... end;
   procedure P3;
   var  x,y: real;
   begin ... end;
begin ... P3; P2; P1; ... end.
```

Вариант 7

```
program P;
var  a: real;
   procedure P1;
   var  a,n: real;
      procedure P2;
```

Вариант 8

```
program P;
var  y,z: real;
   procedure P1;
   var  x,y: real;
   begin ... end;
```

```
      var  x: real;
      begin ... end;
      procedure P3;
      var  z: real;
         procedure P4;
         begin ... P2; ... end;
      begin ... P4; ... end;
   begin ... P3; P2; ... end;
begin ... P1; ... end.
```

Вариант 9

```
program P;
var  a: real;
   procedure P1;
   var  x,y,a: real;
      procedure P3;
      var  b: real;
      begin ... P2; ... end;
   begin ... P2; P3; ... end;
   procedure P2;
   var  g: real;
      procedure P4;
      begin ... end;
   begin ... P4; ... end;
begin ... P1; ... end.
```

Вариант 11

```
program P;
var  d,k: real;
   procedure P1;
   var  m,h: real;
      procedure P2;
      begin ... end;
      procedure P3;
      var  z: real;
         procedure P4;
         var  a,t: real;
         begin ... P2; ... end;
      begin ... P4; ... end;
```

```
      procedure P2;
      var  a,b,c: real;
      begin ... P3; P1; ... end;
      procedure P3;
         procedure P4;
         var  a: real;
         begin ... P1; ... end;
      begin ... P4; ... end;
 begin ... P2; ... end.
```

Вариант 10

```
program P;
var  a,b,c: real;
   procedure P1;
   var  a,x: real;
      procedure P3;
      var  y: real;
      begin ... P2; P4; ... end;
      procedure P4;
      begin ... end;
   begin ... P3; P4; ... end;
   procedure P2;
   var  x: real;
   begin ... end;
begin ... P2; P1; ... end.
```

Вариант 12

```
 program P;
    var  s,n: real;
    procedure P1;
    var  a,f,h: real;
       procedure P2;
       var  b,c: real;
          procedure P4;
          begin ... P3; ... end;
       begin ...P4; P3; ... end;
       procedure P3;
       var  x,k: real;
       begin ... end;
```

```
   begin ... P2; P3; ... end;
begin ... P1; ... end.
```

Вариант 13

```
program P;
var  s,y: real;
   procedure P1;
   var  a,d,v: real;
      procedure P4;
      var  z: real;
      begin ... P3; ... end;
   begin ... P4; ... end;
   procedure P2;
   var  g,f: real;
   begin ... P3; ... end;
   procedure P3;
   begin ... end;
begin ... P1; P2; ... end.
```

Вариант 15

```
program P;
var  d: real;
   procedure P1;
   begin ... end;
   procedure P2;
   var  q,s,c: real;
      procedure P4;
      var  z,k: real;
      begin ... P3; ... end;
   begin ...P4; P1; ... end;
   procedure P3;
   var  e,h: real;
   begin ... end;
begin ... P3; P2; ... end.
```

```
   begin ... P2; ... end;
begin ... P1; ... end.
```

Вариант 14

```
program P;
var  s,h,k: real;
   procedure P1;
   var  i,j: real;
   begin ... end;
   procedure P2;
   var  a: real;
      procedure P3;
      begin ... P4; ... end;
      procedure P4;
      var  a,b: real;
      begin ... P1; ... end;
   begin ... P3; ... end;
begin ... P2; P1; ... end.
```

Литература

1 Ахо, А. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Д. Ульман. – М.: Вильямс, 2003. – 768 с.

2 Опалева, Э.А. Языки программирования и методы трансляции / Э.А. Опалева, В.П. Самойленко. – СПб.: BHV, 2005. – 480 с.

3 Свердлов, С.З. Языки программирования и методы трансляции / С.З. Свердлов. – СПб.: Питер, 2007. – 640 с.

4 Рейуорд-Смит, В.Дж. Теория формальных языков / В.Дж. Рейуорд-Смит. – М.: Радио и связь, 1988. – 378 с.

Для записей

Учебное издание

Короткевич Владимир Аполлонович
Короткевич Людмила Ивановна

ЭВМ И ПРОГРАММИРОВАНИЕ

ПРАКТИЧЕСКОЕ РУКОВОДСТВО
по изучению темы «Методы разработки компиляторов для языков программирования» для студентов специальности 1-31 03 03-01 «Прикладная математика (научно-производственная деятельность)»

В авторской редакции

Подписано в печать 24.04.2009 (43). Формат 60x80 1/16. Бумага писчая №1. Гарнитура «Таймс». Усл. печ.л. 4,8. Уч.-изд.л. 3,7. Тираж 25 экз.

Отпечатано в учреждении образования
«Гомельский государственный университет
имени Франциска Скорины»
246019, г. Гомель, ул. Советская, 104