Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет информационных технологий и управления

Кафедра интеллектуальных информационных технологий

# *ЯЗЫКОВЫЕ ПРОЦЕССОРЫ ИНТЕЛЛЕКТУАЛЬНЫХ СИСТЕМ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ*

*Рекомендовано УМО по образованию в области
информатики и радиоэлектроники в качестве пособия
для специальности 1-40 03 01 «Искусственный интеллект»*

Минск БГУИР 2018

Библиотека БГУИР

УДК 004.43:004.8(076.5)
ББК 32.973.26-018.1я73+32.813я73
Я41

Авторы:
В. В. Голенков, Н. А. Гулякина, И. Т. Давыденко, Д. В. Шункевич

Рецензенты:
кафедра интеллектуальных систем Белорусского государственного университета (протокол №8 от 24.01.2017);

ведущий научный сотрудник государственного научного учреждения «Объединенный институт проблем информатики Национальной академии наук Беларуси», кандидат технических наук, доцент В. И. Романов

Я41 **Языковые** процессоры интеллектуальных систем. Лабораторный практикум : пособие / В. В. Голенков [и др.]. – Минск : БГУИР, 2018. – 111 с. : ил.
ISBN 978-985-543-357-7.

Сформулированы основные положения, касающиеся лабораторного практикума по дисциплине «Языковые процессоры интеллектуальных систем», даны рекомендации по выполнению лабораторных работ, рассмотрены примеры решения задач, поставленных в рамках лабораторных работ.

**УДК 004.43:004.8(076.5)**
**ББК 32.973.26-018.1я73+32.813я73**

**ISBN 978-985-543-357-7**

© УО «Белорусский государственный университет информатики и радиоэлектроники», 2018

Библиотека БГУИР

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ ....................................................................................................4
ЛАБОРАТОРНАЯ РАБОТА №1 ПОСТРОЕНИЕ КОНЕЧНОГО АВТОМАТА...5
1.1 Теоретические сведения...............................................................................5
1.2 Варианты индивидуальных заданий.........................................................18
ЛАБОРАТОРНАЯ РАБОТА №2 ПОСТРОЕНИЕ КС-ГРАММАТИКИ.............19
2.1 Теоретические сведения.............................................................................19
2.2 Варианты индивидуальных заданий.........................................................28
ЛАБОРАТОРНАЯ РАБОТА №3 ПОСТРОЕНИЕ ТАБЛИЦЫ
ПРЕДИКТИВНОГО АНАЛИЗАТОРА.............................................................32
3.1 Теоретические сведения.............................................................................32
3.2 Варианты индивидуальных заданий.........................................................35
ЛАБОРАТОРНАЯ РАБОТА №4 ПОСТРОЕНИЕ ТАБЛИЦЫ
SLR-АНАЛИЗАТОРА ......................................................................................39
4.1 Теоретические сведения.............................................................................39
4.2 Варианты индивидуальных заданий.........................................................54
ЛАБОРАТОРНАЯ РАБОТА №5 ГЕНЕРАЦИЯ И ОПТИМИЗАЦИЯ КОДА.....57
5.1 Теоретические сведения.............................................................................57
5.2 Варианты индивидуальных заданий.........................................................75
ЛАБОРАТОРНАЯ РАБОТА №6 РАЗРАБОТКА ФОРМАЛЬНОЙ
СПЕЦИФИКАЦИИ ЯЗЫКА ПРОГРАММИРОВАНИЯ...................................78
6.1 Теоретические сведения.............................................................................78
6.2 Варианты индивидуальных заданий.........................................................82
ЛАБОРАТОРНАЯ РАБОТА №7 РАЗРАБОТКА КОМПИЛЯТОРА ЯЗЫКА
ПРОГРАММИРОВАНИЯ.................................................................................84
7.1 Теоретические сведения.............................................................................84
7.2 Практическая часть.....................................................................................87
Приложение А Содержимое файла грамматики на первом этапе.....................100
Приложение Б Содержимое файла грамматики на заключительном этапе.......104
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ ..........................................110

Библиотека БГУИР

## ВВЕДЕНИЕ

В настоящее время активно ведется разработка новых и улучшение существующих языков программирования и соответствующих им технологий компиляции и интерпретации, что приводит к необходимости понимания специалистом в области информационных технологий принципов построения и функционирования таких языков и соответствующих инструментальных средств. Изучение конкретных языков программирования и средств проектирования программ без понимания принципов их работы приводит к возникновению трудностей при освоении новых технологий и, как следствие, увеличению времени, необходимого для адаптации к тем или иным незнакомым средствам, что негативно сказывается на общем уровне профессионализма молодого специалиста. Основной задачей дисциплины «Языковые процессоры интеллектуальных систем» является решение данной проблемы.

**Целью** изучения дисциплины «Языковые процессоры интеллектуальных систем» является рассмотрение теории построения языковых процессоров, методов анализа и проектирования языков программирования различного назначения, в том числе используемых в интеллектуальных системах.

В результате изучения дисциплины студенты должны

**знать:**

- основные концепции языков программирования;
- теорию построения компиляторов;
- методы и способы формального определения синтаксиса и семантики языков различного назначения;

**уметь:**

- разрабатывать основные блоки языковых процессоров;
- разрабатывать формальные спецификации языков различного назначения.

Библиотека БГУИР

# ЛАБОРАТОРНАЯ РАБОТА №1
# ПОСТРОЕНИЕ КОНЕЧНОГО АВТОМАТА

*Цель работы*: получить навык построения конечного автомата.

## 1.1 Теоретические сведения

***Регулярные выражения*** (англ. *regular expressions*) – формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов (символов-джокеров, англ. *wildcard characters*). По сути это строка-образец (англ. pattern, по-русски ее часто называют «шаблоном», «маской»), состоящая из символов и метасимволов и задающая правило поиска.

***Регулярный язык (регулярное множество)*** в теории формальных языков – это формальный язык, который может быть выражен средствами регулярных выражений. Класс регулярных множеств удобно изучать в целом, а полученные результаты оказываются применимы для достаточно широкого спектра формальных языков.

Пусть Σ – конечный алфавит. Регулярный язык R(Σ) в алфавите Σ определяется следующими рекурсивными свойствами (таблица 1.1).

Таблица 1.1 – Рекурсивные свойства трегулярного языка R(Σ) в алфавите Σ

| Свойство | Описание |
|---|---|
| $\varnothing \in R(\Sigma)$ | Пустое множество является регулярным множеством в алфавите Σ |
| $\{\varepsilon\} \in R(\Sigma)$ | Множество, состоящее из одной лишь пустой строки, является регулярным множеством в алфавите Σ |
| Для любого $a \in \Sigma: \{a\} \in R(\Sigma)$ | Множество, состоящее из одного любого символа алфавита Σ, является регулярным множеством в алфавите Σ |
| $P \in R(\Sigma) \wedge Q \in R(\Sigma) \rightarrow (PUQ) \in R(\Sigma)$ | Если два какие-либо множества являются регулярными в алфавите Σ, то и их объединение тоже является регулярным множеством в алфавите Σ |
| $P \in R(\Sigma) \wedge Q \in R(\Sigma) \rightarrow PQ \in R(\Sigma)$ | Если два какие-либо множества являются регулярными в алфавите Σ, то и множество, составленное из всевозможных сцеплений пар их элементов, тоже является регулярным множеством в алфавите Σ |
| $P \in R(\Sigma) \rightarrow P^* \in R(\Sigma)$ | Если какое-либо множество является регулярным в алфавите Σ, то множество всевозможных сцеплений его элементов тоже является регулярным множеством в алфавите Σ |

Библиотека БГУИР

Ничто другое, кроме следующего из перечисленного, не является регулярным множеством в алфавите Σ.

Регулярное выражение строится из более простых регулярных выражений с использованием набора правил. Каждое регулярное выражение r обозначает (задает) язык L(r). Правила определяют, каким образом из языков, заданных подвыражениями r, формируется L(r).

После любого элемента регулярного выражения может следовать очень важный тип метасимвола – повторитель.

```
*      нуль или более раз, то же, что {0,…,∞}
+      один или более раз, то же, что {1,…,∞}
?      нуль или один раз, то же, что {0,1}
{n}    точно n раз
{n,}   не менее n раз
{n,m}  не менее n но не более m раз
```

Рассмотрим правила, которые определяют регулярные выражения над алфавитом E.

1. E представляет собой регулярное выражение, обозначающее {e}, т. е. множество, содержащее пустую строку.

2. Если a является символом из E, то a – регулярное выражение, обозначающее {a}, т. е. множество, содержащее строку a. Хотя мы используем одну и ту же запись, технически регулярное выражение a отличается от строки a и символа a. Говорим мы о регулярном выражении, строке или символе – становится понятно из контекста.

3. Предположим, что r и s – регулярные выражения, обозначающие языки L(r) и L(s), тогда:

- (r)|(s) представляет собой регулярное выражение, обозначающее L(r) и L(s);
- (r)(s) – регулярное выражение, обозначающее L(r)L(s);
- (r)* – регулярное выражение, обозначающее (L(r))*;
- (r) – регулярное выражение, обозначающее L(r).

Излишние скобки в регулярном выражении могут быть устранены, если принять следующие соглашения.

1. Унарный оператор * имеет высший приоритет и левоассоциативен.

2. Конкатенация имеет второй по значимости приоритет и левоассоциативна.

3. | (объединение) имеет низший приоритет и левоассоциативно.

При этих соглашениях запись (a)|((b)*(c)) эквивалентна a|b*c.

Библиотека БГУИР

Рассмотрим следующий пример.

Пусть E = {a,b}.

1. Регулярное выражение a|b обозначает множество {a,b}.

2. Регулярное выражение (a|b)(a|b) – {aa,ab,ba,bb} обозначает множество всех строк из a и b длиной 2. Другое регулярное выражение для того же множества – aa|ab|ba|bb.

3. Регулярное выражение a* обозначает множество всех строк из нуля или более a, т.е. {e,a,aa,aaa,…}.

4. Регулярное выражение (a|b)* обозначает множество всех строк, содержащих нуль или несколько экземпляров a и b, т. е. множество всех строк из a и b. Другое регулярное выражение для этого множества – (a*b*)*.

5. Регулярное выражение a|a*b обозначает множество, содержащее строку a и все ее строки, состоящие из нуля или нескольких a, за которыми следует b.

6. Регулярное выражение (a|b)*(a|b)(a|b)* обозначает множество всех непустых цепочек, состоящих из a и b, т. е. множество {a,b}+.

Если два регулярных выражения r и s задают один и тот же язык, то r и s называются эквивалентными, т. е. r = s. Например, (a|b) = (b|a).

Имеется ряд алгебраических законов, используемых для преобразования регулярных выражений в эквивалентные. Ниже приведены некоторые из этих законов для регулярных выражений r, s и t (таблица 1.2).

Таблица 1.2 – Алгебраические свойства регулярных выражений

<table>
<tr><th>Аксиома</th><th>Описание</th></tr>
<tr><td>r|s = s|r</td><td>Оператор | коммутативен</td></tr>
<tr><td>r|(s|t) = (r|s)|t</td><td>Оператор | ассоциативен</td></tr>
<tr><td>(rs)t = r(st)</td><td>Конкатенация ассоциативна</td></tr>
<tr><td>r(s|t) = rs|rt<br>(s|t)r = sr|tr</td><td>Конкатенация дистрибутивна над |</td></tr>
<tr><td>er = r<br>re = r</td><td>e является единичным элементом по отношению к конкатенации</td></tr>
<tr><td>r* = (r|e)*</td><td>Связь между * и e</td></tr>
<tr><td>r** = r*</td><td>Оператор * идемпотентен</td></tr>
</table>

Регулярные выражения, введенные ранее, используются для описания регулярных множеств. Для распознавания регулярных множеств служат конечные автоматы (КА).

***Конечный автомат*** – это преобразователь, который позволяет сопоставить входу соответствующий выход, причем выход этот может зависеть не только от текущего входа, но и от того, что происходило раньше, предыстории работы конечного автомата.

Библиотека БГУИР

КА на рисунке 1.1 состоит:

- из ленты, представленной входной цепочкой;
- считывающего устройства;
- блока управления, который содержит список правил переходов.

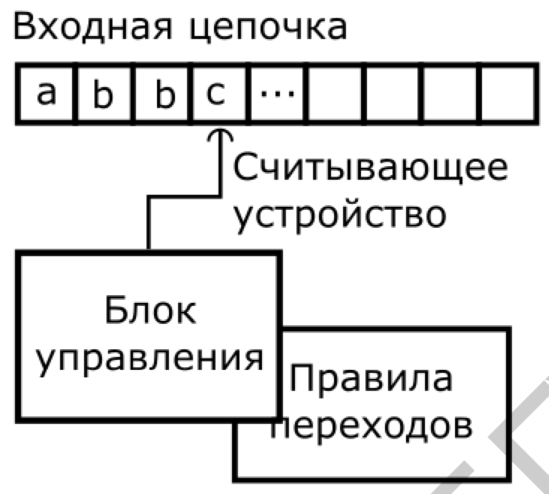



Рисунок 1.1 – Пример работы примитивного КА

Считывающее устройство может двигаться в одном направлении, как правило, слева направо, тем самым считывая символы входной цепочки. За каждое такое движение оно может считать один символ. Далее считанный символ передается блоку управлений. Блок управления изменяет состояние автомата на основе правил переходов. Если список правил переходов не содержит правила для считанного символа, то автомат «умирает».

***Способы задания конечного автомата.*** КА могут задаваться в виде графов или в виде управляющих таблиц.

В виде графа КА задается следующим способом:

- вершины графа соответствуют состояниям КА;
- направленные ребра соответствуют функциям переходов (возле каждого такого ребра указывается символ, по которому выполняется переход);
- вершина с входящим в него ребром, которое не выходит ни из одного состояния, соответствует начальному состоянию;
- конечные состояния КА помечаются двойным контуром.

В виде управляющей таблицы КА задается так:

- состояния КА располагаются в строках таблицы;
- символы распознаваемого языка располагаются в столбцах;
- на пересечении указывается состояние, в которое можно попасть из данного состояния по данному символу.

Библиотека БГУИР

***Распознавателем*** языка называется программа, которая получает на вход строку x и отвечает «да», если x – предложение языка, или в противном случае «нет».

***Недетерминированный конечный автомат (НКА)*** – это пятерка M = = (Q, T, D, $q_0$, F), где:

- Q – конечное множество состояний;
- T – конечное множество допустимых входных символов (входной алфавит);
- D – функция переходов (отображающая множество Q(T ∪ {e}) во множество подмножеств множества Q), которая определяет поведение управляющего устройства;
- $q_0$ ∈ Q – начальное состояние управляющего устройства;
- F ⊆ Q – множество заключительных состояний.

*Недетерминизм* автомата заключается в том, что, во-первых, находясь в некотором состоянии и обозревая текущий символ, автомат может перейти в одно из нескольких возможных состояний и, во-вторых, автомат может делать переходы по e.

***Детерминированный конечный автомат (ДКА)*** является специальным случаем недетерминированного конечного автомата, в котором:

- отсутствуют состояния, имеющие e-переходы;
- для каждого состояния s и входного символа a существует не более одной дуги, исходящей из s и помеченной как a.

ДКА имеет для любого входного символа не более одного перехода из каждого состояния. Если для представления функции переходов ДКА используется таблица, то каждая запись в ней представляет собой единственное состояние. Следующий алгоритм имитирует поведение ДКА при обработке входной строки.

*Основное отличие ДКА от НКА* состоит в том, что ДКА в процессе работы может находиться только в одном состоянии, а НКА в нескольких состояниях одновременно.

**Алгоритм «Моделирование ДКА»**

**Вход:** входная строка x, завершаемая символом конца файла eof, и ДКА D со стартовым состоянием $s_0$ и множеством заключительных состояний F.

**Выход:** «да», если D допускает x, и «нет» в противном случае.

**Метод.** Ко входной строке x применяется алгоритм. Функция move(s,c) дает состояние, в которое происходит переход из состояния s при входном символе c. Функция nextchar возвращает очередной символ строки x:

Библиотека БГУИР

```
S:=s0;
C:= nextchar;
While c≠ eof do begin
S:= move(s,c);
C:= nextchar;
End;
If s ∈ F then
Return “yes”
Else return “no”
```

**Построение ДКА из НКА (построение подмножества)**

**Вход.** НКА N.

**Выход.** ДКА D, допускающий тот же язык.

**Метод.** Данный алгоритм строит таблицу переходов Dtran так, чтобы D описывал «параллельно» все возможные перемещения N по данной входной строке.

В таблице переходов НКА каждая запись представляет собой множество состояний. В таблице переходов ДКА – единственное состояние. Общая идея преобразования НКА в ДКА состоит в том, что каждое состояние ДКА соответствует множеству состояний НКА.

ДКА использует свои состояния для отслеживания всех возможных состояний, в которых НКА может находиться после чтения очередного входного символа. Таким образом, после чтения входного потока, $a_1$, $a_2 \ldots a_n$. ДКА находится в состоянии, которое представляет подмножество T состояний НКА, достижимых из стартового состояния НКА по пути, помеченному как $a_1$, $a_2 \ldots a_n$. Количество состояний ДКА может оказаться экспоненциально зависящим от количества состояний НКА, но на практике этот наихудший случай встречается редко.

**Операции над состояниями НКА**

Таблица 1.3 – Операции над состояниями НКА

| Операция | Описание |
|---|---|
| e-closure(s)<br>(e-closure(s)) | Множество состояний НКА, достижимых из состояния s по e-переходам |
| e-closure(T)<br>(e-closure(T)) | Множество состояний НКА, достижимых из какого-либо состояния s из T только по e-переходам |
| Move(T,a) | Множество состояний НКА, в которые имеется переход из какого-либо состояния s из T по входному символу a |



Библиотека БГУИР

**Алгоритм построения подмножества**

Изначально e-closure($s_0$) является единственным состоянием в Dstates и непомечено:

***While*** *в* ***Dstates*** *имеется непомеченное состояние* ***T do begin***
  *Пометить* ***T;***
  ***For*** *каждый входной символ* ***a do begin***
    ***U:=e-closure(move(T,a));***
    ***If U not ∈Dstates then***
      *Добавить* ***U*** *как непомеченное состояние в* ***Dstates;***
    ***Dtran[T,a]:=U***
  ***End***
***End***

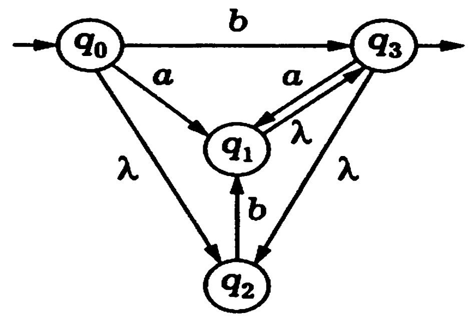
Множество состояний Dstates автомата D и таблицу его переходов Dtran можно создать следующим образом. Каждое состояние D соответствует множеству состояний НКА, в которых может находиться N после чтения некоторой последовательности входных символов, включая все возможные e-переходы до и после считанных символов. Стартовое состояние D – e-closure($s_0$). Состояния и переходы добавляются в D согласно алгоритму. Состояние D является допускающим, если оно представляет собой множество состояний НКА, содержащих как минимум одно допускающее состояние N.

***Вычисление e-замыкания***
*Внести все состояния множества* ***T*** *в стек* ***stack;***
*Инициализировать* ***e-closure(T)*** *множеством* ***T;***
***While stack*** *не пуст* ***do begin***
  *Снять со стека верхний элемент* ***t***
  ***For*** *каждое состояние* ***u*** *с дугой*
    *От* ***t*** *к* ***u****, помеченной* ***e do***
    ***If u not ∈ e-closure(T) then begin***
      *Добавить* ***u*** *к* ***e-closure(T);***
      *Поместить* ***u*** *в* ***stack***
  ***End***
***End***

Библиотека БГУИР

**Метод «вытягивания»**

Пусть у нас имеется конечный автомат в виде, представленном на рисунке 1.2.




λ – е-переход

Рисунок 1.2 – Конечный автомат

Детерминируем его, избавившись от е-переходов (рисунок 1.3).

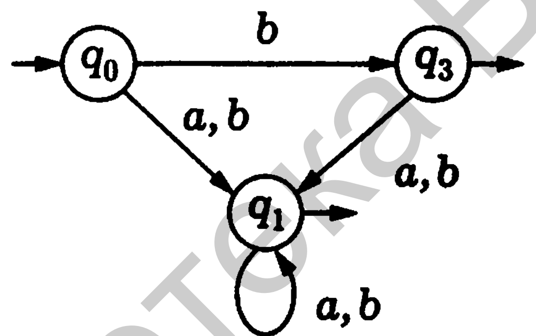



Рисунок 1.3 – Конечный автомат без е-переходов

Заметим, что состояние $q_2$ исчезает, так как в нее заходят только «пустые» дуги.

Чтобы детерминизировать полученный автомат, совершенно не обязательно выписывать все его $2^3 = 8$ состояний, среди которых многие могут оказаться недостижимыми из начального состояния $q_0$. Чтобы получить только достижимые из $q_0$ состояния, воспользуемся так называемым **методом «вытягивания».**

Этот метод в общем случае можно описать так.

В исходном конечном автомате (без пустых дуг) определяем все множества состояний, достижимых из начального, т. е. для каждого входного символа a находим множество $\delta(q_0, a)$. Каждое такое множество в новом автомате является состоянием, непосредственно достижимым из начального.



Библиотека БГУИР

Для каждого из определенных состояний-множеств S и каждого входного символа a находим множество U δ(q, a). Все полученные на этом шаге состояния будут состояниями нового (детерминированного) автомата, достижимыми из начальной вершины по пути длиной 2. Описанную процедуру повторяем до тех пор, пока не перестанут появляться новые состояния-множества (включая пустое!). Можно показать, что при этом получаются все такие состояния конечного автомата $M_1$, которые достижимы из начального состояния $q_0$.

Для конечного автомата на рисунке 1.3 выше имеем:

$$\delta_1(\{q_0\},a) = \{q_1\}; \qquad \delta_1(\{q_0\},b) = \{q_1, q_3\};$$

$$\delta_1(\{q_1\},a) = \{q_1\}; \qquad \delta_1(\{q_1\},b) = \{q_1\};$$

$$\delta_1(\{q_1, q_3\},a) = \delta(q_1,a) \cup \delta(q_3,a) = \{q_1\} \cup \{q_1\} = \{q_1\};$$

$$\delta_1(\{q_1, q_3\},b) = \delta(q_1,b) \cup \delta(q_3,b) = \{q_1\} \cup \{q_1\} = \{q_1\}.$$

Так как новых состояний-множеств больше не появилось, процедура «вытягивания» заканчивается, и мы получаем результирующий КА, представленный на рисунке 1.4.

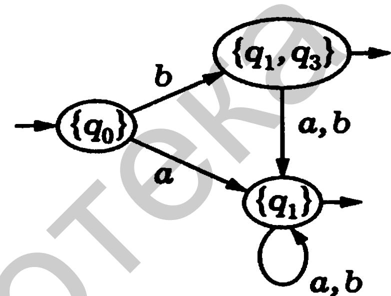



Рисунок 1.4 – Результирующий конечный автомат

**Алгоритм «Минимизация количества состояний ДКА»**

**Вход:** ДКА M с множеством состояний S; множеством входных символов E; переходами, определенными для всех состояний и входных символов; стартовым состоянием $s_0$; множеством заключительных состояний F.

**Выход:** ДКА M’, допускающий тот же язык, что и M, и имеющий наименьшее возможное количество состояний.

**Метод**

1. Построить начальное разбиение П множества состояний с двумя группами, заключительные состояния F и незаключительные состояния S–F.

2. Применить процедуру к разбиению П для построения нового разбиения $\Pi_{new}$.

3. Если $\Pi_{new} = \Pi$, то $\Pi_{final} = \Pi$, и следует перейти к шагу 4. В противном случае повторить шаг 2 с $\Pi := \Pi_{new}$.



Библиотека БГУИР

4. Выбрать одно состояние в каждой группе разбиения $П_{final}$ в качестве представителя этой группы. Представители будут состояниями ДКА M'. Пусть s является представителем. Предположим, что для входного символа a в M существует переход из s в t. Пусть r – представитель группы, в которой находится t (r может являться t), тогда M' имеет переход из s в r по a. Стартовым состоянием M' сделать представителя группы, содержащей стартовое состояние $s_0$ автомата M, а заключительными состояниями M' – представителей в F.

5. Если M' имеет мертвое состояние, т. е. состояние d, которое не является заключительным и имеет переходы в себя для всех входных символов, удалить его из M'. Удалить также все состояния, недостижимые из стартового.

**Построение $П_{new}$:**

***For*** *каждая группа* ***G ∊ П do begin***

*Разделить* ***G*** *на подгруппы, такие, что два состояния* ***s*** *и* ***t*** *из* ***G*** *находятся в одной и той же подгруппе тогда и только тогда, когда для всех входных символов a состояния* ***s*** *и* ***t*** *переходы по* ***a*** *в состояния из одной и той же группы* ***П***

*Заменить* ***G*** *в* $\boldsymbol{П_{new}}$ *множеством всех созданных групп*

***End***

**1.1.1 Условие**

1. По регулярному выражению построить диаграмму переходов конечного автомата.

2. По построенной диаграмме построить таблицу состояний.

3. Проверить, является ли построенный конечный автомат недетерминированным, записать объяснение.

4. Если конечный автомат является недетерминированным, то преобразовать его в детерминированный.

5. Проверить, является ли построенный конечный автомат минимальным, записать объяснение.

6. Если конечный автомат не является минимальным, то минимизировать его.

7. Если производились преобразования построенного конечного автомата, то построить соответствующее ему регулярное выражение

**1.1.2 Пример построения конечного автомата**

1. Формулировка задания: $a^2(ab)^*(c|d)^+$.

Библиотека БГУИР

2. Диаграмма переходов конечного автомата представлена на рисунке 1.5.

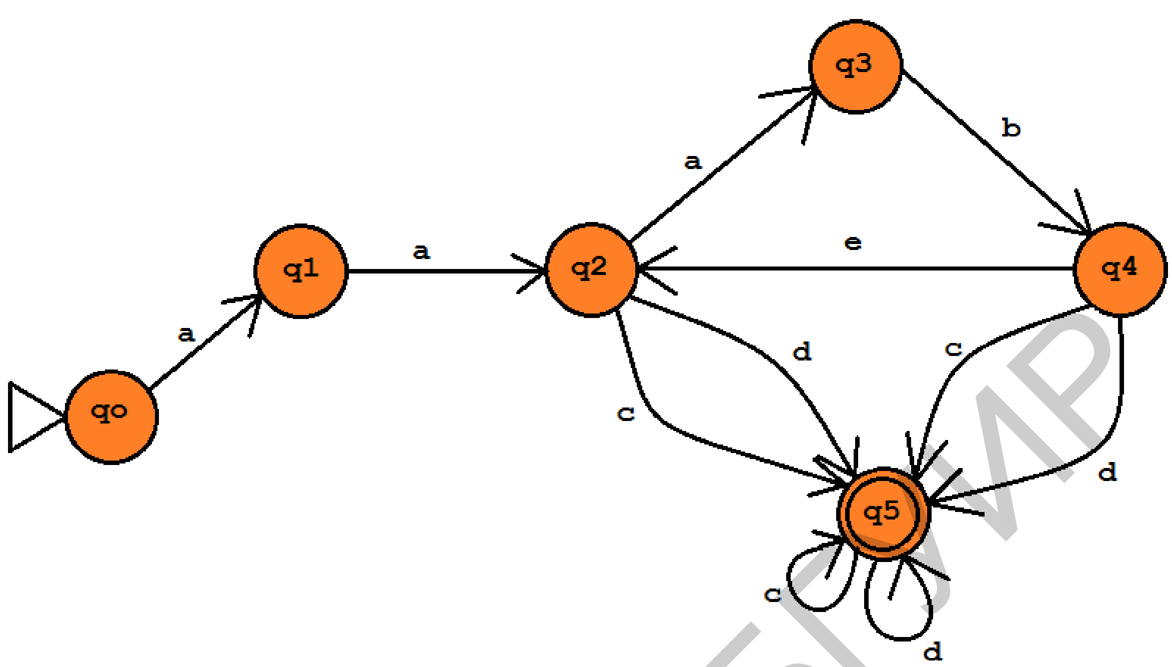



Рисунок 1.5 – Диаграмма переходов конечного автомата

*Примечания*

3. Построенный автомат является недетерминированным, потому что есть е-переходы. Детерминируем конечный автомат (рисунок 1.6).

Библиотека БГУИР

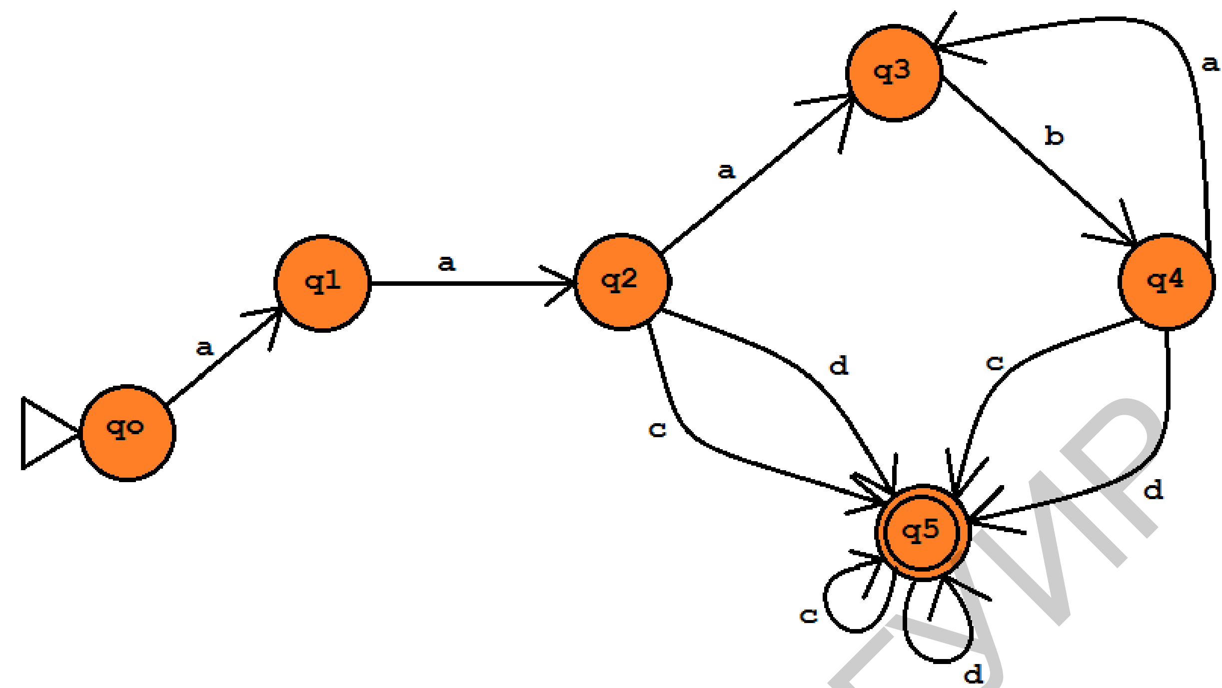



Рисунок 1.6 – Детерминированный конечный автомат

4. Для полученного конечного автомата построим таблицу состояний.

Таблица 1.4 – Таблица состояний конечного автомата

| | a | b | c | d |
|---|---|---|---|---|
| $q_0$ | 1 | – | – | – |
| $q_1$ | 2 | – | – | – |
| $q_2$ | 3 | – | 5 | 5 |
| $q_3$ | – | 4 | – | – |
| $q_4$ | 3 | | 5 | 5 |
| $q_5$ | – | – | 5 | 5 |

5. Конечный автомат является неминимизированным. Следовательно, минимизируем его и построим дерево разбора.

**Алгоритм минимизации**

1. Пусть множество П(0,1,2,3,4,5) – множество всех состояний. Разобьем его на два подмножества согласно условию с состояниями (0,1,2,3,4) и (5), где подмножество (0,1,2,3,4) содержит незаключительные состояния, а подмножество (5) – заключительное состояние.

2. Первое подмножество разбиваем еще на 2 с состояниями (0,1,3) и (2,4), потому что для всех входных символов переход по входному символу происходит в заключительное состояние из одной и той же группы.

3. Первое подмножество делим на 2 с состояниями (0,1) и (3).

4. Первое подмножество делим на 2 с состояниями (0) и (1).

Библиотека БГУИР

В результате получили, что из состояний 2 и 4 можно оставить только одно, например 2. Количество состояний конечного автомата уменьшилось на одно.

Преобразуем конечный автомат (рисунок 1.7).

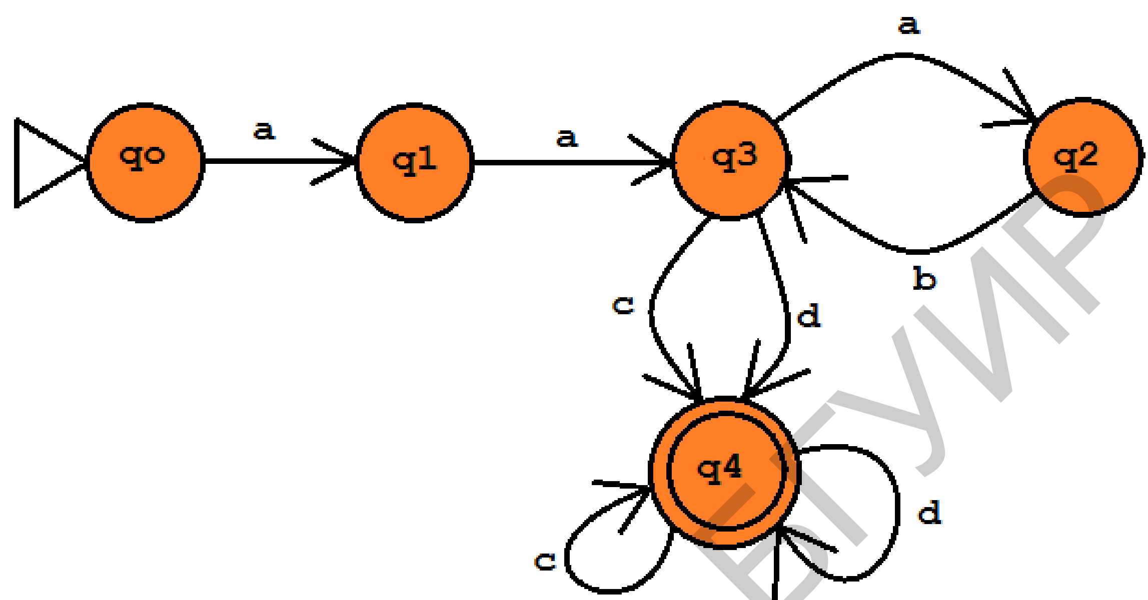



Рисунок 1.7 – Преобразованный конечный автомат

Докажем, что он минимизирован:

$\{q_0,q_1,q_2,q_3\}$ $\{q_4\}$
$\{q_0,q_1,q_2\}$ $\{q_3\}$ $\{q_4\}$
$\{q_0\}$ $\{q_1\}$ $\{q_2\}$ $\{q_3\}$ $\{q_4\}$

Получили детерминированный и минимизированный конечный автомат.
Примеры правильных строк конечного автомата:

*aaababccdcdc*
*aacddc*
*aaabccc*

Примеры неправильных строк:

*abacdcd*
*bcda*
*abacb*



Библиотека БГУИР

## 1.2 Варианты индивидуальных заданий

Таблица 1.5 – Варианты индивидуальных заданий

| № п/п | Формулировка варианта задания |
|---|---|
| 1 | a2(ab)*c\|d |
| 2 | d\|a2(ab)*c |
| 3 | (d\|a2)(ab)*c |
| 4 | (d\|a)2(ab)*c |
| 5 | ((d\|a)2ab)*c |
| 6 | c+((d\|a)2ab)* |
| 7 | (c+)+((d\|a)2ab)* |
| 8 | a2(ab)*(c\|d)+ |
| 9 | (a\|b\|c)+abc(a\|b\|c)* |
| 10 | (a\|b\|c)+abc4(a\|b\|c)* |
| 11 | (d\|a)*(ab)*c* |
| 12 | (d\|a)*\|(ab)*c* |
| 13 | (d\|a)*\|(ab)*\|c* |
| 14 | (d\|a)*(ab)*\|c* |
| 15 | (d\|a+)*(ab)*\|c* |
| 16 | (d\|a*)*(ab)*\|c* |
| 17 | a*b*c*\|a+b+c+ |
| 18 | a*b*(c*\|c+)b+a+ |
| 19 | a*b*(c*\|c+)*b+a+ |
| 20 | a*b*(c*\|c+)cb+a+ |
| 21 | abc\|cbaa+(bc\|cb)a* |
| 22 | ab(c*\|c)baa+(bc\|cb)a* |
| 23 | (abc\|cb)a*a+(bc\|cb)a* |
| 24 | a(bc\|cba)a+(bc\|cb)a* |
| 25 | abc\|cbaa+(bc\|cb)a* |
| 26 | (abc\|cba)*a+(bc\|cb)a* |
| 27 | abc\|cbaa+(bc\|cb)a* |
| 28 | abc\|cba(a+(bc\|cb)a)* |
| 29 | (abc\|cbaa)+(bc\|cb)a* |
| 30 | (abc\|cbaa+(bc\|cb)a)* |

Библиотека БГУИР

# ЛАБОРАТОРНАЯ РАБОТА №2
# ПОСТРОЕНИЕ КС-ГРАММАТИКИ

*Цель работы*: получить навык построения КС-грамматики.

## 2.1 Теоретические сведения

***Порождающая грамматика G*** – это четверка (VT, VN, P, S), где:

- VT – алфавит терминальных символов (терминалов);
- VN – алфавит нетерминальных символов (нетерминалов), не пересекающийся с VT;
- P – конечное подмножество множества (VT ∪ VN)+*(VT ∩ VN)*; элемент (α, β) множества P называется правилом вывода и записывается в виде α → β;
- S – начальный символ (цель) грамматики, S ∈ VN.

### Типы грамматик

По иерархии Хомского грамматики делятся на четыре типа, каждый последующий является более ограниченным подмножеством предыдущего (но и легче поддающимся анализу).

**Тип 0**

В неограниченных грамматиках возможны любые правила, т. е. грамматика G = (VT, VN, P, S) называется грамматикой типа 0, если на правила вывода не накладывается никаких ограничений (кроме тех, которые указаны в определении грамматики).

**Тип 1**

В контекстно-зависимых грамматиках (КЗ-грамматики) левая часть может содержать один нетерминал, окруженный «контекстом» (последовательности символов, в том же виде присутствующие в правой части). Сам нетерминал заменяется непустой последовательностью символов в правой части.

Другими словами, грамматика G = (VT, VN, P, S) называется контекстно-зависимой, если каждое правило из P имеет вид α → β, где α = = ξ1Aξ2; β = ξ1γ ξ2; A ∈ VN; γ ∈ (VT ∪ VN)+; ξ1, ξ2 ∈ (VT ∪ VN)*.

Грамматика G = (VT, VN, P, S) называется неукорачивающей грамматикой, если каждое правило P имеет вид α → β, где α ∈ (VT ∪ VN)+, β ∈ (VT ∪ ∪ VN)+ и |α| <= |β|.

Библиотека БГУИР

**Тип 2**

В <u>контекстносвободных грамматиках (КС-грамматиках)</u> левая часть состоит из одного нетерминала.

Грамматика G = (VT, VN, P, S) называется контекстносвободной, если каждое правило из P имеет вид $A \rightarrow \beta$, где $A \in VN$, $\beta \in (VT \cup VN)+$.

В частном случае грамматика G = (VT, VN, P, S) называется <u>укорачивающей контекстносвободной</u> (УКС), если каждое правило из P имеет вид $A \rightarrow \beta$, где $A \in VN$, $\beta \in (VT \cup VN)^*$.

Контекстносвободные грамматики широко применяются для определения грамматической структуры в грамматическом анализе.

**Тип 3**

<u>Регулярные грамматики</u> – более простые, эквивалентны конечным автоматам.

Грамматика G = (VT, VN, P, S) называется <u>праволинейной</u>, если каждое правило из P имеет вид $A \rightarrow tB$ либо $A \rightarrow t$, где $A \in VN$, $B \in VN$, $t \in VN$.

Грамматика G = (VT, VN, P, S) называется <u>леволинейной</u>, если каждое правило из P имеет вид $A \rightarrow Bt$ либо $A \rightarrow t$, где $A \in VN$, $B \in VN$, $t \in VN$.

**Соглашения по обозначениям**

*Терминалами* являются следующие символы:

а) строчные буквы из начала алфавита, такие как a, b, c;

б) цифры 0, 1, 2, …, 9;

в) символы пунктуации, такие как запятые, скобки и т. п.;

г) символы операторов, такие как +, – и т. п.;

д) строки, выделенные полужирным шрифтом, такие как **id** или **if**.

*Нетерминалами* считаются следующие символы:

а) прописные буквы из начала алфавита, такие как A, B, C;

б) буква S, которая обычно обозначает стартовый символ;

в) имена из строчных букв, выделенные курсивом, такие как *stmt* или *expr*.

*Другие соглашения*:

- прописные буквы из конца алфавита, такие как X, Y, Z, представляют грамматические символы, т. е. являются либо терминалами, либо нетерминалами;
- строчные буквы из конца алфавита, такие как u, v, …, z, обозначают строки терминалов;
- строчные греческие буквы, такие как α, β, μ, представляют собой строки грамматических символов. То есть в общем виде продукция может быть



Библиотека БГУИР

записана как $A \to \alpha$, в которой одиночный нетерминал A располагается слева от стрелки, а строка грамматических символов $\alpha$ – справа;

- если $A \to \alpha_1$, $A \to \alpha_2$, …, $A \to \alpha_k$ представляют собой продукции с A в левой части, то можем записать $A \to \alpha_1| \alpha_2| \dots| \alpha_k$. Тогда $\alpha_1$, $\alpha_2$, …, $\alpha_k$ назовем альтернативами A;
- если иное не указано явно, левая часть первой продукции является стартовым символом;
- продукция обозначается символом $\to$ .

***Контекстносвободная грамматика*** состоит из четырех компонентов:

- терминалов;
- нетерминалов;
- стартового символа;
- продукций.

**Основные определения**

***Терминал (или терминальный символ)*** – это объект, непосредственно присутствующий в словах языка, соответствующего грамматике, и имеющий конкретное, неизменяемое значение (обобщение понятия «буква»). В формальных языках, используемых на компьютере, в качестве терминалов обычно берут все или часть стандартных символов ASCII – латинские буквы, цифры и специальные символы.

***Нетерминал (или нетерминальный символ)*** – это объект, обозначающий какую-либо сущность языка (например, формула, арифметическое выражение, команда) и не имеющий конкретного символьного значения.

***Стартовым символом*** считается один из нетерминалов грамматики, а также множество строк, которые он обозначает и которые являются языком, определяемым грамматикой.

***Продукции*** грамматики определяют способ, которым терминалы и нетерминалы могут объединяться для создания строк. Каждая продукция состоит из нетерминала, за которым следует стрелка, и строка нетерминалов и терминалов.

Продукция называется ***продукцией нетерминала***, если нетерминал записан в левой части.

Дерево называется ***деревом вывода (или деревом разбора)*** в КС-грамматике G = (VT, VN, P, S), если выполнены следующие условия:

Библиотека БГУИР

- каждая вершина помечена символом из множества (VT ∪ VN ∪ ε), при этом корень дерева помечен символом S, листья – символами из (VT ∪ ε);
- если вершина дерева помечена символом A ∈ VN, а ее непосредственные потомки – символами $a_1$, $a_2$, ..., $a_n$, где каждое $a_i$ ∈ (VT ∪ VN), то A → $a_1 a_2 ... a_n$ – правило вывода в этой грамматике;
- если вершина дерева помечена символом A ∈ VN, а ее единственный непосредственный потомок помечен символом ε , то A → ε – правило вывода в этой грамматике.

КС-грамматика называется ***неоднозначной***, если существует хотя бы одна цепочка α ∈ L(G), для которой могут быть построены два или более различных деревьев вывода.

**Свойства КС-грамматик**

*Рекурсивность грамматики*:

- левая рекурсивность при A → AB*;
- правая рекурсивность при A → B*A.

*Факторизация грамматики*:

- левая при A → aB* и B → aC*;
- правая при A → B*a и B → C*a.

Наиболее часто встречаются левая рекурсии и левая факторизация.

***Левая факторизация*** представляет собой преобразование грамматики в пригодную для предикативного анализа.

Грамматика является ***леворекурсивной***, если в ней имеется нетерминал A, такой, что существует порождение A → Aα для некоторой строки α.

**Алгоритм «Устранение левой рекурсии КС-грамматики»**

**Вход:** грамматика G без циклов и ε-продукций.

**Выход:** эквивалентная грамматика без левой рекурсии.

**Метод.** Применить алгоритм устранения левой рекурсии. Результирующая грамматика без левой рекурсии может иметь ε-продукции.

**1.** *Расположить нетерминалы в некотором порядке* $\boldsymbol{A_1, A_2, ..., A_n}$

**2.** ***for i:=1 to n do begin***

***for j:=1 to i–1 do begin***

*Заменить каждую продукцию вида* $\boldsymbol{A_i \rightarrow A_{j\gamma}}$

*Продукциями* $\boldsymbol{A \rightarrow \delta_{1\gamma} | \delta_{2\gamma} | \dots | \delta_{k\gamma}}$

*Где* $\boldsymbol{A_j\ A \rightarrow \delta_1 | \delta_2 | \dots | \delta_k}$ *– все текущие* $\boldsymbol{A_j}$ *– продукции*



Библиотека БГУИР

*End*

*Устранить непосредственную левую рекурсию*

*Среди $A_i$ – продукций*

***End***

**Пример устранения левой рекурсии КС-грамматики**

Перед тем как рассмотреть пример, опишем процедуру, устраняющую все правила вида A → Aα для фиксированного нетерминала A.

1. Запишем все правила вывода из A в виде $A \rightarrow A\alpha \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid\mid \beta_n$, где:

- α – это непустая последовательность терминалов и нетерминалов (α → → ε);
- β – это непустая последовательность терминалов и нетерминалов, не начинающаяся с A.

2. Заменим правила вывода из A на $A \rightarrow \beta_1 A' \mid \ldots \mid \beta_m A' \mid \beta_1 \mid \ldots \mid \beta_m$ .
3. Создадим новый нетерминал $A' \rightarrow \alpha_1 A', \mid \ldots \mid \alpha_n A' \mid \alpha_1 \mid \ldots \mid \alpha_n$ .

Изначально нетерминал A порождает сроки вида $\beta\alpha_{i0}\alpha_{i1}\ldots\alpha_{ik}$.

В новой грамматике нетерминал A порождает βA’, а A’ порождает строки вида $\alpha_{i0}\alpha_{i1}\ldots\alpha_{ik}$.

Перейдем непосредственно к примеру. Пусть у нас имеется грамматика:

A → Sα | Aα

S → Aβ

Есть непосредственная левая рекурсия A → Aα. Добавим нетерминал A’ и правила A → SαA’ A’ → αA’.

Новая грамматика:

A → SαA’Sα

A’ → αA’ | α

S → Aβ

**Алгоритм «Левая факторизация КС-грамматики»**

**Вход:** грамматика G.

**Выход:** эквивалентная левофакторизованная грамматика.

**Метод:** для каждого нетерминала A находим наидлиннейший префикс α, общий для двух или большего числа альтернатив.

Если α ≠ ε, т. е. имеется нетривиальный общий префикс, заменим все продукции $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$, где γ представляет все альтернативы, не начинающиеся с α, продукциями:

Библиотека БГУИР

$A \rightarrow \alpha A' \mid \gamma$

$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Здесь A' – новый нетерминал.

Выполняем это преобразование до тех пор, пока никакие две альтернативы не будут иметь общий префикс.

### 2.1.1 Условие

1. По описанию языка построить порождающую грамматику.
2. Определить тип построенной грамматики и свойства.
3. Построить для трех примеров деревья разбора.
4. Если грамматика является леворекурсивной, то устранить ее.
5. Если грамматика не является левофакторизованной, то левофакторизовать ее.
6. Для модифицированной грамматики построить деревья разбора по тем же примерам, что и в пункте 3.

### 2.1.2 Пример построения КС-грамматики

1. Дано описание языка:

```
<P> ::=  <H> (<B>)…
<H> ::= 'h' ('t' 'i')…
<B> ::= 'b' (',' 'b')… ';'
```

2. Построим грамматику по описанию языка:

```
P → H          P → HG        G → BG       G → ε
H → hN         N → tiN       N → ε
B → bM;  M → ,bM      M → ε
```

3. Грамматика является <u>*контекстносвободной*</u>, потому что символы левой части – нетерминалы, а символы правой части – терминалы или нетерминалы.

4. Построим для трех примеров деревья разбора (рисунки 2.1–2.3).

Библиотека БГУИР

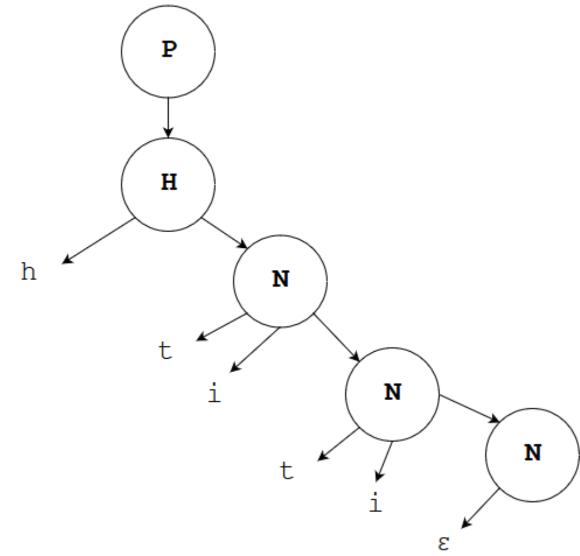



Рисунок 2.1 – Разбор htiti

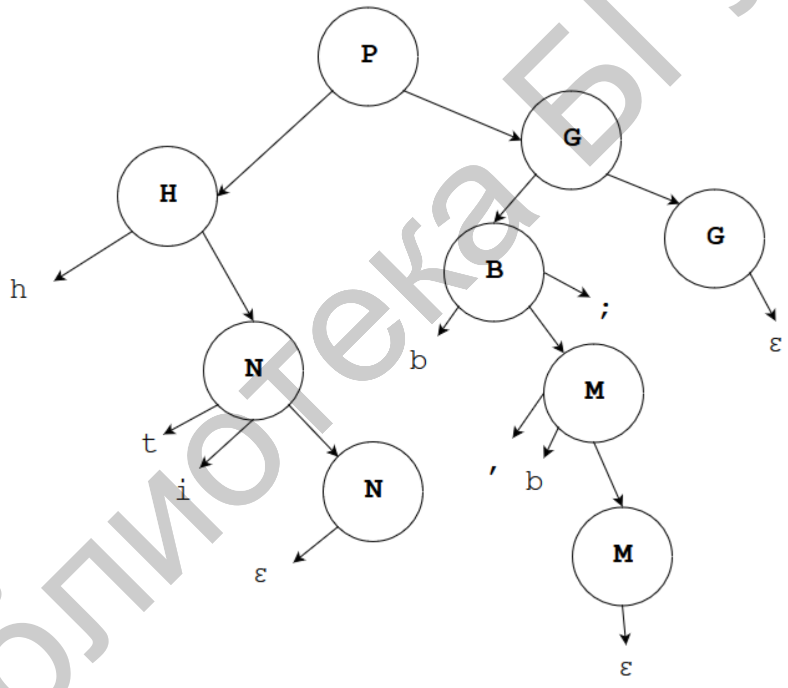



Рисунок 2.2 – Разбор htib,b;

Библиотека БГУИР

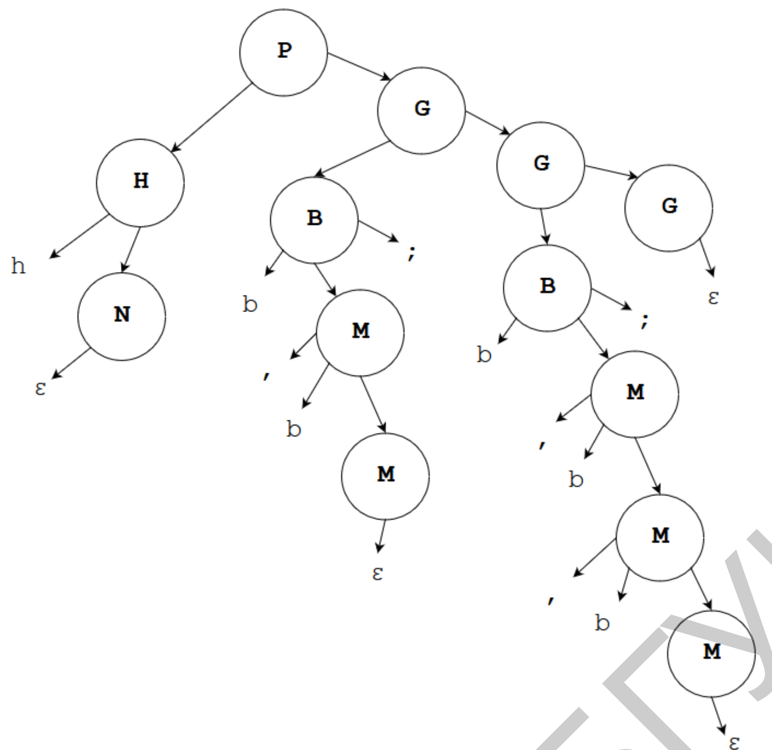



Рисунок 2.3 – Разбор htib,b;b,b,b

5. Грамматика не является леворекурсивной, потому что не встречается правило вида A → Aα.

6. Грамматика не является левофакторизованной, потому что встречаются правила вида:

A → aC

A → aD

Левофакторизуем грамматику: заменим P → H и P → HG на P → HC, где C → G, C → ε.

Преобразованная грамматика будет выглядеть следующим образом:

P → HC  C → G  C → ε  G → BG  G → ε

H → hN  N → tiN  N → ε

B → bM;  M → ,bM  M → ε

Так как C → G, C → ε, а G → BG, G → ε, то можно опустить C и записать так:

P → HG  G → BG  G → ε

H → hN  N → tiN  N → ε

B → bM;  M → ,bM  M → ε

7. По преобразованной грамматике построим деревья разбора (рисунки 2.4–2.6).

Библиотека БГУИР

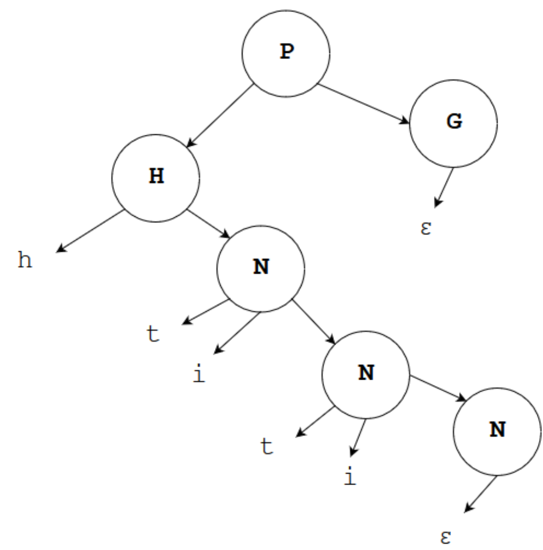


Рисунок 2.4 – Дерево разбора htiti

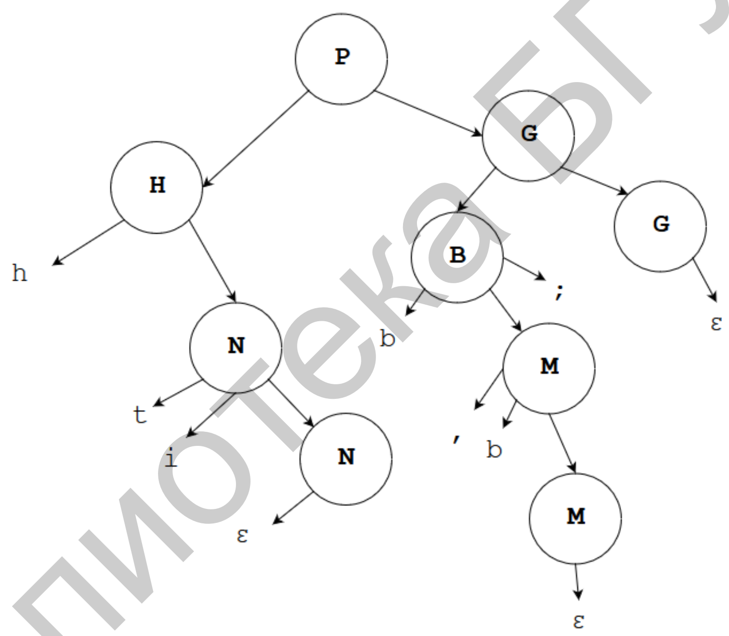


Рисунок 2.5 – Дерево разбора htib,b

Библиотека БГУИР

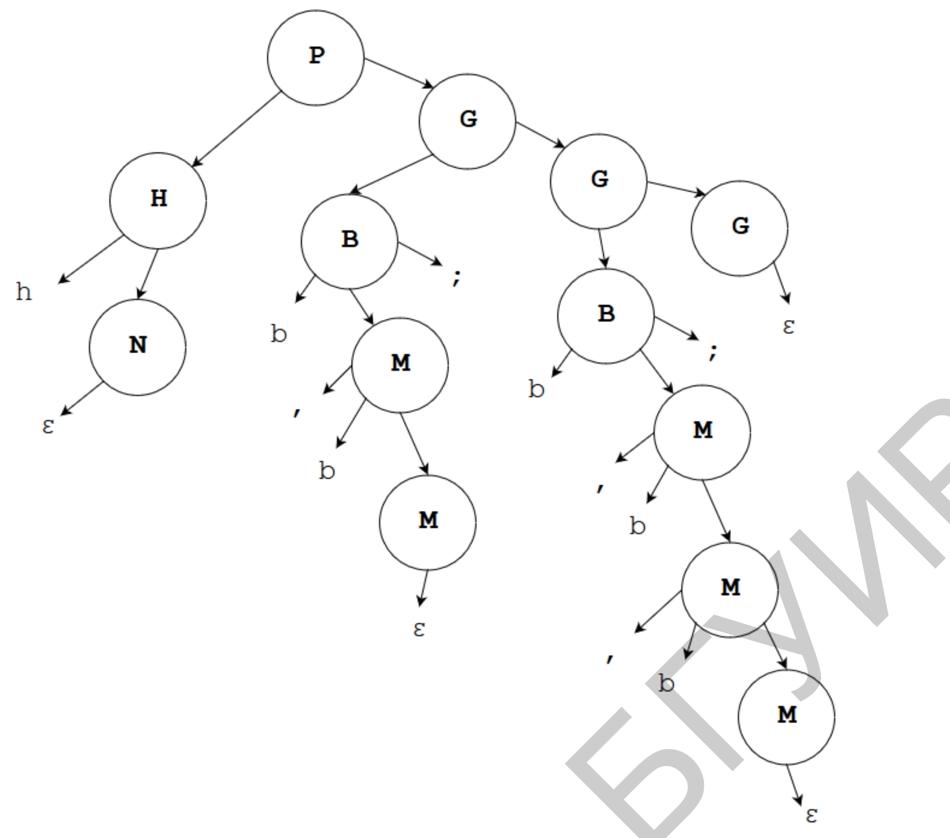
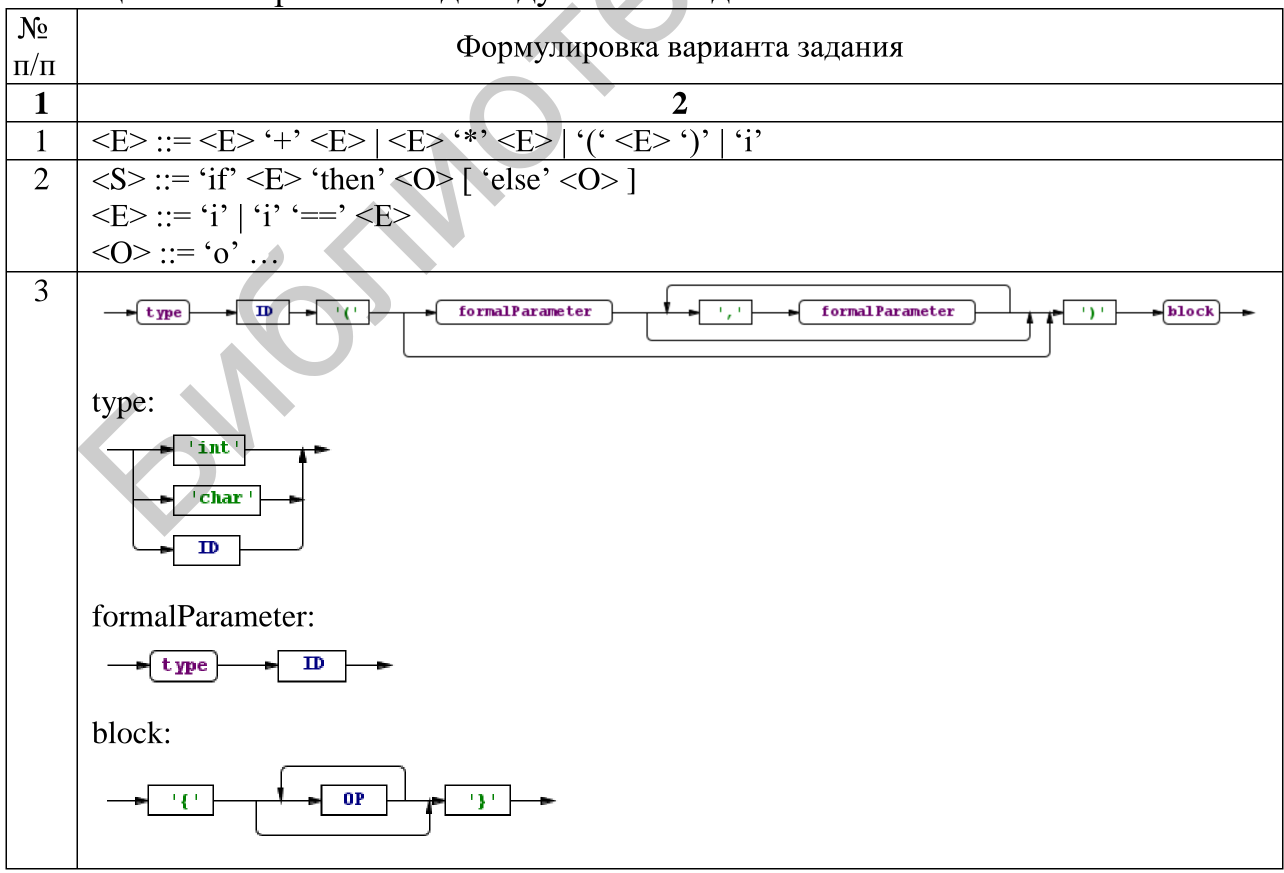



Рисунок 2.6 – Дерево разбора htib,b;b,b,b

## 2.2 Варианты индивидуальных заданий

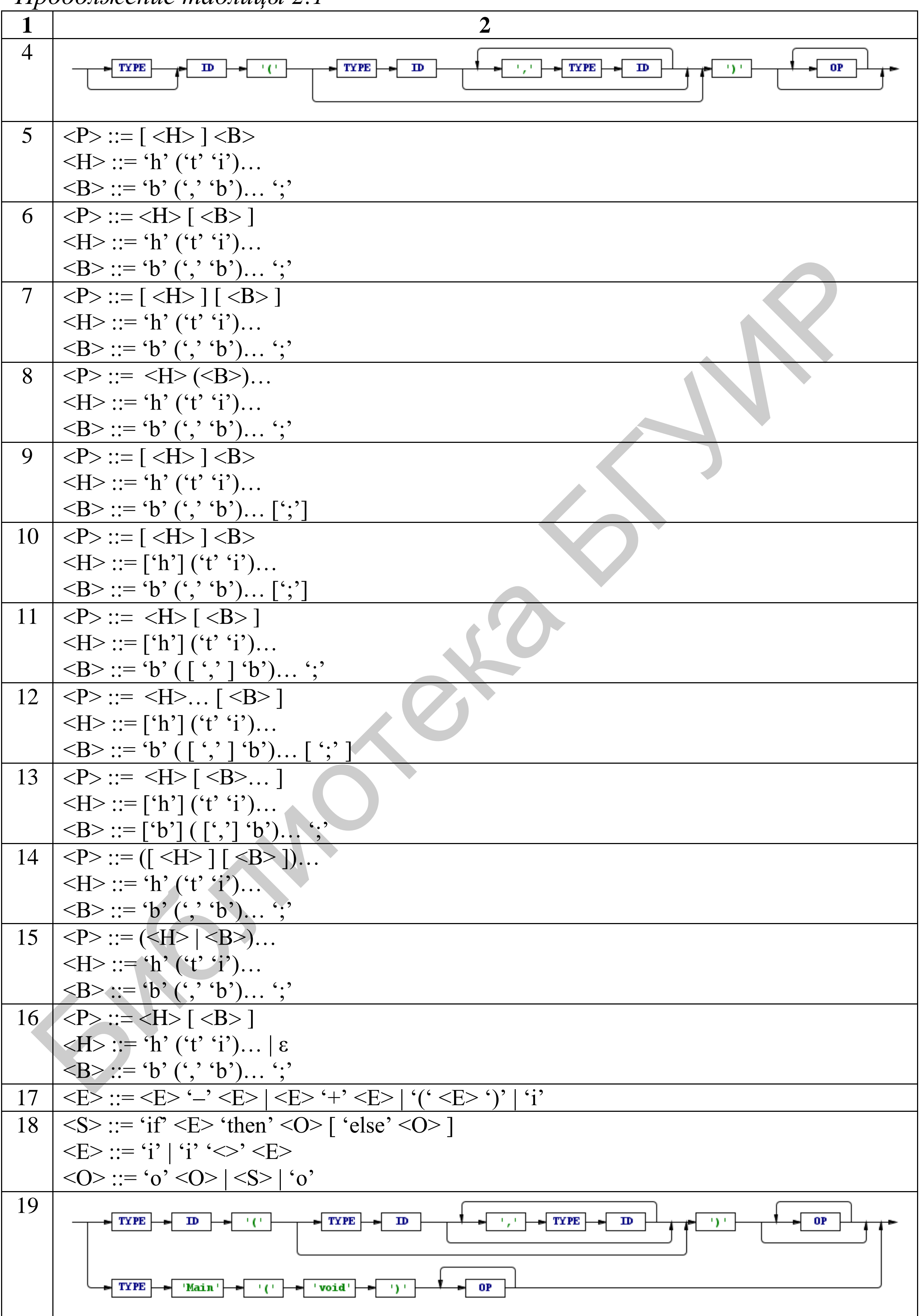
Таблица 2.1 – Варианты индивидуальных заданий



| № п/п | Формулировка варианта задания |
|---|---|
| **1** | **2** |
| 1 | <E> ::= <E> ‘+’ <E> \| <E> ‘*’ <E> \| ‘(‘ <E> ‘)’ \| ‘i’ |
| 2 | <S> ::= ‘if’ <E> ‘then’ <O> [ ‘else’ <O> ]<br><E> ::= ‘i’ \| ‘i’ ‘==’ <E><br><O> ::= ‘o’ … |
| 3 | type ID '(' formalParameter ',' formalParameter ')' block<br>type:<br>'int' 'char' ID<br>formalParameter:<br>type ID<br>block:<br>'{' OP '}' |

Библиотека БГУИР

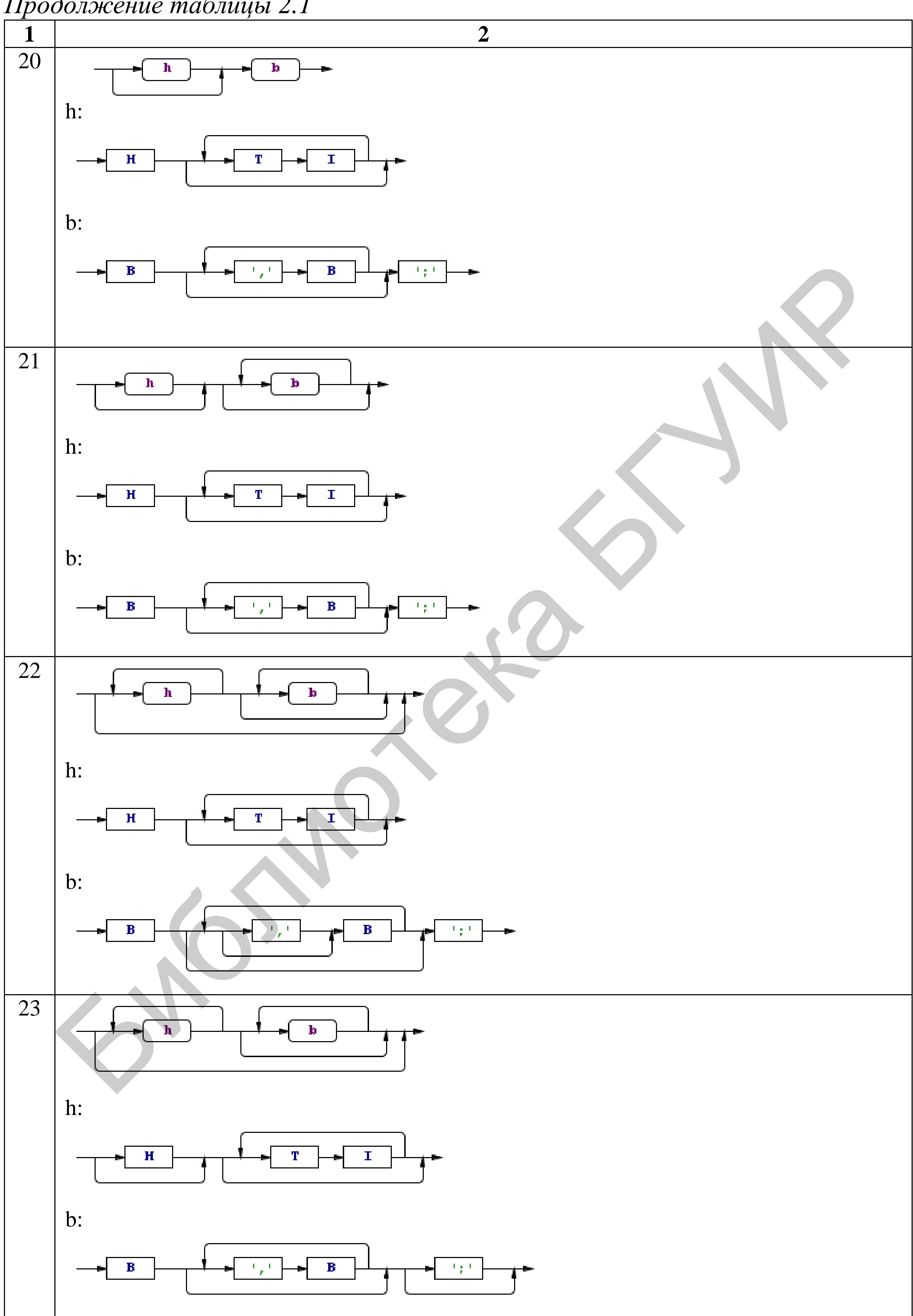
*Продолжение таблицы 2.1*



| 1 | 2 |
|---|---|
| 4 | TYPE ID '(' TYPE ID ',' TYPE ID ')' OP |
| 5 | <P> ::= [ <H> ] <B><br><H> ::= ‘h’ (‘t’ ‘i’)…<br><B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
| 6 | <P> ::= <H> [ <B> ]<br><H> ::= ‘h’ (‘t’ ‘i’)…<br><B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
| 7 | <P> ::= [ <H> ] [ <B> ]<br><H> ::= ‘h’ (‘t’ ‘i’)…<br><B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
| 8 | <P> ::= <H> (<B>)…<br><H> ::= ‘h’ (‘t’ ‘i’)…<br><B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
| 9 | <P> ::= [ <H> ] <B><br><H> ::= ‘h’ (‘t’ ‘i’)…<br><B> ::= ‘b’ (‘,’ ‘b’)… [‘;’] |
| 10 | <P> ::= [ <H> ] <B><br><H> ::= [‘h’] (‘t’ ‘i’)…<br><B> ::= ‘b’ (‘,’ ‘b’)… [‘;’] |
| 11 | <P> ::= <H> [ <B> ]<br><H> ::= [‘h’] (‘t’ ‘i’)…<br><B> ::= ‘b’ ( [ ‘,’ ] ‘b’)… ‘;’ |
| 12 | <P> ::= <H>… [ <B> ]<br><H> ::= [‘h’] (‘t’ ‘i’)…<br><B> ::= ‘b’ ( [ ‘,’ ] ‘b’)… [ ‘;’ ] |
| 13 | <P> ::= <H> [ <B>… ]<br><H> ::= [‘h’] (‘t’ ‘i’)…<br><B> ::= [‘b’] ( [‘,’] ‘b’)… ‘;’ |
| 14 | <P> ::= ([ <H> ] [ <B> ])…<br><H> ::= ‘h’ (‘t’ ‘i’)…<br><B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
| 15 | <P> ::= (<H> \| <B>)…<br><H> ::= ‘h’ (‘t’ ‘i’)…<br><B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
| 16 | <P> ::= <H> [ <B> ]<br><H> ::= ‘h’ (‘t’ ‘i’)… \| ε<br><B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
| 17 | <E> ::= <E> ‘–’ <E> \| <E> ‘+’ <E> \| ‘(‘ <E> ‘)’ \| ‘i’ |
| 18 | <S> ::= ‘if’ <E> ‘then’ <O> [ ‘else’ <O> ]<br><E> ::= ‘i’ \| ‘i’ ‘<>’ <E><br><O> ::= ‘o’ <O> \| <S> \| ‘o’ |
| 19 | TYPE ID '(' TYPE ID ',' TYPE ID ')' OP<br>TYPE 'Main' '(' 'void' ')' OP |

Библиотека БГУИР

*Продолжение таблицы 2.1*




Библиотека БГУИР

*Продолжение таблицы 2.1*

<table>
<tr><th>1</th><th>2</th></tr>
<tr><td>24</td><td><S> ::= ‘if’ [ <E> ] ( [ ‘i’ ‘:’ ] ‘then’ <O> )…<br><E> ::= ‘i’ | ‘i’ ‘<>’ <E><br><O> ::= ‘o’ <O> | <S> | ‘o’</td></tr>
<tr><td>25</td><td><S> ::= ‘if’ [ <E> ] ( ‘i’ ‘:’ ‘then’ <O> )…<br><E> ::= ‘i’ | ‘i’ ‘<>’ <E><br><O> ::= ‘o’ <O> | ‘o’</td></tr>
<tr><td>26</td><td><S> ::= ‘if’ [ <E> ] ( ‘i’ ‘:’ ‘then’ <O> )…<br><E> ::= ‘i’ | ‘i’ ‘<>’ <E><br><O> ::= ‘o’ <O> | <S> | ‘o’</td></tr>
<tr><td>27</td><td><S> ::= ‘if’ [ <E> ] ‘then’ <O> | <O><br><E> ::= ‘i’ | ‘i’ ‘<>’ <E><br><O> ::= ‘o’ <O> | <S> | ‘o’</td></tr>
<tr><td>28</td><td><S> ::= ‘if’ [ <E> ] ‘then’ <O> [ ‘else’ <O> ] | <O><br><E> ::= ‘i’ | ‘i’ ‘<>’ <E><br><O> ::= ‘o’ <O> | <S> | ‘o’</td></tr>
<tr><td>29</td><td><S> ::= ‘if’ [ <E> ] ‘then’ <O> [ ‘else’ <O> ]<br><E> ::= ‘i’ | ‘i’ ‘<>’ <E><br><O> ::= ‘o’ <O> | <S> | ‘o’</td></tr>
<tr><td>30</td><td><E> ::= <E> ‘*’ <E> | <E> ‘=’ <E> | ‘(‘ <E> ‘)’ | ‘i’</td></tr>
</table>

Библиотека БГУИР

# ЛАБОРАТОРНАЯ РАБОТА №3
# ПОСТРОЕНИЕ ТАБЛИЦЫ ПРЕДИКТИВНОГО АНАЛИЗАТОРА

*Цель работы*: получить навык построения таблицы предикативного анализатора.

## 3.1 Теоретические сведения

***Предиктивный анализ*** – анализ, при котором сканируемый символ однозначно определяет процедуру, выбранную для каждого нетерминала.

***Предиктивный анализатор*** представляет собой программу, содержащую процедуры для каждого нетерминального символа.

При построении предиктивного синтаксического анализатора можно создать его план в виде ***диаграммы переходов***.

**Диаграмма переходов** – стилизованная блок-схема, которая изображает действия, выполняемые лексическим анализатором при вызове его синтаксическим анализатором для получения очередного токена.

Для построения диаграммы переходов предиктивного синтаксического анализатора на основе грамматики вначале следует устранить из нее левые рекурсии, а затем провести левую факторизацию. После этого для каждого нетерминала A выполняется следующее:

1) создаем начальное и заключительное состояния;

2) для каждой продукции $A \rightarrow X_1, X_2, \ldots, X_n$, создаем путь от начального к заключительному состоянию с дугами, помеченными как $X_1, X_2, \ldots, X_n$.

### FIRST и FOLLOW

Если α – произвольная строка символов грамматики, то определим FIRST(α) как множество терминалов, с которых начинаются строки, выводимые из α. Если $\alpha \rightarrow \varepsilon$, то $\varepsilon \in$ FIRST(α).

Определим FOLLOW(A) для нетерминала A как множество терминалов, которые могут располагаться непосредственно справа от A в некоторой сентенциальной форме, т. е. множество терминалов a, таких, что существует порождение вида $S \rightarrow \alpha A a \beta$ для некоторых α и β.

Чтобы вычислить FOLLOW(A) для всех нетерминалов A, будем применять следующие правила до тех пор, пока ко всем множествам FOLLOW нельзя будет добавить ни одного символа.

1. Поместим $ в FOLLOW(S), где S – стартовый символ, а $ – правый ограничитель входного потока.

Библиотека БГУИР

2. Если имеется продукция $A \rightarrow \alpha B\beta$, то все элементы множества FIRST($\beta$), кроме $\varepsilon$, помещаются в множество FOLLOW(B).

3. Если имеется продукция $A \rightarrow \alpha B$ или $A \rightarrow \alpha B\beta$, где FIRST($\beta$) содержит $\varepsilon$, то все элементы из множества FOLLOW(A) помещаются во множество FOLLOW(B).

Чтобы вычислить FIRST(X) для всех символов грамматики X, будем применять следующее правило до тех пор, пока ко всем множествам FIRST не смогут быть добавлены ни терминалы, ни $\varepsilon$:

1) если X – терминал, то FIRST(X) = {X};

2) если имеется продукция $X \rightarrow \varepsilon$, добавим $\varepsilon$ к FIRST(X);

3) если X – нетерминал и имеется продукция $X \rightarrow Y_1, Y_2, \ldots, Y_k$, то поместим a в FIRST(X) при условии, что для некоторого i a $\in$ FIRST($Y_i$) и $\varepsilon$ входит во все множества FIRST($Y_1$), …, FIRST($Y_{i-1}$), т. е. $Y_1 \ldots Y_{i-1} \rightarrow \varepsilon$. Если $\varepsilon$ имеется во всех FIRST($Y_i$), i = 1…k, то добавляем $\varepsilon$ к FIRST(X).

Теперь можно вычислить FIRST для любой строки $X_1, X_2, \ldots, X_n$ следующим образом. Добавим к FIRST($X_1, X_2, \ldots, X_n$) все «не-$\varepsilon$-символы» из FIRST($X_1$). Добавим также все «не-$\varepsilon$-символы» из FIRST($X_2$), если $\varepsilon \in$ FIRST($X_1$); все «не-$\varepsilon$-символы» из FIRST($X_3$), если $\varepsilon$ имеется как в FIRST($X_1$), так и в FIRST($X_2$) и т. д.; FIRST($X_1, X_2, \ldots, X_n$), если для всех i FIRST($X_i$) содержит $\varepsilon$.

**Алгоритм «Построение таблицы предиктивного анализатора»**

**Вход:** грамматика G.

**Выход:** таблица анализа M.

**Метод**

1. Для каждой продукции грамматики $A \rightarrow \alpha$ выполняем шаги 2 и 3.

2. Для каждого терминала a из FIRST($\alpha$) добавляем $A \rightarrow \alpha$ в ячейку M[A, a].

3. Если в FIRST($\alpha$) входит $\varepsilon$, для каждого терминала b из FOLLOW(A) добавим $A \rightarrow \alpha$ в ячейку M[A, b]. Если $\varepsilon$ входит в FIRST($\alpha$), а \$ − в FOLLOW(A), добавим $A \rightarrow \alpha$ в ячейку M[A, \$].

4. Сделаем каждую неопределенную ячейку таблицы M указывающей на ошибку.

### 3.1.1 Условие

1. По описанию языка построить КС-грамматику.

2. Определить свойства полученной грамматики.

3. Если грамматика не обладает свойствами, требуемыми для построения таблицы предиктивного анализатора, то преобразовать грамматику к требуемой форме.

4. Определить значение функций FIRST и FOLLOW для разработанной грамматики.

5. Построить таблицу предиктивного анализатора.

6. Проверить правильность построения на трех примерах (один правильный, два неправильных).

### 3.1.2 Пример построения таблицы предиктивного анализатора

1. Дано описание языка:

```
<P> ::=  <H> (<B>)…
<H> ::= 'h' ('t' 'i')…
<B> ::= 'b' (',' 'b')… ';'
```

2. По описанию языка построили КС-грамматику:

```
P → HG          G → BG          G → ε
H → hN          N → tiN         N → ε
B → bM;  M → ,bM        M → ε
```

3. Грамматика нелеворекурсивная и левофакторизованная.

4. Определим значение функций FIRST и FOLLOW для разработанной грамматики:

| **FIRST** | **FOLLOW** |
|---|---|
| ***P = {h}*** | ***P ={$}*** |
| ***H = {h}*** | ***H = {b, $}*** |
| ***N = {t, ε}*** | ***N = {b, $, t}*** |
| ***M = { , , ε}*** | ***M = { , , ; }*** |
| ***B = {b}*** | ***B = {b, $}*** |
| ***G = {b, ε}*** | ***G = {b, $}*** |

5. Построим таблицу предиктивного анализатора.

Таблица 3.1 – Таблица предиктивного анализатора

| | h | ti | b | , | ; | $ |
|---|---|---|---|---|---|---|
| P | P → HG | – | – | – | – | – |
| H | H → hN | – | – | – | – | – |
| N | – | N → tiN | N → ε | – | – | N → ε |
| G | – | – | G → BG | – | – | G → ε |
| B | – | – | B → bM; | – | – | – |
| M | – | – | – | M → ,bM | M → ε | – |

Библиотека БГУИР

6. Проверим правильность построения на трех примерах.

**Правильные примеры:**

htib;$
P$
HG$
hNG$
NG$
tiNG$
NG$
G$
BG$
bM;G$
M;G$
;G$
G$
$

htitib,b;$
P$
HG$
hNG$
NG$
tiNG$
NG$
tiNG$
NG$
G$
BG$
bM;G$
M;G$
,bM;G$
M;G$
;G$
G$
$

Строки разложены.

**Неправильный пример:**

***hbtib,b;$***
P$
HG$
hNG$
NG$
tiNG$ – ошибка, так как в строке есть лишний терминал b.

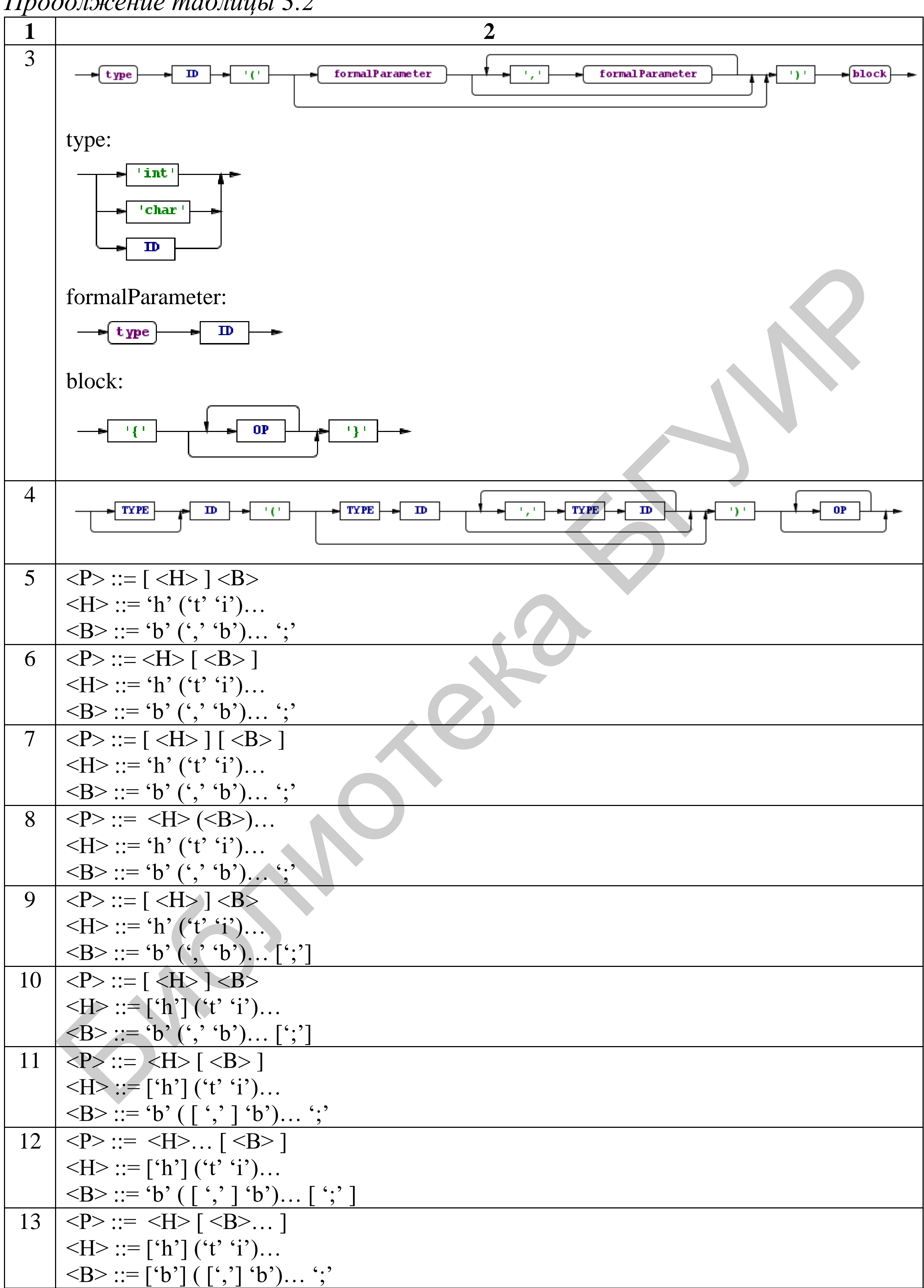
### 3.2 Варианты индивидуальных заданий

Таблица 3.2 – Варианты индивидуальных заданий

| № п/п | Формулировка варианта задания |
|---|---|
| **1** | **2** |
| 1 | <E> ::= <E> ‘+’ <E> \| <E> ‘*’ <E> \| ‘(‘ <E> ‘)’ \| ‘i’ |
| 2 | <S> ::= ‘if’ <E> ‘then’ <O> [ ‘else’ <O> ]<br><E> ::= ‘i’ \| ‘i’ ‘==’ <E><br><O> ::= ‘o’ … |

Библиотека БГУИР

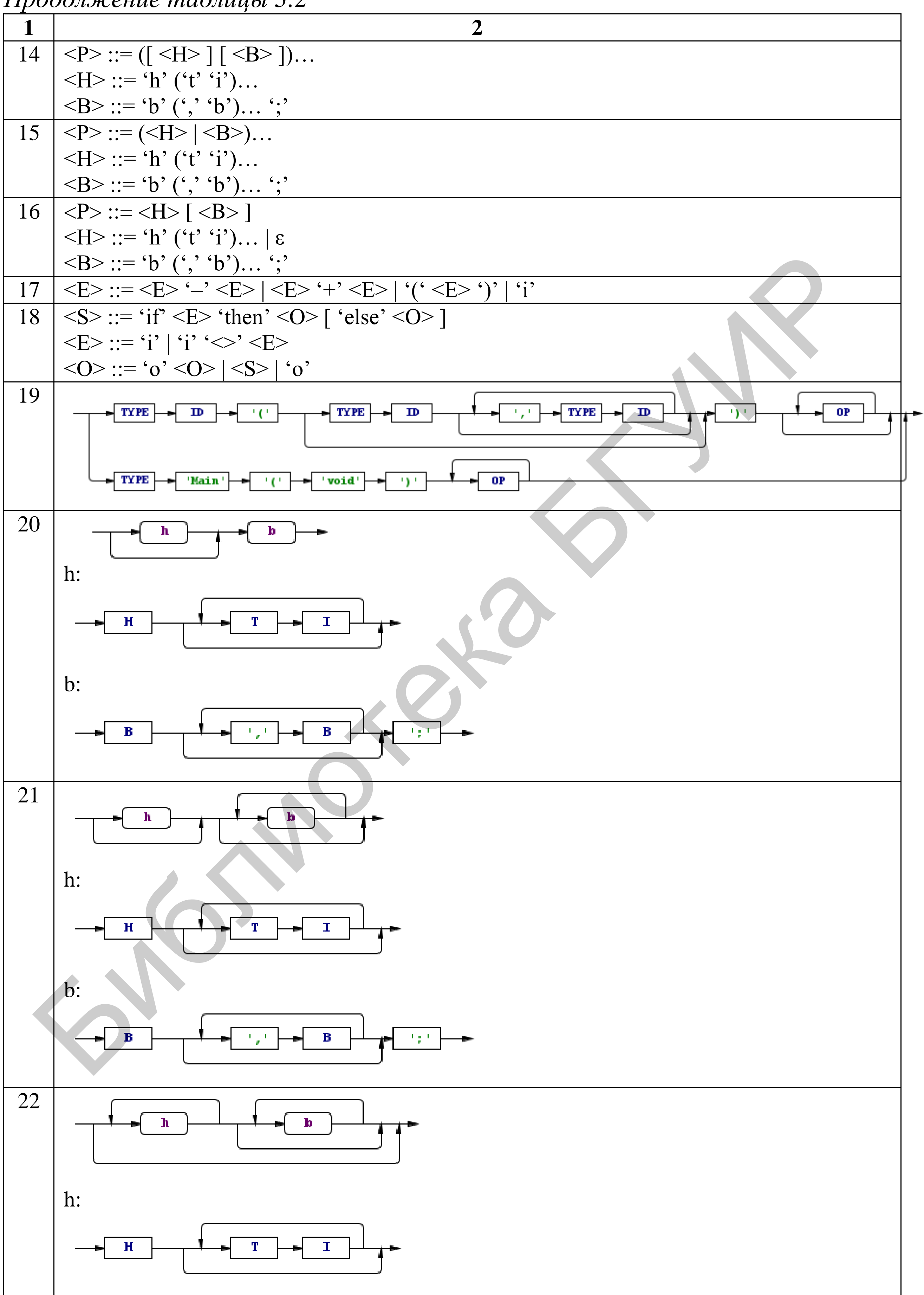
*Продолжение таблицы 3.2*



| 1 | 2 |
|---|---|
| 3 | type ID '(' formalParameter ',' formalParameter ')' block<br>type:<br>'int' 'char' ID<br>formalParameter:<br>type ID<br>block:<br>'{' OP '}' |
| 4 | TYPE ID '(' TYPE ID ',' TYPE ID ')' OP |
| 5 | <P> ::= [ <H> ] <B><br><H> ::= 'h' ('t' 'i')…<br><B> ::= 'b' (',' 'b')… ';' |
| 6 | <P> ::= <H> [ <B> ]<br><H> ::= 'h' ('t' 'i')…<br><B> ::= 'b' (',' 'b')… ';' |
| 7 | <P> ::= [ <H> ] [ <B> ]<br><H> ::= 'h' ('t' 'i')…<br><B> ::= 'b' (',' 'b')… ';' |
| 8 | <P> ::= <H> (<B>)…<br><H> ::= 'h' ('t' 'i')…<br><B> ::= 'b' (',' 'b')… ';' |
| 9 | <P> ::= [ <H> ] <B><br><H> ::= 'h' ('t' 'i')…<br><B> ::= 'b' (',' 'b')… [';'] |
| 10 | <P> ::= [ <H> ] <B><br><H> ::= ['h'] ('t' 'i')…<br><B> ::= 'b' (',' 'b')… [';'] |
| 11 | <P> ::= <H> [ <B> ]<br><H> ::= ['h'] ('t' 'i')…<br><B> ::= 'b' ( [ ',' ] 'b')… ';' |
| 12 | <P> ::= <H>… [ <B> ]<br><H> ::= ['h'] ('t' 'i')…<br><B> ::= 'b' ( [ ',' ] 'b')… [ ';' ] |
| 13 | <P> ::= <H> [ <B>… ]<br><H> ::= ['h'] ('t' 'i')…<br><B> ::= ['b'] ( [','] 'b')… ';' |

Библиотека БГУИР

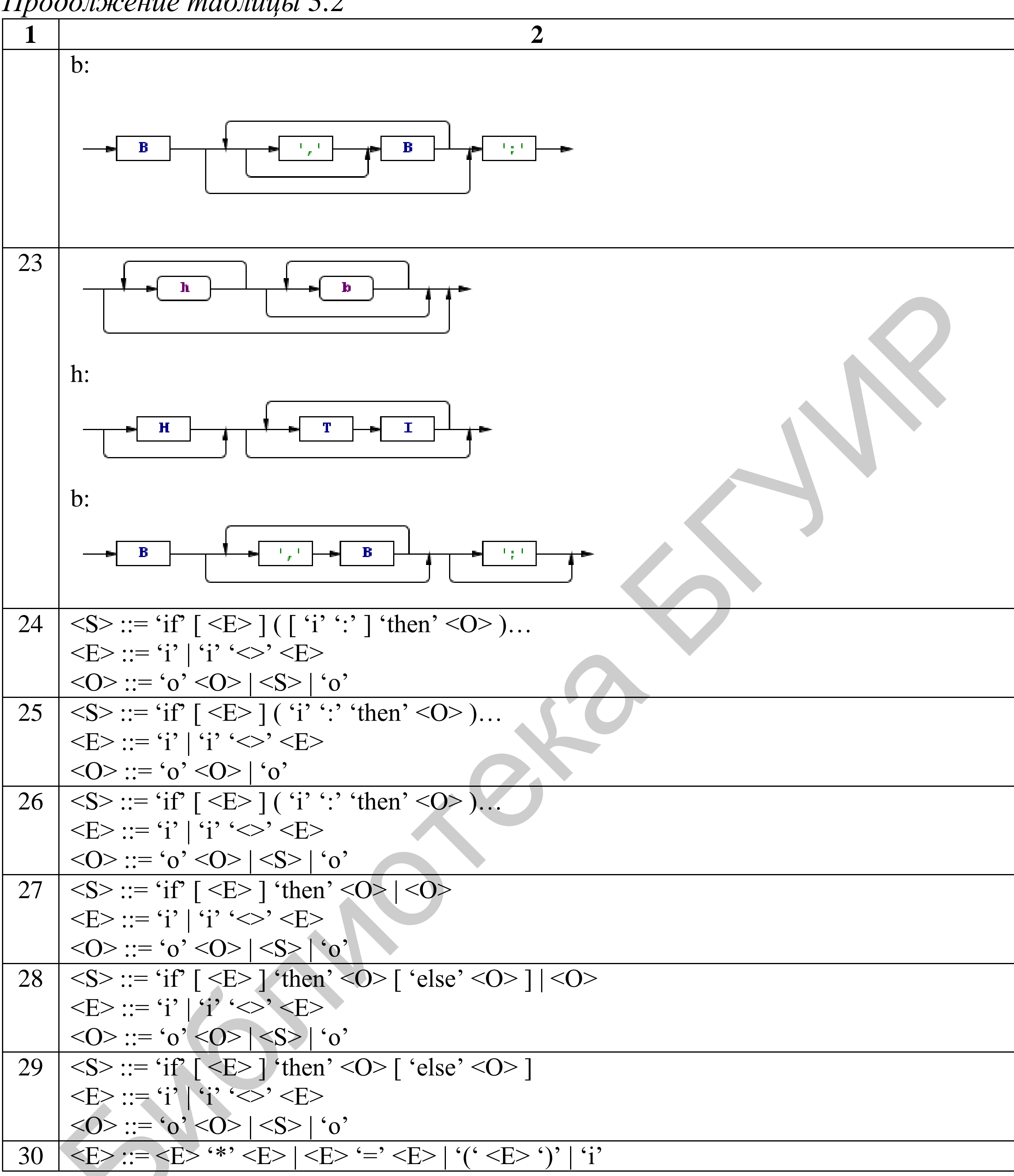
*Продолжение таблицы 3.2*




<table>
<tr><th>1</th><th>2</th></tr>
<tr><td>14</td><td><P> ::= ([ <H> ] [ <B> ])…<br><H> ::= ‘h’ (‘t’ ‘i’)…<br><B> ::= ‘b’ (‘,’ ‘b’)… ‘;’</td></tr>
<tr><td>15</td><td><P> ::= (<H> | <B>)…<br><H> ::= ‘h’ (‘t’ ‘i’)…<br><B> ::= ‘b’ (‘,’ ‘b’)… ‘;’</td></tr>
<tr><td>16</td><td><P> ::= <H> [ <B> ]<br><H> ::= ‘h’ (‘t’ ‘i’)… | ε<br><B> ::= ‘b’ (‘,’ ‘b’)… ‘;’</td></tr>
<tr><td>17</td><td><E> ::= <E> ‘–’ <E> | <E> ‘+’ <E> | ‘(‘ <E> ‘)’ | ‘i’</td></tr>
<tr><td>18</td><td><S> ::= ‘if’ <E> ‘then’ <O> [ ‘else’ <O> ]<br><E> ::= ‘i’ | ‘i’ ‘<>’ <E><br><O> ::= ‘o’ <O> | <S> | ‘o’</td></tr>
<tr><td>19</td><td></td></tr>
<tr><td>20</td><td>h:<br>b:</td></tr>
<tr><td>21</td><td>h:<br>b:</td></tr>
<tr><td>22</td><td>h:</td></tr>
</table>

Библиотека БГУИР

*Продолжение таблицы 3.2*



<table>
<tr><th>1</th><th>2</th></tr>
<tr><td></td><td>b:<br>B ',' B ';'</td></tr>
<tr><td>23</td><td>h b<br>h:<br>H T I<br>b:<br>B ',' B ';'</td></tr>
<tr><td>24</td><td><S> ::= ‘if’ [ <E> ] ( [ ‘i’ ‘:’ ] ‘then’ <O> )…<br><E> ::= ‘i’ | ‘i’ ‘<>’ <E><br><O> ::= ‘o’ <O> | <S> | ‘o’</td></tr>
<tr><td>25</td><td><S> ::= ‘if’ [ <E> ] ( ‘i’ ‘:’ ‘then’ <O> )…<br><E> ::= ‘i’ | ‘i’ ‘<>’ <E><br><O> ::= ‘o’ <O> | ‘o’</td></tr>
<tr><td>26</td><td><S> ::= ‘if’ [ <E> ] ( ‘i’ ‘:’ ‘then’ <O> )…<br><E> ::= ‘i’ | ‘i’ ‘<>’ <E><br><O> ::= ‘o’ <O> | <S> | ‘o’</td></tr>
<tr><td>27</td><td><S> ::= ‘if’ [ <E> ] ‘then’ <O> | <O><br><E> ::= ‘i’ | ‘i’ ‘<>’ <E><br><O> ::= ‘o’ <O> | <S> | ‘o’</td></tr>
<tr><td>28</td><td><S> ::= ‘if’ [ <E> ] ‘then’ <O> [ ‘else’ <O> ] | <O><br><E> ::= ‘i’ | ‘i’ ‘<>’ <E><br><O> ::= ‘o’ <O> | <S> | ‘o’</td></tr>
<tr><td>29</td><td><S> ::= ‘if’ [ <E> ] ‘then’ <O> [ ‘else’ <O> ]<br><E> ::= ‘i’ | ‘i’ ‘<>’ <E><br><O> ::= ‘o’ <O> | <S> | ‘o’</td></tr>
<tr><td>30</td><td><E> ::= <E> ‘*’ <E> | <E> ‘=’ <E> | ‘(‘ <E> ‘)’ | ‘i’</td></tr>
</table>

Библиотека БГУИР

## ЛАБОРАТОРНАЯ РАБОТА №4
## ПОСТРОЕНИЕ ТАБЛИЦЫ SLR-АНАЛИЗАТОРА

*Цель работы*: получить навык построения таблицы SLR-анализатора.

### 4.1 Теоретические сведения

***Основная идея SLR-метода*** – построение на базе грамматики ДКА для распознавания активных префиксов. Мы группируем пункты в множества, которые приводят к состояниям SLR-анализатора. Пункты могут рассматриваться как состояния НДКА, распознающего активные префиксы.

Каноническая система LR(0)-пунктов обеспечивает основу для построения SLR-анализаторов.

В данной задаче будет рассмотрен метод восходящего синтаксического анализа, известный как синтаксический анализ типа «перенос/свертка» (ПС-анализ).

***Восходящий синтаксический анализ*** – такой вид синтаксического анализа, который пытается строить дерево разбора для входной строки, начиная с листьев (снизу) и работая по направлению к корню дерева (вверх).

Пусть $G = (V, N, S, P)$ – КС-грамматика, $\xi$ – цепочка в объединенном алфавите, $A \rightarrow \alpha \in P$.

При $\alpha \sqsubseteq \xi$ вхождение $\alpha$ в $\xi$ $(\beta, \alpha, \gamma)$ называют основой, если цепочка $\beta A \gamma$ выводима из аксиомы S при условии, что цепочка $\xi$ выводима из аксиомы S.

Таким образом, по определению основа – это такое вхождение правой части некоторого правила КС-грамматики в некоторую выводимую из аксиомы цепочку, что после замены этой правой части соответствующим нетерминалом полученная цепочка снова будет выводимой из аксиомы. Заметим, что для не выводимой из аксиомы цепочки любое вхождение правой части некоторого правила в нее можно считать основой.

***LR-грамматика*** – такая грамматика, для которой возможно построить таблицу синтаксического анализа. По сути для того чтобы грамматика была LR-грамматикой, достаточно, чтобы ПС-анализатор, читающий входной поток слева направо, был способен распознавать основы только при их появлении в стеке.

Рассмотрим наиболее простой метод LR-анализа, который называется ***SLR-анализом*** (Simple LR, простой LR). Недостатком этого метода является достаточно небольшое число грамматик, с которыми он работает, однако этот метод наиболее прост в реализации. Таблицу, построенную таким методом,



Библиотека БГУИР

будем называть SLR-таблицей, а синтаксический анализатор, работающий с SLR-таблицей, – SLR-анализатором, а соответствующую грамматику – SLR-грамматикой.

***LR(0)-пункт***, или ***элемент*** грамматики G, – продукция G с точкой в некоторой позиции правой части. Следовательно, продукция, $A \rightarrow XYZ$ дает четыре пункта:

1) $A \rightarrow \cdot XYZ$;
2) $A \rightarrow X \cdot YZ$;
3) $A \rightarrow XY \cdot Z$;
4) $A \rightarrow XYZ$.

Продукция $A \rightarrow \varepsilon$ генерирует только один пункт, $A \rightarrow \cdot$ . По сути, пункт указывает, какую часть продукции мы уже увидели в данной точке в процессе синтаксического анализа. Например, первый пункт, приведенный выше, определяет, что во входном потоке мы ожидаем встретить строку, порождаемую XYZ. Второй пункт указывает, что у нас уже есть строка, порожденная X, и мы ожидаем получить из входного потока строку, порождаемую YZ.

Система LR(0)-пунктов, которую назовем *канонической*, обеспечивает основу для построения SLR-анализаторов. Для построения канонической LR(0)-системы грамматики мы определим расширенную грамматику и две функции – ***closure*** и ***goto***.

Если G – грамматика со стартовым символом S, то G', *расширенная грамматика (augmented grammar)*, представляет собой G с новым стартовым символом S' и продукцией $S' \rightarrow S$. Назначение этой новой стартовой продукции – указать синтаксическому анализатору, когда он должен прекратить разбор и объявить о допущении входной строки. Таким образом, допуск строки происходит только тогда, когда синтаксический анализатор выполняет свертку, соответствующую продукции $S' \rightarrow S$.

**Операция замыкания (closure)**

Если I – множество пунктов грамматики G, то closure(I) – множество пунктов, построенное из I по следующим правилам.

1. Изначально в closure(I) входят все пункты из I.

2. Если $A \rightarrow \alpha \cdot B\beta$ входит в closure(I) и $B \rightarrow \gamma$ представляет собой продукцию, то добавляем в closure(I) пункт $B \rightarrow \cdot\gamma$ (если его там еще нет). Мы применяем это правило до тех пор, пока не внесем все возможные пункты в closure(I).



Библиотека БГУИР

Наличие A → α·Bβ в closure(I) указывает, что в некоторый момент в процессе синтаксического анализа мы полагаем, что можем встретить во входном потоке подстроку, выводимую из Bβ. Но если имеется продукция B → γ, то, естественно, мы также можем встретить в этот момент строку, выводимую из γ, поэтому включаем B → ·γ в closure(I).

```
function closure(I);
begin
        J := I;
        repeat
for каждый элемент A → α·Bβ из J и каждая продукция B → γ грамматики
G такая, что B → · γ в J не входит в J do
        добавить B →  γ в J
        until больше добавлять в J нечего;
        return J
end
```

**Операция goto**

Второй полезной функцией является goto(I, X), где I является множеством пунктов, а X – символом грамматики. Функция goto(I, X) определяется как замыкание (closure) множества всех пунктов [A → α·Bβ] ϵ I. По сути, если I является множеством пунктов, допустимых для некоторого активного префикса γ, то goto(I, X) есть множество пунктов, допустимых для активного префикса γX.

```
goto (I, X)
{
        J = {}; /* {} обозначает пустое множество */
        for (каждой ситуации [A → w.Xv] из I) {
                J += [A → wX.v];
        }
        return closure (J);
}
```

***Алгоритм построения С-системы множества пунктов для расширенной грамматики G'***

Приведем алгоритм, который позволяет построить каноническую систему LR(0)-пунктов для некоторой расширенной грамматики. Данный алгоритм использует описанные выше операции closure и goto:

Библиотека БГУИР

```
procedure items(G');
begin
    C := {closure( { [S' → ·S] } ) };
    repeat
        for каждое множество пунктов I в C и каждый символ
            грамматики X, такой, что goto(I, X) не является пустым
            и не принадлежит C do
              добавить goto(I, X) к C
        until больше нет множеств, которые можно добавить к C
end
```

***Алгоритм построения таблицы SLR-анализатора***

Данный алгоритм строит функции ***action*** и ***goto*** SLR-анализа (эти функции будут описаны ниже при описании алгоритма LR-анализа). Он не дает однозначно определенные таблицы действий для всех грамматик, но успешно работает для многих грамматик языков программирования. Данную грамматику G необходимо расширить до грамматики G' и на основе G' строить C – каноническую систему множества пунктов для G' (алгоритм 4.1). По системе C строим action, функцию действий синтаксического анализа, и goto, функцию переходов, в соответствии с приведенным далее алгоритмом. Он требует знания FOLLOW(A) для каждого нетерминала A грамматики.

**Вход:** расширенная грамматика G'.

**Выход:** функции action и goto таблицы SLR-анализа для грамматики G'.

**Алгоритм**

1. Построим $C = \{I_0, I_1, \ldots, I_n\}$ – систему множества LR(0)-пунктов для грамматики G'.

2. Состояние i строится на основе $I_i$. Действия синтаксического анализа для состояния i определяются следующим образом:

а) если $[A \rightarrow \alpha{\cdot}a\beta] \in I_i$ и $goto(I_i, a) = I_j$, то определить action[i, a] как «перенос j»; здесь a должно быть нетерминалом;

б) если $[A \rightarrow \alpha\cdot] \in I_i$, то определить action[i, a] как «свертка $A \rightarrow \alpha$» для всех a из FOLLOW(A); здесь A не должно быть S';

в) если $[S' \rightarrow S\cdot] \in I_i$, то определить action[i, $] как «допуск».

Если по этим правилам генерируются конфликтующие действия, мы говорим, что грамматика не является SLR(1). Алгоритм не в состоянии построить синтаксический анализатор для нее. В этом случае необходимо преобразовать грамматику (возможно изменение языка) для SLR(1)-анализа.



Библиотека БГУИР

3. Переходы ***goto*** для состояния i и всех нетерминалов A строятся по правилу: если goto($I_i$, A) = $I_j$, то goto[i, A] = j.

4. Все записи, не определенные по правилам 2 и 3, указываются как «ошибка».

5. Начальное состояние синтаксического анализатора представляет собой состояние, построенное из множества пунктов, содержащего [S' → S].

Таблица синтаксического анализа, состоящая из функций ***action*** и ***goto***, определяемых алгоритмом 4.2, называется *SLR(1)-таблицей грамматики G*. LR-анализатор, использующий SLR(1)-таблицу для грамматики G, называется SLR(1)-анализатором для G, а соответствующая грамматика – SLR(1)-грамматикой. Обычно при указании SLR(1) часть (1) опускается, поскольку мы не работаем более чем с одним символом предпросмотра.

**Алгоритм LR-анализа**

Анализатору требуется несколько пунктов для работы:

- сам входной поток, который будет анализироваться;
- стек – структура данных, которая отвечает правилу LIFO (Last In First Out) для состояний парсера;
- таблица действий, которая подсказывает, что делать в текущем состоянии и с текущим символом на входе;
- таблица переходов – вспомогательная таблица, которая используется в одном из действий.

Необходимо уточнить, как будет работать анализатор. Текущим состоянием считается состояние на вершине стека. Смотрим в таблицу действий (индексами являются текущий входной символ и текущее состояние). В этой таблице могут быть четыре типа записей:

- успех (accepted(accept)) – входная строка принадлежит данной грамматике;
- пустота (error(e)) – нет никаких действий, мы в тупике, пользователь ошибся с текущим символом;
- перенос (shift(s)) – на вершину стека переносится состояние, которое отвечает входному символу, далее читается следующий символ;
- свертка (reduce(r)) – в стеке набрались состояния, которые мы можем заменить одним, используя правило грамматики, здесь мы берем значение в таблице переходов. Первый индекс – текущее состояние, второй – левая часть правила, то есть то, во что мы свернули последовательность состояний (рисунок 4.1).

Библиотека БГУИР

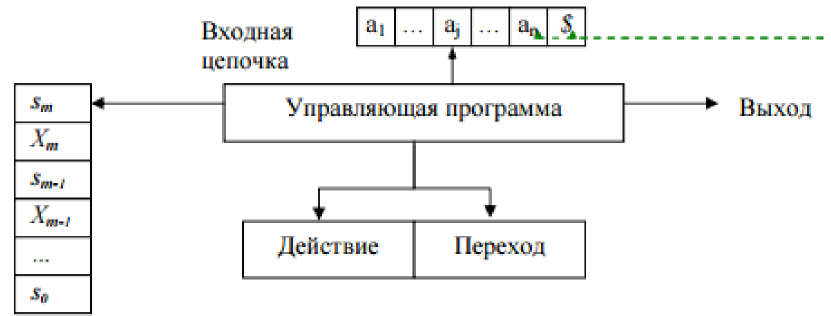



Рисунок 4.1 – Свернутая последовательность состояний

**Модель LR-анализатора**

Управляющая программа для всех LR-анализаторов одна и та же; изменяются только таблицы синтаксического анализа. Программа синтаксического анализа считывает символы из входного буфера по одному и использует стек для хранения строк вида $s_0X_1s_1X_2s_2…X_ms_m$ ($s_m$ находится на вершине стека). Каждый символ $X_i$ является символом грамматики, а каждый $s_i$ – символом, именуемым *состоянием*. Каждый символ состояния обобщает информацию, содержащуюся в стеке ниже его. Комбинация символа и состояния на вершине стека и текущего входного символа используется в качестве индекса таблицы синтаксического анализа и определяет дальнейшее действие – перенос или свертку. При реализации грамматические символы не обязаны появляться в стеке.

Таблица синтаксического анализа состоит из двух частей – функций действий синтаксического анализа ***action*** и функции переходов ***goto.*** Управляющая программа LR-анализатора функционирует следующим образом. Она определяет $s_m$, текущее состояние на вершине стека, и $a_i$, текущий входной символ. Затем программа обращается к action[$s_m$, $a_i$], ячейке таблицы действий синтаксического анализа, определяемой состоянием $s_m$ и символами $a_i$, которая может иметь одно из четырех значений:

1) перенос s, где s – состояние;
2) свертка в соответствии с продукцией $A \rightarrow \beta$;
3) допуск (accept);
4) ошибка (error).

Функция goto получает в качестве аргументов состояние и символ грамматики и возвращает новое состояние.

*Конфигурация* LR-анализатора представляет собой пару, первый компонент которой – содержимое стека, а второй – непросмотренная часть входного потока: ($s_0X_1s_1X_2s_2…X_ms_m$, $a_ia_{i+1}…a_n$\$).

Библиотека БГУИР

Следующий шаг синтаксического анализатора определяется текущим входным символом $a_i$ и состоянием на вершине стека $s_m$ в соответствии со значением ячейки таблицы action[$s_m$, $a_i$]. Конфигурации, получаемые после каждого из четырех типов действий, следующие.

1. Если action[$s_m$, $a_i$] = “перенос s”, синтаксический анализатор выполняет перенос, переходя в конфигурацию ($s_0X_1s_1X_2s_2\ldots X_ms_ma_is$, $a_{i+1}\ldots a_n\$$). Синтаксический анализатор переносит в стек текущий входной символ $a_i$ и очередное состояние s, определяемое значением action[$s_m$, $a_i$]; текущим входным символом становится $a_{i+1}$.

2. Если action[$s_m$, $a_i$] = “свертка $A \rightarrow \beta$”, то синтаксический анализатор выполняет свертку в конфигурацию ($s_0X_1s_1X_2s_2\ldots X_{m-r}s_{m-r}As$, $a_ia_{i+1}\ldots a_n\$$), где s = goto[$s_{m-r}$, $a_i$], а r – длина β правой части продукции. Здесь синтаксический анализатор вначале снимает со стека 2r символов (r символов состояний и r символов грамматики), выводя на вершину стека состояние $s_{m-r}$. Затем вносит в стек A (левую часть продукции) и s запись из ячейки goto[$s_{m-r}$, A]. Текущий входной символ при этом не изменяется. Последовательность снимаемых со стека символов грамматики $X_{m-r+1}\ldots X_m$ всегда соответствует β, правой части продукции свертки.

3. Если action[$s_m$, $a_i$] = “допуск”, синтаксический анализ завершается.

4. Если action[$s_m$, $a_i$] = “ошибка”, синтаксический анализатор обнаружил ошибку и вызывает подпрограмму восстановления после нее.

Полностью алгоритм LR-анализа приведен ниже. Все LR-анализаторы ведут себя одинаково, единственная разница между ними заключается в таблицах action и goto.

**Вход:** входная строка w и таблица LR-анализа с функциями action и goto для грамматики G.

**Выход:** если $w \in L(G)$, выдается восходящий разбор для w; в противном случае выводится сообщение об ошибке.

**Алгоритм.** Изначально синтаксический анализатор содержит в стеке начальное состояние $s_0$, а во входном буфере – w$. Затем анализатор выполняет приведенный ниже алгоритм до тех пор, пока не будет достигнуто успешное завершение анализа или обнаружена ошибка:

*Установить указатель ip на первый символ w$;*

**repeat forever begin**

*Пусть s – состояние на вершине стека, а a – символ, на который указывает ip*

**if** action[s,a] = “перенос s” **then begin**



Библиотека БГУИР

Поместить в стек a, затем s’; переместить ip к следующему входному символу

**end**

**else if** action[s,a] = “свертка A → β” **then begin**

Снять со стека 2*|β| символов. Пусть s’ – текущее состояние на вершине стека. Поместить в стек A, затем goto[s’,A]; вывести продукцию A → β.

**end**

**else if** action[s,a] = “допуск” **then**

**return**

**else** error()

**end**

**Пример части таблицы LR-анализа**

Таблица 4.1 – Часть таблица LR-анализа

| Состояние | action | | | | | | goto | | |
|---|---|---|---|---|---|---|---|---|---|
| | Id | + | * | ( | ) | $ | E | T | F |
| 0 | s5 | – | – | s4 | – | – | 1 | 2 | 3 |
| 1 | – | s6 | – | – | – | accept | – | – | – |
| 2 | – | r2 | s7 | – | r2 | r2 | – | – | – |

Поясним обозначения в данной таблице:

- $s_i$ означает перенос и i-е состояние на вершине стека;
- $r_j$ означает свертку в соответствии с продукцией номер j;
- accept означает допуск входной строки;
- пустая ячейка означает ошибку;
- в части ***action*** находятся терминальные символы грамматики, в части ***goto***, соответственно, нетерминальные.

### 4.1.1 Условие

1. Определить значение функций FIRST и FOLLOW для разработанной грамматики.
2. Построить множество пунктов.
3. Построить диаграмму переходов.
4. Построить таблицу SLR-анализатора с учетом вариантов восстановления после ошибок.
5. Проверить правильность построения на трех примерах (один правильный, два неправильных).



Библиотека БГУИР

### 4.1.2 Пример построения таблицы SLR-анализатора

1. Исходная грамматика для декларации функции:

```
S → Aa
A → Ac|Bb|ε
B → cB| ε
```

2. Видим, что присутствует левая рекурсия. Избавляемся от нее:

```
S → Aa
A → cA|Bb|ε
B → cB| ε
```

3. Построим функции FIRST и FOLLOW для грамматики, заданной в условии:

FIRST(S) = {a, b, c}  FOLLOW(S) = {$}
FIRST(A) = {c, b, ε}  FOLLOW(A) = {a}
FIRST(B) ={c, ε}  FOLLOW(B) = {a}

4. Построим множество пунктов. Для этого пронумеруем правила и добавим фиктивное нулевое правило. После этого грамматика примет вид:

```
1) S′ → S;
2) S → Aa;
3) A → cA;
4) A → bB;
5) A → ε;
6) B → cB;
7) B → ε.
```

1. Нулевой пункт $I_0$ будет состоять из результата функции closure( { [S’ → ·S] } ).

2. Подсчитываем результат функции closure({[S’ → ·S]}):

i) $I_0$={ S’ → ·S}. Поскольку пункт стоит перед нетерминалом S, то пытаемся добавить в $I_0$ пункт S → ·Aa. Такого пункта в $I_0$ нет, поэтому добавляем его;

ii) $I_0$ = { S’ → ·S, S → ·Aa}. Поскольку пункт стоит перед нетерминалом A, то пытаемся добавить в $I_0$ пункты A → ·cA, A → ·bB, A → ·; таких пунктов нет в $I_0$, поэтому добавляем их;

iii) $I_0$ = { S’ → ·S, S → ·Aa, A → ·cA, A → ·bB, A → ·}. Поскольку пункты стоят перед уже разобранным терминалом и нетерминалами, то множество пунктов $I_0$ сформировано.

3. Подсчитываем $I_1$ = goto($I_0$, S) = closure({[S’ → S·]}):

i) $I_1$ = {S’ → S·}. Поскольку пункт стоит только перед символами ε, то множество пунктов $I_1$ сформировано.

Библиотека БГУИР

4. Подсчитываем $I_2$ = goto($I_0$, A) = closure({[S → A·a]}):

i) $I_2$ = {S → A·a}. Поскольку пункт стоит только перед терминалами, то множество пунктов $I_2$ сформировано.

5. Подсчитываем $I_3$ = goto($I_2$, a) = closure ({[S → Aa·]}):

i) $I_3$ = {S → Aa·}. Поскольку пункт стоит только перед символами ε, то множество пунктов $I_3$ сформировано.

6. Подсчитываем $I_4$ = goto($I_0$, c) = closure ({[A → c·A]}):

i) $I_4$ = {A → c·A}. Поскольку пункт стоит перед нетерминалом A, то пытаемся добавить в $I_4$ пункты A → ·cA, A → ·bB, A → ·; таких пунктов нет в $I_4$, поэтому добавляем их;

ii) $I_4$ = {A → c·A, A → ·cA, A → ·bB, A → ·}. Поскольку пункт стоит перед уже разобранным нетерминалом A и терминалами, то множество пунктов $I_4$ сформировано.

7. Подсчитываем $I_5$ = goto($I_4$, A) = closure({[A → cA·]}):

i) $I_5$ = {A → cA·}. Поскольку пункт стоит только перед символами ε, то множество пунктов $I_5$ сформировано.

8. Подсчитываем $I_6$ = goto($I_0$, b) = closure({[A → b·B]}):

i) $I_6$ = {A → b·B}. Поскольку пункт стоит перед нетерминалом B, то пытаемся добавить в $I_6$ пункты B → ·cB, B → ·; таких пунктов нет в $I_6$, поэтому добавляем их;

ii) $I_6$ = {A → b·B, B → ·cB, B → ·}. Поскольку пункт стоит перед уже разобранным нетерминалом B и терминалами, то множество пунктов $I_6$ сформировано.

9. Подсчитываем $I_7$ = goto($I_6$, B) = closure({[A → bB·]}):

i) $I_7$ = {A → bB·}. Поскольку пункт стоит только перед символами ε, то множество пунктов $I_5$ сформировано.

10. Подсчитываем $I_8$ = goto($I_6$, c) = closure({[B →c·B]}):

i) $I_8$ = {B → c·B}. Поскольку пункт стоит перед нетерминалом B, то пытаемся добавить в $I_8$ пункты B → ·cB, B → ·; таких пунктов нет в $I_8$, поэтому добавляем их;

ii) $I_8$ = {B → c·B, B → ·cB, B → ·}. Поскольку пункт стоит перед уже разобранным нетерминалом B и терминалами, то множество пунктов $I_8$ сформировано.

11. Подсчитываем $I_9$ = goto($I_8$, B) = closure({[B →cB·]}):

i) $I_9$ = {B → cB·}. Поскольку пункт стоит только перед символами ε, то множество пунктов $I_9$ сформировано.

Библиотека БГУИР

12. Больше множеств пунктов построить нельзя, поскольку либо все пункты расположены либо после символов ε, либо не порождают новых состояний.

Множество пунктов для данной грамматики имеет вид:

- $I_0$ ={S′ → ·S, S → ·Aa, A → ·cA, A → ·bB, A → ·}
- $I_1$ = {S′ → S·}
- $I_2$ = {S → A· a}
- $I_3$ = {S → Aa·}
- $I_4$ = {A → c·A, A → ·cA, A → ·bB, A → ·}
- $I_5$ = {A → cA·}
- $I_6$ = {A → b·B, B → ·cB, B → ·}
- $I_7$ = {A → bB·}
- $I_8$ = {B → c·B, B → ·cB, B →· }
- $I_9$ = {B → cB·}

5. Диаграмма переходов представляет собой ориентированный граф, вершинами которого являются состояния, а дугами – переходы между этими состояниями.

Матрица смежности этого графа будет иметь следующий вид (таблица 4.2).

Таблица 4.2 – Матрица смежности графа

| Состояние | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | – | S | A | – | c | – | B | – | – | – |
| 1 | – | – | – | – | – | – | – | – | – | – |
| 2 | – | – | – | a | – | – | – | – | – | – |
| 3 | – | – | – | – | – | – | – | – | – | – |
| 4 | – | – | – | – | c | A | b | – | – | – |
| 5 | – | – | – | – | – | – | – | – | – | – |
| 6 | – | – | – | – | – | – | – | B | c | – |
| 7 | – | – | – | – | – | – | – | – | – | – |
| 8 | – | – | – | – | – | – | – | – | c | B |
| 9 | – | – | – | – | – | – | – | – | – | – |

Диаграмма переходов, построенная на основании приведенной выше матрицы смежности, имеет вид, представленный на рисунке 4.2.

Библиотека БГУИР

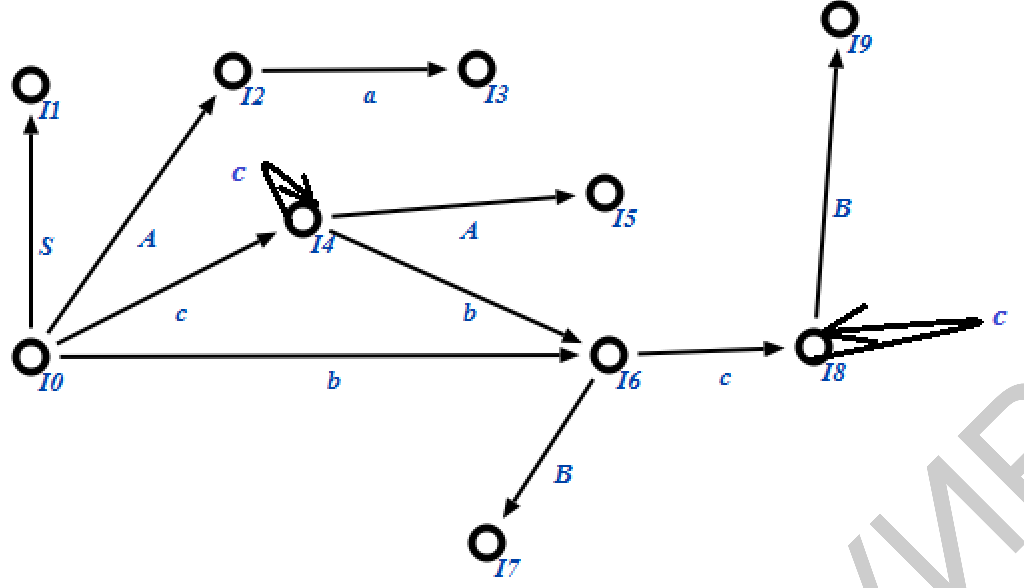



Рисунок 4.2 – Диаграмма переходов

6. Строим таблицу SLR-анализатора.

Рассматриваем последовательно каждое состояние, заполняя функции ***action*** и ***goto*** в таблице 4.3.

Таблица 4.3 – Таблица SLR-анализатора

| Состояние | action | | | | goto | | |
|---|---|---|---|---|---|---|---|
| | A | B | c | $ | A | B | S |
| 0 | r4 | S6 | S4 | – | 2 | – | 1 |
| 1 | – | – | – | accept | – | – | – |
| 2 | S3 | – | – | – | – | – | – |
| 3 | – | – | – | r1 | – | – | – |
| 4 | r4 | S6 | S4 | – | 5 | – | – |
| 5 | r2 | – | – | – | – | – | – |
| 6 | r6 | – | S8 | – | – | 7 | – |
| 7 | r3 | – | – | – | – | – | – |
| 8 | r6 | – | S8 | – | – | 9 | – |
| 9 | r5 | – | – | – | – | – | – |

7. Примеры разбора текстов, порожденных данной грамматикой, с помощью построенной таблицы SLR-анализатора.



Библиотека БГУИР

***Правильный пример*:**
ccbcca

Таблица 4.4 – Разбор текст ccbcca

| Стек | Входной поток | Комментарии |
|---|---|---|
| **1** | **2** | **3** |
| $0 | ccbcca$ | action[I0, c] = S4 Осуществляем перенос c в стек, а также добавляем в него состояние 4 |
| $0c4 | cbcca$ | action[I4, c] = S4 Осуществляем перенос c в стек, а также добавляем в него состояние 4 |
| $0c4c4 | bcca$ | action[I4, b] = S6 Осуществляем перенос b в стек, а также добавляем в него состояние 6 |
| $0c4c4b6 | cca$ | action[I6, c] = S8 Осуществляем перенос c в стек, а также добавляем в него состояние 8 |
| $0c4c4b6c8 | ca$ | action[I8, c] = S8 Осуществляем перенос c в стек, а также добавляем в него состояние 8 |
| $0c4c4b6c8c8 | a$ | action[I8, a] = r6 Осуществляем свертку по правилу номер 6. Снимаем со стека 2*0 символов, добавляем в него сворачиваемый нетерминал, а также состояние, которое указано в ячейке goto[I8, B] = 9 |

Библиотека БГУИР

*Продолжение таблицы 4.4*

| 1 | 2 | 3 |
|---|---|---|
| $0c4c4b6c8c8B9 | a$ | action[I8, a] = r6<br>Осуществляем свертку по правилу номер 6. Снимаем со стека 2*0 символов, добавляем в него сворачиваемый нетерминал, а также состояние, которое указано в ячейке goto[I8, B] = 9 |
| $0c4c4b6c8B9 | a$ | action[I6, a] = r6<br>Осуществляем свертку по правилу номер 6. Снимаем со стека 2*0 символов, добавляем в него сворачиваемый нетерминал, а также состояние, которое указано в ячейке goto[I6, B] = 7 |
| $0c4c4b6B7 | a$ | action[I4, a] = r4<br>Осуществляем свертку по правилу номер 4. Снимаем со стека 2*0 символов, добавляем в него сворачиваемый нетерминал, а также состояние, которое указано в ячейке goto[I4, A] = 5 |
| $0c4c4A5 | a$ | action[I4, a] = r4<br>Осуществляем свертку по правилу номер 4. Снимаем со стека 2*0 символов, добавляем в него сворачиваемый нетерминал, а также состояние, которое указано в ячейке goto[I4, A] = 5 |

Библиотека БГУИР

*Продолжение таблицы 4.4*

| 1 | 2 | 3 |
|---|---|---|
| $0c4A5 | a$ | action[I0, a] = r4<br>Осуществляем свертку по правилу номер 0. Снимаем со стека 2*0 символов, добавляем в него сворачиваемый нетерминал, а также состояние, которое указано в ячейке<br>goto[I0, A] = 2 |
| $0A2 | a$ | action[I2, a] = S3<br>Осуществляем перенос a в стек. Так же добавляем в него состояние 3 |
| $0A2a3 | $ | action[I3, $] = r1<br>Осуществляем свертку по правилу номер 3. Снимаем со стека 2*0 символов, добавляем в него сворачиваемый нетерминал, а также состояние, которое указано в ячейке<br>goto[I3, $] = 1 |
| $0S1 | $ | action[I1, $] = accept<br>Попадаем в ячейку accept. Строка успешно разобрана |

***Неправильный пример*:**

cacbca

Таблица 4.5 – Пример неправильного разбора текста

| Стек | Входной поток | Комментарии |
|---|---|---|
| $0 | cacbca$ | action[I0, c] = S4<br>Осуществляем перенос c в стек, а также добавляем в него состояние 4 |
| $0c4 | acbca$ | action[I4, a] = ???<br>Ошибка, так как ожидается переход по a, т. е. строка неправильная |

Библиотека БГУИР

## 4.2 Варианты индивидуальных заданий

Таблица 4.6 – Варианты индивидуальных заданий

| № п/п | Формулировка варианта задания |
|---|---|
| **1** | **2** |
| 1 | S → Aa<br>A → Ab \| Bb<br>B → cB \| ε |
| 2 | S → ABC<br>A → BC \| a<br>B → A \| b<br>C → c \| ε |
| 3 | S → AB<br>A → aAb \| Ab \| ε<br>B → bB \| b |
| 4 | S → aAb<br>A → BC \| bc<br>B → Aa<br>C → c \| ε |
| 5 | S → AS \| c<br>A → Aa \| Ca<br>C → cA \| c |
| 6 | S → BS \| c<br>A → cB \| c<br>B → Ba \| Aa |
| 7 | S → ABc<br>A → aAb \| Ab \| ε<br>B → bB \| bA |
| 8 | S → Aa<br>A → Ac \| Bb \| ε<br>B → cB \| ε |

Библиотека БГУИР

*Продолжение таблицы 4.6*

| 1 | 2 |
|---|---|
| 9 | S → aBS \| c<br>A → cB \| c<br>B → Ba \| Aa \| ε |
| 10 | S → AC \| c<br>A → Aa \| Ca<br>C → cA \| c |
| 11 | S → SA \| c<br>A → Aa \| Ca<br>C → cA \| c |
| 12 | S → aAb<br>A → BS \| bc<br>B → AaC<br>C → c \| ε |
| 13 | S → aB \| c<br>A → cB \| c<br>B → Ba \| Aa \| ε |
| 14 | S → ABC<br>A → BC \| a<br>B → A \| b<br>C → c \| ε |
| 15 | S → AB<br>A → aAb \| Ab \| ε<br>B → bB \| b |
| 16 | S → aAb<br>A → BC \| bc<br>B → Aa<br>C → c \| ε |
| 17 | S → AS \| c<br>A → Aa \| Ca<br>C → cA \| c |
| 18 | S → BS \| c<br>A → cB \| c<br>B → Ba \| Aa |
| 19 | S → ABc<br>A → aAb \| Ab \| ε<br>B → bB \| bA |

Библиотека БГУИР

*Продолжение таблицы 4.6*

| 1 | 2 |
|---|---|
| 20 | S → Aa<br>A → Ac \| Bb \| ε<br>B → cB \| ε |
| 21 | S → aBS \| c<br>A → cB \| c<br>B → Ba \| Aa \| ε |
| 22 | S → AC \| c<br>A → Aa \| Ca<br>C → cA \| c |
| 23 | S → AC \| c<br>A → Aa \| Ca<br>C → cA \| c |
| 24 | S → SA \| c<br>A → Aa \| Ca<br>C → cA \| c |
| 25 | S → aAb<br>A → BS \| bc<br>B → AaC<br>C → c \| ε |
| 26 | S → aB \| c<br>A → cB \| c<br>B → Ba \| Aa \| ε |
| 27 | S → ABC<br>A → BC \| a<br>B → A \| b<br>C → c \| ε |
| 28 | S → AB<br>A → aAb \| Ab \| ε<br>B → bB \| b |
| 29 | S → aAb<br>A → BC \| bc<br>B → Aa<br>C → c \| ε |
| 30 | S → AS \| c<br>A → Aa \| Ca<br>C → cA \| c |

Библиотека БГУИР

# ЛАБОРАТОРНАЯ РАБОТА №5
# ГЕНЕРАЦИЯ И ОПТИМИЗАЦИЯ КОДА

*Цель работы*: получить навык генерации и оптимизации кода.

## 5.1 Теоретические сведения

***Синтаксически управляемая трансляция*** (СУТ, англ. *Syntax-directed translation*, *SDT*) – преобразование текста в последовательность команд, через добавление таких команд в правила грамматики. Во время обработки строки синтаксический анализатор находит последовательность применений правил. СУТ предоставляет простой способ связи такого синтаксиса с семантикой.

Синтаксически управляемая трансляция работает за счет добавления действий в контекстносвободную грамматику. Эти действия будут осуществляться, когда соответствующее правило используется в выводе. Описание грамматики с такими действиями называется *схемой синтаксически управляемой трансляции* (или просто *схемой трансляции*).

Каждый символ в грамматике может иметь *атрибуты*, которые содержат данные. Обычно такие атрибуты могут включать в себя тип переменной, значение выражения и т. п. Для символа $X$ с атрибутом $t$ обращение к атрибуту может выглядеть как $X.t$.

Таким образом, с использованием действий и атрибутов грамматика может быть применена для перевода текста с языка, порождаемого ею, выполнением действий и переносом информации через атрибуты символов.

Промежуточный код может иметь три **типа представления:**

- синтаксические деревья;
- постфиксная запись;
- трехадресный код.

Далее подробнее о каждом типе представления. Постфиксная запись в данной задаче рассматриваться не будет.

### Синтаксические деревья

Синтаксические деревья бывают двух видов: **собственно синтаксическое дерево** и **даг**. Синтаксическое дерево изображает естественную иерархическую структуру исходной программы. В свою очередь, даг дает ту же информацию, но в более компактном виде (одинаковые подвыражения в нем объединены).

Приведем пример графического представления синтаксического дерева (рисунок 5.1) и дага (рисунок 5.2) для инструкции присваивания a := b * –c + + b * –c. Данный пример не описывает всю специфику представления промежуточного кода в виде синтаксических деревьев. Более подробный пример будет приведен в разборе задачи.

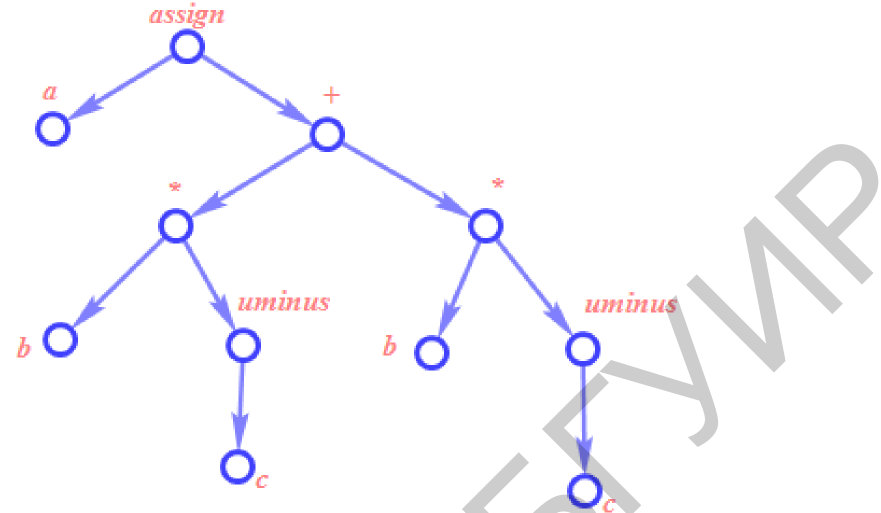



Рисунок 5.1 – Синтаксическое дерево для инструкции a := b * –c + b * –c

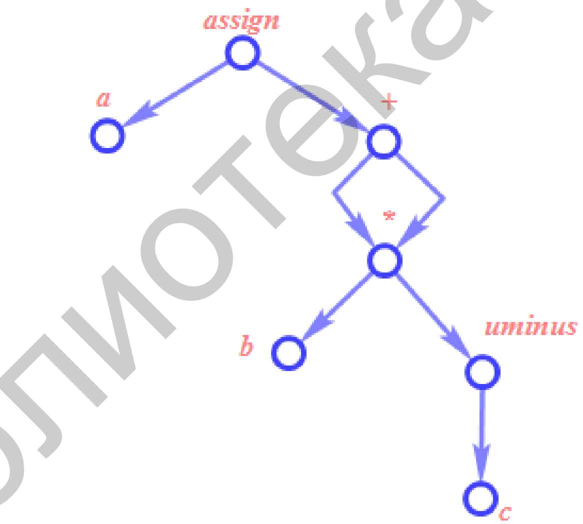



Рисунок 5.2 – Даг для инструкции a := b * –c + b * –c

Опишем примерные семантические правила, которые необходимы для построения промежуточного кода в виде синтаксического дерева для грамматики арифметических выражений (таблица 5.1).

Библиотека БГУИР

Таблица 5.1 – Семантические правила для построения промежуточного кода

| Продукция | Семантическое правило |
|---|---|
| S → id := E | S.nptr := mknode(‘assign’, mkleaf(id, id.place), E.nptr) |
| E → E1 + E2 | E.nptr := mknode(‘+’, E1.nptr, E2.nptr) |
| E → E1 * E2 | E.nptr := mknode(‘*’, E1.nptr, E2.nptr) |
| E → – E1 | E.nptr := mkunode(‘uminus’, E1.nptr) |
| E → (E1) | E.nptr := E1.nptr |
| E → id | E.nptr := mkleaf(id, id.place) |

Поле нетерминального символа ***nptr*** обозначает указатель на некоторый узел дерева.

Операция ***mknode(name, ptr1, ptr2)*** позволяет создать узел дерева, который имеет название name и ссылается на узлы, обозначаемые указателями *ptr1* и *ptr2* соответственно.

Операция ***mkunode(name, ptr)*** позволяет создать узел дерева, который имеет название *name* и ссылается только на узел, обозначаемый указателем *ptr*.

Операция ***mkleaf(name, value)*** позволяет создать лист дерева, который имеет название name и значение value.

Построение дага будет осуществлено в том случае, если модифицировать правила, описанные выше, таким образом, что операции ***mknode(name, ptr1, ptr2)*** и ***mkunode(name, ptr)*** будут не сразу создавать новый узел дерева, а проверять, существует ли такой узел, и если существует, то возвращать указатель на него.

Приведем возможный вариант реализации структуры хранения для синтаксического дерева в компиляторе.

Первый вариант реализации в виде записей с указателями представлен на рисунке 5.3.

Библиотека БГУИР

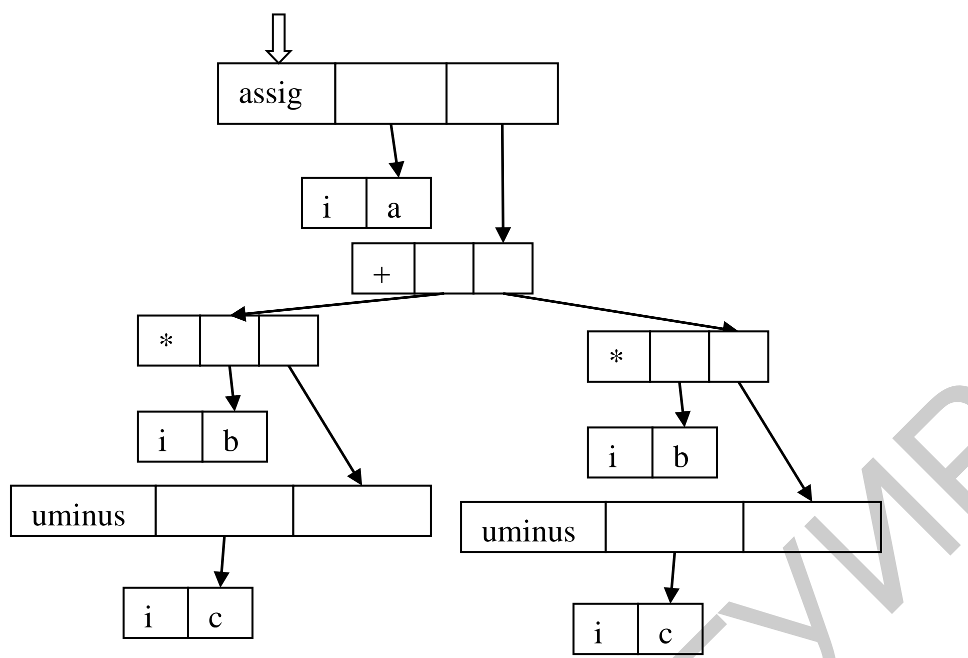


Рисунок 5.3 – Структура хранения для синтаксического дерева
в компиляторе в виде записей с указателями

Второй вариант реализации в виде массива представлен в таблице 5.2.

Таблица 5.2 – Структура хранения для синтаксического дерева в компиляторе в виде массива

| 0 | id | b | |
|---|---|---|---|
| 1 | id | c | |
| 2 | uminus | 1 | |
| 3 | * | 0 | 2 |
| 4 | id | b | |
| 5 | id | c | |
| 6 | uminus | 5 | |
| 7 | * | 4 | 6 |
| 8 | + | 3 | 7 |
| 9 | id | a | |
| 10 | assign | 9 | 8 |

Приведем также аналогичные варианты хранения дага (рисунок 5.4 и таблица 5.3).

Библиотека БГУИР

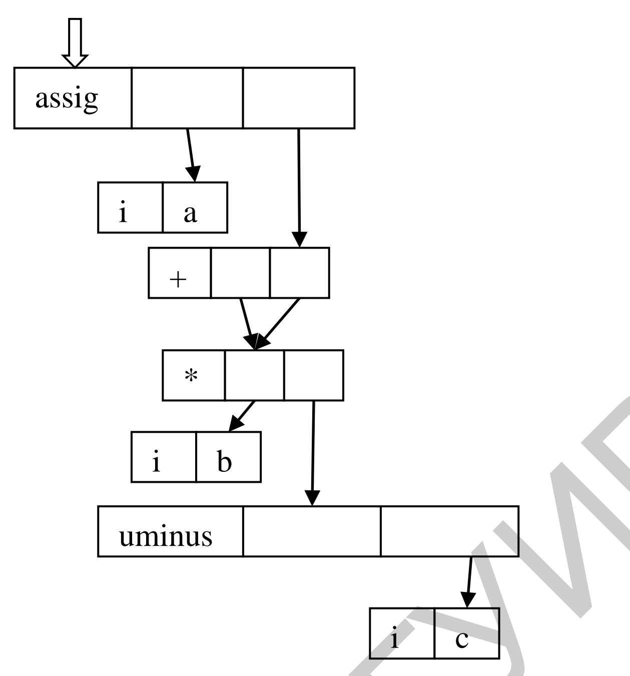



Рисунок 5.4 – Структура хранения для синтаксического дерева
в компиляторе в виде дага

Таблица 5.3 – Структура хранения для синтаксического дерева в компиляторе
в виде дага

| | | | |
|---|---|---|---|
| 0 | id | b | – |
| 1 | id | c | – |
| 2 | uminus | 1 | – |
| 3 | * | 0 | 2 |
| 4 | + | 3 | 3 |
| 5 | id | a | – |
| 6 | assign | 5 | 4 |

Как видно из размера структур данных, даг занимает меньше памяти, чем синтаксическое дерево.

**Трехадресный код**

Трехадресный код представляет собой последовательность инструкций вида x := y op z, где x, y, z – имена, константы, временные переменные, генерируемые компилятором; op означает некоторый оператор, например, арифметический оператор для работы с логическими значениями. Заметим, что не разрешены никакие встроенные арифметические выражения, и в правой части инструкции имеется только один оператор. Следовательно, выражение исходного языка наподобие x + y * z может быть транслировано в следующую последовательность:



Библиотека БГУИР

$t_1 := y * z$

$t_2 := x + t_1$

Здесь $t_1$ и $t_2$ – сгенерированные компилятором временные имена. Такое разложение сложных арифметических выражений и вложенных инструкций потока управления делает трехадресный код подходящим для генерации целевого кода и оптимизации.

Трехадресный код является линеаризованным представлением синтаксического дерева или дага, в котором внутренним узлам графа соответствуют явные имена. Покажем, как при помощи трехадресного кода представить синтаксическое дерево и даг, представленные на рисунках 5.1 и 5.2 соответственно.

Код для синтаксического дерева:

$t_1 := - c$

$t_2 := b * t_1$

$t_3 := - c$

$t_4 := b * t_3$

$a := t_2 + t_4$

Код для дага:

$t_1 := - c$

$t_2 := b * t$

$t_3 := t_2 + t_2$

$a := t_3$

**Типы трехадресных инструкций**

Трехадресные инструкции похожи на ассемблерный код.

Ниже приведен список основных трехадресных инструкций, предлагаемых для использования в данной задаче.

1. Инструкция присваивания вида x := y op z, где op – бинарная арифметическая операция или логическая операция.

2. Инструкция присваивания вида x := op y, где op – унарная операция. Основные унарные операции включают унарный минус, логическое отрицание, операторы сдвига и операторы преобразования, например, преобразуют число с фиксированной точкой в число с плавающей точкой.

3. Инструкции копирования вида x := y, в которых значение y присваивается x.

4. Безусловный переход goto L. После этой инструкции будет выполнена трехадресная инструкция с меткой L.

Библиотека БГУИР

5. Условный переход типа if x relop y goto L. Эта инструкция применяет оператор отношения relop (<, >, = и т. п.) к x и y, и следующей выполняется инструкция с меткой L, если соотношение x relop y верно. В противном случае выполняется следующая за условным переходом инструкция.

6. Инструкции param x и call p, n для вызова процедур и return y для возврата из них, где y обозначает необязательное возвращаемое значение. Обычно они используются в виде следующей последовательности трехадресных инструкций:

param $x_1$,
param $x_2$,
param $x_3$,
…,
param $x_n$
call p, n

Данная последовательность генерируется в качестве части вызова процедуры p($x_1$, $x_2$, …, $x_n$). В инструкции call p, n целое число n, указывающее количество действительных параметров, не является излишним в силу того, что вызовы могут быть вложенными.

7. Индексированные присвоения типа x := y[i] и x[i] := y. Первая инструкция присваивает x значение, находящееся в i-й ячейке памяти по отношению к y. Инструкция x[i] := y заносит в i-ю ячейку памяти по отношению к x значение y. В обеих инструкциях x, y и i ссылаются на объекты данных.

8. Присваивание адресов и указателей вида x := &y, x := *y и *x := y. Первая инструкция устанавливает значение x равным положению y в памяти. Предположительно, y представляет собой имя, возможно временное, обозначающее выражение с l-значением типа A[i, j], а x – имя указателя или временное имя. Таким образом, r-значение x становится равным содержимому этой ячейки. И, наконец, инструкция *x := y устанавливает r-значение объекта, указываемого x, равным r-значению y.

Опишем примерные семантические правила, которые необходимы для построения промежуточного кода в виде трехадресного кода для грамматики арифметических выражений (таблица 5.4).

Библиотека БГУИР

Таблица 5.4 – Семантические правила для построения промежуточного кода в виде трехадресного кода для грамматики арифметических выражений

| Продукция | Семантическое правило |
|---|---|
| S → id := E | S.code := E.code \|\| gen(id.place ‘:=’ E.place) |
| E → E1 + E2 | E.place := newtemp;<br>E.code := E1.code \|\| E2.code \|\| gen(E.place ‘:=’ E1.place ‘+’ E2.place) |
| E → E1 * E2 | E.place := newtemp;<br>E.code := E1.code \|\| E2.code \|\| gen(E.place ‘:=’ E1.place ‘*’ E2.place) |
| E → – E1 | E.place := newtemp;<br>E.code := E1.code \|\| gen(E.place ‘:=’‘uminus’ E1.place) |
| E → (E1) | E.place := E1.place;<br>E.code := E1.code |
| E → id | E.place := id.placel<br>E.code := ‘’ |

Поле нетерминального символа ***place*** обозначает имя, которое хранит нетерминал.

Поле нетерминального символа ***code*** обозначает последовательность трехадресных инструкций, вычисляющих E.

Операция ***newtemp*** последовательно возвращает имена временных переменных $t_1$, $t_2$, …, $t_k$.

Операция *//* позволяет добавить к последовательности трехадресных инструкций новые, сгенерированные при срабатывании данного правила.

Операция ***gen(instruction)*** позволяет генерировать трехадресную инструкцию, которая является конкатенацией строк, записанных в параметрах.

**Реализация трехадресных инструкций**

Трехадресные инструкции представляют собой абстрактный вид промежуточного кода. В компиляторе эти инструкции могут быть реализованы как записи с полями для операторов и операндов. Три возможных представления – это *четверки*, *тройки* и *косвенные тройки*.

**Четверки**

Четверка (quadruple) представляет собой запись с четырьмя полями, которые назовем op, arg1, arg2, result. Поле op содержит внутренний код оператора. Трехадресная инструкция x := y op z представляется размещением y в arg1, z – в arg2 и x – в result. Инструкции с унарным оператором наподобие



Библиотека БГУИР

x := –y или x := y не используют arg2. Операторы типа param не используют ни arg2, ни result. Условные и безусловные переходы помещают в result целевую метку.

Покажем представление в виде четверок операции присваивания (таблица 5.5):

a := b * – c + b * – c

Таблица 5.5 – Представление в виде четверок присваивания

| | op | arg1 | arg2 | result |
|---|---|---|---|---|
| (0) | uminus | c | – | t1 |
| (1) | * | b | t1 | t2 |
| (2) | uminus | c | – | t3 |
| (3) | * | b | t3 | t4 |
| (4) | + | t2 | t4 | t5 |
| (5) | assign | t5 | – | a |

**Тройки**

Для того чтобы избежать вставки временных имен в таблицу символов, мы можем ссылаться на временные значения по номеру инструкции, которая вычисляет значение, соответствующее этому имени. Если мы поступим таким образом, трехадресные инструкции можно будет представить записями только с тремя полями: op, arg1 и arg2. Поля arg1 и arg2 для аргументов op представляют собой либо указатели в таблицу символов, либо указатели на тройки. Поскольку здесь используется три поля, этот вид промежуточного кода известен как тройки (triple). За исключением представления имен, определенных в программе, тройки соответствуют представлению синтаксического дерева или графа в виде массива вершин.

Покажем представление в виде троек операции присваивания (таблица 5.6):

a := b * –c + b * –c

Таблица 5.6 – Представление в виде троек присваивания

| | op | arg1 | arg2 |
|---|---|---|---|
| (0) | uminus | c | – |
| (1) | * | b | (0) |
| (2) | uminus | c | – |
| (3) | * | b | (2) |
| (4) | + | (1) | (3) |
| (5) | assign | a | (4) |

Библиотека БГУИР

**Косвенные тройки**

Еще одно представление трехадресного кода состоит в использовании списка указателей на тройки вместо списка самих троек (таблицы 5.7 и 5.8). Естественно, такая реализация названа косвенными тройками (indirect triples).

Таблица 5.7 – Таблица, содержащая непосредственно тройки

| | op | arg1 | arg2 |
|---|---|---|---|
| (14) | uminus | c | – |
| (15) | * | b | (14) |
| (16) | uminus | c | – |
| (17) | * | b | (16) |
| (18) | + | (15) | (17) |
| (19) | assign | a | (18) |

Таблица 5.8 – Таблица, содержащая ссылки на тройки

| | statement |
|---|---|
| (0) | (14) |
| (1) | (15) |
| (2) | (16) |
| (3) | (17) |
| (4) | (18) |
| (5) | (19) |

**Оптимизация**

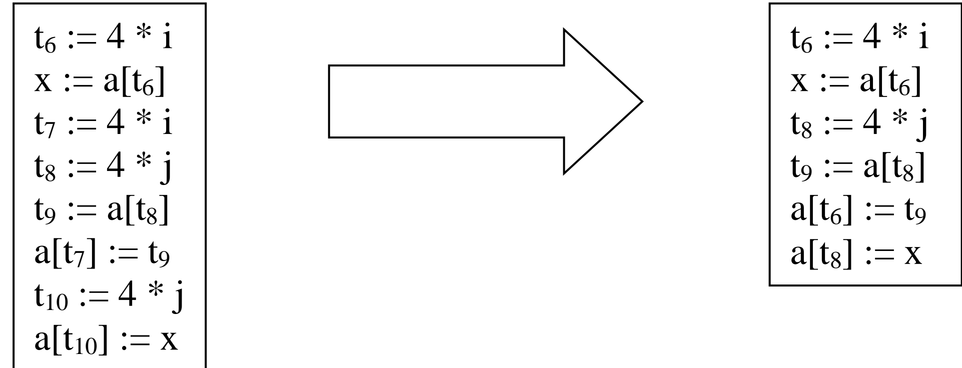
Существует ряд способов, которыми компилятор может улучшить программу без изменения вычисляемой функции. Устранение общих подвыражений, размножение копий, удаление недоступного кода и дублирование констант – вот основные примеры таких преобразований, сохраняющих функции.

**Общие подвыражения**

Выражение E называется **общим подвыражением**, если E было ранее вычислено и значения переменных в E с того времени не изменились. Повторного вычисления можно избежать, если использовать ранее вычисленные значения.

Приведем фрагмент кода с подвыражениями, которые будут устранены в результате оптимизации.

Библиотека БГУИР

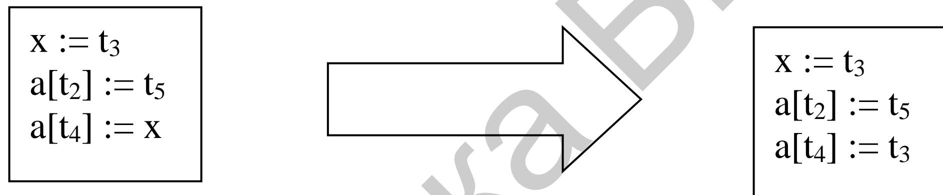



**Распространение копий**

Идея преобразования распространения копий заключается в использовании после присваивания f := g переменной g вместо f везде, где это только возможно.

Сам по себе этот прием не несет улучшения качества кода, однако это возможно совместно с удалением бесполезного кода, которое будет описано ниже.

Приведем фрагмент кода, в котором проведем распространение копий.

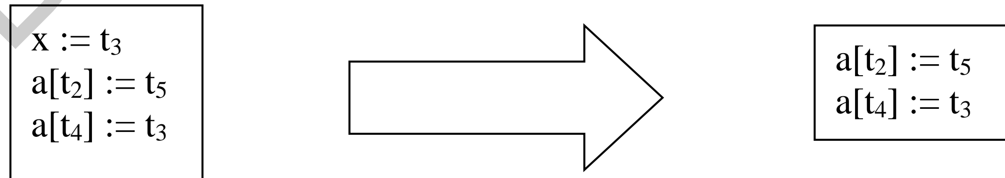



**Удаление бесполезного кода**

Переменная в некоторой точке программы считается «живой», или активной, если ее значение будет использовано в программе в последующем; в противном случае она считается «мертвой». Очередная оптимизация касается «мертвого», или бесполезного кода, т. е. кода, вычисляющего значения, которые никогда не будут использованы.

Таким образом, данная оптимизация совместно с распространением копий превращает инструкции копирования в мертвый код, который впоследствии удаляется.

Приведем фрагмент кода, в котором удалим бесполезный код.




Построение формальной КС-грамматики, а также ее атрибутивной грамматики с семантическими действиями более подробно описано в лабораторном практикуме по курсу.

Библиотека БГУИР

### 5.1.1 Условие

1. Построить промежуточный код в указанном формате (C?) для фрагмента исходного кода (F?) на языке C.

2. Описать типологию команд промежуточного кода. Она должна соответствовать той, что описана выше в теории.

3. Над полученным промежуточным кодом провести оптимизацию. Записать протокол оптимизирующих действий.

4. Полученный после оптимизации код записать в другом формате (→ C?).

### 5.1.2 Примеры решения задачи

**Пример 1**: F3, C1(C2) → C3

1. Фрагмент кода:

```
for (int i = 0; i < 12; i++)
{
  scanf(“%d”, f);
  int d = i * f * 3;
  printf(“%d”, d);
}
```

2. Построение промежуточного кода:

- в виде синтаксического дерева (рисунок 5.5);

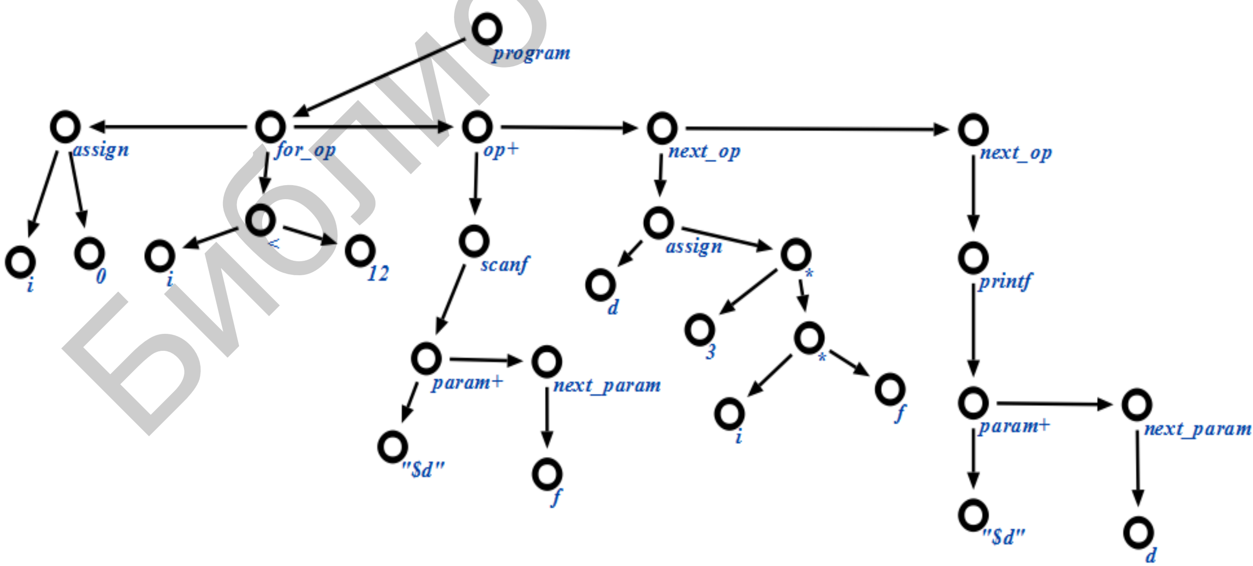



Рисунок 5.5 – Промежуточный код в виде синтаксического дерева

- в виде дага (рисунок 5.6):

Библиотека БГУИР

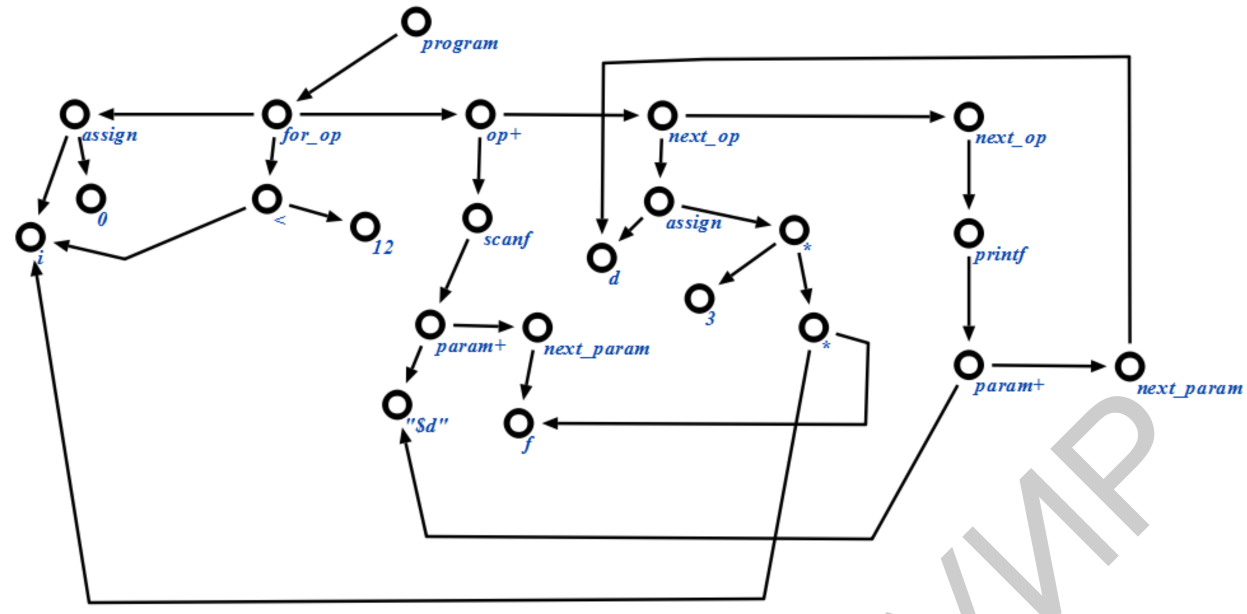



Рисунок 5.6 – Промежуточный код в виде дага

3. Типология команд промежуточного кода.

Для синтаксического дерева и для дага:

- program – программа. Содержит список операторов программы;
- op+ – список операторов программы. Содержит оператор и ссылку на следующий;
- next_op – ссылка на следующий оператор списка. Содержит оператор и ссылку на следующий (ссылка может отсутствовать);
- assign – оператор присваивания. Содержит объект, которому присваивается значение и значение, которое присваивается;
- for_op – оператор итерационного цикла. Состоит из оператора присваивания, условного оператора и списка операторов тела цикла;
- < – логическая операция «меньше». Состоит из двух объектов, которые сравниваются этим оператором;
- strlen – оператор нахождения длины строки. Содержит строку, длину которой находит;
- scanf – оператор ввода. Содержит список параметров;
- param+ – список параметров функции. Содержит параметр и ссылку на следующий параметр;
- next_param – ссылка на следующий параметр функции. Содержит параметр функции и ссылку на следующий (ссылка может отсутствовать);
- & – оператор взятия адреса. Содержит объект, адрес которого берется;
- * – оператор умножения. Содержит умножаемые числа;

Библиотека БГУИР

• printf – оператор вывода. Содержит список параметров;

• [] – оператор обращения к элементу массива по индексу. Содержит массив, к которому осуществляется обращение и индекс запрашиваемого элемента.

4. Оптимизация промежуточного кода.

Для наглядности оптимизацию проведем над исходным кодом, затем покажем, как она отразилась на каждом виде промежуточного кода.

Проведем распространие копий:

```
for (int i = 0; i < 12; i++) {
  scanf("%d", f);
  int d = i * f * 3;
  printf("%d", d);
}
```

→

```
for (int i = 0; i < 12; i++) {
  scanf("%d", f);
  int d = i * f * 3;
  printf("%d", i * f * 3);
}
```

Проведем удаление бесполезного кода:

```
for (int i = 0; i < 12; i++) {
  scanf("%d", f);
  int d = i * f * 3;
  printf("%d", i * f * 3);
}
```

→

```
for (int i = 0; i < 12; i++) {
  scanf("%d", f);
  printf("%d", i * f * 3);
}
```

5. Запишем полученный оптимизированный промежуточный код в виде четверок (таблицы 5.9 и 5.10).

Таблица 5.9 – Ячейки памяти для констант

| Номер ячейки | Значение |
|---|---|
| D01 | "%d" |

Таблица 5.10 – Оптимизированный промежуточный код в виде четверок

| | Op | arg1 | arg2 | result |
|---|---|---|---|---|
| (0) | assign | 0 | – | i |
| (1) | < | i | 12 | t1 |
| (2) | if | t1 | (4) | – |
| (3) | goto | (15) | – | – |



Библиотека БГУИР

*Продолжение таблицы 5.10*

| | | | | |
|---|---|---|---|---|
| (4) | param | D01 | – | – |
| (5) | param | f | – | – |
| (6) | call | scanf | 2 | – |
| (7) | param | D01 | – | – |
| (8) | * | i | f | t2 |
| (9) | * | t2 | 3 | t3 |
| (10) | param | t3 | – | – |
| (11) | call | printf | 2 | – |
| (12) | + | i | 1 | t4 |
| (13) | assign | t4 | – | i |
| (14) | goto | (1) | – | – |
| (15) | – | – | – | – |

**Пример 2:** F2, C4(C5) → C1

1. Фрагмент кода:

```
int f(int n) { return n * 3 + 1;}
int main(void) {
  int d = f (4);
  return f(5) + d;
}
```

2. Построение промежуточного кода:

2.1. В виде троек (таблица 5.11).

Таблица 5.11 – Ячейки памяти для констант

| Номер ячейки | Значение |
|---|---|
| D01 | 4 |
| D02 | 5 |
| D03 | 3 |
| D04 | 1 |

Для функции main (таблица 5.12).

Таблица 5.12 – Тройки для функции main

| | op | arg1 | arg2 |
|---|---|---|---|
| (0) | param | D01 | – |
| (1) | call | f | 1 |
| (2) | return | – | – |
| (3) | assign | d | (2) |

Библиотека БГУИР

*Продолжение таблицы 5.12*

| | | | |
|---|---|---|---|
| (4) | param | D02 | |
| (5) | call | f | 1 |
| (6) | return | – | – |
| (7) | + | (6) | d |
| (8) | return | (7) | – |

Для функции f (таблица 5.13).

Таблица 5.13 – Тройки для функции f

| | op | arg1 | arg2 |
|---|---|---|---|
| (9) | * | n | D03 |
| (10) | + | (9) | D04 |
| (11) | return | (10) | – |
| (12) | – | – | – |

2.2. В виде косвенных троек (таблица 5.14).

Таблица 5.14 – Ячейки памяти для констант

| Номер ячейки | Значение |
|---|---|
| D01 | 4 |
| D02 | 5 |
| D03 | 3 |
| D04 | 1 |

Для функции main (таблица 5.15).

Таблица 5.15 – Тройки для функции main

| | op | arg1 | arg2 |
|---|---|---|---|
| (20) | param | D01 | – |
| (21) | call | f | 1 |
| (22) | return | – | – |
| (23) | assign | d | (22) |
| (24) | param | D02 | |
| (25) | call | f | 1 |
| (26) | return | – | – |
| (27) | + | (26) | d |
| (28) | return | (27) | – |

Для функции f (таблица 5.16).



Библиотека БГУИР

Таблица 5.16 – Тройки для функции f

| | op | arg1 | arg2 |
|---|---|---|---|
| (30) | * | n | D03 |
| (31) | + | (30) | D04 |
| (32) | return | (31) | – |
| (33) | – | – | – |

Таблица 5.17 – Ссылки на тройки

| (0) | (20) |
|---|---|
| (1) | (21) |
| (2) | (22) |
| (3) | (23) |
| (4) | (24) |
| (5) | (25) |
| (6) | (26) |
| (7) | (27) |
| (8) | (28) |
| (9) | (30) |
| (10) | (31) |
| (11) | (32) |
| (12) | (33) |

3. Типология команд промежуточного кода.

Для четверок, троек и косвенных троек:

- assign – оператор присваивания. Содержит объект, которому присваивается значение, и значение, которое присваивается;
- param – оператор занесения параметра в стек. Содержит параметр, заносимый в стек;
- call – вызов функции. Содержит имя вызываемой функции и число параметров, извлекаемых из стека;
- < – логическая операция «меньше». Состоит из двух объектов, которые сравниваются этим оператором;
- if – оператор условного перехода. Содержит условие перехода и метку, по которой осуществляется переход. В случае выполнения условия осуществляется переход по метке, в противном случае осуществляется переход к следующей инструкции;
- goto – оператор безусловного перехода. Содержит метку, по которой осуществляется переход;

Библиотека БГУИР

• & – оператор взятия адреса. Содержит объект, адрес которого берется;

• * – оператор умножения. Содержит умножаемые числа;

• [] – оператор обращения к элементу массива по индексу. Содержит массив, к которому осуществляется обращение, и индекс запрашиваемого элемента;

• + – оператор сложения. Содержит складываемые числа.

4. Оптимизация промежуточного кода.

Для наглядности оптимизацию проведем над исходным кодом, затем покажем, как она отразилась на каждом виде промежуточного кода.

Проведем распространение копий:

```
int f(int n) { return n * 3 + 1;}
int main(void) {
  int d = f (4);
  return f(5) + d;
}
```

→

```
int f(int n) { return n * 3 + 1;}
int main(void) {
  int d = f (4);
  return f(5) + f(4);
}
```

```
int f(int n) { return n * 3 + 1;}
int main(void) {
  int d = f (4);
  return f(5) + f(4) ;
}
```

→

```
int f(int n) { return n * 3 + 1;}
int main(void) {
  return f(5) + f(4);
}
```

```
int f(int n) { return n * 3 + 1;}
int main(void) {
  return f(5) + f(4);
}
```

→

```
int main(void) {
  return f(5 * 3 + 1) + f(4 * 3 +
1);
}
```

5. Запишем полученный оптимизированный промежуточный код в виде синтаксического дерева (рисунок 5.7).



Библиотека БГУИР

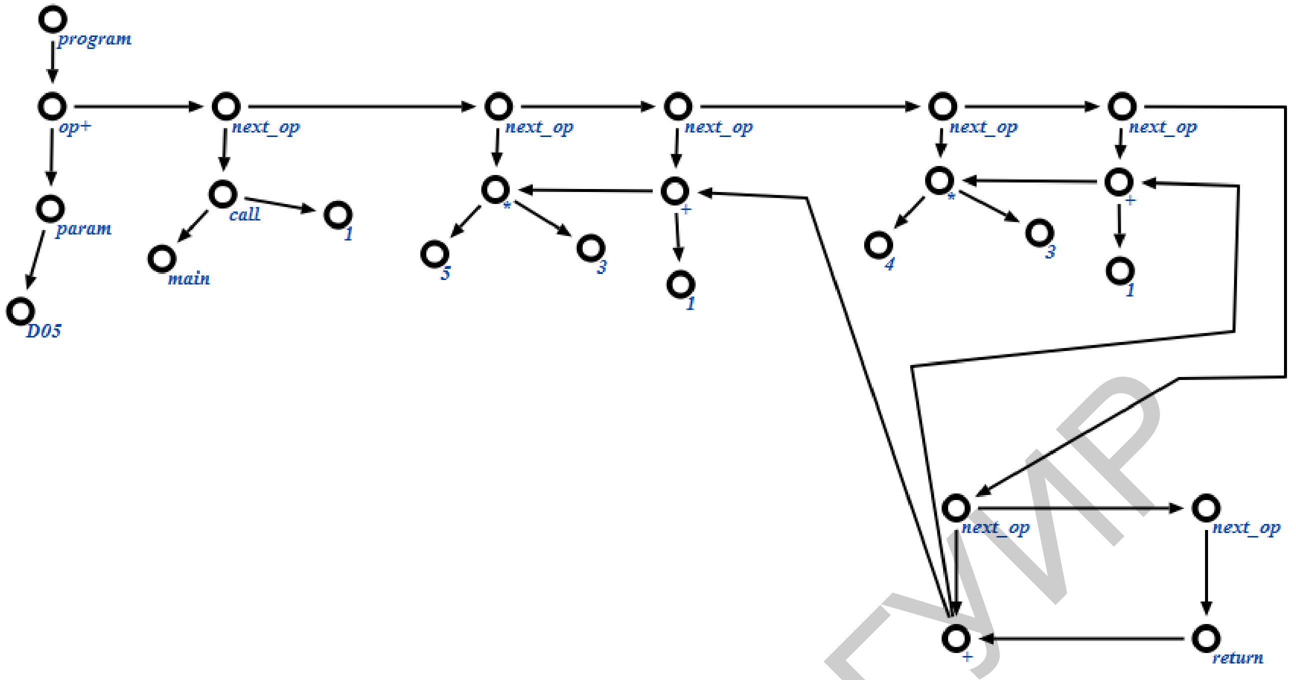



Рисунок 5.7 – Синтаксическое дерево промежуточного кода

### 5.2 Варианты индивидуальных заданий

Таблица 5.18 – Варианты индивидуальных заданий

| № | Формулировка варианта задания | № | Формулировка варианта задания |
|---|---|---|---|
| 1 | F1,C1 → C3 | 16 | F4,C1 → C3 |
| 2 | F1,C2 → C3 | 17 | F4,C2 → C3 |
| 3 | F1,C3 → C1 | 18 | F4,C3 → C1 |
| 4 | F1,C4 → C1 | 19 | F4,C4 → C1 |
| 5 | F1,C5 → C1 | 20 | F4,C5 → C1 |
| 6 | F2,C1 → C3 | 21 | F5,C1 → C3 |
| 7 | F2,C2 → C3 | 22 | F5,C2 → C3 |
| 8 | F2,C3 → C1 | 23 | F5,C3 → C1 |
| 9 | F2,C4 → C1 | 24 | F5,C4 → C1 |
| 10 | F2,C5 → C1 | 25 | F5,C5 → C1 |
| 11 | F3,C1 → C3 | 26 | F6,C1 → C3 |
| 12 | F3,C2 → C3 | 27 | F6,C2 → C3 |
| 13 | F3,C3 → C1 | 28 | F6,C3 → C1 |
| 14 | F3,C4 → C1 | 29 | F6,C4 → C1 |
| 15 | F3,C5 → C1 | 30 | F6,C5 → C1 |

Библиотека БГУИР

Таблица 5.19 – F* в таблице 5.18

| | |
|---|---|
| F1 | int a, b, c = 1, d = c * 3;<br>char * s = “строка символов”;<br>if (s[c + d] == 0) {<br>  a = c + 1;<br>  b = c + 2;<br>} else {<br>  a = d + 1;<br>  b = d + 2;<br>}<br>printf(“Ответ: %d”, a + b); |
| F2 | int f(int n) { return n * 3 + 1;}<br>int main(void) {<br>  int d = f (4);<br>  return f(5) + d;<br>} |
| F3 | for (int i = 0; i < 12; i++) {<br>  scanf(“%d”, f);<br>  int d = i * f * 3;<br>  printf(“%d”, d);<br>} |
| F4 | int a, b;<br>scanf(“%d”, a);<br>switch(a) {<br>  case 1: b = a + 1;<br>  case 2: b = a + 2; break;<br>  default: b = 0;<br>}<br>exit(b); |
| F5 | struct {int a; char b} s, f;<br>s.a = 1;<br>s.b = ‘3’;<br>strcpy(&f, &s, sizeof(s));<br>printf(“%d – %d”, f.a, f.b); |
| F6 | char[] str = “string”;<br>int a, b, c;<br>a = 3;<br>if (str[a] == ‘\n’) {<br>  b = str[a–1]; c = 0;<br>} else {<br>  b = str[a+1]; c = 1;<br>}<br>str[b] = 0;<br>printf (“%s”, str); |

Библиотека БГУИР

Таблица 5.20 – С* в таблице 5.18

| | |
|---|---|
| С1 | Синтаксическое дерево |
| С2 | Даг |
| С3 | Трехадресный код. Четверки |
| С4 | Трехадресный код. Тройки |
| С5 | Трехадресный код. Косвенные тройки |

Библиотека БГУИР

# ЛАБОРАТОРНАЯ РАБОТА №6<br>РАЗРАБОТКА ФОРМАЛЬНОЙ СПЕЦИФИКАЦИИ<br>ЯЗЫКА ПРОГРАММИРОВАНИЯ

*Цель работы*: получить навык в разработке формальной спецификации языка программирования.

## 6.1 Теоретические сведения

**Требования к разрабатываемому языку**

1. Встроенные типы.

2. Возможность инициализация переменных всех типов при объявлении: <тип> <имя_переменной> = <выражение>:

- инициализирующее выражение может быть константным.

3. Встроенные операции.

4. Встроенные функции:

- встроенные функции ввода/вывода для работы со встроенными типами.

5. Использование сложных выражений (составных и со скобками).

6. Блочный оператор.

7. Управляющие структуры:

- условный оператор (if-then-else);
- операторы цикла (while и until);
- оператор цикла с итерациями (for).

8. Пользовательские подпрограммы:

- передача и возврат параметров;
- задание локальной и глобальной области видимости для имен переменных.

***Варианты свойств языка***

1. Объявление переменных:

- явное;
- неявное.

2. Преобразование типов:

- явное, например, a = (int) b;
- неявное.

3. Оператор присваивания:

- одноцелевой, например, a = b;
- многоцелевой, например, a, b = c, d.

4. Структуры, ограничивающие область видимости:

- подпрограммы;



Библиотека БГУИР

- подпрограммы и блочные операторы.

5. Маркер блочного оператора:
- явный, например, { } или begin end;
- неявный, например как в python.

6. Условные операторы:
- двухвариантный оператор if-then-else;
- двухвариантный оператор и многовариантный switch-case.

7. Перегрузка подпрограмм:
- отсутствует;
- присутствует.

8. Передача параметров в подпрограмму:
- только по значению и возвращаемому значению;
- по значению и результату;
- по ссылке.

9. Допустимое место объявления подпрограмм:
- в начале программы;
- в любом месте программы, также внутри другой подпрограммы.

***Варианты языков***

1. Язык, описывающий математические вычисления:
- встроенные типы: int, float;
- операции: +, –, *, \, %, ^, ==, !=, <, >, <=, >=.

2. Язык для работы с векторами и матрицами:
- встроенные типы: vector, matrix;
- операции: v + v, v – v, n * v, v * v, |v|, m + m, m – m, m * n, m * m, m[n], m[n] * n, |m|, v * m.

3. Язык для работы с графовыми структурами:
- встроенные типы: node, arc, graph;
- операции: переопределить +, –, *, \ и т. д. для встроенных типов.

4. Язык для работы с множествами:
- встроенные типы: element, set;
- операции: переопределить +, –, *, \ и т. д. для встроенных типов.

5. Язык для работы со строками:
- встроенные типы: char, string, массив string;
- операции: переопределить +, –, *, \ и т. д. для встроенных типов.

6. Язык для работы со списковыми структурами:
- встроенные типы: element, list;

Библиотека БГУИР

- операции: переопределить +, –, *, \ и т. д. для встроенных типов.

7. Язык для работы с xml данными:

- встроенные типы: document, node, attribute.

8. Язык для работы с реляционными данными:

- встроенные типы: table, row, column.

**Варианты целевого кода**

1. Исполняемый файл (.exe), формат промежуточного кода – язык C++, генерация целевого кода стандартным компилятором.

2. Байт-код JVM, формат промежуточного кода – язык Java, генерация целевого кода стандартным компилятором (javac).

3. Байт-код JVM, формат промежуточного кода – ассемблер для JVM.

4. Байт-код .NET, формат промежуточного кода – ассемблер для .NET (CIL).

5. Исполняемый файл (.exe), для генерации целевого кода использовать LLVM (http://llvm.org).

6. Исполняемый файл (.exe), для генерации целевого код использовать GCC Front end (входит в состав GNU GCC).

Таблица 6.1 – Порядок выполнения и промежуточная отчетность

| № занятия | Задания |
|---|---|
| **1** | **2** |
| 1 | *Задание на занятии:*<br>• выбрать вариант разрабатываемого языка;<br>• записать пример какого-нибудь алгоритма на придуманном языке.<br>*Домашнее задание:*<br>• разработать минимум три примера логически цельных алгоритма на придуманном языке;<br>• начать описывать синтаксис языка, используя формат ANTLR |
| 2 | *Исходные материалы:*<br>• три текстовых файла с примерами на придуманном языке;<br>• первая версия синтаксиса языка, описанная в формате ANTLR.<br>*Задание на занятии:*<br>• дополнить синтаксис языка;<br>• сгенерировать код синтаксического анализатора и добавить «main».<br>*Домашнее задание:*<br>• дописать синтаксический анализатор и протестировать его на имеющихся примерах |



Библиотека БГУИР

*Продолжение таблицы 6.1*

| 1 | 2 |
|---|---|
| 3 | *Исходные материалы:*<br>• примеры, файл грамматики, программный код синтаксического анализатора.<br>*Задание на занятии:*<br>• определить набор семантических правил для языка;<br>• добавить в грамматику действия по выполнению семантических правил;<br>• выбрать вариант целевого кода.<br>*Домашнее задание:*<br>• дописать семантический анализатор и проверить на примерах выдачу синтаксических и семантических ошибок;<br>• изучить формат выбранного целевого кода |
| 4 | *Исходные материалы:*<br>• примеры, файл грамматики, программный код синтаксического и семантического анализатора.<br>*Задание на занятии:*<br>• добавить в грамматику действия по формированию целевого кода.<br>*Домашнее задание:*<br>• доделать программный код компилятора и проверить его на примерах |
| 5 | *Исходные материалы:*<br>• версия компилятора с некоторыми небольшими недочетами.<br>*Задание на занятии:*<br>• устранить недоделки в компиляторе;<br>• протестировать компилятор;<br>• сформировать отчет по лабораторной работе.<br>*Домашнее задание:*<br>• сформировать отчет по лабораторной работе;<br>• подготовиться к защите лабораторной работы |
| 6 | *Исходные материалы:*<br>• программный код компилятора, собранный компилятор;<br>• примеры, отчет.<br>*Задание на занятии:*<br>• защитить лабораторную работу;<br>• выбрать вариант задания для второй лабораторной работы.<br>*Домашнее задание:*<br>• подготовиться ко второй лабораторной работе |

Библиотека БГУИР

**Форма итоговой отчетности**

Защита лабораторного практикума состоит из следующих частей.

1. Демонстрация практических результатов и проверка корректности работы и соответствия документации.

2. Проверка документации и результатов аналитического задания.

3. Ответ на контрольные вопросы.

Содержание отчета по первой части лабораторного практикума следующее:

1. Спецификация разработанного языка программирования:

– синтаксис объявления переменных и подпрограмм;

– синтаксис операций над данными (их должно быть не менее 10 штук);

– синтаксис всех управляющих конструкций.

2. Оформленный (отформатированный и прокомментированный) файл грамматики (.g).

3. Описание дополнительно разработанных классов.

4. Перечень генерируемых ошибок.

5. Примеры работы компилятора.

Требования к оформлению отчета аналогичны требованиям оформления научно-технической документации. Допускается оформления фрагментов исходных текстов и примеров шрифтом меньшего размера.

## 6.2 Варианты индивидуальных заданий

Таблица 6.2 – Варианты индивидуальных заданий

| Вариант | Язык | Свойства | | | | | | | | | Целевой код |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
| 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 2 |
| 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 3 |
| 4 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| 6 | 4 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 3 |
| 7 | 4 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 |
| 8 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 |
| 9 | 5 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3 |
| 10 | 6 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| 11 | 6 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 2 |
| 12 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 3 |
| 13 | 1 | 2 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| 14 | 1 | 2 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 2 |

Библиотека БГУИР

*Продолжение таблицы 6.2*

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 3 |
| 16 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 17 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| 18 | 4 | 2 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 3 |
| 19 | 4 | 2 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 |
| 20 | 4 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 |
| 21 | 5 | 2 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 3 |
| 22 | 6 | 2 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| 23 | 6 | 2 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 2 |
| 24 | 6 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 3 |
| 25 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 26 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| 27 | 4 | 2 | 2 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 3 |
| 28 | 7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 |
| 29 | 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| 30 | 8 | 2 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 3 |

Библиотека БГУИР

# ЛАБОРАТОРНАЯ РАБОТА №7
# РАЗРАБОТКА КОМПИЛЯТОРА ЯЗЫКА ПРОГРАММИРОВАНИЯ

*Цель работы*: получить навык в разработке компилятора языка программирования.

## 7.1 Теоретические сведения

**Постановка задачи**

Придумать несколько исходных текстов разрабатываемого языка. На основании предыдущего написать грамматику и код шаблона трансляции и, используя ANTLR, сгенерировать код парсера и лексера на целевом языке (Java). Разработать дополнительные классы для обработки входных данных и конструкций. Экспортировать проект в запускной файл (в нашем случае jar).

**Описание структуры файла грамматики**

В ANTLR-грамматике есть несколько ключевых блоков и понятий.

Блок параметров:

```
options{
     language = Java;           //Язык выходных файлов парсера и лексера
     output = template;      //Указываем на использование шаблонов
}
```

Блок токенов:

```
tokens{     //В этом блоке содержатся константные терминалы
     FUN_PRINT    =      'print';
     FUN_PERIM    =      'perimeter';
     FUN_LENGTH   =      'length';
     FUN_SQUARE   =      'square';
     PROG         =      'Program'    ;
}
```

Блок заголовков парсера:

```
@header {
  package grammar;              //Имя пакета для парсера
  import types.*;                        //
  import types.NamesTable.Name; //import других пакетов и классов
  import org.antlr.stringtemplate.*;  //
}
```

Блок заголовков лексера:

```
@lexer::header {
   package grammar;
```



Библиотека БГУИР

```
}
```

Блок глобальных переменных и методов:

```
@members {
   public static ArrayList<String> errors = new ArrayList<String>();
   public static NamesTable names = new NamesTable();
}
```

Пример терминала:

```
'denis'
```

Пример нетерминала:

```
ID :
     ('a'..'z'|'A'..'Z')+
;
```

Пример правила:

```
rule:
     ID '=' ID;
;
```

Каждый нетерминал транслируется в метод на целевом языке, который может возвращать какие-либо параметры. Для этого нужно в квадратных скобках указать тип и имя переменной, которую нужно вернуть/принять.

Общее правило передачи параметров следующее:

```
rule[int a, String b] returns[int c, String d]
```

То есть после имени нетерминала в квадратных скобках указываются принимаемые аргументы, а после returns – возвращаемые параметры. Вместо int и String могут использоваться любые другие типы. Количество параметров также может быть различным.

В описании нетерминала можно делать вставки Java кода (если вы пишете на Java):

```
rule returns[String hokkey]:
l_id=ID '=' r_id=ID ';'{
//В l_id,r_id занесены значения  соответствующих ID
             if($l_id.text.equals($r_id.text)){
       $hokkey = "Ice hokkey world championship";
}};
```

Библиотека БГУИР

Для того чтобы взять значение параметра hokkey в нетерминале rule, нужно получить конструкцию вида

```
{ String s = $rule.hokkey; }
```

**Описание структуры файла шаблонов**

Для преобразования кода на разрабатываемом языке в код на языке Java (в данном случае) используются шаблоны. Для начала работы с шаблонами необходимо создать файл с расширением .stg. и в первой строке записать следующее:

```
group название_файла_шаблонов;
```

Общая структура шаблона приведена ниже:

```
название_шаблона (список_аргументов) ::=
<<
Тело шаблона
>>
```

Теперь перейдем к практике. Рассмотрим небольшой пример:

```
program(nameProgram)  ::=
<<
public class <nameProgram.text>{}
>>
```

Шаблон с названием program получает на вход аргумент nameProgram. Метод .text возвращает текст переданного аргумента. Допустим, nameProgram.text возвращает текст MyProgram. Тогда результат действия шаблона следующий:

```
public class MyProgram{}
```

Для того чтобы взять значение переданного аргумента, нужно взять название аргумента в треугольные скобки:

```
<название_аргумента>
```

Теперь приведем пример, который демонстрирует работу условного оператора:

```
operation(a) ::=
<<
<if(a)>id1+id2 <else>id1-id2<endif>
>>
```

Если a=true, то результатом действия шаблона operation будет выражение id1+id2, иначе выполнится код, идущий за <else>.



Библиотека БГУИР

Рассмотрим пример, в котором операнд хранит в себе несколько элементов.

1. Исходный файл с кодом на разрабатываемом языке:

```
double a,b,c;
```

2. В файле с расширением .g дан следующий код:

```
initialization      :
       t=DOUBLE      var+=ID (',' var+=ID)* ';'

        } →   init(type={$t},var={$var})
```

3. Файл с расширением .stg:

```
init(type, var) ::=
<<
<var:{v|<type.text> <v.text>;};separator="\n ">
>>
```

В этом примере <var> хранит в себе несколько элементов. Благодаря оператору separator, который разделяет элементы с помощью символа перехода на новую строку, мы получили следующий код:

```
Double a;
Double b;
Double c;
```

Обратите внимание на строку в файле с грамматикой:

```
→      init(type={$t},var={$var})
```

После символа → пишется название вызываемого шаблона с данными параметрами. Эту конструкцию можно заменить следующей:

```
{$st = %init(type={$t}, var={$var});}
```

## 7.2 Практическая часть

**Настройка среды разработки**

Для выполнения лабораторной используется Eclipse 3.7 Indigo (можно использовать 3.3–3.7) и ANTLRv3. Инструкция по установке плагина ANTLRv3 под Eclipse представлена на электронном ресурсе http://antlrv3ide.sourceforge.net. Необходимо перейти по ссылке, зайти в Download/Install, выбрать вкладку с вашей версией Eclipse и следовать инструкции. Видеоинструкция представлена по ссылке https://vimeo.com/8001326/.

Библиотека БГУИР

### Примеры исходных текстов программы

Изначально следует обдумать структуру, константы, базовые конструкции разрабатываемого языка и придумать несколько примеров исходных текстов программы.

Поскольку мы разрабатываем язык для планиметрии, то к основным типам объектов можно отнести точку, линию, треугольник. Операциями над объектами являются длина (относится к линии), площадь и периметр (к треугольнику). Линию и треугольник можно задавать как наборы точек.

Пример 1

```
Program program_v_1{
        Point p1=(5,4), p2=(3,2), p3=(4,6);
        Triangle t1=[p1,p2,p3];
        if( 30>square(t1)){
        print(t1);
        }
}
```

Пример 2

```
Program PR1 {
        Decimal i,j;
        Point p1=(5,4), p2=(3,2), p3=(4,6);
        Triangle t1=[p1,p2,p3];
        i=5;
        j=10;
        if(i>j){
        print(square(t1));
        }
}
```

### Создание грамматики разрабатываемого языка

Определим возможные ключевые слова языка и занесем их в блок токенов:

```
tokens{
        FUN_PRINT       =       'print';
        FUN_PERIM       =       'perimeter';
        FUN_LENGTH      =       'length';
        FUN_SQUARE      =       'square';
        PROG            =       'Program';
        L_FBR           =       '{';
        R_FBR           =       '}';
        L_CBR           =       '[';
        R_CBR           =       ']';
```

Библиотека БГУИР

```
            L_BR                    =       '(';
            R_BR                    =       ')';
            COMMA                   =       ',';
            DOT_COMMA               =       ';';
            ASSIGN                  =       '=';
            AND                     =       'and';
            NOT                     =       'not';
            OR                      =       'or';
            IF                      =       'if';
            TYPE_POINT              =       'Point' ;
            TYPE_LINE               =       'Line' ;
            TYPE_TRIANGLE           =       'Triangle' ;
            TYPE_DEC                =       'Decimal' ;
            PLUS                    =       '+';
            MINUS                   =       '-';
            MULT                    =       '*';
            NULL                    =       'null';
}
```

Опишем возможные нетерминалы языка. Некоторые из них можно сгенерировать автоматически, а другие придется описывать вручную:

```
LOG_SIGN :   '<'|'>'|'==';
ID  :  ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
    ;
INT :  '0'..'9'+
    ;
FLOAT
    :   ('0'..'9')+ '.' ('0'..'9')* EXPONENT?
    |   '.' ('0'..'9')+ EXPONENT?
    |   ('0'..'9')+ EXPONENT
    ;
COMMENT
    :   '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
    |   '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
    ;
WS  :   ( ' '
        | '\t'
        | '\r'
        | '\n'
        ) {$channel=HIDDEN;}
    ;
STRING
    :  '"' ( ESC_SEQ | ~('\\'|'"') )* '"'
    ;
CHAR:  '\'' ( ESC_SEQ | ~('\''|'\\') ) '\''
    ;
fragment
EXPONENT : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;
fragment
HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;
fragment
ESC_SEQ
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
    |   UNICODE_ESC
    |   OCTAL_ESC
    ;
```

Библиотека БГУИР

```
fragment
OCTAL_ESC
    :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7')
    ;
fragment
UNICODE_ESC
    :   '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
;
```

Атрибут **fragment** указывает, что следующий нетерминал будет использоваться в составе другого нетерминала.

Содержимое файла грамматики на первом этапе работы приведено в приложении А.

Далее добавим заголовочные блоки, которые укажут, где будут размещаться классы лексера и парсера. Также определим глобальные переменные и методы:

```
@header {
package grammar;
  import types.*;
  import types.Name;
  import org.antlr.stringtemplate.*;
}


@lexer::header {
package grammar;
}
@members {
public static NamesTable names = new NamesTable();
public boolean isDefined(String id){
       if(names.get(id)!=null){
                     return true;
       }
       return false;
       }
}
```

Приступим к описанию нетерминалов. Стартовым нетерминалом будет program:

```
program :    PROG ID L_FBR statement+ R_FBR;
```

Это правило соответствует следующей записи:

```
Program PRG1 {…}
```

Программа будет состоять из определения переменных, инициализации, функции печати и условного оператора:

```
statement    :      ((var_define | assign | fun_print) DOT_COMMA ) | if_stat;
```



Библиотека БГУИР

Опишем каждую операцию по отдельности.
Фрагмент грамматики:

```
if_stat      :      IF L_BR log_expr R_BR L_FBR if_statement+ R_FBR DOT_COMMA?;
if_statement : (assign DOT_COMMA | fun_print DOT_COMMA | if_stat);
```

Ему соответствует:

```
if(i>j){
       …
}
```

Фрагмент грамматики:

```
var_define   :      type ID (ASSIGN var_value) (COMMA ID (ASSIGN var_value))*;
var_value    :
        ID
       |point
       |triangel
       |line
       |dec
       |NULL
;
```

Ему соответствует:

```
Decimal i=null,j=2;
Point p1=(5,4), p2=(3,2), p3=(4,6);
Triangle t1=[p1,p2,p3];
```

Фрагмент грамматики:

```
assign       :      ID ASSIGN expr;
```

Ему соответствует:

```
i=5;
j=10;
```

Получившаяся  грамматика представлена в приложении А.

**Добавление функциональности**

Добавим в нашу грамматику немного функциональности. Начнем с объявления переменных:

```
var_define  :
t=type ids+=ID (ASSIGN values+=var_value)
                           (COMMA ids+=ID (ASSIGN values+=var_value))*
{for(Token id : (List<Token>)$ids){
      Name n = new Name(id.getText(),$t.type,id.getLine());
      names.add(n);
    }}
;
```

Библиотека БГУИР

В этом блоке кода происходит объявление переменных. Пользователь может ввести что-то подобное:

```
Decimal i=null, j=20, k=11, p=null;
```

Поэтому нужно ввести переменную ids, которая будет списком всех идентификаторов (ID). Формально в Java переменная ids будет представлена как

```
List<Object> ids;
```

ID будет представлена таким образом:

```
Token ID;
```

Класс Token имеет несколько базовых методов:

- getText() – возвращает текстовое содержимое нетерминала;
- getLine() – возвращает номер строки, на которой находится данный нетерминал в исходном тексте программы.

Поэтому, чтобы взять значение конкретного элемента списка ids, нужно для начала привести его к списку из токенов:

```
(List<Token>)$ids
```

Затем следует добавить в список переменных программы names:

```
for(Token id : (List<Token>)$ids){
Name n = new Name(id.getText(),$t.type,id.getLine());
names.add(n);
}
```

Объект класса Name будет хранить имя переменной, тип и линию в исходном тексте нашей программы.

Для того чтобы проанализировать код на наличие необъявленных переменных, в некоторых местах грамматики вставим следующий код:

```
var_value   :
 ID         {
 if(!isDefined($ID.text)){
       throw new NotDefinedVariable($ID.text, $ID.line);
 }
….
```

Метод isDefined(String variableId) проверяет, есть ли в списке переменных программы переменная с идентификатором variableId. Если есть – true, в противном случае – false. Поэтому если пользователь вдруг напишет

```
Line line = {(2,3),p2};,
```



Библиотека БГУИР

а переменная p2 нигде выше не была объявлена, то будет брошено исключение NotDefineVariable(String variableId, int lineInSource)

**Создание шаблонов**

Рассмотрим создание файла шаблонов на некоторых правилах грамматики. Добавим в файл Template.stg шаблонный метод для работы с условным оператором:

```
if_template(cond, stat)::=<<
if(<cond>){
  <stat;separator="\n">
}
>>
```

В метод if_template поступают следующие параметры cond и stat, где cond условие, stat хранит в себе тело условного оператора. Вызываем шаблонный метод следующим образом:

```
  if_stat    :
IF L_BR log_expr R_BR L_FBR (stat+=if_statement)+ R_FBR DOT_COMMA?
-→ if_template(cond={$log_expr.st}, stat={$stat})
  ;
```

Для инициализации создадим метод:

```
var_define_template(type, list_id, list_value)::=<<
Element <list_id,list_value:{v, t|<v.text> = <t>};separator=","> ;
>>
```

В грамматике вышеуказанный метод будет вызван такими аргументами, как тип, список переменных и список их значений:

```
var_define  :
t=type ids+=ID (ASSIGN values+=var_value)
                        (COMMA ids+=ID (ASSIGN values+=var_value))*
{for(Token id : (List<Token>)$ids){
      Name n = new Name(id.getText(),$t.type,id.getLine());
      names.add(n);
}}                                                →
var_define_template(type={$type.type},list_id={$ids},list_value={$values})

;
```

Для оператора инициализации вызовем шаблонный метод assign_template с параметрами: имя переменной и выражение, которое ей присваивается.

```
assign      :
ID ASSIGN expr → assign_template(id = {$ID}, expr = {$expr.st})
;
```



Библиотека БГУИР

Соответствующий метод в .stg файле:

```
assign_template(id, expr)::=<<
<id.text> = <expr>;
>>
```

Содержимое файла грамматики на заключительном этапе работы приведено в приложении Б.

**Разработка вспомогательных классов**

После того как грамматика и шаблоны были успешно созданы, мы имеем программу, транслирующую исходный код созданного нами языка в Java–код, который впоследствии мы должны запустить. Вот пример того, что может получиться на выходе:

```
package main;
import operations.Operation;
import types.Decimal;
import types.Element;
import types.Point;
import types.Line;
import types.Triangel;
import utls.CompareType;
import exceptions.*;

public class Main{
  public static void begin() throws NotComparableTypes, NotSupportedValues{
    Element t1 = new Triangel(new Point(7, 2),new Point(4, 1),new Point(9, 9)) ;
    Operation.print(Operation.perimeter(t1));
  }
  public static void main(String[]args) throws Exception{
     try{
        begin();
     } catch(NotComparableTypes e){
        System.out.println(e.info());
     } catch(NotSupportedValues e){
        System.out.println(e.info());
     } catch(Exception e){
        System.out.println("Unknown error");
     }
  }
}
```

Поскольку у нас нету почти ни одного из тех классов, которые задействованы в этом исходном тексте, то следует приступить к их реализации.

Для обработки данных и вычисления результатов операций были разработаны классы и пакеты, представленные на рисунке 7.1.



Библиотека БГУИР

Рисунок 7.1 – Классы и пакеты проекта

В пакете:

- grammar – находятся *.java файлы парсера и лексера;
- exceptions – классы исключений;
- operations – класс реализованных операций;
- types – классы базовых типов и конструкций;
- utls – перечисление операторов сравнения;
- main – запускной класс.

Для того чтобы вывести транслированный исходный текст в файл, нужно создать запускной файл следующего содержания:

```
package main;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectOutputStream;
import java.io.PrintWriter;

import org.antlr.runtime.ANTLRInputStream;
import org.antlr.runtime.CommonTokenStream;
import org.antlr.runtime.RecognitionException;
import org.antlr.stringtemplate.StringTemplate;
import org.antlr.stringtemplate.StringTemplateGroup;

import exceptions.NotDefinedVariable;
```

Библиотека БГУИР

```
import grammar.PlanimetryLexer;
import grammar.PlanimetryParser;

public class EntryPoint {

	public static void main(String[] args) throws IOException,
RecognitionException{
		if(args.length==0) return;
		FileReader gf = new FileReader("src\\Template.stg");
		StringTemplateGroup stg = new StringTemplateGroup(gf);
		gf.close();
		File file = new File(args[0]);
		InputStream fis = new FileInputStream(file);
		try{
			ANTLRInputStream input = new ANTLRInputStream(fis);
			PlanimetryLexer lexer = new PlanimetryLexer(input);
			CommonTokenStream tokens = new CommonTokenStream(lexer);
			PlanimetryParser parser = new PlanimetryParser(tokens);
			parser.setTemplateLib(stg);
			PlanimetryParser.program_return r = parser.program();
			StringTemplate output = (StringTemplate) r.getTemplate();
			System.out.println(output.toString());
			PrintWriter printer =
			 new PrintWriter("src/main/Main.java", "cp1251");
			printer.print(output.toString());
			printer.close();
		} catch(NotDefinedVariable e){
			System.out.println(e.info());
		} catch(Exception e){
			System.out.println(e.toString());
		}

	}
}
```

Метод main принимает в качестве аргументов путь к файлу исходных текстов, после чего создается объект класса группы шаблонов:

```
StringTemplateGroup stg.
```

Далее, после инициализации парсера и лексера запускается стартовое правило program. В переменную output заносится транслированный исходный текст, который мы записываем в файл как `src/main/Main.java`.

**Запуск и тестирование**

Для того чтобы увидеть результаты работы, необходимо экспортировать проект в jar-файл. Для этого заходим в меню File и нажимаем Export. В результате появится окно, которое отображено на рисунке 7.2.

Библиотека БГУИР

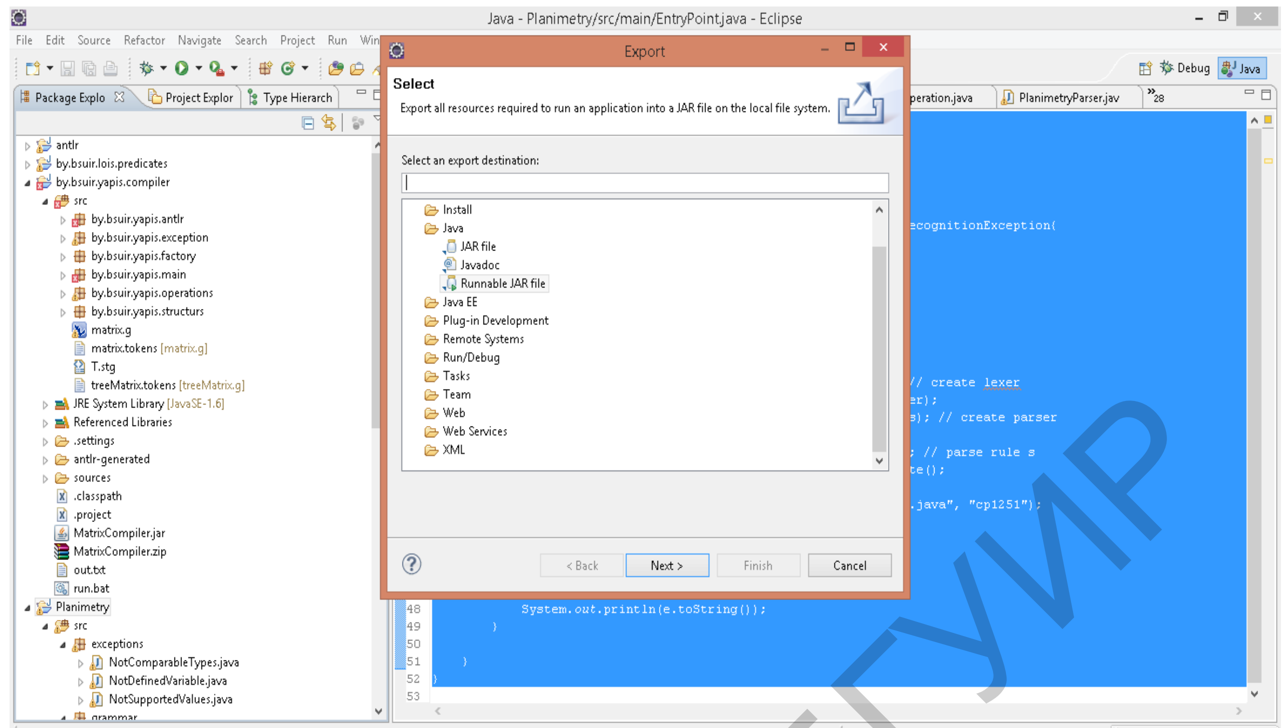



Рисунок 7.2 – Выбор вида экспортируемого файла

Далее выбираем пункт Runnable JAR file и нажимаем кнопку Next. После проделанных действий должно появиться окно, в котором необходимо выставить все параметры, аналогичные параметрам на рисунке 7.3.

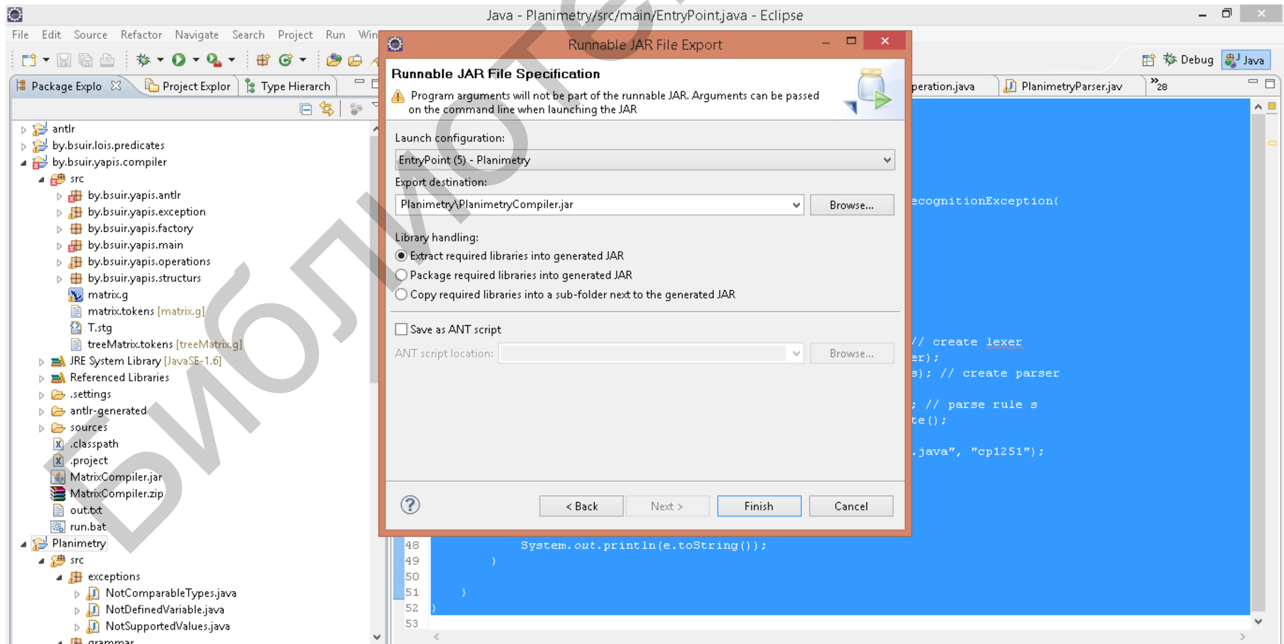



Рисунок 7.3 – Экспорт файла

Затем создадим файл с расширением .bat и запишем в него следующий код:

Библиотека БГУИР

```
@echo off
java -jar PlanimetryCompiler.jar %~f1
if exist .\src\main\Main.java (
javac -d bin -sourcepath src -cp c:\antlr-3.3\lib\antlr-3.2.jar src\main\Main.java
java -cp bin;c:\antlr-3.3\lib\antlr-3.2.jar main.Main
) else ( echo System message : source java file doesn't exist. Exiting compilation)
pause > NUL
```

Теперь запустим этот файл. Откроем консоль в папке с проектом и напишем следующую строку:

```
run.bat source1.txt
```

В .bat файле параметр source1.txt будет подставлен вместо `%~f1`.

```
java -jar PlanimetryCompiler.jar %~f1
```

Эта строка соответствует запуску `PlanimetryCompiler.jar` с параметром `%~f1`. В результате будет сгенерирован файл `.\src\main\Main.java`.

```
if exist .\src\main\Main.java
```

Далее проверяем, существует ли сгенерированный файл `.\src\main\Main.java`.

Если существует, то компилируем java-файл:

```
javac -d bin -sourcepath src -cp c:\antlr-3.3\lib\antlr-3.2.jar src\main\Main.java
```

Здесь `c:\antlr-3.3\lib\antlr-3.2.jar` – это путь к библиотеке antlr, а `src\main\Main.java` – это путь к сгенерированному java-файлу. После этой операции будет сгенерирован class-файл, который мы запустим следующей командой:

```
java -cp bin;c:\antlr-3.3\lib\antlr-3.2.jar main.Main
```

В этом случае `main.Main` – это наш class-файл, но расширение здесь не указывается. После выполнения операции будет выведен результат, представленный на рисунке 7.4.

Библиотека БГУИР

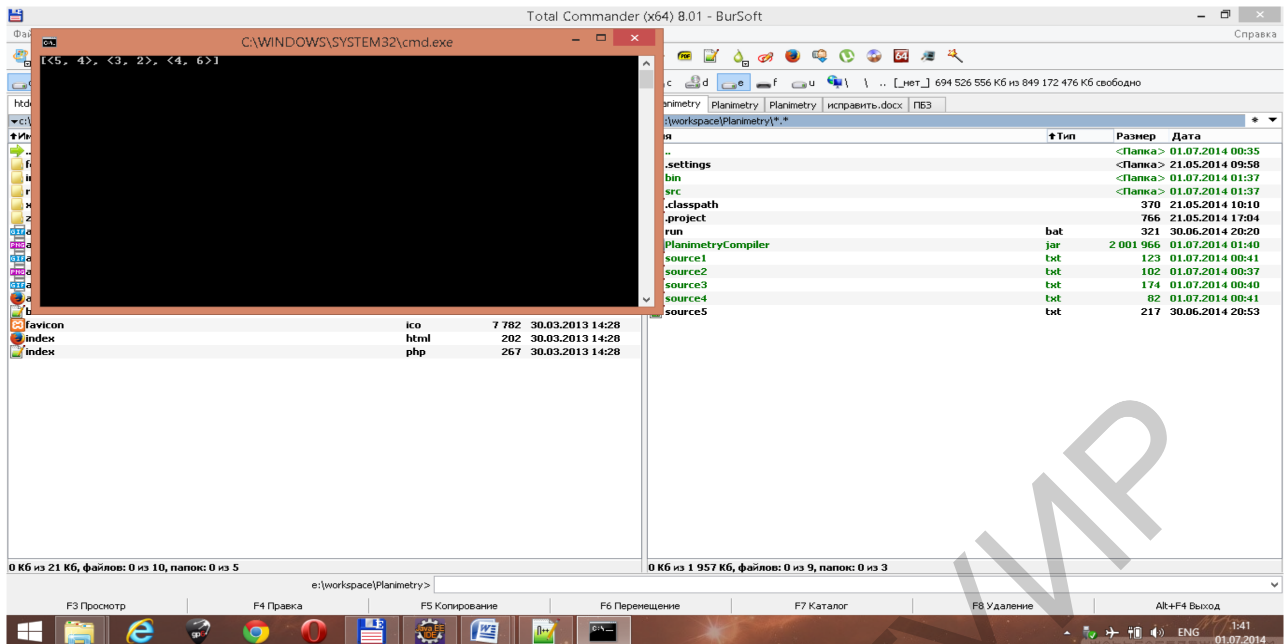



Рисунок 7.4 – Результат

Библиотека БГУИР

## ПРИЛОЖЕНИЕ А
(*справочное*)

## Содержимое файла грамматики на первом этапе

На первом этапе файл грамматики Planimetry.g выглядит так:

```
grammar Planimetry;
options {
  language = Java;
  output = template;
}
tokens{
FUN_PRINT          =      'print';
FUN_PERIM          =      'perimeter';
FUN_LENGTH         =      'length';
FUN_SQUARE         =      'square';
PROG               =   'Program';
L_FBR              =      '{';
R_FBR              =      '}';
L_CBR              =      '[';
R_CBR              =      ']';
L_BR               =      '(';
R_BR               =      ')';
COMMA              =      ',';
DOT_COMMA          =      ';';
ASSIGN             =      '=';
AND                        =      'and';
NOT                        =      'not';
OR                         =      'or';
IF                         =      'if';
TYPE_POINT   =  'Point' ;
TYPE_LINE          =  'Line' ;
TYPE_SEG           =  'Segment' ;
TYPE_TRIANGLE      =  'Triangle' ;
TYPE_CIRCLE  =  'Circle' ;
TYPE_DEC           =  'Decimal' ;
PLUS               =  '+';
MINUS              =  '-';
MULT               =  '*';
DIV                        =  '/';
NULL               =  'null';
}
@header {
package grammar;
  import types.*;
  import types.Name;
  import org.antlr.stringtemplate.*;
}

@lexer::header {
package grammar;
}

LOG_SIGN :   '<'|'>'|'==';
ID  :  ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
    ;
```

Библиотека БГУИР

```
INT :  '0'..'9'+
    ;
FLOAT
    :   ('0'..'9')+ '.' ('0'..'9')* EXPONENT?
    |   '.' ('0'..'9')+ EXPONENT?
    |   ('0'..'9')+ EXPONENT
    ;

COMMENT
    :   '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
    |   '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
    ;
WS  :   ( ' '
        | '\t'
        | '\r'
        | '\n'
        ) {$channel=HIDDEN;}
    ;
STRING
    :  '"' ( ESC_SEQ | ~('\\'|'"') )* '"'
    ;
CHAR:  '\'' ( ESC_SEQ | ~('\''|'\\') ) '\''
    ;
fragment
EXPONENT : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;

fragment
HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment
ESC_SEQ
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
    |   UNICODE_ESC
    |   OCTAL_ESC
    ;
fragment
OCTAL_ESC
    :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7')
    ;

fragment
UNICODE_ESC
    :   '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
    ;

program :
       PROG ID L_FBR statement+ R_FBR
;

statement    :
       (   ( var_define
             | assign
             | fun_print
             ) DOT_COMMA )
       | if_stat
;
```

Библиотека БГУИР

```
if_stat      :
       IF L_BR log_expr R_BR L_FBR if_statement+ R_FBR DOT_COMMA?
;

if_statement :
       (      (assign
              |fun_print
              ) DOT_COMMA
       | if_stat
       )
;

var_define    :
       type ID ASSIGN var_value (COMMA ID ASSIGN var_value)*
;
var_value     :
        ID
       |point
       |triangel
       |line
       |dec
       |NULL
;
assign        :
       ID ASSIGN expr
;
subexpr :
       ( expr_fun
       | L_BR expr R_BR
       | var_value
       )
;
sublog_expr :
       expr LOG_SIGN expr
;
expr :
       mult ((PLUS|MINUS) mult)*
;
mult :
       subexpr (MULT subexpr)*
;
log_expr:
       ((sublog_expr ((AND|OR) sublog_expr)*) | (NOT sublog_expr))
;
expr_fun returns [String type]
:       fun_length
       |fun_perimeter
       |fun_square
       ;
fun_length
:
       FUN_LENGTH L_BR var_value R_BR
;
fun_perimeter
:
       FUN_PERIM L_BR var_value R_BR
;
fun_square
```



Библиотека БГУИР

```
	:
		FUN_SQUARE L_BR var_value R_BR
	;
	fun_print
	:
		FUN_PRINT L_BR expr R_BR
	;

	triangel	:
		L_CBR (point|name_point) COMMA (point|name_point) COMMA (point|name_point)
R_CBR
	;
	line :
		'{' (point|name_point) COMMA (point|name_point) '}'
	;
	name_point	:
		ID
	;
	point  :
		(L_BR dec COMMA dec R_BR)
	;

	type	returns [String type]
	:	  TYPE_POINT
		| TYPE_LINE
		| TYPE_TRIANGLE
		| TYPE_DEC
	;
	dec	:
		  INT
	;
```

Библиотека БГУИР

## ПРИЛОЖЕНИЕ Б

(*справочное*)

### Содержимое файла грамматики на заключительном этапе

Окончательный файл грамматики Planimetry.g со вставками Java-кода и templates:

```
grammar Planimetry;

options {
  language = Java;
  output = template;
}

tokens{
FUN_PRINT         =     'print';
FUN_PERIM         =     'perimeter';
FUN_LENGTH        =     'length';
FUN_SQUARE        =     'square';
PROG              =    'Program'     ;
L_FBR             =     '{';
R_FBR             =     '}';
L_CBR             =     '[';
R_CBR             =     ']';
L_BR              =     '(';
R_BR              =     ')';
COMMA             =     ',';
DOT_COMMA         =     ';';
ASSIGN                  =     '=';
AND                     =     'and';
NOT                     =     'not';
OR                      =     'or';
IF                      =     'if';
TYPE_POINT        =   'Point' ;
TYPE_LINE         =   'Line' ;
TYPE_SEG          =   'Segment' ;
TYPE_TRIANGLE     =   'Triangle' ;
TYPE_CIRCLE       =   'Circle' ;
TYPE_DEC          =   'Decimal' ;
PLUS              =   '+';
MINUS             =   '-';
MULT              =   '*';
DIV                     =  '/';
NULL              =   'null';
}

scope MScope {
      Hashtable<String, Name> symbols;
}

@header {
```

Библиотека БГУИР

```
package grammar;
  import types.*;
  import types.Name;
  import java.util.Hashtable;
  import exceptions.NotDefinedVariable;
  import org.antlr.stringtemplate.*;
}

@lexer::header {
package grammar;
}

@members {
public static ArrayList<String> errors = new ArrayList<String>();
public static NamesTable names = new NamesTable();
public boolean isDefined(String id){
      if(names.get(id)!=null){
                  return true;
      }
      return false;
      }
}

LOG_SIGN :  '<'|'>'|'==';
ID   : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
     ;

INT : '0'..'9'+
     ;

FLOAT
     :    ('0'..'9')+ '.' ('0'..'9')* EXPONENT?
     |    '.' ('0'..'9')+ EXPONENT?
     |    ('0'..'9')+ EXPONENT
     ;

COMMENT
     :    '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
     |    '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
     ;

WS   :    ( ' '
          | '\t'
          | '\r'
          | '\n'
          ) {$channel=HIDDEN;}
     ;

STRING
     :  '"' ( ESC_SEQ | ~('\\'|'"') )* '"'
     ;

CHAR:  '\'' ( ESC_SEQ | ~('\''|'\\') ) '\''
```

Библиотека БГУИР

```
    ;

fragment
EXPONENT : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;

fragment
HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment
ESC_SEQ
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
    |   UNICODE_ESC
    |   OCTAL_ESC
    ;

fragment
OCTAL_ESC
    :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7')
    ;

fragment
UNICODE_ESC
    :   '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
    ;

program
:
        PROG ID L_FBR (list_stat += statement)+ R_FBR
-> init(stat = {$list_stat})
;

statement   :
        (   ( var_define  {$st = $var_define.st;}
              | assign          {$st = $assign.st;}
              | fun_print       {$st = $fun_print.st;}
              ) DOT_COMMA )
        | if_stat               {$st = $if_stat.st;}
;

if_stat     :
        IF L_BR log_expr R_BR L_FBR (stat+=if_statement)+ R_FBR DOT_COMMA?

-> if_template(cond={$log_expr.st}, stat={$stat})
;

if_statement     :
        (     (assign     {$st = $assign.st;}
              |fun_print  {$st = $fun_print.st;}
              ) DOT_COMMA
        | if_stat         {$st = $if_stat.st;}
        )
```

Библиотека БГУИР

```
	;

	var_define	:
		t=type ids+=ID (ASSIGN values+=var_value)
					    (COMMA ids+=ID (ASSIGN values+=var_value))*
		{for(Token id : (List<Token>)$ids){
			Name n = new Name(id.getText(),$t.type,id.getLine());
			names.add(n);
		}}
	→
var_define_template(type={$type.type},list_id={$ids},list_value={$values})

	;
	var_value	:
		 ID		{
		 if(!isDefined($ID.text)){
			 throw new NotDefinedVariable($ID.text, $ID.line);
			}
		 $st = new StringTemplate($ID.text);}
		|point		{$st = $point.st;}
		|triangel	{$st = $triangel.st;}
		|line {$st = $line.st;}
		|dec	{$st = new StringTemplate("new Decimal("+$dec.text+")");}
		|NULL {$st = new StringTemplate($NULL.text);}
	;
	assign		:
		ID ASSIGN expr{
		if(!isDefined($ID.text)){
			 throw new NotDefinedVariable($ID.text, $ID.line);
			}
		}
	→ assign_template(id = {$ID}, expr = {$expr.st})
	;
	subexpr :
		( expr_fun			{$st = $expr_fun.st;}
		| L_BR expr R_BR
	{$st = new StringTemplate("("+$expr.st.toString()+")");}
		| var_value			{$st = $var_value.st;}
		)
	;
	sublog_expr :
		( left_e=expr LOG_SIGN right_e=expr		{
			if($LOG_SIGN.text.equals("<")){
				$st = new
StringTemplate("Operation.compare("+$left_e.st+","+$right_e.st+","+"CompareType.
LESS)");
			} else if($LOG_SIGN.text.equals(">")){
				$st = new
StringTemplate("Operation.compare("+$left_e.st+","+$right_e.st+","+"CompareType.
GR)");
			} else if($LOG_SIGN.text.equals("==")){
				$st = new
StringTemplate("Operation.compare("+$left_e.st+","+$right_e.st+","+"CompareType.
EQ)");
```

Библиотека БГУИР

```
                }
        }
        )
;
expr returns [String type]
:
        opd+=mult ((op=PLUS|op=MINUS) opd+=mult)*
        {if($op!=null){
                        if($op.text.equals("+")){
                                $st = %plus_template(operands={$opd});
                        } else if($op.text.equals("-")){
                            $st = %minus_template(operands={$opd});
                        }
                } else $st = %atom_template(operands={$opd});
        }

;
mult :
        opd+=subexpr (op=MULT opd+=subexpr)*
        {if($op!=null){
                        if($op.text.equals("*")){
                                $st = %mult_template(operands={$opd});
                        }
                } else $st = %atom_template(operands={$opd});
        }
;
log_expr
@init{
        String result = "";
}:
        (( f=sublog_expr
{result+=$f.st.toString();}((AND{result+="&&";}|OR{result+="||";})
s=sublog_expr{result+=$s.st.toString();})*)
        | (NOT t=sublog_expr {result+="!"+t.st.toString();})) {
$st = new StringTemplate(result);}
;
expr_fun returns [String type]
:       fun_length          {$st = $fun_length.st;}
        |fun_perimeter      {$st = $fun_perimeter.st;}
        |fun_square         {$st = $fun_square.st;}
        ;
fun_length returns [String type]
:
        FUN_LENGTH L_BR var_value R_BR {$type="decimal";}
→       fun_length_template(f={$var_value.st})
;
fun_perimeter returns [String type]
:
        FUN_PERIM L_BR var_value R_BR {$type="decimal";}
→       fun_perimeter_template(f={$var_value.st})
;
fun_square  returns [String type]
:
        FUN_SQUARE L_BR var_value R_BR {$type="decimal";}
```

Библиотека БГУИР

```
	→ fun_square_template(f={$var_value.st})
	;
	fun_print	returns [String type]
	:
		FUN_PRINT L_BR expr R_BR {$type="void";}
	→ fun_print_template(e = {$expr.st})
	;

	triangel	:
		L_CBR (p+=point|p+=name_point) COMMA (p+=point|p+=name_point) COMMA
(p+=point|p+=name_point) R_CBR	→	triangel_template(list_point={$p})
	;

	line :
		'{' (p+=point|p+=name_point) COMMA (p+=point|p+=name_point) '}'
	→	line_template(points={$p})
	;

	name_point	:
		p=ID{
		if(!isDefined($ID.text)){
			throw new NotDefinedVariable($ID.text,$ID.line);
			}
		}
	→name_point(point={$p})
	;

	point :
		(L_BR f=dec COMMA s=dec R_BR)
	→	point_template(f={$f.text}, s={$s.text})
	;

	type	returns [String type]
	:	TYPE_POINT {$type="Point";}
		| TYPE_LINE {$type="Line";}
		| TYPE_TRIANGLE {$type="Triangle";}
		| TYPE_DEC {$type="Decimal";}
	;

	dec	:
		INT
	;
```

Библиотека БГУИР

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1. Компиляторы: принципы, технологи и и инструменты / А. Ахо [и др.] ; пер. с англ. – 2-е изд. – М. : Издат. дом «Вильямс», 2015. – 1178 с.
2. Белоусов, А. И. Дискретная математика / А. И. Белоусов, С. Б. Ткачев. – М. : МГТУ им. Н. Э. Баумана, 2004. – 743 с.
3. Себеста, Р. У. Основные концепции языков программирования / Р. У. Себеста ; пер. с англ. – 5-е изд. – М. : Издат. дом «Вильямс», 2001. – 672 с.
4. Страуструп, Б. Дизайн и эволюция C++ / Б. Страуструп ; пер. с англ. – М. : ДМК Пресс ; СПб. : Питер, 2006. – 448 с.
5. Карпов, Ю. Основы построения трансляторов. Теория и технология программирования / Ю. Карпов. – М. : BHV, 2005. – 272 с.
6. Пратт, Т. Языки программирования: разработка и реализация / Т. Пратт, М. Зелковиц ; пер. с англ. – 4-е изд. – СПб. : Питер, 1979. – 688 с.
7. Ахо, А. Структуры данных и алгоритмы / А. Ахо, Д. Хопкрофт, Д. Ульман. – М. : Изд. дом «Вильямс», 2003. – 384 с.
8. Фаулер, М. Рефакторинг. Улучшение существующего кода / М. Фаулер. – М. : Символ, 2007. – 432 с.
9. Касьянов, В. Н. Графы в программировании: обработка, визуализация и применение / В. Н. Касьянов, В. А. Евстигнеев. – СПб. : БХВ–Петербург, 2003. – 1104 с.


Библиотека БГУИР

Св. план 2017, поз. 17

*Учебное издание*

**Голенков** Владимир Васильевич
**Гулякина** Наталья Анатольевна
**Давыденко** Ирина Тимофеевна
**Шункевич** Даниил Вячеславович

# *ЯЗЫКОВЫЕ ПРОЦЕССОРЫ ИНТЕЛЛЕКТУАЛЬНЫХ СИСТЕМ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ*

ПОСОБИЕ

Редактор *А. К. Мяделко*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *В. М. Задоля*

Подписано в печать 21.02.2018. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 6,63. Уч.-изд. л. 6,5. Тираж 80 экз. Заказ 144.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
ЛП №02330/264 от 14.04.2014.
220013, Минск, П. Бровки, 6

Библиотека БГУИР