

Московский государственный технический университет  
имени Н.Э. Баумана

=====

Факультет «Информатика и системы управления»

Кафедра «Компьютерные системы и сети»

**Г.С. Иванова, Т.Н. Ничушкина**

# **ОСНОВЫ КОНСТРУИРОВАНИЯ КОМПИЛЯТОРОВ**

Учебное пособие

Москва

(С) 2021 МГТУ им. Н.Э. БАУМАНА

УДК 004.4'41

ББК 32.973.2

*Рецензенты:*

**Иванова Г.С., Ничушкина Т.Н.**

Основы конструирования компиляторов: учебное пособие. – М.: МГТУ имени Н.Э. Баумана, 2021. – 104 с.

Учебное пособие содержит сведения из математической логики и теории формальных языков, составляющие основу для построения лексических и синтаксических анализаторов. Приведены математические определения формального языка и формальной грамматики, описана классификация формальных грамматик Хомского. Рассмотрены способы построения распознающих конечных автоматов и автоматов с магазинной памятью, а также метод рекурсивного спуска и метод грамматического разбора, основанный на свойствах грамматик с операторным предшествованием.

Издание предназначено для студентов второго курса кафедры «Компьютерные системы и сети» направлений 09.03.01 «Информатика и вычислительная техника» и 09.03.03 «Прикладная информатика», изучающих модуль 3 дисциплины «Машинно-зависимые языки и основы компиляции».

*Рекомендовано учебно-методическим советом МГТУ им. Н.Э. Баумана*

*Учебное пособие*

**Иванова Галина Сергеевна**

**Ничушкина Татьяна Николаевна**

**Основы конструирования компиляторов**

© 2021 МГТУ имени Н.Э. Баумана

[Оглавление](#)

Г.С. Иванова, Т.Н. Ничушкина. Основы конструирования компиляторов

## Оглавление

Предисловие.....	4
Глоссарий.....	6
Введение.....	8
<b>1. Структура компилятора.....</b>	<b>9</b>
1.1. Основные понятия и определения .....	9
1.2. Этапы процесса компиляции .....	10
1.3. Метод Рутисхаузера .....	13
Контрольные вопросы и задания .....	16
<b>2. Основные положения теории формальных языков.....</b>	<b>17</b>
2.1. Формальная грамматика и формальный язык.....	17
2.2. Понятие грамматического разбора .....	22
2.2.1. Левосторонний нисходящий грамматический разбор («сверху–вниз») .....	24
2.2.2. Левосторонний восходящий грамматический разбор («слева–направо»).....	28
2.3. Расширенная классификация грамматик Хомского .....	30
Контрольные вопросы и задания .....	33
<b>3. Распознавание регулярных грамматик.....</b>	<b>35</b>
3.1. Конечный автомат и его программная реализация .....	35
3.2. Построение лексических анализаторов с использованием конечных автоматов .....	37
3.3. Построение синтаксических анализаторов с использованием конечных автоматов.....	48
Контрольные вопросы и задания .....	51
<b>4. Распознавание контекстно-свободных грамматик .....</b>	<b>52</b>
4.1. Автомат с магазинной памятью .....	52
4.2. Синтаксические анализаторы $LL(k)$ -грамматик. Метод рекурсивного спуска .....	57
4.3. Синтаксические анализаторы $LR(k)$ -грамматик. Грамматики предшествования.....	68
4.4. Польская запись. Алгоритм Бауэра–Замельзона.....	76
Контрольные вопросы и задания .....	79
<b>5. Генерация и оптимизация кодов.....</b>	<b>81</b>
5.1. Распределение памяти под программы и данные.....	81
5.2. Генерация кодов.....	82
5.3. Машинно-независимая оптимизация.....	83
5.3.1. Удаление недостижимого кода .....	83
5.3.2. Оптимизация линейных участков программы.....	84
5.3.3. Подстановка кода функции вместо ее вызова в объектный код .....	85
5.3.4. Оптимизация вычислений в циклах.....	85
5.4. Машинно-зависимая оптимизация.....	86
5.4.1. Распределение регистров процессора.....	86
5.4.2. Оптимизация передачи параметров в процедуры и функции .....	87
5.4.3. Оптимизация кода для процессоров, допускающих распараллеливание вычислений .....	87
Контрольные вопросы и задания .....	88
Заключение .....	90
Литература .....	91
Ответы на задания .....	92

## Предисловие

Учебное пособие составлено в соответствии с самостоятельно устанавливаемыми образовательными стандартами (СУОС), основными образовательными программами по направлениям подготовки бакалавров 09.03.01 «Информатика и вычислительная техника» и 09.03.03 «Прикладная информатика», а также программой дисциплины. Издание предназначено для самостоятельного изучения студентами модуля 3 дисциплины «Машинно-зависимые языки и основы компиляции» и подготовки к выполнению домашнего задания указанного модуля.

Цель учебного пособия – подготовка бакалавра в области разработки компилирующих программ для специализированных языков программирования.

В результате изучения материала, представленного в пособии, студент будет знать: основные положения теории формальных языков и формальных грамматик, классификацию грамматик формальных языков Хомского, определение и способы грамматического разбора, методы распознавания конструкций языков регулярного типа с использованием конечных автоматов, методы распознавания конструкций контекстно-свободных грамматик и алгоритмы разработки соответствующих программ.

Для успешного освоения материала необходимо предварительное изучение следующих дисциплин: «Основы программирования», «Информатика», «Объектно-ориентированное программирование».

Задача издания предоставить студентам материал, который будет способствовать более качественному усвоению материала модуля 3 дисциплины «Машинно-зависимые языки и основы компиляции».

Для успешного усвоения материала необходимо внимательно разбирать предлагаемые примеры, также целесообразно синтезировать и прораба-

### [Оглавление](#)

тывать аналогичные задачи. Ответы на вопросы позволят обучающемуся оценить степень понимания и усвоения теоретических положений, способов и методов. Проработка материала обеспечит прохождение рубежного контроля по модулю и качественное выполнение домашнего задания по дисциплине, связанного с темой конструирования компиляторов.

Примеры программ в пособии написаны с использованием RAD Studio 10 (проприетарная лицензия получена МГТУ им. Н.Э. Баумана 01.01.2017).

## Глоссарий

**Аксиома грамматики** – нетерминальный символ, из которого можно вывести все предложения языка.

**Алфавит** – непустое конечное множество символов, используемых для записи предложений языка.

**Бэкуса–Наура форма (БНФ)** – нотация для записи правил синтаксиса формального языка.

**Грамматический разбор** – процесс построения синтаксического дерева для конкретного предложения формального языка.

**Компилятор** – транслятор с языка высокого уровня на машинный язык или язык ассемблера.

**Лексема** – относительно простое понятие языка, которое соответствует его слову.

**Лексический анализ** – процесс преобразования исходного текста в строку однородных символов (токенов).

**Нетерминальный символ** – элемент грамматики (символ), имеющий собственные имена и структуру.

**Семантика** – это совокупность правил, определяющих логическое соответствие между элементами и значением синтаксически корректных предложений, т. е. *содержание* языка.

**Сентенциальная форма** – промежуточная форма грамматического разбора, как правило, содержащая терминальные и нетерминальные символы.

**Синтаксис** – это совокупность правил, определяющих допустимые конструкции языка, т. е. его *форму*.

**Синтаксический анализ** – процесс распознавания конструкций языка в строке токенов.

**Сканер** – программа, выполняющая лексический анализ.

## [Оглавление](#)

**Терминальный символ** (терминал, символ алфавита) – минимальный элемент грамматики языка – символ, считающийся в рамках языка не имеющим структуры (неделимым, конечным).

**Токен** – символ строки результата лексического анализа.

**Транслятор** – программа, которая переводит программу, написанную на одном языке, в эквивалентную ей программу, написанную на другом языке.

**Формальная грамматика** – это математическая система, определяющая язык посредством порождающих правил.

**Формальный язык**  $L$  в алфавите  $A$  – подмножество множества  $A^* = A^+ \cup \{e\}$ , где  $A^*$  – множество всех составленных из символов алфавита  $A$  строк, в которое входит пустая строка,  $A^+$  – множество всех составленных из символов алфавита  $A$  строк, в которое не входит пустая строка,  $e$  – пустая строка.

## **Введение**

Построение транслирующих и компилирующих программ базируется на глубоком понимании теории формальных языков. Однако соответствующий раздел математики сложен для усвоения, и прежде всего это происходит из-за большого количества терминов, которые используются только в рамках указанной теории.

При изучении дисциплины «Машинно-зависимые языки и основы компиляции» студент должен получить минимально необходимые теоретические и практические знания, достаточные для разработки простейших распознающих программ, что и определило подбор материала:

- основные положения теории формальных языков и формальных грамматик;
- классификация грамматик формальных языков Хомского;
- определение и способы грамматического разбора;
- распознавание языков регулярного типа с использованием конечных автоматов;
- распознавание контекстно-свободных языков методом рекурсивного спуска и стековым методом;
- польская запись и алгоритм Бауэра–Замельзона.

Учебное пособие также содержит тексты программ, демонстрирующих применение излагаемой теории на практике.



## 1. Структура компилятора

В настоящем разделе перечисляются основные виды существующих компилирующих программ и рассмотрены этапы компиляции, определяющие структуру программы. Также поясняется основная проблема, связанная с разработкой компилирующих программ, и приводится исторически первый метод генерации последовательности операций, выполняемых при вычислении сложного выражения.

### 1.1. Основные понятия и определения

Любой язык (формальный или естественный) обязательно подчиняется правилам, которые определяют его синтаксис и семантику. *Синтаксис* – это совокупность правил, задающих допустимые конструкции языка, т. е. его *форму*. *Семантика* – это совокупность правил, определяющих логическое соответствие между элементами и значением синтаксически корректных предложений, т. е. *содержание* языка.

Существует несколько программ различных видов, в которых используются элементы теории формальных языков, правила описания синтаксиса или семантики. Наиболее известными из них являются: трансляторы с одних языков программирования на другие, компиляторы, ассемблеры, интерпретаторы, макрогенераторы, препроцессоры и т. п. Эти программы следует различать.

*Транслятор* – программа, которая переводит программу, написанную на одном языке, в эквивалентную ей программу, написанную на другом языке.

*Компилятор* – транслятор с языка высокого уровня на машинный язык или язык ассемблера.

*Ассемблер* – транслятор с языка ассемблера на машинный язык.

*Интерпретатор* – программа, которая принимает исходную программу и выполняет ее, не создавая программы на другом языке.

*Макропроцессор (препроцессор* для компиляторов языков высокого уровня) – программа, которая принимает исходную программу как текст и выполняет в нем замены определенных символов на подстроки. Макропроцессор обрабатывает программу до трансляции/компиляции и генерирует текст программы в соответствии с командами макрогенерации/командами препроцессора.

Чаще всего приходится иметь дело с компиляторами.

## ***1.2. Этапы процесса компиляции***

Процесс компиляции предполагает распознавание конструкций исходного языка (анализ) и сопоставление каждой правильной конструкции семантически эквивалентной конструкции другого языка (синтез).

Процесс компиляции включает несколько этапов, основными среди которых являются:

- лексический анализ,
- синтаксический анализ,
- семантический анализ,
- распределение памяти,
- генерация и оптимизация объектного кода.

***Лексический анализ*** – преобразование исходного текста в строку однородных символов. Каждый символ строки (*токен*) соответствует языковой единице (символу или «слову» языка), т. е. *лексеме* и характеризуется набором атрибутов, таких как тип, адрес и т. п. Поэтому строку токенов часто представляют таблицей, каждая строка которой описывает один токен.

Лексема – термин, обозначающий относительно простое понятие языка, которое соответствует его слову. Всего существует два типа лексем:

а) для символов алфавита языка, таких как «Служебные слова» и «Служебные символы»;

б) для базовых понятий языка, таких как «Идентификатор» и «Литерал».

**Пример 1.1.** Выполнить лексический анализ предложения языка:

```
if Sum>5 then pr:= true;
```

Отдельные части («слова» предложения) разделяются одним или несколькими пробелами или точкой с запятой.

Кроме этого, если следующее слово – служебный символ, то никаких дополнительных разделяющих символов синтаксис языка не требует.

Программа лексического анализа (сканер), используя указанные варианты разделения слов, должна сформировать таблицу токенов (табл. 1.1) и проверить таблицы переменных (табл. 1.2) и констант (табл. 1.3).

**Таблица 1.1** Таблица токенов

Лексема	Тип лексемы	Значение	Ссылка
if	Служебное слово	Код «if»	—
Sum	Идентификатор	—	Адрес в таблице идентификаторов
>	Служебный символ	Код «>»	—
5	Литерал	—	Адрес в таблице литералов
then	Служебное слово	Код «then»	—
pr	Идентификатор	—	Адрес в таблице идентификаторов
:=	Служебный символ	Код «:=»	—
true	Литерал	—	Адрес в таблице литералов
;	Служебный символ	Код «;»	—

**Таблица 1.2** Таблица идентификаторов переменных

Имя	Тип	Адрес
Sum	Integer	∅ (пока не распределена)
pr	Boolean	∅ (пока не распределена)

**Таблица 1.3** Таблица констант

Константа	Тип	Значение
5	Byte	00000101
true	Boolean	00000001

Следует иметь в виду, что лексический анализ может выполняться двумя способами: с построением строки токенов, как описано выше, и без построения строки токенов в явном виде.

Программа, которая строит строку токенов в явном виде, называется *сканером*. Она обрабатывает исходное предложение языка или сразу весь исходный текст до вызова программы синтаксического анализа. Формат создаваемой сканером строки токенов зависит от особенностей формального языка.

Во втором способе анализ лексем встроен в синтаксический анализ. При этом сканеры не создают единой программы, вызывая подпрограммы лексического анализа различных лексем по мере их обнаружения при разборе исходного предложения, как это происходит, например, при использовании метода рекурсивного спуска (см. раздел 4.2.).

**Синтаксический анализ** –распознавание конструкций языка в строке токенов, полученной после работы сканера или формируемой по мере разбора предложения. Главными результатами являются отождествление строки или подстроки с некоторой конструкцией и, при необходимости, получение информации об обнаруженных при этом ошибках.

**Пример 1.2.** Выполнить синтаксический анализ строки токенов, описанной табл. 1.1.

На этом этапе должны быть распознаны конструкции: <Логическое выражение>, <Оператор присваивания>, <Оператор *if*>.

**Семантический анализ** –распознавание/проверка смысла конструкции. По результатам распознавания строится последовательность, приближенная

к последовательности операторов будущей программы, и выполняются предусмотренные проверки правильности программы.

**Пример 1.3.** На этапе семантического анализа для рассматриваемого примера, например, может быть проверена инициализация переменной *Sum*.

**Распределение памяти** – назначение адресов для размещения именованных и неименованных констант, а также переменных программы.

**Генерация и оптимизация объектного кода** – формирование программы на выходном языке, которая семантически эквивалентна исходной программе. На этом этапе также обычно оптимизируется генерируемый код.

Лексический и синтаксический анализы предполагают грамматический разбор. При этом используют специальный математический аппарат – формальные грамматики. Однако до появления указанного математического аппарата существовали методы грамматического разбора, построенные по эвристическому принципу.

### 1.3. Метод Рутисхаузера

Первые компилирующие программы обеспечивали перевод формульной записи выражений в машинный язык. В основе такого перевода лежит представление выражения в виде последовательности *троек*, где каждая тройка включает два адреса операндов и операцию:

$E_L \text{ } \odot \text{ } E_R$ , где  $E_L$ ,  $E_R$  – левый и правый операнды,  $\odot$  – операция.

Основной проблемой при этом является необходимость учета *приоритетов* операций. Например, для выражения  $a+b*c$  должны быть построены тройки  $T_1=b*c$ ,  $T_2=a+T_1$ .

Для решения этой задачи первым был предложен метод Рутисхаузера.

**Метод Рутисхаузера** требует, чтобы выражение было записано в полной скобочной записи, когда порядок выполнения операций указывается не

правилами приоритета, а скобками. Так, выражение  $d=a+b*c$  должно быть записано в виде  $d=a+(b*c)$ , в противном случае сначала будет выполняться операция сложения.

Метод заключается в следующем.

Шаг 1. Каждому символу строки выражения  $S_i$  ставится в соответствие индекс  $N_i$  по следующему алгоритму:

**$N[0]:=0$**

**$J:=1$**

**Цикл-пока  $S[J] \neq ' '$**

**Если  $S[J] = '('$  или  $S[J] = \langle \text{операнд} \rangle$**

**то  $N[J]:=N[J-1] + 1$**

**иначе  $N[J]:=N[J-1] - 1$**

**Все-если**

**$J:=J+1$**

**Все-цикл**

**$N[J]:=0$**

Шаг 2. Определяется наибольшее значение индекса  $k$  в строке индексов. Это значение входит в подстроку вида  $k(k-1)k$  или при наличии скобок – в подстроку вида  $(k-1)k(k-1)k(k-1)$ . После этого строится соответствующая подстроке тройка.

Шаг 3. Обработанные символы вместе со скобками удаляются, и на их место ставится значение  $n = k-1$ .

Шаг 4. Шаги 2, 3 повторяются до завершения преобразования выражения в последовательность троек.

**Пример 1.4.** Получить тройки для выражения  $((a+b)*c)+d)/k$ .

Процесс получения троек представлен в табл. 1.4.

**Таблица 1.4** Получение троек из выражения методом Рутисхаузера

№ шага	Строка / Индексы	Тройка
1	S:     ( ( ( a + b ) * c ) + d ) / k N: 0 1 2 <u>3 4 3 4 3</u> 2 3 2 1 2 1 0 1 0	$T_1 = a + b$
2	S:     ( ( T <sub>1</sub> * c ) + d ) / k N: 0 1 <u>2 3 2 3 2</u> 1 2 1 0 1 0	$T_2 = T_1 * c$
3	S:     ( T <sub>2</sub> + d ) / k N: 0 <u>1 2 1 2 1</u> 0 1 0	$T_3 = T_2 + d$
4	S:     T <sub>3</sub> / k N: <u>0 1 0 1 0</u>	$T_4 = T_3 / k$

Основной недостаток метода – требование полной скобочной записи. Как правило, для ее получения используют специальную программу, которая вставляет скобки, следуя принятым в математике приоритетам операций. Так, например, выражение  $a + b / c$  программой расстановки скобок должно быть преобразовано к виду  $a + (b / c)$ , поскольку операция деления по правилам арифметики имеет более высокий приоритет и, следовательно, должна выполняться в первую очередь.

Метод Рутисхаузера – сравнительно простой метод разбора выражений, базирующийся на определяющих приоритет скобках, оказался непосредственно не применим для разбора не связанных с выражениями операторов языков программирования. Решение проблем компиляции долгое время было непростым. Первые компиляторы являлись крайне сложными для того времени программами, написание которых требовало неординарных способностей. Вследствие своей сложности компиляторы содержали большое количество ошибок и требовали значительного времени на тестирование и отладку. Так, создание первого компилятора языка программирования *Fortran* потребовало 18 человеко-лет.

В математической теории формальных языков предложен соответствующий инструментарий, в результате чего небольшой компилятор может стать темой курсовой работы студента.

### ***Контрольные вопросы и задания***

- 1.1. Назовите основные типы программ, включающих элементы лексического и синтаксического анализа. Поясните их различия.

[Ответ.](#)

- 1.2. Назовите основные этапы процесса компиляции. Уточните задачи каждого этапа.

[Ответ.](#)

- 1.3. В чем заключается метод Рутисхаузера?

[Ответ.](#)

- 1.4. Почему метод Рутисхаузера не используют при разборе операторов языков программирования?

[Ответ.](#)

- 1.5. Для заданного предложения на языке C++ покажите, какие операции будут выполняться на этапах лексического и синтаксического анализа:

```
for (i1=0, s=0; i1<=5; i1++)  
    if (i1%3==0) s+=abc[i1];
```

[Решение.](#)

- 1.6. Выполните построение троек методом Рутисхаузера для выражения  $r12 * (h + gn) + f * tt1$ .

[Решение.](#)



## 2. Основные положения теории формальных языков

Для решения проблем, связанных с анализом исходных текстов программ, написанных на языках высокого уровня, в том числе специализированных, в 70-е годы прошлого века была создана математическая теория формальных языков. Появление этой теории существенно упростило разработку языков программирования и, таким образом, положило начало созданию большого количества новых языков как универсальных, так и специализированных.

### 2.1. Формальная грамматика и формальный язык

Предложения любого языка (формального или естественного) строятся из символов алфавита.

*Алфавит* – непустое конечное множество символов, используемых для записи предложений языка. Например, множество арабских цифр, знаки «+» и «-», т. е.  $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -\}$  – это алфавит для записи целых десятичных чисел, например: «-365», «78», «+45».

*Строка* – любая последовательность символов алфавита, расположенных один за другим. Строки в теории формальных языков, как правило, обозначают строчными греческими буквами:  $\alpha, \beta, \gamma, \dots$ . Строка, не содержащая символов, называется *пустой* и далее будет обозначаться латинской буквой  $e$ . В литературе с этой целью также используют греческие буквы  $\varepsilon$  или  $\lambda$ .

Множество всех составленных из символов алфавита  $A$  строк, в которое входит пустая строка, обозначают  $A^*$ . Множество всех составленных из символов алфавита  $A$  строк, в которое не входит пустая строка, обозначают  $A^+$ . Откуда:  $A^* = A^+ \cup \{e\}$ , где  $e$  – пустая строка.

*Формальным языком  $L$  в алфавите  $A$*  называют подмножество множества  $A^*$ . Таким образом, язык  $L(A)$  – это совокупность допустимых предложений, составленных из  $A$ .

Задать язык  $L(A)$  означает: либо перечислить все включаемые в него предложения, либо указать правила образования допустимых предложений.

Если язык содержит бесконечное количество предложений, то задать его перечислением невозможно.

Определение правил образования предложений осуществляют с использованием абстракций формальных грамматик.

*Формальная грамматика* – это математическая система, определяющая язык посредством порождающих правил – *правил продукции*. При задании правил используют понятия: терминальные символы (терминалы) и нетерминальные символы (нетерминалы).

*Терминалы* – это минимальные элементы грамматики языка – символы его алфавита, считающиеся в рамках языка не имеющими структуры (неделимыми, конечными). При записи правил грамматики в общем виде терминалы обычно обозначают строчными буквами латинского алфавита, а при описании правил конкретного языка используют непосредственно символы алфавита  $A$  языка.

*Нетерминалы* – это элементы грамматики (символы), имеющие собственные имена и структуру. В общем случае они могут включать в определенном порядке другие нетерминалы, терминалы или представлять собой пустую строку. При записи правил в общем виде нетерминалы обычно обозначают прописными буквами латинского алфавита, а в правилах конкретного языка записывают их имена в угловых скобках, например **<Целое>**.

Полное описание грамматики представляет собой набор правил, которые определяют все нетерминальные символы грамматики так, что каждый нетерминальный символ может быть сведён к комбинации терминальных символов путём последовательного применения правил. Особую роль при этом

играет *аксиома грамматики* – нетерминал, из которого можно вывести все предложения языка.

Математически формальная грамматика определяется как «четверка»

$$G = (V_T, V_N, P, S),$$

где  $V_T$  – алфавит языка или множество терминальных (незаменяемых) символов;

$V_N$  – множество нетерминальных (заменяемых) символов, являющееся вспомогательным алфавитом, символы которого обозначают допустимые понятия языка;

$$V_T \cap V_N = \emptyset; V = V_T \cup V_N \text{ – словарь грамматики};$$

$P$  – множество порождающих правил. Каждое правило состоит из пары строк  $(\alpha, \beta)$ , где  $\alpha \in V^+$  – левая часть правила,  $\beta \in V^*$  – правая часть правила, что обозначается как  $\alpha \rightarrow \beta$ , где строка  $\alpha$  должна содержать хотя бы один нетерминал;

$$S \in V_N \text{ – начальный символ (аксиома грамматики)}.$$

Для описания синтаксиса языков с бесконечным количеством различных предложений используют *рекурсию*. При этом если нетерминал в порождающем правиле расположен справа, то рекурсию называют *правосторонней*, если слева, то *левосторонней*, а если между двумя подстроками, то *вложенной*.

**Пример 2.1.** Определить грамматику записи десятичных чисел  $G_0$ .

Граматику записи десятичных чисел можно определить следующим образом:

$$V_T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -\};$$

$$V_N = \{\langle \text{Целое} \rangle, \langle \text{ЦелоеБезЗнака} \rangle, \langle \text{Цифра} \rangle, \langle \text{Знак} \rangle\};$$

$$P = \{\langle \text{Целое} \rangle \rightarrow \langle \text{Знак} \rangle \langle \text{ЦелоеБезЗнака} \rangle,$$

$$\langle \text{Целое} \rangle \rightarrow \langle \text{Целое без знака} \rangle,$$

$$\langle \text{ЦелоеБезЗнака} \rangle \rightarrow \langle \text{Цифра} \rangle \langle \text{ЦелоеБезЗнака} \rangle,$$

$$\langle \text{ЦелоеБезЗнака} \rangle \rightarrow \langle \text{Цифра} \rangle,$$

$$\langle \text{Цифра} \rangle \rightarrow 0, \langle \text{Цифра} \rangle \rightarrow 1, \langle \text{Цифра} \rangle \rightarrow 2,$$

$\langle \text{Цифра} \rangle \rightarrow 3, \langle \text{Цифра} \rangle \rightarrow 4, \langle \text{Цифра} \rangle \rightarrow 5,$   
 $\langle \text{Цифра} \rangle \rightarrow 6, \langle \text{Цифра} \rangle \rightarrow 7, \langle \text{Цифра} \rangle \rightarrow 8,$   
 $\langle \text{Цифра} \rangle \rightarrow 9,$   
 $\langle \text{Знак} \rangle \rightarrow +, \langle \text{Знак} \rangle \rightarrow - \};$

$S = \langle \text{Целое} \rangle.$

*Примечание.* Грамматика для описания последовательностей цифр, входящих в запись целого числа, использует правило с правосторонней рекурсией:

$\langle \text{ЦелоеБезЗнака} \rangle \rightarrow \langle \text{Цифра} \rangle \langle \text{ЦелоеБезЗнака} \rangle.$

Благодаря наличию этого правила целые числа, соответствующие правилам языка, могут содержать любое количество цифр  $k \geq 1$ .

Нетрудно видеть, что описание правил продукции в каноническом виде очень длинное. Для описания синтаксиса обычно используют более компактные и наглядные формы (модели), например, форму Бэкуса–Наура (БНФ), расширенную форму Бэкуса–Наура (РБНФ) или синтаксические диаграммы.

**Форма Бэкуса–Наура.** БНФ связывает терминальные и нетерминальные символы, используя две операции: « $::=$ » – «можно заменить на» и « $|$ » – «или». Основное достоинство БНФ – группировка правил, определяющих каждый нетерминал. Например, правила продукции грамматики целых чисел, рассмотренной выше, с использованием БНФ можно записать следующим образом:

$\langle \text{Целое} \rangle ::= \langle \text{Знак} \rangle \langle \text{ЦелоеБезЗнака} \rangle | \langle \text{ЦелоеБезЗнака} \rangle,$   
 $\langle \text{ЦелоеБезЗнака} \rangle ::= \langle \text{Цифра} \rangle \langle \text{ЦелоеБезЗнака} \rangle | \langle \text{Цифра} \rangle,$   
 $\langle \text{Цифра} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9,$   
 $\langle \text{Знак} \rangle ::= + | -.$

**Расширенная форма Бэкуса–Наура.** РБНФ в отличие от БНФ существует в нескольких вариантах. Чаще прочих встречается вариант, в котором нетерминалы записывают без угловых скобок, а терминалы – в двойных или одинарных кавычках, используя следующие операции:

[Оглавление](#)

« = » – «это есть» (для отделения левой части правила);

« , » – конкатенация (сцепление) символов (для записи последовательных символов в строке);

«[...]» – условное вхождение (для указания необязательной части строки);

«{...}» – повтор (для записи повторяющейся подстроки, в том числе возможен повтор ноль раз);

« | » – выбор, т. е. операция «или» (если в строку может входить один из указанных вариантов);

«(...)» – группировка символов (если необходимо сгруппировать несколько символов при описании строки).

Записи правил в РФБН обычно получаются более сложными, чем в БНФ, так как используют больше операций, но и более компактными. Так, с применением перечисленных обозначений правила грамматики целых чисел можно записать следующим образом:

**Целое** = ["+"|"-" ] **Цифра**{**Цифра**}.

**Цифра** = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".

Нетрудно видеть, что для описания правил потребовалось всего два нетерминала, но запись использует больше операций, чем в БНФ.

**Синтаксическая диаграмма.** Математики синтаксическую диаграмму часто называют «графом Глени», хотя, согласно определению, это очевидно не граф. Синтаксическая диаграмма – наглядное графическое представление правил продукции. На ней терминальные символы рисуют в округлых или овальных блоках, нетерминальные – в прямоугольных или квадратных. Стрелками показывают пути формирования допустимых строк из терминальных и нетерминальных символов (см. далее рис. 3.2–3.5, 4.2).

В целом формальная грамматика определяет правила формирования допустимых конструкций языка, т. е. его синтаксис.

## 2.2. Понятие грамматического разбора

Принадлежность строки конкретному языку распознается его выводом из аксиомы. *Вывод* представляет собой последовательность подстановок, при выполнении которых левая часть правила заменяется правой.

Строки, получаемые в процессе вывода, и аксиому грамматики в процессе вывода называют *сентенциальными формами*. Сентенциальная форма, содержащая только терминальные символы, называется *допустимой* и представляет собой предложение языка.

Для обозначения подстановки в процессе вывода используют символ « $\Rightarrow$ », справа от которого указывают номер подстановки.

**Пример 2.2.** Вывод строки «-45» из аксиомы <Целое>:

<Целое>  $\Rightarrow_1$

$\Rightarrow_1$  <Знак><ЦелоеБезЗнака>  $\Rightarrow_2$

$\Rightarrow_2$  <Знак><Цифра><ЦелоеБезЗнака>  $\Rightarrow_3$

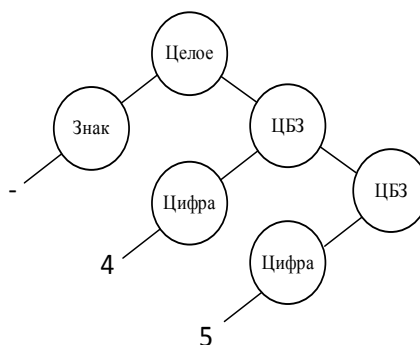
$\Rightarrow_3$  <Знак><Цифра><Цифра>  $\Rightarrow_4$

$\Rightarrow_4$  - <Цифра><Цифра>  $\Rightarrow_5$

$\Rightarrow_5$  - 4<Цифра>  $\Rightarrow_6$

$\Rightarrow_6$  - 45

Вывод можно представить графически в виде *синтаксического дерева* или *дерева грамматического разбора*, в корне которого находится аксиома, а листья образуют допустимое предложение языка (рис. 2.1).



**Рис. 2.1.** Синтаксическое дерево

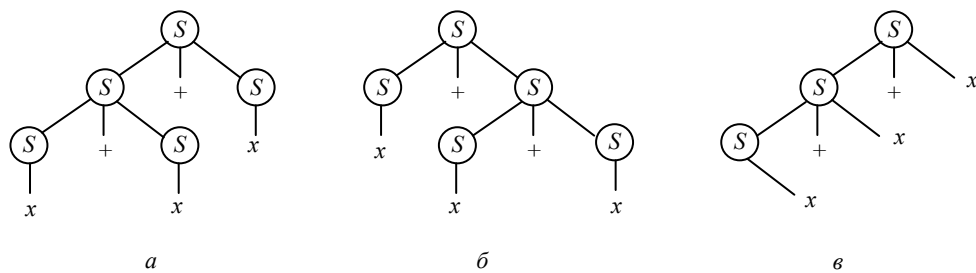
**Неоднозначные грамматики.** Существуют грамматики, в которых для одной строки можно построить несколько деревьев. Такие грамматики называются *неоднозначными*. Разбор неоднозначных грамматик затруднен, поэтому их, если возможно, преобразуют в однозначные или ограничивают правилами.

**Пример 2.3.** Строка  $x + x + x$ , порождаемая грамматикой с правилами:

$$S \rightarrow S + S,$$

$$S \rightarrow x,$$

имеет два дерева разбора (рис. 2.2, а, б).



**Рис. 2.2.** Деревья разбора неоднозначной грамматики (а, б) и единственное дерево соответствующей однозначной грамматики (в)

Описывающая тот же язык однозначная грамматика (рис. 2, в) использует правила

$$S \rightarrow S + x,$$

$$S \rightarrow x.$$

**Пример 2.4.** Применение правила описания конструкции

```
if <ЛогичВыраж> then <Оператор> [else <Оператор>];
```

при разборе предложения

```
if a>5 then if f< = -6 then s: = 0 else s: = -1;
```

приводит к неоднозначности результата разбора (неясно, к какому if относится else). Поскольку преобразование правила грамматики конструкции невозможно, разбор ограничен *правилом вложенности*, которое всегда относит else к ближайшему if.

**Грамматический разбор.** *Грамматический разбор* – построение синтаксического дерева для конкретного предложения языка. Построение такого дерева позволяет однозначно доказать, что анализируемая строка языка является *допустимой*, т. е. принадлежит ему.

Грамматический разбор может осуществляться в разной последовательности, при этом дерево можно строить как «сверху», от корня – аксиомы, так и «снизу», от листьев – терминалов, входящих в допустимое предложение языка. Соответственно различают *нисходящий* и *восходящий* методы разбора. При этом предложения языка можно рассматривать слева направо и справа налево. Соответственно получаем *левосторонний* и *правосторонний* разборы. Левосторонний разбор используют чаще, поскольку мы читаем и пишем слева направо и, следовательно, такое направление разбора совпадает с привычным нам направлением письма.

### **2.2.1. Левосторонний нисходящий грамматический разбор («сверху–вниз»)**

Метод предполагает формирование последовательности правил, которая позволит из аксиомы вывести разбираемую строку языка.

Вначале в строку правил заносят аксиому. Затем на каждом шаге вместо самого левого нетерминала строки правил записывают правую часть определяющего этот нетерминал правила. Замены продолжают, пока слева не оказывается терминал или нетерминал, не разделяемый в рамках используемого множества или подмножества правил.

После этого самые левые символы разбираемой строки и строки правил сравнивают. Если указанные символы совпадают, то их считают распознанными. Распознанные символы удаляют из исходной строки и строки правил и продолжают выполнять подстановку правил.



В случае несовпадения символов фиксируют ошибку и возвращаются к ближайшему шагу замены левого нетерминала, на котором можно выбирать другое правило для подстановки.

Разбор завершается успешно, если все символы строки распознаны и удалены из разбираемой строки, а строка правил пустая.

Если все альтернативные замены правил проверены, но часть символов строки остались нераспознанными, то разбираемая строка не соответствует аксиоме, т. е. содержит ошибку. Аналогичная ситуация, если все символы строки распознаны, а в строке правил остаются символы.

Метод применим, если *грамматика не содержит правил с левосторонней рекурсией*. При наличии таких правил процедура разбора становится бесконечной. Правила должны подставляться, начиная с самых сложных, иначе цель разбора не будет достигнута.

**Пример 2.5.** Разобрать строку «-45».

Правила грамматики:

- а)  $\langle \text{Целое} \rangle ::= \langle \text{Знак} \rangle \langle \text{ЦелоеБезЗнака} \rangle | \langle \text{ЦелоеБезЗнака} \rangle$ ,
- б)  $\langle \text{ЦелоеБезЗнака} \rangle ::= \langle \text{Цифра} \rangle \langle \text{ЦелоеБезЗнака} \rangle | \langle \text{Цифра} \rangle$ ,
- в)  $\langle \text{Цифра} \rangle ::= 0|1|2|3|4|5|6|7|8|9$ ,
- г)  $\langle \text{Знак} \rangle ::= +|-$ .

Для уменьшения длины строки правил заменим имя нетерминала  $\langle \text{ЦелоеБезЗнака} \rangle$  сокращением  $\langle \text{ЦБЗ} \rangle$ . Процесс разбора представлен в табл. 2.1. При этом полужирным выделен заменяемый на следующем шаге нетерминал. Кроме того, альтернативные правила, записанные в одной форме Бекуса-Наура, пронумерованы, так, например, правило  $\langle \text{Целое} \rangle ::= \langle \text{Знак} \rangle \langle \text{ЦелоеБезЗнака} \rangle$  получило номер а1, а альтернативное ему правило  $\langle \text{Целое} \rangle ::= \langle \text{ЦелоеБезЗнака} \rangle$  – номер а2. Эти номера и указаны в таблице.

**Таблица 2.1** Пример разбора строки левосторонним нисходящим методом

№ шага	Распознано	Распознаваемая строка	Строка правил	Действие
1		—45	<Целое>	Подстановка правила а1
2		—45	<Знак><ЦБЗ>	Подстановка правила г1
		—45	+<ЦБЗ>	Ошибка, возврат к шагу 2
3		—45	<Знак><ЦБЗ>	Подстановка правила г2
		—45	-<ЦБЗ>	Символ распознан и удален
4	—	45	<ЦБЗ>	Подстановка правила б1
5	—	45	<Цифра><ЦБЗ>	Подстановка правила в1
	—	45	0 <ЦБЗ>	Ошибка, возврат к шагу 5
6	—	45	<Цифра><ЦБЗ>	Подстановка правила в2
	—	45	1 <ЦБЗ>	Ошибка, возврат к шагу 6
7	—	45	<Цифра><ЦБЗ>	Подстановка правила в3
	—	45	2 <ЦБЗ>	Ошибка, возврат к шагу 7
8	—	45	<Цифра><ЦБЗ>	Подстановка правила в4
	—	45	3 <ЦБЗ>	Ошибка, возврат к шагу 8
9	—	45	<Цифра><ЦБЗ>	Подстановка правила в5
	—	45	4 <ЦБЗ>	Символ распознан
10	—4	5	<ЦБЗ>	Подстановка правила б1
11	—4	5	<Цифра><ЦБЗ>	Подстановка правила в1
	—4	5	0 <ЦБЗ>	Ошибка, возврат к шагу 11
12	—4	5	<Цифра><ЦБЗ>	Подстановка правила в2
	—4	5	1 <ЦБЗ>	Ошибка, возврат к шагу 12

№ шага	Распознано	Распознаваемая строка	Строка правил	Действие
13	–4	5	<Цифра><ЦБЗ>	Подстановка правила в3
	–4	5	2 <ЦБЗ>	Ошибка, возврат к шагу 13
14	–4	5	<Цифра><ЦБЗ>	Подстановка правила в4
	–4	5	3 <ЦБЗ>	Ошибка, возврат к шагу 14
15	–4	5	<Цифра><ЦБЗ>	Подстановка правила в5
	–4	5	4 <ЦБЗ>	Ошибка, возврат к шагу 15
16	–4	5	<Цифра><ЦБЗ>	Подстановка правила в6
	–4	5	5 <ЦБЗ>	Символ распознан и удален
17	–45	∅	<ЦБЗ>	Подстановка правила б1
18..27	–45	∅	<Цифра><ЦБЗ>	Подстановки правил в1–в10
	–45	∅	0 ... 9 <ЦБЗ>	Ошибки, возвраты, возврат к шагу 17
28	–45	∅	<ЦБЗ>	Подстановка правила б2
29..38	–45	∅	<Цифра>	Подстановки правил в1–в10
	–45	∅	0 ... 9	Ошибки, возвраты, возврат к шагу 10
39	–4	5	<ЦБЗ>	Подстановка правила б2
40..44	–4	5	<Цифра>	Подстановки правил в1–в5
	–4	5	0 ... 4	Ошибки и возвраты
45	–4	5	<Цифра>	Подстановка правила в6
	–4	5	5	Символ распознан и удален
46	–45	∅	∅	Конец «Строка распознана»

При разборе многократно фиксировались ситуации, когда его продолжение становилось невозможным, так как распознаваемый терминальный символ отличался от левого терминального символа строки правил. Это было связано с неверным выбором правил в процессе разбора. В такой ситуации

#### [Оглавление](#)

приходилось возвращаться и выбирать альтернативные правила. В табл. 2.1 часть шагов, приводящих к возвратам в очевидных случаях сгруппированы.

Также встретилась ситуация, когда все альтернативные правила в ближайших точках подстановки правила были испробованы (см. шаг 38). В этом случае возврат был выполнен к предыдущей точке, в которой возможен выбор альтернативного правила (см. шаг 10).

*Недостаток метода* заключается в том, что программе необходимо хранить всю информацию о каждом его шаге разбора, чтобы обеспечить возможность возврата.

### **2.2.2. Левосторонний восходящий грамматический разбор (разбор «слева–направо»)**

Метод разбора применим, если *грамматика не содержит правил с правосторонней рекурсией*.

При грамматическом разборе восходящим методом сентенциальные формы, начиная с исходной строки, просматривают слева направо и последовательно заменяют подстроки, совпадающие с правой частью правил на левую часть того же правила. Процесс называют *сверткой*, а заменяемую часть сентенциальной формы – *основой*.

Разбор заканчивается, когда вместо исходной строки мы получаем аксиому. Если все варианты подстановки проверены, а получить аксиому не удалось, то строка содержит ошибку.

Как и в предыдущем случае, правила подстановки должны проверяться, начиная с самого сложного, иначе сложные правила никогда не будут применены, и разбор выполнить не получится. В общем случае при разборе возможны возвраты и попытки выбора другой основы, поскольку на очередном шаге было использовано неподходящее правило.

**Пример 2.6.** Выполнить грамматический разбор строки «–45».

Правила грамматики:

- а)  $\langle \text{Целое} \rangle ::= \langle \text{Знак} \rangle \langle \text{ЦелоеБезЗнака} \rangle | \langle \text{ЦелоеБезЗнака} \rangle$ ,
- б)  $\langle \text{ЦелоеБезЗнака} \rangle ::= \langle \text{ЦелоеБезЗнака} \rangle \langle \text{Цифра} \rangle | \langle \text{Цифра} \rangle$ ,
- в)  $\langle \text{Цифра} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$ ,
- г)  $\langle \text{Знак} \rangle ::= + | -$ .

Будем использовать сокращение  $\langle \text{ЦБЗ} \rangle$  для имени нетерминала  $\langle \text{ЦелоеБезЗнака} \rangle$ , как в предыдущем пункте. Полученная рассматриваемым методом последовательность сентенциальных форм представлена в табл. 2.2 (основа на каждом шаге выделена жирным шрифтом).

**Таблица 2.2** Пример разбора строки левосторонним восходящим методом

№ шага	Распознаваемая строка	Основа	Операция
1	–45	–	Свертка по правилу г2
2	$\langle \text{Знак} \rangle$ 45	$\langle \text{Знак} \rangle$	Нет правила для свертки. Форми-рование следующей основы
3	$\langle \text{Знак} \rangle$ 45	$\langle \text{Знак} \rangle$ 4	Нет правила для свертки. Форми-рование следующей основы
4	$\langle \text{Знак} \rangle$ 45	<b>4</b>	Свертка по правилу в5
5	$\langle \text{Знак} \rangle \langle \text{Цифра} \rangle$ 5	$\langle \text{Знак} \rangle$	Нет правила для свертки. Форми-рование следующей основы
6	$\langle \text{Знак} \rangle \langle \text{Цифра} \rangle$ 5	$\langle \text{Знак} \rangle \langle \text{Цифра} \rangle$	Нет правила для свертки. Форми-рование следующей основы
7	$\langle \text{Знак} \rangle \langle \text{Цифра} \rangle$ 5	$\langle \text{Цифра} \rangle$	Свертка по правилу б2
8	$\langle \text{Знак} \rangle \langle \text{ЦБЗ} \rangle$ 5	$\langle \text{Знак} \rangle$	Нет правила для свертки. Форми-рование следующей основы
9	$\langle \text{Знак} \rangle \langle \text{ЦБЗ} \rangle$ 5	$\langle \text{Знак} \rangle \langle \text{ЦБЗ} \rangle$	Свертка по правилу а1
10	$\langle \text{Целое} \rangle$ 5	<b>Аксиома! Тупик!</b>	<b>Возврат к шагу 9. Форми-рование следующей основы</b>
11	$\langle \text{Знак} \rangle \langle \text{ЦБЗ} \rangle$ 5	$\langle \text{Цбз} \rangle$	Свертка по правилу а2
12	$\langle \text{Знак} \rangle \langle \text{Целое} \rangle$ 5	<b>Аксиома! Тупик!</b>	<b>Возврат к шагу 11. Форми-рование следующей основы</b>
13	$\langle \text{Знак} \rangle \langle \text{ЦБЗ} \rangle$ <b>5</b>	$\langle \text{Знак} \rangle \langle \text{ЦБЗ} \rangle$ 5	Нет правила для свертки. Формирование следующей основы
14	$\langle \text{Знак} \rangle \langle \text{ЦБЗ} \rangle$ <b>5</b>	$\langle \text{ЦБЗ} \rangle$ 5	Нет правила для свертки. Формирование следующей

			основы
15	<Знак> <ЦБЗ> 5	5	Свертка по правилу гб
16	<Знак> <ЦБЗ><Цифра>	<Знак> <ЦБЗ>	Свертка по правилу а1
17	<Целое> <Цифра>	Аксиома! Тупик!	Возврат к шагу 16. Формирование следующей основы
18	<Знак> <ЦБЗ><Цифра>	<ЦБЗ>	Свертка по правилу а2
19	<Знак> <Целое> <Цифра>	Аксиома! Тупик!	Возврат к шагу 18. Формирование следующей основы
20	<Знак> <ЦБЗ> <Цифра>	<ЦБЗ> <Цифра>	Свертка по правилу б1
21	<Знак> <ЦБЗ>	<Знак> <ЦБЗ>	Свертка по правилу а1
22	<Целое>	Конец	-

Недостатком метода, как и в предыдущем случае, является необходимость предусмотреть возможность возвратов на предыдущие шаги, где есть альтернатива выбора основы. Это предполагает хранение всей информации о каждом шаге процесса.

Таким образом, реализация обоих рассмотренных методов сложна и требует достаточно много памяти. Поэтому для теории формальных языков большое значение имеет классификация грамматик по виду используемых правил, что непосредственно связано со сложностью грамматического разбора предложений соответствующих языков.

### 2.3. Расширенная классификация грамматик Хомского

В зависимости от вида правил различают грамматики следующих типов:

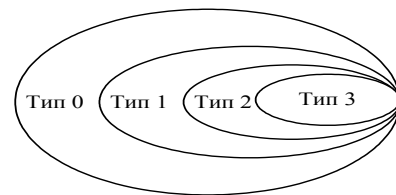
**тип 0** – фразовой структуры или «без ограничений», т. е. использующие правила вида  $\alpha \rightarrow \beta$ , где  $\alpha \in V^+$ ;  $\beta \in V^*$ ;  $V = V_T \cup V_N$ . В таких грамматиках допустимо наличие любых правил продукции, что свойственно грамматикам естественных языков;

**тип 1** – контекстно-зависимые, использующие правила вида  $\alpha X \beta \rightarrow \alpha x \beta$ , где  $X \in V_N$ ;  $x \in V_T$ ;  $\alpha, \beta \in V^*$ ;  $V = V_T \cup V_N$ . Причем  $\alpha, \beta$  одновременно не являются пустыми, а значит возможность подстановки  $x$  вместо символа  $X$  определяется присутствием хотя бы одной из подстрок  $\alpha$  и  $\beta$ , т. е. *контекста*, откуда и происходит название типа грамматик;

**тип 2** – контекстно-свободные (КС-грамматики), использующие правила вида  $A \rightarrow \beta$ , где  $A \in V_N$ ;  $\beta \in V^*$ . Поскольку в левой части правила стоит единственный нетерминал, т. е. отсутствуют подстроки слева и справа, подстановки не зависят от контекста;

**тип 3** – регулярные или автоматные, использующие правила вида  $A \rightarrow \alpha$ ,  $A \rightarrow \alpha B$  или  $A \rightarrow B\alpha$ , где  $A, B \in V_N$ ;  $\alpha \in V_T$ , т. е. правая часть правила содержит не более одного терминала и может содержать нетерминал. Эти грамматики описывают правила синтаксиса самых простых языков.

Классификация построена по правилам иерархии, т. е. грамматики третьего типа являются частным случаем грамматик второго типа и т. д. (рис. 2.3).



Тип грамматик определяется по ее самым сложным правилам. Так, самые сложные правила грамматик третьего типа могут содержать лево- или правостороннюю (но не вложенную!) рекурсию.

**Рис. 2.3.** Классификация типов грамматик

Граматики второго типа отличаются от грамматик третьего типа по наличию в правилах явной или косвенной *вложенной* рекурсии, т. е. правил вида  $A \rightarrow \alpha A \beta$ , где  $\alpha$  и  $\beta \in V^+$ , т. е. подстроки  $\alpha$  и  $\beta$  не являются пустыми.

Определяющей характеристикой грамматик первого типа, отличающей их от грамматик второго типа, является наличие в правилах продукции *контекстной зависимости*, т. е. правил типа  $\alpha X \beta \rightarrow \alpha x \beta$ , где  $\alpha, \beta \in V^*$ , в которых только одна из подстрок  $\alpha$  или  $\beta$  может быть пустой.

**Пример 2.7.** Язык скобок, описываемый правилами: а)  $S \rightarrow (S)$ , б)  $S \rightarrow SS$ , в)  $S \rightarrow e$ .

Поскольку слева стоит единственный нетерминал, грамматика не относится к грамматикам первого типа. Однако в ней присутствует правило с вложенной рекурсией (см. правило а). Следовательно, грамматика принадлежит к классу грамматик второго типа.

**Леммы о разрастании.** Существует простой метод проверки принадлежности языка к регулярным, основанный на *лемме о разрастании для регулярных грамматик*: в строке регулярного языка всегда можно найти непустую подстроку, повторение которой произвольное количество раз порождает новые строки того же языка.

**Пример 2.8.** Язык описания целых чисел. Из строки «+23» можно построить строки: 22222, 23232323 и т. д. (того же языка).

*Лемма о разрастании КС языков* формулируется следующим образом: в строке, принадлежащей КС языку, всегда можно найти две подстроки с ненулевой суммарной длиной, одновременное повторение которых произвольное количество раз порождает новую строку того же языка.

**Пример 2.9.** Язык скобок. Из строки «()» можно построить строку: ((( ))) и другие строки того же языка.

**Распознаватели.** Соответствие строк языку проверяется специальной программой – *распознавателем*, который может использоваться для определения языка так же, как и грамматики. Чем шире класс распознаваемых грамматик, тем сложнее класс соответствующих распознавателей. Доказано, что грамматики третьего типа распознаются *конечными автоматами*, грамматики второго типа – *автоматами с магазинной памятью*, грамматики первого типа – *линейными ограниченными автоматами*, грамматики нулевого типа – *машинами Тьюринга*.



### ***Контрольные вопросы и задания***

2.1. Сформулируйте, что такое алфавит языка.

[Ответ.](#)

2.2. Дайте определение формального языка. Почему формальные языки обычно не задают перечислением допустимых предложений?

[Ответ.](#)

2.3. Дайте определение формальной грамматики.

[Ответ.](#)

2.4. Что такое «Форма Бэкуса – Наура»? Для чего она используется? Чем отличается от «Расширенной формы Бэкуса– Наура»?

[Ответ.](#)

2.5. В чем заключается цель грамматического разбора предложений языка.

[Ответ.](#)

2.6. В чем заключается левосторонний нисходящий грамматический разбор? Назовите, в каких случаях он не применим, и укажите его основной недостаток.

[Ответ.](#)

2.7. В чем заключается левосторонний восходящий грамматический разбор? Назовите, в каких случаях он не применим, и назовите его основной недостаток.

[Ответ.](#)

2.8. Назовите основные типы грамматик по Хомскому. Какие грамматики используют в языках программирования?

[Ответ.](#)

2.9. Разработайте грамматику, позволяющую описывать строки, содержащие запись вещественных чисел в форме с фиксированной точкой с обязательным присутствием знака, например:

– 0.1, + 25.356, + 0.0

[Оглавление](#)

К какому типу по Хомскому относится данная грамматика? Почему?

[Решение.](#)

2.10. Выполните разбор строки, содержащей запись числа  $-4.5$ , по левостороннему нисходящему методу, используя правила, сформулированные в задании 2.9.

[Решение.](#)

2.11. Выполните разбор строки, содержащей запись числа  $-4.5$ , по левостороннему восходящему методу, используя соответствующим образом скорректированные правила, сформулированные в задании 2.9.

[Решение.](#)

### 3. Распознавание регулярных грамматик

Правила, удовлетворяющие ограничениям регулярных грамматик, при-  
сущи относительно простым конструкциям, таким как идентификаторы, ли-  
тералы, описания переменных и т. п. Для их распознавания достаточно по-  
строить и реализовать конечный автомат.

#### 3.1. Конечный автомат и его программная реализация

*Конечный автомат* – это математическая модель, свойства и поведение  
которой полностью определяются «пятеркой»:

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

где  $Q$  – конечное множество состояний;

$\Sigma$  – конечное множество входных символов;

$\delta(q_i, c_k)$  – функция переходов, где  $q_i \in Q$  – текущее состояние;  $c_k \in \Sigma$  – оче-  
редной символ;

$q_0 \in Q$  – начальное состояние;

$F \subset Q$  – подмножество допускающих состояний.

Конечный автомат – одна из базовых моделей, используемых при проек-  
тировании программного и аппаратного обеспечения средств вычислитель-  
ной техники.

**Пример 3.1.** Математически автомат «Чет–нечет» описывается следую-  
щим образом:

$$Q = \{\text{Чет}, \text{Нечет}\};$$

$$\Sigma = \{0, 1\};$$

$$\delta(\text{Чет}, 0) = \text{Чет}, \delta(\text{Нечет}, 0) = \text{Нечет}, \delta(\text{Чет}, 1) = \text{Нечет},$$

$$\delta(\text{Нечет}, 1) = \text{Чет};$$

$$q_0 = \text{Чет};$$

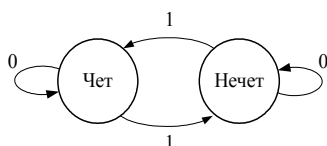
$$F = \{\text{Чет}\}$$

Входная строка «10110» приведет такой автомат в состояние «Нечет», т. е. будет отвергнута, а строка «110011» – в состояние «Чет», т. е. будет принята.

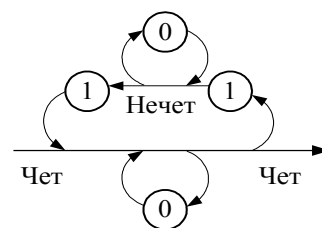
Для представления функции переходов конечного автомата помимо аналитической формы могут использоваться: таблица (табл. 3.1), граф переходов состояний (рис. 3.1) и синтаксическая диаграмма (рис. 3.2).

**Таблица 3.1** Функция переходов

Состояние $q$	Входной символ $s$	
	0	1
Чет	Чет	Нечет
Нечет	Нечет	Чет



**Рис. 3.1.** Граф переходов



**Рис. 3.2.** Синтаксическая диаграмма

Разработчик автомата обычно использует его представление графом или синтаксической диаграммой. При реализации в виде программы автомат задают таблицей переходов.

Для идентификации завершения строки обычно применяют специальный символ или множество символов, которые включают во входной алфавит и учитывают в таблице. Кроме этого, в таблице предусматривают реакцию автомата на символы, не включенные в алфавит и не являющиеся завершающими. Появление таких символов в строке означает, что предложение языка содержит ошибку (табл. 3.2).

**Таблица 3.2** Дополненная таблица конечного автомата

Состояние $q$	Входной символ $s$			
	0	1	Символы завершения	Другие символы

[Оглавление](#)

Чет	Чет	Нечет	Конец	Ошибка
Нечет	Нечет	Чет	Ошибка	Ошибка

Пусть  $S$  – строка на входе автомата,  $Ind$  – номер очередного символа,  $q$  – текущее состояние автомата,  $Table$  – таблица, учитывающая символы завершения и другие символы.

Тогда алгоритм работы программы, реализующей автомат, можно описать на псевдоязыке следующим образом:

**$Ind := 1$**

**$q := q_0$**

**Цикл-пока  $q \neq \text{«Ошибка»}$  и  $q \neq \text{«Конец»}$**

**$q = Table[q, S[Ind]]$**

**$Ind := Ind + 1$**

**Все-цикл**

**Если  $q = \text{«Конец»}$**

**то «Строка принята»**

**иначе «Строка отвергнута»**

**Все-если**

В приведенном описании запись  $q = Table[q, S[Ind]]$  означает, что следующее состояние автомата  $q$  определяется текущим состоянием  $q$ , задающим строку таблицы переходов, и текущим входным символом строки  $S[Ind]$ , указывающим столбец той же таблицы.

### **3.2. Построение лексических анализаторов с использованием конечных автоматов**

Цель лексического анализа – разбиение текста на отдельные «слова» (лексемы) и проверка правильности их написания. Лексемами языка могут

являться служебные слова/символы или слова, обозначающие базовые понятия языка, например, идентификаторы.

Для разделения лексем в языке могут использоваться специальные разделители. Чаще всего такими разделителями являются пробелы. Однако разделителями могут служить и служебные слова или символы, например, запятые.

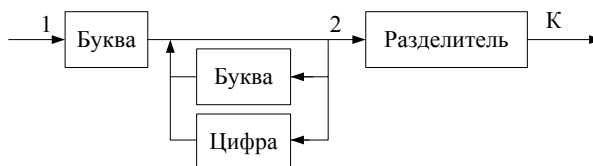
Служебные слова и служебные символы выявляют сравнением с учетом всех возможных вариантов написаний. Причем, если язык не различает строчные и прописные буквы (например, Паскаль или Ассемблер), то перед обработкой все символы букв заменяют прописными буквами, поскольку это сразу сокращает количество сравнений. Если язык чувствителен к регистру (например, С или С++), то в нем обычно определено единственное написание служебных слов.

Для распознавания лексем второго типа – базовых понятий языка – при построении лексических анализаторов используют конечные автоматы. Поскольку подобные лексемы, как правило, можно описать в рамках самых простых (регулярных) грамматик.

Алфавит автомата лексического анализатора – все множество однобайтовых (*ANSI*) или двухбайтовых (*Unicode*) символов. При записи правил обычно используют обобщающие нетерминалы вида «Буквы», «Цифры». В процессе распознавания может формироваться описываемый объект, например, число или идентификатор.

**Пример 3.2.** Разработать функцию распознавания идентификатора в операторе. Функция должна проверять, является ли следующая лексема программы идентификатором, и если является, то строить идентификатор и возвращать в качестве результата true.

Будем строить лексический анализатор с использованием конечного автомата, работающего по синтаксической диаграмме на рис. 3.3.



**Рис. 3.3.** Синтаксическая диаграмма лексемы «Идентификатор»:

1 – исходное состояние; 2 – распознан хотя бы один символ; K – конец (распознан идентификатор)

Таблица автомата (табл. 3.3), построенная по синтаксической диаграмме, учитывает возможное присутствие неразрешенных символов.

**Таблица 3.3** Таблица автомата, распознающего идентификаторы

<i>q</i>	<i>c</i>			
	Буква	Цифра	Разделитель	Другой символ
1	2	Ошибка	Ошибка	Ошибка
2	2	2	Конец	Ошибка

Для удаления из анализируемой строки пробелов в процессе анализа будем использовать процедуру:

```

Procedure Probef (Var St:shortstring);
Begin
  While (St<>'') and (St[1]=' ') do Delete(St,1,1);
End;

```

Текст, реализующей автомат программы, выглядит следующим образом:

```

Function Id(
  Var St:shortstring; // исходная строка
  Razd:setofChar      // множество разделителей
):boolean;
Var S:shortString;
  State,      // состояние
  Ind,        // номер символа в исходной строке
  Colon:byte; // столбец таблицы
Const TableId:array[1..2,1..4] of Byte=
  ((2,0,0,0),(2,2,10,0)); // таблица автомата
Begin
  Probef (St); // процедура удаления разделяющих пробелов
  State:=1;   // исходное состояние
  S:='';

```

```

while (State<>0) and (State<>10) and
      (Length(St)<>0) do
begin
  if St[1] in ['A'..'Z','a'..'z'] then Colon:=1
  else if St[1] in ['0'..'9'] then Colon:=2
      else if (St[1] in Razd) then Colon:=3
          else Colon:=4;
  State:=TableId[State,Colon];
  if (State<>0) and (State<>10) then
  begin
    S:=S+St[1];
    Delete(St,1,1);
  end;
end;
if length(st)=0 then State:=10;
if (State=10) and (S<>'') then
  Begin
    Result:=true; WriteLn('Identify=',S);
  End
else
  if (State=0) then
    Begin
      Result:=false;
      WriteLn('Wrong symbol *',St[1],'*');
    End
  else
    Begin
      Result:=false;
      WriteLn('Identifier waits...', St);
    End;
End;
End;

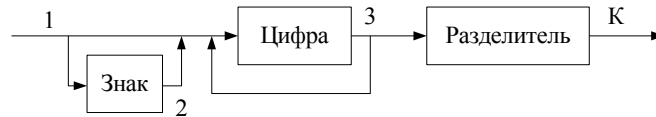
```

*Примечание.* В процедуре состояние ошибки кодируется числом 0, а состояние завершения разбора (конец) – числом 10.



**Пример 3.3.** Разработать функцию распознавания целых десятичных литералов. Функция должна проверять правильность записи значения и строить распознанное значение в процессе ее распознавания.

Грамматика записи целых чисел задается синтаксической диаграммой (рис. 3.4).



**Рис. 3.4.** Синтаксическая диаграмма лексемы «Целое число»:

1 – исходное состояние; 2 – распознан знак; 3 – распознана хотя бы одна цифра; К – конец (распознано целое число)

В табл. 3.4 цифрами обозначены разрешенные переходы синтаксической диаграммы. Запрещенные переходы (ошибки) обозначены символом «Е».

**Таблица 3.4** Таблица переходов распознающего автомата

Состояние $q$	Входной символ $c$			
	Знак	Цифра	Разделитель	Другие символы
1	2, А1	3, А2	Е, Д1	Е, Д4
2	Е, Д2	3, А2	Е, Д3	Е, Д4
3	К, А3	3, А2	К, А3	Е, Д4

Кроме переходов в таблице распознавателя указаны подпрограммы обработки, которые должны быть выполнены в исходном состоянии и при переходах:

**А0:** Инициализация: Целое := 0; Знак\_числа := «+»

**А1:** Знак\_числа := Знак

**А2:** Целое := Целое\*10 + Цифра

**А3:** Если Знак\_числа = «-» то Целое := -Целое Все-если

Каждая подпрограмма вызывается при соответствующем переходе автомата, и при этом формируются значения целочисленных литералов.

Кроме этого, в табл. 3.4 показаны обозначения диагностических сообщений, которые должны выводиться при выявлении той или иной ошибки:

**Д1:** «Строка не является десятичным числом»;

**Д2:** «Два знака рядом»;

**Д3:** «В строке отсутствуют цифры»;

**Д4:** «В строке встречаются недопустимые символы».

В программе:  $S$  – строка на входе автомата;  $Ind$  – номер очередного символа;  $q$  – текущее состояние автомата;  $Table$  – таблица, учитывающая символы завершения и другие символы.

Тогда алгоритм сканера-распознавателя можно представить следующим образом.

**$Ind := 1$**

**$q := 1$**

**Выполнить  $A0$**

**Цикл-пока  $q \neq \langle E \rangle$  и  $q \neq \langle K \rangle$**

**Если  $S[Ind] = \langle + \rangle$  или  $S[Ind] = \langle - \rangle$ ,**

**то  $j := 1$**

**иначе Если  $S[Ind] \geq \langle 0 \rangle$  и  $S[Ind] \leq \langle 9 \rangle$ ,**

**то  $j := 2$ ,**

**иначе  $j := 3$**

**Все-если**

**Все-если**

**Выполнить  $A_i := Table[q, j]. A0$**

**$q := Table[q, j]$**

**$Ind := Ind + 1$**

**Все-цикл**

**Если  $q = \langle K \rangle$**

**то Выполнить  $A3$**

**Вывести сообщение «Это число»**

## иначе Вывести сообщение Ді

### Все-если

**Пример 3.4.** Разработать подпрограмму-сканер, осуществляющую лексический анализ идентификаторов и операций для разбора арифметических выражений, запись которых содержит идентификаторы, знаки операций и скобки. Сканер должен строить строку токенов следующего вида:

$I$  – идентификатор переменной или поименованной константы;

$+$ ,  $-$ ,  $*$ ,  $/$ ,  $($ ,  $)$  – знаки операций.

Так для выражения  $aa2+j1*(a+one)$  по правилам должна быть построена строка токенов  $I+I*(I+I)$ .

При выполнении синтаксического анализа стековым методом (см. раздел 4.3) строка будет разделена на фрагменты из одного–двух символов, завершающиеся символами операций  $I+ I* ( I+ I)$ .

Анализ строк токенов показал, что существует единственный случай, когда в строке идут два символа операций подряд – символ арифметической операции и открывающаяся скобка. Соответственно в этом случае для дальнейшего анализа выбирается один символ – открывающаяся скобка.

Для исключения особого случая будем сразу при лексическом анализе включать перед открывающейся скобкой вспомогательный токен – пустой операнд  $@$ . Тогда строка токенов будет выглядеть так:  $I+I*@(I+I)$ , и ее разбиение будет всегда выполняться по два символа во фрагменте:

$I+ I* @( I+ I)$ .

Анализ допустимых предложений языка показывает, что при сканировании должны быть использованы следующие разделители:

```
Const Razd:setofChar=[' ', '+', '-', '*', '/', '(', ')'];
```

При разработке сканера использованы две вспомогательные подпрограммы:

подпрограмма удаления пробелов

```
Procedure Probel(Var St:shortstring);
```

### [Оглавление](#)

и функция распознавателя идентификаторов

```
Function Id(Var St:shortstring;Razd:setofChar):Boolean;
```

Тела обеих подпрограмм рассмотрены в примере 3.2.

Реализацию сканера выполним в виде функции, которая возвращает true, если при распознавании идентификаторов ошибок не обнаружено, или false – иначе.

Текст сканера для арифметических выражений будет выглядеть следующим образом:

```
Function Scan(
    St:shortstring; {исходная строка}
    Razd:setofChar; {множество разделителей}
    Var StS:shortstring {строка токенов}
    ):boolean;
Var R:boolean;
Begin
    R:=true;
    StS:='';
    Probel(St);
    while (length(St)<>0) and R do
    begin
        if not (St[1] in Razd) then
        begin
            R:=Id(St,Razd);
            if R then StS:=StS+'I';
        end
        else
        begin
            if St[1]='(' then StS:= StS+'@';
            StS:=StS+St[1];
```

[Оглавление](#)

```

Delete (St, 1, 1);
end;
Probel (St);
end;
WriteLn ('After Scan:', StS);
Result:=R;
End;

```

Пример результата работы подпрограммы:

```

Input Strings:
Qr34+ghj*(hj+yi)
Identify=Qr34
Identify=ghj
Identify=hj
Identify=yi
After Scan:I+I*@(I+I)

```

Рассмотрим более сложный вариант сканирующей программы.

**Пример 3.5.** Разработать сканер, осуществляющий лексический анализ идентификаторов и служебных слов для операторов условной передачи управления и присваивания в синтаксисе языка Паскаль. Для упрощения примера в качестве логического выражения можно использовать только простые сравнения, а служебные слова – вводить только строчными символами.

На выходе сканер должен формировать строку токенов длиной два символа каждый:

```

V_ – идентификатор (операнд);
@@ – пустой операнд, используемый при отсутствии операнда перед
      служебным словом или операцией;
if – служебное слово if;
th – служебное слово then;
el – служебное слово else;

```

$>_, <_, =_, <>, >=, <=$  – операции сравнения;

$+_, -_, *__, /_, (, )$  – операторы выражения;

$:=$  – служебное слово «присвоить»;

$;$  – токен, обозначающий конец оператора.

Таким образом, если на вход сканера поступает строка вида

```
if aaaa>vvvv then j:=hhhh
```

```
else
```

```
    if h then ffff:=hhh+(ppp+yyy);
```

то на выходе мы должны получить строку токенов

@@	if	V_	>_	V_	th	V_	:=	V_	el
----	----	----	----	----	----	----	----	----	----

@@	if	V_	th	V_	:=	V_	*_	@@	(	V_	+	V_	)	;	_
----	----	----	----	----	----	----	----	----	---	----	---	----	---	---	---

Перед токенами *if* и открывающейся скобкой вставлены пустые операнды @@, позволяющие при синтаксическом анализе считывать строго по два токена <операнд-оператор> (границы пар токенов выделены жирной линией)

Анализ допустимых предложений языка показывает, что при сканировании должны быть использованы следующие разделители:

```
Const Razd:setofChar=
```

```
[' ', '+', '-', '*', '/', '(', ')', ':', ';', '<', '>', '='];
```

Вспомогательные функции используем те же, что и в примере 3.4.:

```
Procedure Probel(Var St:shortstring); // удаление пробелов
```

```
Function Id(Var St:shortstring;Razd:setofChar):boolean;
```

```
//распознаватель идентификаторов
```

Текст функции-сканера:

```
Function Scanner(
```

```
    St:shortstring;           // исходная строка
```

```
    Razd:setofChar;          // множество разделителей
```

```
    Var StS:shortstring      // строка токенов
```

[Оглавление](#)

```

        ):boolean;
Const SlSl:array[1..3] of shortString=
        ('if','then','else'); // служебные слова
Var R:boolean; i,k:byte; St1:shortstring;
Begin
    R:=true;
    StS:='';
    Probel(St); // удаляем пробелы, если они есть
    while (length(St)<>0) and R do
        // пока оператор не кончился и не обнаружили ошибку
    begin
        if not(St[1] in Razd) then // если не разделитель
        begin
            k:=Pos(' ',St);
            if k<>0 then
            begin
                St1:=Copy(St,1,k-1);
                i:=1;
                while (St1<>SlSl[i]) and (i<4) do i:=i+1;
                if i<>4 then // служебное слово
                begin
                    WriteLn('Slugeb slovo ',St1);
                    Delete(St,1,k-1);
                    if St1[1]='i' then
                        StS:=StS+'@@';
                    StS:=StS+Copy(St1,1,2);
                    R:=true;
                end
            else // идентификатор?
            begin
                R:=Id(St,Razd); // анализатор идентификаторов
                if R then StS:=StS+'V ';
            end;
        end
    else
    begin
        R:=Id(St,Razd); // анализатор идентификаторов
        if R then StS:=StS+'V ';
    end;
end
end

```

```

        end
    end
    else // если найден разделитель
        begin
            if St[1]='(' then Sts:=StS+'@@';
            Sts:=StS+St[1];
            Delete(St,1,1);
            if St[1] in ['=', '>'] then
                begin
                    Sts:=StS+St[1];
                    Delete(St,1,1);
                end
            else Sts:=StS+' ';
            WriteLn('Slugeb simbol ',
                    Copy(StS,length(StS)-1,2));
        end;
        Probel(St);
    end;
    WriteLn('After Scan:',StS);
    Result:=R;
End;
```

В результате выполнения функции-сканера исходная строка будет преобразована в строку токенов, пригодную для синтаксического анализа.

### ***3.3. Построение синтаксических анализаторов с использованием конечных автоматов***

Синтаксический анализ – распознавание конструкций языка в строке токенов. Важным результатом, помимо распознавания заданной конструкции, является информация об ошибках в выражениях, операторах и описаниях программы.

Способ построения синтаксического анализатора определяется типом грамматики языка:

- для регулярных грамматик используют конечные автоматы;

[Оглавление](#)



- для КС-грамматик – автоматы с магазинной памятью.

Аналогично лексическим анализаторам на конечных автоматах синтаксические анализаторы языков с регулярными грамматиками строят по синтаксическим диаграммам.

**Пример 3.6.** Разработать синтаксический анализатор списка объявлений целых скаляров, массивов и функций (для упрощения без параметров).

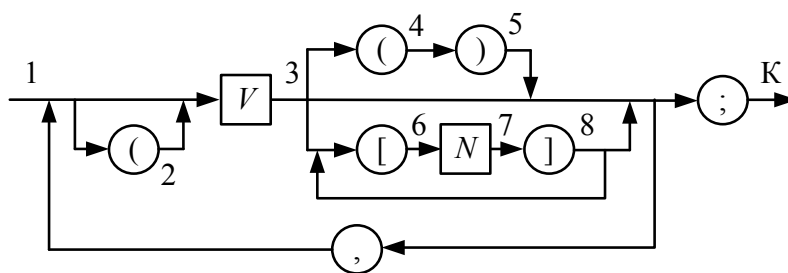
Например, на вход компилятора может поступить исходная строка вида:

```
int xaf, y22[5], zrr[2][4], re[N], fun(), *g;
```

Во время лексического анализа в строке будут выделены лексемы, проверены идентификаторы, а сама строка будет преобразована в строку токенов вида `int V, V[N], V[N][N], V[N], V(), *V;` где  $V$  – идентификатор;  $N$  – целочисленная константа; `[ ] ( ) , ; * – служебные символы.`

Задача синтаксического анализатора проверить правильность задания конструкции по строке токенов, т. е. определить допустимость исходной строки. Для этого построим конечный автомат.

Функции переходов конечного автомата определяются синтаксической диаграммой (рис. 3.5).



**Рис. 3.5.** Синтаксическая диаграмма конструкции «Объявление»:

1 – исходное состояние; 2 – распознан символ «\*»; 3 – распознан идентификатор; 4 – распознана открывающая скобка; 5 – распознано имя функции; 6 – распознана открывающая квадратная скобка; 7 – распознано целое число; 8 – распознана размерность массива; К – конец (распознана конструкция «Объявление»)

По диаграмме построим таблицу автомата (табл. 3.5), где символом «E» обозначено состояние ошибки.

**Таблица 3.5** Таблица переходов распознающего автомата

Состояние $q$	Входной символ $c$									
	$V$	$N$	$*$	(	)	[	]	,	;	Другие символы
1	3	$E$	2	$E$	$E$	$E$	$E$	$E$	$E$	$E$
2	3	$E$	$E$	$E$	$E$	$E$	$E$	$E$	$E$	$E$
3	$E$	$E$	$E$	4	$E$	6	$E$	1	K	$E$
4	$E$	$E$	$E$	$E$	5	$E$	$E$	$E$	$E$	$E$
5	$E$	$E$	$E$	$E$	$E$	$E$	$E$	1	K	$E$
6	$E$	7	$E$	$E$	$E$	$E$	$E$	$E$	$E$	$E$
7	$E$	$E$	$E$	$E$	$E$	$E$	8	$E$	$E$	$E$
8	$E$	$E$	$E$	$E$	$E$	6	$E$	1	K	$E$

Алгоритм распознавателя:

**$Ind := 1$**

**$q := 1$**

**Цикл-пока  $q \neq \langle E \rangle$  и  $q \neq \langle K \rangle$**

**$q := Table[q, Pos(S[Ind], \langle VN^*()[]; \rangle)]$**

**$Ind := Ind + 1$**

**Все-цикл**

**Если  $q = \langle K \rangle$**

**то Вывести сообщение «Строка корректна»**

**иначе Вывести сообщение «В строке обнаружена ошибка»**

**Все-если**

В описании алгоритма функция  $Pos(S[Ind], \langle VN^*()[]; \rangle)$  определяет номер позиции текущего токена исходной строки в последовательности токенов. В результате получаем номер столбца таблицы (последовательность токенов совпадает с заголовками столбцов таблицы переходов).

***Контрольные вопросы и задания***

3.1. Определите конечный автомат. В чем особенность его использования при грамматическом разборе?

[Ответ.](#)

3.2. Какие способы описания конечных автоматов существуют? Какой способ используется при программной реализации и почему?

[Ответ.](#)

3.3. Как построить конечный автомат для лексического анализа? В чем заключается его особенность?

[Ответ.](#)

3.4. Как построить конечный автомат для синтаксического анализа? Что подается на вход синтаксического анализатора?

[Ответ.](#)

3.5. Постройте синтаксическую диаграмму и таблицу конечного автомата. На вход автомата поступает последовательность непустых строк, содержащих группы букв и цифр, при этом цифры есть обязательно. Каждая строка завершается символом «#». Автомат должен распознавать и допускать строки, в которых встречается только четное количество цифр в группах, например, допустимы строки:

**а) fhgfhfj56787979tuytuuyu45cvbcv#**

**б) 56плорл354356#**

**в) 6789#**

[Решение.](#)

#### 4. Распознавание контекстно-свободных грамматик

Распознавание КС-грамматик выполняется автоматом с магазинной памятью. Однако построение такого автомата уже не самая простая задача, поэтому среди контекстно-свободных языков были выявлены две подгруппы языков, для которых разбор левосторонним восходящим или левосторонним нисходящим методом выполняется без возвратов. Это  $LR(k)$  и  $LL(k)$ -грамматики соответственно. Для разбора предложений языков с грамматиками указанных типов используют стековый метод или метод рекурсивного спуска.

##### 4.1. Автомат с магазинной памятью

Распознавание КС-грамматик выполняется автоматом с магазинной памятью. Математическая модель *Автомат с магазинной памятью* определяется «семеркой»:

$$P_M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F),$$

где  $Q$  – конечное множество состояний автомата;

$\Sigma$  – конечный входной алфавит;

$\Gamma$  – конечное множество магазинных символов;

$\delta(q_i, c_k)$  – функция переходов ( $q_i \in Q, c_k \in \Sigma, z_j \in \Gamma$ );

$q_0 \in Q$  – начальное состояние автомата;

$z_0 \in \Gamma$  – символ, находящийся в магазине в начальный момент,

$F \subset Q$  – множество заключительных (допускающих) состояний.

**Пример 4.1.** Разработать синтаксический анализатор арифметических выражений.

Как уже указывалось ранее, синтаксический анализ подразумевает, что арифметическое выражение уже было обработано сканером, и в результате

лексического анализа была сформирована строка токенов. При анализе арифметических выражений обычно используют токены:

<Идентификатор> или <Ид> – операнд – идентификатор переменной, константы или литерал;

+, −, \*, /, (, ) – операции.

Так, для выражения  $A + (c - 5) * k - S / (A + X)$  строка токенов должна выглядеть так: <Ид>+(<Ид>-<Ид>)\*<Ид>-<Ид>/(<Ид>+<Ид>).

Граматику арифметических выражений, учитывающую приоритеты арифметических операций, опишем в БНФ и синтаксическими диаграммами.

Поскольку операции сложения и вычитания имеют низший приоритет, выражение представляем, как сумму (разность) выражения и слагаемого (терма). Каждый терм – произведение (результат деления) терма на множитель. В качестве множителей могут служить операнды (идентификаторы) или более приоритетное выражение в скобках.

Добавляем аксиому <ПростоеВыражение> и получаем:

**<ПростоеВыражение> ::= ► <Выражение> ◄**

**<Выражение> ::= <Выражение>+<Терм> |**

**<Выражение>-<Терм> |**

**<Терм>**

**<Терм> ::= <Терм> \* <Множитель> |**

**<Терм> / <Множитель> |**

**<Множитель>**

**<Множитель> ::= (<Выражение>) |**

**<Идентификатор>**

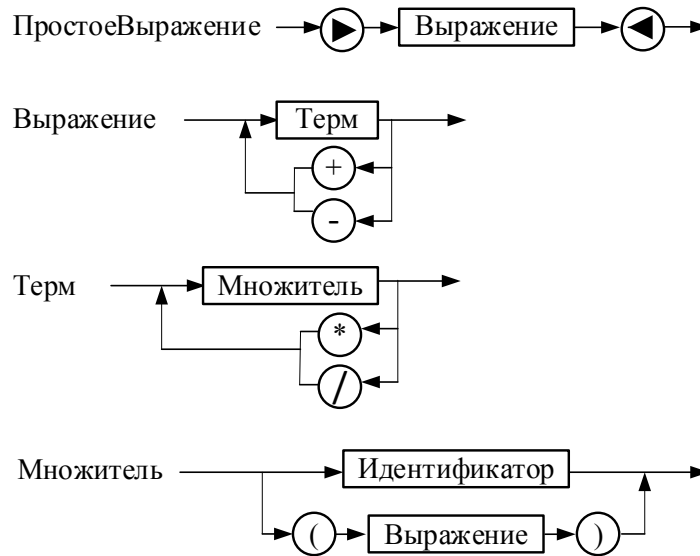
где ► – начало выражения;

◄ – конец выражения.

Правила, определяющие структуру нетерминалов «Выражение» и «Терм», содержат левостороннюю рекурсию, правило построения нетерминала «Множитель» включает неявное рекурсивное вложение: «Выражение» в

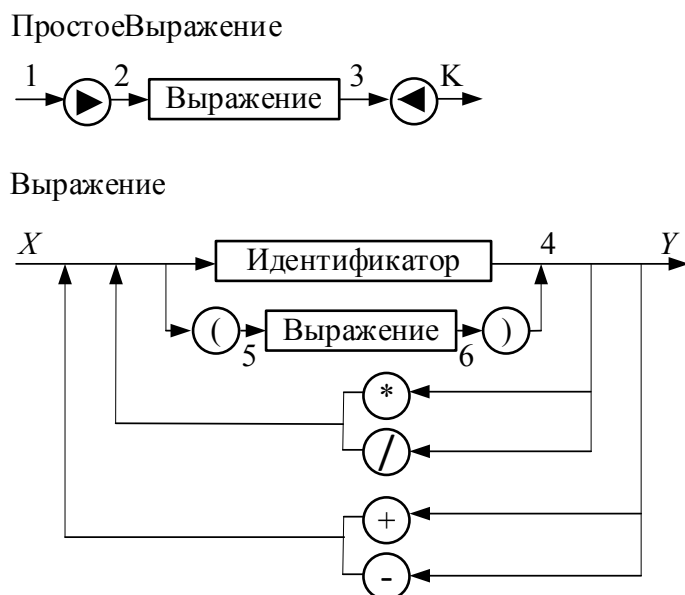
конечном счете определяется через «Выражение». Следовательно, это грамматика второго типа по Хомскому.

Синтаксические диаграммы правил представлены на рис. 4.1.



**Рис. 4.1.** Синтаксические диаграммы грамматики выражений

Теперь подставим диаграммы одна в другую, исключая промежуточные конструкции «Терм» и «Множитель». В результате получим полную синтаксическую диаграмму (рис. 4.2), которая состоит из основной и рекурсивной частей. Наличие двух частей связано с имеющимся рекурсивным вложением в правилах продукции грамматики, описывающей синтаксис арифметического выражения. По диаграмме построим таблицу автомата, которая также будет состоять из двух частей: основной и рекурсивной (табл. 4.1).



**Рис. 4.2.** Объединенная синтаксическая диаграмма конструкции «Выражение»:

1 – исходное состояние; 2 – распознан символ «Начало выражения»; 3 – распознано выражение; 4 – распознан идентификатор; 5 – распознана открывающаяся круглая скобка; 6 – распознано вложенное выражение;  $X$  – заменяется на состояние входа в рекурсию;  $Y$  – заменяется на состояние выхода из рекурсии на каждом уровне вложения

**Таблица 4.1** Таблица переходов распознающего автомата

Состояние $q$	Входной символ $c$								
	<Ид>	+	-	*	/	(	)	▶	◀
1								2	
2						↓ $Y=3$			
3									К

Состояние $q$	Входной символ $c$								
	<Ид>	+	-	*	/	(	)	▶	◀
$X$	4					5			
4		$X$	$X$	$X$	$X$		↑ $Y$		↑ $Y$
5						↓ $Y=6$			
6							4		

В таблице использованы следующие условные обозначения:

К – конец разбора;

$E$  – состояние ошибки (все пустые ячейки таблицы должны содержать значение  $E$ );

$\downarrow Y = \langle \text{состояние} \rangle$  – рекурсивный вызов распознавателя (справа указан номер состояния после возврата из данного вызова);

$\uparrow Y$  – возврат из рекурсии (следующее состояние определяется значением  $Y$ ).

Необходимость запоминания состояния возврата при рекурсивном вызове подтверждает, что описываемый табл. 4.1 автомат относится к автоматам с магазинной (стековой) памятью.

Возможны два варианта реализации такого автомата: рекурсивная и нерекурсивная. При рекурсивной реализации для запоминания адреса возврата из рекурсии используется неявно организуемый программный стек, адрес возврата в который пишется автоматически при вызове подпрограммы. При нерекурсивной – стек реализуют в программе.

При разработке алгоритма нерекурсивной реализации используем следующие обозначения:

*Mag* – магазин (стек) распознающего автомата, элементы, записываемые в стек, – терминальные и нетерминальные символы;  $\downarrow$  – запись в стек,  $\uparrow$  – чтение из стека;

$q$  – текущее состояние;

*Table* ( $\langle \text{Текущее состояние} \rangle$ ,  $\langle \text{Анализируемый символ} \rangle$ ) – элемент таблицы (матрицы) переходов, состоящий из полей:  $q$  – следующее состояние, в том числе « $\downarrow$ » – рекурсивный вызов, « $\uparrow$ » – возврат из рекурсии;  $S$  – состояние после возврата из рекурсии, если необходим рекурсивный вызов, или  $\emptyset$ ;

*String* – исходная строка;

*Ind* – номер символа исходной строки.

Соответственно алгоритм программы, реализующей автомат, будет выглядеть следующим образом:

**$q := 1$**

**$Ind := 1$**

**$Mag := \emptyset$**



Цикл-пока  $q \neq \langle E \rangle$  и  $q \neq \langle K \rangle$   
 $q := \text{Table}[q, \text{String}[Ind]].q$   
 Если  $q = \downarrow$   
     то  $\text{Mag} \downarrow \text{Table}[q, \text{String}[Ind]].S$   
          $q := X$   
 иначе Если  $q = \uparrow$   
     то  $\text{Mag} \uparrow q$   
         иначе  $Ind := Ind + 1$   
 Все-если  
 Все-если  
 Если  $q = \langle K \rangle$   
     то «Строка принята»  
     иначе «Строка отвергнута»  
 Все-если

Построение автомата с магазинной памятью для сложного языка – задача непростая, поскольку таблица получается длинной – сотни строк. Поэтому для разбора контекстно-свободных языков в теории формальных языков разработаны другие методы. Для этого из класса КС-грамматик выделяют подклассы грамматик, обладающих определенными свойствами, основываясь на которых можно строить конкретные распознаватели.

#### **4.2. Синтаксические анализаторы $LL(k)$ -грамматик. Метод рекурсивного спуска**

Проанализируем процесс левостороннего нисходящего грамматического разбора (см. раздел 2.2.1), обращая внимание на ситуации, когда появляется необходимость возврата.

**Пример 4.2.** Осуществить левосторонний восходящий грамматический разбор строки токенов:

[Оглавление](#)



6	<Ид>+(<Ид>-Ид)*<Ид> ◀	<Терм> + <Выр> ◀	Подстановка правила в2
7	<Ид>+(<Ид>-<Ид>)*<Ид> ◀	<Множ>/<Терм>-<Выр> ◀	Подстановка правила г1
	<Ид>+(<Ид>-<Ид>)*<Ид> ◀	(<Выр>)/<Терм>-<Выр> ◀	Ошибка, воз- врат к шагу 7
8	<Ид>+(<Ид>-<Ид>)*<Ид> ◀	<Множ>/<Терм>-<Выр> ◀	Подстановка правила г2
	<Ид>+(<Ид>-<Ид>)*<Ид> ◀	<Ид>/<Терм>-<Выр> ◀	Символ распо- знан и удален
	+(<Ид>-<Ид>)*<Ид> ◀	/<Терм>-<Выр> ◀	Ошибка, воз- врат к шагу 6
9	<Ид>+(<Ид>-<Ид>)*<Ид> ◀	<Терм>+<Выр> ◀	Подстановка правила в3
10	<Ид>+(<Ид>-<Ид>)*<Ид> ◀	<Множ>+<Выр> ◀	Подстановка правила г1
	<Ид>+(<Ид>-<Ид>)*<Ид> ◀	(<Выр>)+<Выр> ◀	Ошибка, воз- врат к шагу 10
11	<Ид>+(<Ид>-<Ид>)*<Ид> ◀	<Множ>+<Выр> ◀	Подстановка правила г2
	<Ид>+(<Ид>-<Ид>)*<Ид> ◀	<Ид>+<Выр> ◀	Символ распо- знан и удален
	+(<Ид>-<Ид>)*<Ид> ◀	+<Выр> ◀	Символ распо- знан и удален
12	(<Ид>-<Ид>)*<Ид> ◀	<Выр> ◀	Подстановка правила б1
13	(<Ид>-<Ид>)*<Ид> ◀	<Терм>+<Выр> ◀	Подстановка правила в1
14	(<Ид>-<Ид>)*<Ид> ◀	<Множ>*<Терм>+<Выр> ◀	Подстановка правила г1
	(<Ид>-<Ид>)*<Ид> ◀	(<Выр>)*<Терм>+<Выр> ◀	Символ распо- знан и удален
15	<Ид>-<Ид>)*<Ид> ◀	<Выр>)*<Терм>+<Выр> ◀	Подстановка правила б1
16	<Ид>-<Ид>)*<Ид> ◀	<Терм>+<Выр>)*<Терм>+<Выр> ◀	Подстановка правила в1
17	<Ид>-<Ид>)*<Ид> ◀	<Множ>*<Терм>+<Выр>)*<Терм>+<Выр> ◀	Подстановка правила г1
	<Ид>-<Ид>)*<Ид> ◀	(<Выр>)*<Терм>+<Выр>)*<Терм>+<Выр> ◀	Ошибка, воз- врат к шагу 17
18	<Ид>-<Ид>)*<Ид> ◀	<Множ>*<Терм>+<Выр>)*<Терм>+<Выр> ◀	Подстановка правила г2
	<Ид>-<Ид>)*<Ид> ◀	<Ид>*<Терм>+<Выр>)*<Терм>+<Выр> ◀	Символ распо- знан и удален
	-<Ид>)*<Ид> ◀	*<Терм>+<Выр>)*<Терм>+<Выр> ◀	Ошибка, воз- врат к шагу 16
19	<Ид>-<Ид>)*<Ид> ◀	<Терм>+<Выр>)*<Терм>+<Выр> ◀	Подстановка правила в2
20	<Ид>-<Ид>)*<Ид> ◀	<Множ>/<Терм>+<Выр>)*<Терм>+<Выр> ◀	Подстановка правила г1

## Оглавление

	$\langle \text{Ид} \rangle - \langle \text{Ид} \rangle^* \langle \text{Ид} \rangle \blacktriangleleft$	$(\langle \text{Выр} \rangle / \langle \text{Терм} \rangle + \langle \text{Выр} \rangle)^* \langle \text{Терм} \rangle + \langle \text{Выр} \rangle \blacktriangleleft$	Ошибка, возврат к шагу 20
21	$\langle \text{Ид} \rangle - \langle \text{Ид} \rangle^* \langle \text{Ид} \rangle \blacktriangleleft$	$\langle \text{Множ} \rangle / \langle \text{Терм} \rangle + \langle \text{Выр} \rangle^* \langle \text{Терм} \rangle + \langle \text{Выр} \rangle \blacktriangleleft$	Подстановка правила г2
	$\langle \text{Ид} \rangle - \langle \text{Ид} \rangle^* \langle \text{Ид} \rangle \blacktriangleleft$	$\langle \text{Ид} \rangle / \langle \text{Терм} \rangle + \langle \text{Выр} \rangle^* \langle \text{Терм} \rangle + \langle \text{Выр} \rangle \blacktriangleleft$	Символ распознан и удален
	$-\langle \text{Ид} \rangle^* \langle \text{Ид} \rangle \blacktriangleleft$	$/\langle \text{Терм} \rangle + \langle \text{Выр} \rangle^* \langle \text{Терм} \rangle + \langle \text{Выр} \rangle \blacktriangleleft$	Ошибка, возврат к шагу 19
22	$\langle \text{Ид} \rangle - \langle \text{Ид} \rangle^* \langle \text{Ид} \rangle \blacktriangleleft$	$\langle \text{Терм} \rangle + \langle \text{Выр} \rangle^* \langle \text{Терм} \rangle + \langle \text{Выр} \rangle \blacktriangleleft$	Подстановка правила в3
23	$\langle \text{Ид} \rangle - \langle \text{Ид} \rangle^* \langle \text{Ид} \rangle \blacktriangleleft$	$\langle \text{Множ} \rangle + \langle \text{Выр} \rangle^* \langle \text{Терм} \rangle + \langle \text{Выр} \rangle \blacktriangleleft$	...

Анализ разбора показывает, что большинство подстановок лишние, так как выбраны правила, применение которых не позволят завершить разбор. Так, например, очевидно, что интерпретация первого слагаемого как множителя, умноженного на терминал, не имеет смысла, поскольку первое слагаемое выражения не содержит операции умножения.

Однако левосторонний нисходящий метод разбора предполагает последовательную подстановку всех правил в порядке их записи, поэтому рассматриваемая подстановка тоже проверяется, пока не будет обнаружено, что дальнейший разбор невозможен, т. е. не будут зафиксирована ошибка и выполнен возврат.

Разработать же алгоритм выбора правильной подстановки правил для используемой в настоящем примере грамматики достаточно сложно, так как проверка применимости правил зависит от их структуры.

*LL(k)-грамматики* – это подкласс КС-грамматик, гарантирующих существование *детерминированных нисходящих* распознавателей (*L* – левосторонний просмотр исходной строки, *L* – левосторонний разбор, *k* – количество символов, просматриваемых для определения очередного правила), т. е. это такие грамматики, для которых выбор следующего правила определяется *k* сравниваемыми символами строки и правила.

Грамматика записи выражений, рассмотренная в примере 4.2, *LL(k)*-грамматикой не является, однако, легко может быть к ней преобразована.

Перепишем правила таким образом, чтобы терминальные символы операций оказались в начале альтернативных правил, чтобы их можно было использовать для определения нужного правила. При этом возникнет необходимость использования пустых строк, обозначенных  $\epsilon$ :

- а)  $\langle \text{ПростоеВыражение} \rangle ::= \triangleright \langle \text{Выражение} \rangle \blacktriangleleft$ ,  
 б)  $\langle \text{Выражение} \rangle ::= \langle \text{Терминал} \rangle \langle \text{Сложение} \rangle$ ,  
 в)  $\langle \text{Сложение} \rangle ::= + \langle \text{Терминал} \rangle \langle \text{Сложение} \rangle |$   
 $- \langle \text{Терминал} \rangle \langle \text{Сложение} \rangle |$   
 $\epsilon$ ,  
 г)  $\langle \text{Терминал} \rangle ::= \langle \text{Множитель} \rangle \langle \text{Умножение} \rangle$ ,  
 д)  $\langle \text{Умножение} \rangle ::= * \langle \text{Множитель} \rangle \langle \text{Умножение} \rangle |$   
 $/ \langle \text{Множитель} \rangle \langle \text{Умножение} \rangle |$   
 $\epsilon$ ,  
 е)  $\langle \text{Множитель} \rangle ::= (\langle \text{Выражение} \rangle) | \langle \text{Идентификатор} \rangle$ .

Полученная грамматика по-прежнему является грамматикой второго типа по Хомскому, так как в ней присутствует вложенная рекурсия. Однако теперь это  $LL(1)$ -грамматика, так как, чтобы правильно выбрать очередное правило из альтернативных, достаточно проанализировать один терминальный символ – знак операции.

**Пример 4.3.** Осуществить нисходящий разбор выражения, используя  $LL(1)$ -грамматику языка выражений, приведенную выше.

При разборе используем следующие сокращения:

$\langle \text{Выр} \rangle = \langle \text{Выражение} \rangle$ ,  $\langle \text{Терм} \rangle = \langle \text{Терминал} \rangle$ ,  $\langle \text{Слож} \rangle = \langle \text{Сложение} \rangle$ ,  
 $\langle \text{Умн} \rangle = \langle \text{Умножение} \rangle$ ,  $\langle \text{Множ} \rangle = \langle \text{Множитель} \rangle$ ,  
 $\langle \text{Ид} \rangle = \langle \text{Идентификатор} \rangle$ .

Последовательность разбора показана в табл. 4.3.

**Таблица 4.3** Разбор выражения по правилам  $LL(1)$ -грамматики

№	Распознаваемая строка	Строка правил	Действие
1	$\langle \text{Ид} \rangle * (\langle \text{Ид} \rangle + \langle \text{Ид} \rangle) + \langle \text{Ид} \rangle$	$\langle \text{Выр} \rangle$	Подстановка правила а
2	$\langle \text{Ид} \rangle * (\langle \text{Ид} \rangle + \langle \text{Ид} \rangle) + \langle \text{Ид} \rangle$	$\langle \text{Терм} \rangle \langle \text{Слож} \rangle$	Подстановка

[Оглавление](#)

			правила в
3	<Ид>*(<Ид>+<Ид>)+<Ид>	<Множ><Умн><Слож>	Подстановка правила д2
4	<Ид>*(<Ид>+<Ид>)+<Ид>	<Ид><Умн><Слож>	Символ распознан
5	*(<Ид>+<Ид>)+<Ид>	<Умн><Слож>	Подстановка правила г1
6	*(<Ид>+<Ид>)+<Ид>	*<Множ><Умн><Слож>	Символ распознан
7	(<Ид>+<Ид>)+<Ид>	<Множ><Умн><Слож>	Подстановка правила д1
8	(<Ид>+<Ид>)+<Ид>	(<Выр>)<Умн><Сложение>	Символ распознан
9	<Ид>+<Ид>)+<Ид>	<Выр>)<Умн><Слож>	Подстановка правила а
10	<Ид>+<Ид>)+<Ид>	<Терм><Слож>)<Умн><Слож>	Подстановка правила в
11	<Ид>+<Ид>)+<Ид>	<Множ><Умн><Слож>)<Умн> <Слож>	Подстановка правила д2
12	<Ид>+<Ид>)+<Ид>	<Ид><Умн><Слож>)<Умн> <Слож>	Символ распознан
13	+<Ид>)+<Ид>	<Умн><Слож>)<Умн><Слож>	Подстановка правила г3 (е)
14	+<Ид>)+<Ид>	<Слож>)<Умн><Слож>	Подстановка правила б1
15	+<Ид>)+<Ид>	+<Терм><Слож>)<Умн><Слож> >	Символ распознан
16	<Ид>)+<Ид>	<Терм><Слож>)<Умн><Слож>	Подстановка правила в
17	<Ид>)+<Ид>	<Множ><Умн><Слож>)<Умн> <Слож>	Подстановка правила д2
18	<Ид>)+<Ид>	<Ид><Умн><Слож>)<Умн> <Слож>	Символ распознан
19	)<Ид>	<Умн><Слож>)<Умн><Слож>	Подстановка правила г3 (е)
20	)<Ид>	<Слож>)<Умн><Слож>	Подстановка правила б3 (е)
21	)<Ид>	)<Умн><Слож>	Символ распознан
22	+<Ид>	<Умн><Слож>	Подстановка правила г3 (е)
23	+<Ид>	<Слож>	Подстановка правила б1
24	+<Ид>	+<Терм><Слож>	Символ распознан
25	<Ид>	<Терм><Слож>	Подстановка правила в
26	<Ид>	<Множ><Умн><Слож>	Подстановка правила д2
27	<Ид>	<Ид> <Умн><Слож>	Символ

			распознан
28		<Умн><Слож>	Подстановка правила г3 (е)
29		<Слож>	Подстановка правила б3 (е)
30	∅	∅	Конец «Строка распознана»

Однозначный выбор правила за счет сравнения терминальных символов обеспечивает разбор без «возвратов», что существенно упрощает составление соответствующей программы.

Однако для программирования разбор  $LL(k)$ -грамматик наиболее удобен *метод рекурсивного спуска*, который также позволяет встроить лексический анализ в синтаксический. Соответственно при использовании метода рекурсивного спуска не возникает необходимости предварительного сканирования программы и построения таблицы токенов.

**Метод рекурсивного спуска.** Метод основывается на синтаксических диаграммах, описывающих конструкции языка. Он позволяет для каждого нетерминала разрабатывать рекурсивную процедуру. Основная программа вызывает процедуру аксиомы, которая вызывает процедуры нетерминалов, упомянутые в правой части аксиомы и т. д.

**Пример 4.4.** Разработать программу разбора выражений с использованием метода рекурсивного спуска.

Как указывалось ранее, синтаксис языка выражений можно описать с помощью синтаксических диаграмм (см. рис. 4.1).

По каждой диаграмме пишем рекурсивную процедуру и добавляем основную программу, проверяющую аксиому:

**Функция Выражение:** Boolean:

R:=Терм()

Цикл-пока R=true и (NextSymbol = '+' или NextSymbol = '-')

R:=Терм()

Все-цикл

Result:= R

**Все**

**Функция Терм: boolean:**

Множ()

Цикл-пока R=true и (NextSymbol = '\*' или NextSymbol = '/')

R:=Множ()

Все-цикл

Result:= R

**Все**

**Функция Множ: Boolean:**

Если NextSymbol = '('

то R:=Выражение()

Если NextSymbol ≠ ')' то Ошибка Все-если

иначе R:= Ид()

Все-если

Result:=R

**Все**

**Основная программа:**

Если NextSymbol ≠ '▶'

то R:=false;

иначе R:=Выражение()

Если R=true и NextSymbol ≠ '◀'

то R:=false;

Все-если

Все-если

Если R=false

то «Обнаружена ошибка»

Иначе «Строка распознана»

Все-если

**Конец**



Ниже приведен полный текст программы – распознавателя выражений. При этом распознаватель идентификаторов построен по синтаксической диаграмме (см. рис. 3.3).

```

Program Compiler;

{$APPTYPE CONSOLE}

uses SysUtils;

Type SetofChar=set of AnsiChar;
Const Bukv:setofChar=['A'..'Z','a'..'z'];
Const Cyfr:setofChar=['0'..'9'];
Const Razd:setofChar=[' ','+', '-', '*', '/', ')'];
Const TableId:array[1..2,1..4] of
Byte=((2,0,0,0),(2,2,10,0));
Function Culc(Var
St:shortstring;Razd:setofChar):boolean;forward;
Var St:shortstring; R:boolean;

Procedure Error(St:shortstring); {вывод сообщений об
ошибках}
Begin WriteLn('Error *** ', st, ' ***'); End;

Procedure Probef(Var St:shortstring); {удаление пробелов}
Begin While (St<>' ') and (St[1]=' ') do
Delete(St,1,1);
End;

Function Id(Var St:shortstring;Razd:setofChar):boolean;
{распознаватель идентификатора с помощью конечного автомата, текст см.
пример 3.2 из раздела 3.2}

Function Mult(Var
St:shortstring;Razd:setofChar):boolean;
Var R:boolean;
Begin
Probef(St);
if St[1]='(' then
begin

```

```

Delete(St,1,1); Probel(St);
R:=Culc(St,Razd);
Probel(St);
if R and (St[1]='') then Delete(St,1,1)
                        else Error(St);
end
else R:=Id(St,Razd);
Mult:=R;
End;

Function Term(Var
St:shortstring;Razd:setofChar):boolean;
Var S:shortstring; R:boolean;
Begin
R:=Mult(St,Razd);
if R then
begin
Probel(St);
While ((St[1]='*') or (St[1]='/')) and R do
begin
Delete(St,1,1);
R:=Mult(St,Razd);
end;
end;
Term:=R;
End;

```

```

Function Culc(Var
St:shortstring;Razd:setofChar):boolean;
Var S:shortstring; R:boolean;

```

### [Оглавление](#)

```

Begin
    R:=Term(St,Razd);
    if R then
        begin
            Probel(St);
            While ((St[1]='+') or (St[1]='-')) and R do
                begin
                    Delete(St,1,1);
                    R:=Term(St,Razd);
                end;
            end;
            Culc:=R;
        End;
Begin
    Writeln('Input Strings:'); Readln(St);
    R:=true;
    While (St<>'end') and R do
        Begin
            R:=Culc(St,Razd);
            if R and (length(st)=0) then Writeln('Yes')
                else Writeln('No');
            Writeln('Input Strings:'); Readln(St);
        End;
    Writeln('Input any key');
    Readln;
End.

```

### 4.3. Синтаксические анализаторы $LR(k)$ -грамматик. Грамматики предшествования

В отличие от  $LL(k)$ -грамматик  $LR(k)$ -грамматики обеспечивают восходящий разбор без возвратов. Вспомним, как осуществляется восходящий разбор на примере все той же грамматики выражений. Только теперь будем использовать правостороннюю рекурсию.

**Пример 4.5.** Осуществить восходящим методом разбор выражения:

$\blacktriangleright \langle \text{Ид} \rangle + (\langle \text{Ид} \rangle - \langle \text{Ид} \rangle) * \langle \text{Ид} \rangle \blacktriangleleft$ ,

где  $\langle \text{Ид} \rangle$  –  $\langle \text{Идентификатор} \rangle$ ,  $\blacktriangleright$ ,  $\blacktriangleleft$  – начало и конец выражения соответственно.

Грамматика записи выражений (вариант с правосторонней рекурсией):

а)  $\langle \text{ПростоеВыражение} \rangle ::= \blacktriangleright \langle \text{Выражение} \rangle \blacktriangleleft$ ,

б)  $\langle \text{Выражение} \rangle ::= \langle \text{Выражение} \rangle + \langle \text{Терминал} \rangle \mid$

$\langle \text{Выражение} \rangle - \langle \text{Терминал} \rangle \mid$

$\langle \text{Терминал} \rangle$ ,

в)  $\langle \text{Терминал} \rangle ::= \langle \text{Терминал} \rangle * \langle \text{Множитель} \rangle \mid$

$\langle \text{Терминал} \rangle / \langle \text{Множитель} \rangle \mid$

$\langle \text{Множитель} \rangle$ ,

г)  $\langle \text{Множитель} \rangle ::= (\langle \text{Выражение} \rangle) \mid \langle \text{Идентификатор} \rangle$ .

При записи разбора используем следующие обозначения:

$\langle \text{Выр} \rangle$  –  $\langle \text{Выражение} \rangle$ ,  $\langle \text{Терм} \rangle$  –  $\langle \text{Терминал} \rangle$ ,  $\langle \text{Множ} \rangle$  –  $\langle \text{Множитель} \rangle$ ,  $\langle \text{Ид} \rangle$  –  $\langle \text{Идентификатор} \rangle$ .

Последовательность разбора (жирным шрифтом выделена основа, свертываемая на следующем шаге) показана в табл. 4.4.

**Таблица 4.4** Разбор выражения левосторонним восходящим методом

№ шага	Распознаваемая строка	Основа	Операция
1	$\langle \text{Ид} \rangle + (\langle \text{Ид} \rangle - \langle \text{Ид} \rangle) * \langle \text{Ид} \rangle$	$\langle \text{Ид} \rangle$	Свертка по правилу г2

2	$\langle \text{Множ} \rangle + (\langle \text{Ид} \rangle - \langle \text{Ид} \rangle) * \langle \text{Ид} \rangle$	$\langle \text{Множ} \rangle$	Свертка по правилу в3
3	$\langle \text{Терм} \rangle + (\langle \text{Ид} \rangle - \langle \text{Ид} \rangle) * \langle \text{Ид} \rangle$	$\langle \text{Терм} \rangle$	Свертка по правилу б3
4	$\langle \text{Выр} \rangle + (\langle \text{Ид} \rangle - \langle \text{Ид} \rangle) * \langle \text{Ид} \rangle$	$\langle \text{Ид} \rangle$	Свертка по правилу г2
5	$\langle \text{Выр} \rangle + (\langle \text{Множ} \rangle - \langle \text{Ид} \rangle) * \langle \text{Ид} \rangle$	$\langle \text{Множ} \rangle$	Свертка по правилу в3
6	$\langle \text{Выр} \rangle + (\langle \text{Терм} \rangle - \langle \text{Ид} \rangle) * \langle \text{Ид} \rangle$	$\langle \text{Терм} \rangle$	Свертка по правилу б3
7	$\langle \text{Выр} \rangle + (\langle \text{Выр} \rangle - \langle \text{Ид} \rangle) * \langle \text{Ид} \rangle$	$\langle \text{Ид} \rangle$	Свертка по правилу г2
8	$\langle \text{Выр} \rangle + (\langle \text{Выр} \rangle - \langle \text{Множ} \rangle) * \langle \text{Ид} \rangle$	$\langle \text{Множ} \rangle$	Свертка по правилу в3
9	$\langle \text{Выр} \rangle + (\langle \text{Выр} \rangle - \langle \text{Терм} \rangle) * \langle \text{Ид} \rangle$	$\langle \text{Выр} \rangle - \langle \text{Терм} \rangle$	Свертка по правилу б2
10	$\langle \text{Выр} \rangle + (\langle \text{Выр} \rangle) * \langle \text{Ид} \rangle$	$(\langle \text{Выр} \rangle)$	Свертка по правилу г1
11	$\langle \text{Выр} \rangle + \langle \text{Множ} \rangle * \langle \text{Ид} \rangle$	$\langle \text{Множ} \rangle$	Свертка по правилу в3
12	$\langle \text{Выр} \rangle + \langle \text{Терм} \rangle * \langle \text{Ид} \rangle$	$\langle \text{Выр} \rangle + \langle \text{Терм} \rangle$	Свертка по правилу б1
13	$\langle \text{Выр} \rangle * \langle \text{Ид} \rangle$	Тупик!	Возврат к шагу 12.
14	$\langle \text{Выр} \rangle + \langle \text{Терм} \rangle * \langle \text{Ид} \rangle$	$\langle \text{Ид} \rangle$	Свертка по правилу г2
15	$\langle \text{Выр} \rangle + \langle \text{Терм} \rangle * \langle \text{Множ} \rangle$	$\langle \text{Выр} \rangle + \langle \text{Терм} \rangle$	Свертка по правилу б1
16	$\langle \text{Выр} \rangle * \langle \text{Множ} \rangle$	Тупик!	Возврат к шагу 15.
17	$\langle \text{Выр} \rangle + \langle \text{Терм} \rangle * \langle \text{Множ} \rangle$	$\langle \text{Терм} \rangle * \langle \text{Множ} \rangle$	Свертка по правилу в1
18	$\langle \text{Выр} \rangle + \langle \text{Терм} \rangle$	$\langle \text{Выр} \rangle + \langle \text{Терм} \rangle$	Свертка по правилу б1
19	$\langle \text{Выр} \rangle$	Конец.	-

Тупики возникают при неверном выборе основы.

$LR(k)$ -грамматики – это класс КС-грамматик, гарантирующих существование детерминированных *восходящих* распознавателей ( $L$  – левосторонний просмотр,  $R$  – правосторонний разбор,  $k$  – количество символов, просматриваемых для однозначного определения следующего правила). В грамматиках этого класса отсутствуют правила с правосторонней рекурсией, и обеспечивается однозначное выделение основы.

При восходящем разборе стек используют для накопления основы. Автомат при этом выполняет две основные операции: свертку и перенос. Свертка осуществляется, когда в стеке накоплена вся основа, и заключается в ее замене на левую часть соответствующего правила. Перенос выполняется в процессе накопления основы и заключается в сохранении в стеке очередного распознаваемого символа сентенциальной формы. Основная проблема мето-

[Оглавление](#)

да заключается в нахождении способа выделения очередной основы. Проще всего основу выделить для грамматик, получивших название «грамматики предшествования». Рассмотрим эти грамматики.

Если два символа  $\alpha, \beta \in V$  расположены рядом в сентенциальной форме, то между ними возможны следующие отношения, названные *отношениями предшествования*:

- 1)  $\alpha$  принадлежит основе, а  $\beta$  – нет, т. е.  $\alpha$  – конец основы:  $\alpha > \beta$ ;
- 2)  $\beta$  принадлежит основе, а  $\alpha$  – нет, т. е.  $\beta$  – начало основы:  $\alpha < \beta$ ;
- 3)  $\alpha$  и  $\beta$  принадлежит одной основе, т. е.  $\alpha = \beta$ ;
- 4)  $\alpha$  и  $\beta$  не могут находиться рядом в сентенциальной форме (ошибка).

**Грамматикой с предшествованием** называется грамматика, в которой существует однозначное отношение предшествования между соседними символами. Это отношение позволяет просто определить очередную основу, т. е. момент выполнения каждой свертки.

Различают:

- 1) грамматики с простым предшествованием, для которых  $\alpha, \beta \in V$ ;
- 2) грамматики с операторным предшествованием, для которых  $\alpha, \beta \in V_T$ ;  
т. е. отношение предшествования определено для терминальных символов и не зависит от нетерминальных символов, расположенных между ними;
- 3) грамматики со слабым предшествованием, для которых отношение предшествования не однозначно – оно требует выполнения специальных проверок.

Метод разбора, использующий свойства грамматик предшествования, был назван *стековым*, поскольку в процессе разбора для промежуточного хранения информации используется стек.

**Пример 4.6.** Грамматика описания арифметических выражений, рассмотренная выше, относится к классу грамматик с операторным предшествованием.

Отношения предшествования терминалов (знаков операций), полученные с учетом приоритетов операций, сведены в табл. 4.5.

**Таблица 4.5** Таблица предшествования

Знак операции $\alpha$	Знак операции $\beta$				
	+	*	(	)	◀
▶	<.	<.	<.	?	Выход
+	.>	<.	<.	.>	.>
*	.>	.>	<.	.>	.>
(	<.	<.	<.	=	?
)	.>	.>	?	.>	.>

*Примечание.* В таблице использованы следующие обозначения:

? – ошибка; <. – начало основы; .> – конец основы; = – принадлежат одной основе; ▶ – начало выражения; ◀ – конец выражения.

Пусть задано выражение

▶  $d + c * (a + b)$  ◀,

где  $a, b, c, d$  – обозначения токенов вида <Ид>, соответствующих идентификаторам переменных, констант или литералов чисел.

Результат разбора заданного выражения стековым методом представлен в табл. 4.6.

**Таблица 4.6** Разбор выражения стековым методом

Содержимое стека	Анализируемые символы	Отношение	Операция	Тройка	Результат свертки
▶	$d +$	<.	Перенос	—	—
▶ $d +$	$c *$	<.	Перенос	—	—
▶ $d + c *$	(	<.	Перенос	—	—
▶ $d + c * ($	$a +$	<.	Перенос	—	—
▶ $d + c * (a +$	$b )$	.>	Свертка	$R_1 := a + b$	<Выражение>
▶ $d + c * ($	$R_1 )$	=.	Свертка	$R_1 := (R_1)$	<Множитель>
▶ $d + c *$	$R_1 \blacktriangleleft$	.>	Свертка	$R_2 := c * R_1$	<Терм>
▶ $d +$	$R_2 \blacktriangleleft$	.>	Свертка	$R_3 := d + R_2$	<Выражение>
▶	$R_3 \blacktriangleleft$	Конец			

Текст программы, реализующей стековый метод, представлен ниже:

```

Program Compiler3;
{$APPTYPE CONSOLE}
Uses SysUtils;

Type SetofChar=set of AnsiChar;
      Troyki=array[1..10] of shortstring; {массив троек}
Const Razd:setofChar=[' ', '+', '-', '*', '/', '(', ')'];
Var St, StS:shortstring;
      Comands:Troyki;
      R:boolean;
Function Scan(St:shortstring;Razd:setofChar;
              Var StS:shortstring):boolean;forward;
      { Сканер, текст программы приведен в примере 3.4}
Procedure Probel(Var St:shortstring); { Подпрограмма
      удаления пробелов, текст программы приведен в примере 3.2}
Procedure Error(St:shortstring); {подпрограмма вывода
      сообщений об ошибках}
      Begin      WriteLn('Error *** ', st, ' ***'); End;

Function Id(Var St:shortstring;Razd:setofChar):boolean;
      { Подпрограмма распознавания идентификаторов, текст
      программы приведен в примере 3.2}
Function Stack_metod(
      St:shortstring; // строка токенов
      Var Comands:Troyki // массив троек
      ):boolean;
      Const TablePred:array[0..3,1..6] of byte=
              ((1, 1, 1,10, 4,10),

```



```

        (2, 1, 1, 2, 2,10),
        (2, 2, 1, 2, 2,10),
        (1, 1, 1, 3,10,10));

Var Stack:shortString;
    i,i1,IndStr,IndCol:byte;  Konec:boolean;
Procedure Perenos(Var St,Stack:shortstring); // перенос
Begin
    Stack:=Stack+Copy(St,1,2);
    Delete(St,1,2);
end;

Procedure Svertka(Var i:byte;Var St,Stack:shortstring;
    Var Comands:Troyki); // свертка
begin
    i:=i+1;
    Comands[i]:=Copy(Stack,length(Stack)-1,2);
    Delete(Stack,length(Stack)-1,2);
    Comands[i]:=Comands[i]+St[1];
    St[1]:='R';
end;

Procedure Odn_osnova(Var St,Stack:shortstring); {одна
                                                    основа}
Begin
    Delete(Stack,length(Stack),1);
    if Stack[length(Stack)]='@' then
        Delete(Stack,length(Stack),1);
    Delete(St,2,1);
end;

Begin {текст синтаксического анализатора}
    i:=0;
```

```

Konec:=false;
Result:=false;
Stack:='>';
St:=St+'<';
while not Konec do
begin
    case Stack[length(Stack)] of
        '+','-': IndStr:=1;
        '*','/': IndStr:=2;
        '(': IndStr:=3;
        '>': IndStr:=0;
        else IndStr:=0;
    end;
    case St[2] of
        '+','-': IndCol:=1;
        '*','/': IndCol:=2;
        '(': IndCol:=3;
        ')': IndCol:=4;
        '<': IndCol:=5;
        else IndCol:=6;
    end;
    case TablePred[IndStr,IndCol] of
        1: Perenos(St,Stack);
        2: Svertka(i,St,Stack,Comands);
        3: Odna_osnova(St,Stack);
        4: begin
            Result:=true;
            Konec:=true;
            For i1:=1 to i do
                WriteLn('Comands:',Comands[i1]);
            end;
        end;
    end;
end;

```

### [Оглавление](#)

```

        end;
    10: begin
        Konec:=true;
        For i1:=1 to i do
            WriteLn('Comands:',Comands[i1]);
        WriteLn('St=',St);
        end;
    else
        begin
            Konec:=true;
            For i1:=1 to i do
                WriteLn('Comands:',Comands[i1]);
            WriteLn('St=',St);
        end;
    end;
end;
end;
end;
Begin
    Writeln('Input Strings:'); Readln(St);
    R:=true;
    While (St<>'end') and R do
        Begin
            R:=Scan(St,Razd,StS);
            if R then R:=Stack_metod(StS,Comands);
            if R then Writeln('Yes')
                else Writeln('No');
            Writeln('Input Strings:'); Readln(St);
            R:=true;
        End;
    Writeln('Press Enter');

```

Readln;  
End.

#### **4.4. Польская запись. Алгоритм Бауэра–Замельзона**

Результат синтаксического анализа – дерево грамматического разбора – часто представляют в виде обратной польской записи.

**Польская запись (обратная)** (в честь польского математика Яна Лукасевича, предложившего постфиксную запись выражений) представляет собой последовательность команд двух типов:

$K_I$ , где  $I$  – идентификатор операнда. Эта команда означает: выбрать число по имени  $I$  и заслать его в стек операндов;

$K_\xi$ , где  $\xi$  – операция. Эта команда указывает: выбрать два верхних числа из стека операндов, произвести над ними операцию  $\xi$  и занести результат в стек операндов.

Рассмотрим алгоритм Бауэра–Замельзона, по которому осуществляется представление выражений в виде польской записи.

**Алгоритм Бауэра–Замельзона.** Грамматика, описывающая правила записи арифметических выражений, относится к классу грамматик операторного предшествования, т. е. порядок следования терминальных символов (знаков операций) однозначно определяет порядок выделения троек, причем нетерминальные символы (имена операндов) на этот порядок не влияют.

Синтаксический распознаватель выражений в процессе разбора должен формировать запись, по которой затем выполняется генерация кода. В качестве такой записи часто используют обратную польскую запись.

Согласно алгоритму Бауэра–Замельзона разбор выражения и формирование польской записи выполняется в два этапа:

- 1) разбор выражения и построение эквивалентной польской записи;
- 2) выполнение (или трансляция) польской записи.

При этом используются стек операций (на первом этапе) и стек операндов (на втором этапе).

Польская запись строится следующим образом. Компилятор читает выражение слева направо и вырабатывает последовательность команд по правилу:

- а) если символ – операнд с именем  $I$ , то вырабатывается команда  $K_I$ ;
- б) если символ – операция  $\xi$ , то реализуются действия с учетом верхнего символа  $\eta$  из стека операций согласно табл. 4.7.

**Таблица 4.7** Таблица генератора записи

Верхний символ в стеке операций $\eta$	Очередной символ $\xi$				
	+	*	(	)	◀
▶	I	I	I	?	Выход
+	II	I	I	IV	IV
*	IV	II	I	IV	IV
(	I	I	I	III	?

*Примечание.* Операции: I – заслать  $\xi$  в стек операций и читать следующий символ; II – генерировать  $K_\eta$ , заслать  $\xi$  в стек операций и читать следующий символ; III – удалить верхний символ  $\eta$  из стека операций и читать следующий символ; IV – генерировать  $K_\eta$  и повторить с тем же входным символом  $\xi$ ; ? – ошибка.

На этапе выполнения польская запись читается слева направо и выполняется.

**Пример 4.7.** Построить тройки для выражения  $\blacktriangleright (a+b*c) / d \blacktriangleleft$ .

Этап 1. Последовательность построения польской записи представлена в табл. 4.8.

**Таблица 4.8** Построение польской записи выражения

Стек операций	Символ	Действие	Команда
▶	(	I	

[Оглавление](#)

► (	a		$K_a$
► (	+	I	
► (+	b		$K_b$
► (+	*	I	
► (+ *	c		$K_c$
► (+ *	)	IV	$K_*$
► (+	)	IV	$K_+$
► (	)	III	
►	/	I	
► /	d		$K_d$
► /	◀	IV	$K_/_$
►	◀	Конец	

В результате получаем польскую запись:

$K_a K_b K_c K_* K_+ K_d K_/_$  или  $abc*+d/_$ .

Теперь необходимо выполнить польскую запись в соответствии с ее определением.

Этап 2. Выполнение польской записи представлено в табл. 4.9.

**Таблица 4.9** Выполнение польской записи

Стек операндов	Команда	Тройка
∅	$K_a$	—
a	$K_b$	—
ab	$K_c$	—
abc	$K_*$	$T_1=b*c$
$aT_1$	$K_+$	$T_2=a+T_1$
$T_2$	$K_d$	—
$T_2d$	$K_/_$	$T_3=T_2/_d$
$T_3$	—	—

Польская запись может использоваться как промежуточная форма не только для выражений, но и для других операторов. Соответственно при этом для получения записи должен использоваться модифицированный алгоритм Бауэра–Замельзона.

### Контрольные вопросы и задания

4.1. Определите автомат с магазинной памятью. В чем особенность его использования при грамматическом разборе?

[Ответ.](#)

4.2. Как построить автомат с магазинной памятью по синтаксическим диаграммам? В чем заключается особенность полученной программы?

[Ответ.](#)

4.3. Определите  $LL(k)$ -грамматики. Какими особенностями они обладают? В чем заключается метод рекурсивного спуска?

[Ответ.](#)

4.4. Определите  $LR(k)$ -грамматики. Какими особенностями они обладают? В чем заключается стековый метод?

[Ответ.](#)

4.5. Какой вид имеет польская запись? Для чего она используется?

[Ответ.](#)

4.6. В чем заключается алгоритм Бауэра–Замельзона?

[Ответ.](#)

4.7. Дана грамматика описывающая упрощенный вариант сложных выражений C++:

$\langle \text{Сложное выражение} \rangle ::= \langle \text{Сложное выражение} \rangle, \langle \text{Выражение} \rangle \mid$   
 $\langle \text{Выражение} \rangle$

$\langle \text{Выражение} \rangle ::= \langle \text{Выражение} \rangle + \langle \text{Ид} \rangle \mid \langle \text{Выражение} \rangle - \langle \text{Ид} \rangle \mid \langle \text{Ид} \rangle$

Приведите последовательность выполнения грамматического разбора левосторонним восходящим методом для сентенциальной формы

$\langle \text{Ид} \rangle + \langle \text{Ид} \rangle - \langle \text{Ид} \rangle, \langle \text{Ид} \rangle - \langle \text{Ид} \rangle, \langle \text{Ид} \rangle - \langle \text{Ид} \rangle$

[Решение.](#)

- 4.8. Для грамматики, определенной в задании 4.7, приведите последовательность выполнения грамматического разбора стековым методом для сентенциальной формы

$\text{Ид} > + < \text{Ид} > - < \text{Ид} > , < \text{Ид} > - < \text{Ид} > , < \text{Ид} > - < \text{Ид} > .$

[Решение.](#)

- 4.9. Для грамматики, определенной в задании 4.7, приведите последовательность выполнения грамматического разбора с использованием польской записи для сентенциальной формы

$\text{Ид} > + < \text{Ид} > - < \text{Ид} > , < \text{Ид} > - < \text{Ид} > , < \text{Ид} > - < \text{Ид} > .$

Решение.



## 5. Генерация и оптимизация кодов

После распознавания конструкций и определения таблиц переменных, именованных констант и временных переменных на первом шаге генерации кодов распределяют память под эти данные и программы. Далее выполняется генерация и оптимизация объектного кода.

### 5.1. Распределение памяти под программы и данные

Для универсальных языков программирования различают несколько классов памяти переменных (табл. 5.1).

**Таблица 5.1** Классы памяти переменных

№	Класс памяти C++	В Паскале	Размещение	Видимость	Время жизни	Тип распределения памяти
1	Внешние extern	Глобальные	В основном сегменте данных программы	Программа и подпрограммы, если отсутствует перекрытие имен в подпрограммах	От запуска программы до ее завершения	Статическое
2	Внешние статические extern static	Переменные модуля, объявленные внутри Unit	В сегменте данных модуля	Для подпрограмм файла (модуля)	От подключения файла (модуля) до его отключения	Статическое
3	Автоматические auto	Локальные	В стеке	Из подпрограммы, в которой объявлены, и вызываемых из нее подпрограмм	От момента вызова подпрограммы до ее завершения	Автоматическое
4	Статические static	—	В основном сегменте данных программы	Из подпрограммы, в которой объявлены, и вызываемых из нее подпрограмм	От запуска программы до ее завершения	Статическое

**Статическое распределение** памяти выполняется:

- а) во время компиляции – для подпрограмм и инициализированных переменных;
- б) во время загрузки программы на выполнение – для неинициализированных переменных.

**Автоматическое распределение** памяти осуществляется для переменных подпрограмм, используемых только при активации подпрограммы. Эти переменные обычно размещают в стеке.

Кроме этого, существует *динамическое распределение памяти*, которое может выполняться как управляемое и как базированное.

**Управляемое распределение** памяти реализуется во время выполнения программы специальными подпрограммами, например, new и delete.

**Базированное распределение** памяти осуществляется во время выполнения программы из выделенного буфера. Доступ к такой памяти осуществляется через вычисляемые адреса.

В конечном итоге выбор способов распределения памяти под программы, подпрограммы и данные осуществляется программистом.

## 5.2. Генерация кодов

Машинный код генерируется заменой распознанной конструкции соответствующими машинными командами, например, тройка сложения целых чисел может заменяться последовательностью команд:

```
mov    AX, <Op1>
add    AX, <Op2>
mov    AX, <Result>
```

Затем в полученной последовательности команд подставляются адреса операндов и результат.

Очевидно, что код, полученный таким образом, является не оптимальным, поэтому при генерации он оптимизируется.

[Оглавление](#)

Все способы оптимизации можно разделить на две группы:

- 1) машинно-независимая оптимизация;
- 2) машинно-зависимая оптимизация.

**Машинно-независимая** оптимизация включает:

- а) исключение повторных вычислений одних и тех же операндов;
- б) выполнение операций над константами во время трансляции;
- в) вынесение из циклов вычисления величин, не зависящих от параметров циклов;
- г) упрощение сложных логических выражений и т. п.

**Машинно-зависимая** оптимизация включает:

- а) исключение лишних передач управления типа «память-регистр»;
- б) выбор более эффективных команд и т. п.

Оба типа оптимизации предполагают разработку соответствующих семантических моделей для представления результатов распознавания конструкций. Обычно с этой целью используют таблицы разного типа.

### **5.3. Машинно-независимая оптимизация**

#### **5.3.1. Удаление недостижимого кода**

В этом случае задача компилятора найти и исключить код, на который не передается управление.

**Пример 5.1.** В следующем фрагменте выполняется только оператор S1:

**if (1) S1; else S2;   ⇒    S1;**

Альтернативная ветвь никогда не реализуется потому, что условие всегда истинно.

### 5.3.2. Оптимизация линейных участков программы

В современных системах программирования профилировщик на основе результатов запуска программы выдаёт информацию о том, на какие её линейные участки приходится основное время выполнения. Для этих фрагментов реализуется:

а) удаление бесполезных присваиваний, например,

**$a = b * c; d = b + c; a = d * c; \Rightarrow d = b + c; a = d * c;$**

*Примечание.* В следующем примере операция присваивания значения переменной *a* уже не бесполезна, поскольку результат используется через указатель *p*:

**$p = \&a; a = b * c; d = *p + c; a = d * c;$**

б) исключение избыточных вычислений, например,

**$d = d + b * c; a = d + b * c; c = d + b * c; \Rightarrow$**

**$t = b * c; d = d + t; a = d + t; c = a;$**

в) выполнение во время компиляции тех операций исходной программы, для которых значения операндов уже известны, например,

**$i = 2 + 1; j = 6 * i + i; \Rightarrow i = 3; j = 21;$**

г) перестановка операций для дальнейшей свертки или оптимизации вычислений, например,

**$a = 2 * b * 3 * c; \Rightarrow a = (2 * 3) * (b * c);$**

**$a = (b + c) + (d + e); \Rightarrow a = (b + (c + (d + e)));$**

В первом случае появляется возможность выполнить операцию над константами во время компиляции, во втором – результаты операций можно хранить в одном регистре, что позволяет использовать более быстрый вариант машинной команды «регистр-память»;

д) арифметические преобразования на основе алгебраических и логических тождеств, например,

**$a = b * c + b * d; \Rightarrow a = b * (c + d);$**

$a*1 \Rightarrow a, \quad a*0 \Rightarrow 0, \quad a+0 \Rightarrow a;$

е) оптимизация вычисления логических выражений

$a || b || c || d; \Rightarrow a,$  если  $a = \text{true}.$

*Примечание.*  $a || f(b) || g(c)$  не всегда равно  $a$  при  $a = \text{true}$ , так как возможен побочный эффект из-за операций в теле функций.

### 5.3.3. Подстановка кода функции вместо ее вызова в объектный код

Этот способ оптимизации, как правило, применим к простым функциям и процедурам, вызываемым непосредственно по адресу, без применения косвенной адресации через таблицы RTTI (Run Time Type Information).

Некоторые компиляторы допускают применять указанный способ только к функциям, содержащим последовательные вычисления без циклов. Язык C++ позволяет явно указать (`inline`), для каких функций желательно использовать `inline`-подстановку.

### 5.3.4. Оптимизация вычислений в циклах

Анализируются параметры циклов и циклических операций и осуществляются:

а) вынесение инвариантных вычислений из циклов, например,

```
for (i=1; i<=10; i++) a[i]=b*c*a[i];    =>
d = b*c;  for (i=1; i<=10; i++) a[i]=d*a[i];
```

б) замена операций с индуктивными, т. е. образующими арифметическую прогрессию переменными, например,

```
for (i=1; i<=N; i++) a[i]=i*10;    =>
t = 10; i = 1; while (i<=N) { a[i]=t; t=t+10; i++;}
// заменили умножение на сложение
s = 10; for (i=1; i<=N; i++) {r=r+f(s); s=s+10;} =>
s=10; m=N*10; while (s <= m) { r=r+f(s); s=s+10; }
```

[Оглавление](#)

// избавились от одной индуктивной переменной

в) слияние циклов, например,

```
for (i=1; i<=N; i++)
    for (j=1; j<= M; j++) a[i][j] = 0;      =>
k = N*M;
for (i=1; i<=k; i++) a[i] = 0; // только в объектном коде!
```

г) разворачивание циклов, например,

```
for (i=1; i<=3; i++) a[i]=i;      =>
a [1] = 1; a [2] = 2; a [3] = 3;
```

*Примечание.* Способ применим, если количество повторений цикла известно на этапе компиляции.

#### **5.4. Машинно-зависимая оптимизация**

Машинно-зависимые преобразования результирующей объектной программы зависят от архитектуры вычислительной системы, на которой будет выполняться результирующая программа. При этом может учитываться объем кэш-памяти, методы организации работы процессора и другие особенности системы.

Эти преобразования, как правило, являются "ноу-хау", и именно они существенно повышают эффективность результирующего кода.

##### **5.4.1. Распределение регистров процессора**

Использование регистров общего назначения и специальных регистров (аккумулятор, счетчик цикла, базовый указатель) для хранения значения операндов и результатов вычислений увеличивает быстродействие программы.

Доступных регистров всегда ограниченное количество, поэтому перед компилятором встает вопрос их оптимального распределения и использования при выполнении вычислений.

#### 5.4.2. Оптимизация передачи параметров в процедуры и функции

Обычно параметры процедур и функций передаются через стек. При этом всякий раз при вызове процедуры или функции компилятор создает объектный код для размещения ее фактических параметров в стеке, а при выходе из нее – код для освобождения соответствующей памяти.

Можно уменьшить код и время выполнения результирующей программы за счет оптимизации передачи параметров в процедуру или функцию, передавая их **через регистры** процессора (конвенция *fastcall*).

Реализация данного оптимизирующего преобразования зависит от количества доступных регистров процессора в целевой вычислительной системе и от используемого компилятором алгоритма распределения регистров.

Недостатки метода:

- оптимизированные таким образом процедуры и функции не могут быть использованы в качестве *библиотечных*, так как методы передачи параметров через регистры не стандартизованы и зависят от реализации компилятора;
- этот метод не может быть использован, если где-либо в функции требуется выполнить *операции с адресами* параметров.

Языки Си и C++ позволяют явно указать (конвенция *register*), какие параметры и локальные переменные желательно разместить в регистрах.

#### 5.4.3. Оптимизация кода для процессоров, допускающих параллеливание вычислений

При возможности параллельного выполнения нескольких операций компилятор должен порождать объектный код таким образом, чтобы в нем было максимально возможное количество соседних операций, все операнды которых не зависят друг от друга.

Для каждого оператора надо найти оптимальный порядок выполнения операций (переставить их), например,

[Оглавление](#)

$$a + b + c + d + e + f; \Rightarrow$$

если один поток обработки данных, то последовательно складываются операнды  $((((a + b) + c) + d) + e) + f;$

если два потока обработки данных, то результат рассчитывается как сумма двух последовательных сложений операндов (каждая в своем потоке)  $((a + b) + c) + ((d + e) + f);$

если три потока обработки данных, то аналогично предусматриваем три последовательности сложений операндов

$$(a + b) + (c + d) + (e + f);$$

### ***Контрольные вопросы и задания***

5.1. Какие виды распределения памяти вы знаете? Чем они различаются и для чего используются?

[Ответ.](#)

5.2. Как происходит выбор способа распределения памяти под данные программы?

[Ответ.](#)

5.3. Назовите машинно-независимые способы оптимизации кода. Почему они так названы?

[Ответ.](#)

5.4. Назовите машинно-зависимые способы оптимизации кода. Почему они так названы?

[Ответ.](#)

5.5. Выполните ручную оптимизацию следующих фрагментов программы:

а)  $c = -b * b * b + (g - 1) * b;$

б)  $a = b * 128;$

в)  $a = b + c; b = a - 5; a = b - d;$

### [Оглавление](#)



- г) `a=0;c=5; while (b&&!b) a+=c;`
- д) `for(s=0,i=0;i<=10;i++) a[i]=k*5;`
- е) `a=5;b=(a+1)*j;`

[Решение.](#)

## **Заключение**

Для общения человека с вычислительными системами используется большое количество формальных языков, работа с которыми предполагает наличие в системе транслирующих, компилирующих или интерпретирующих программ.

Построение таких программ базируется на знании основных методов теории формальных грамматик и формальных языков, рассмотренных в настоящем учебном пособии.

Знание этой теории, с одной стороны, существенно углубляет понимание особенностей различных языков программирования, а с другой, позволяет при необходимости осознанно использовать методы грамматического разбора при решении различных практических задач.

## Литература

*Ахо А., Сети Р., Ульман Д.* Компиляторы: принципы, технологии, инструменты. Пер. с англ. М.: Вильямс, 2001. 767 с.

*Грис. Д.* Конструирование компиляторов для цифровых вычислительных машин. М.: Мир, 1975. 544 с.

*Рейуорд-Смит В.Дж.* Теория формальных языков. Вводный курс. М.: Радио и связь, 1988. 127 с.

## Приложение. Решение контрольных заданий

**Задание 1.5.** Для заданного предложения на языке C++ покажите, какие операции будут выполняться на этапах лексического и синтаксического анализа:

```
for (i1=0, s=0; i1<=5; i1++)
    if (i1%3==0) s+=abc[i1];
```

**Решение.**

Результат лексического анализа приведен в табл. П.1 – П.3.

**Таблица П.4** Таблица токенов

Лексема	Тип лексемы	Значение	Ссылка
for	Служебное слово	Код «for»	—
(	Служебный символ	Код «(»	—
i1	Идентификатор	—	Адрес в таблице идентификаторов
=	Служебный символ	Код «=»	—
0	Литерал	—	Адрес в таблице литералов
,	Служебный символ	Код «,»	—
s	Идентификатор	—	Адрес в таблице идентификаторов
=	Служебный символ	Код «=»	—
0	Литерал	—	Адрес в таблице литералов
;	Служебный символ	Код «;»	—
i1	Идентификатор	—	Адрес в таблице идентификаторов
<=	Служебный символ	Код «<=»	—
5	Литерал	—	Адрес в таблице литералов
;	Служебный символ	Код «;»	—
i1	Идентификатор	—	Адрес в таблице идентификаторов
++	Служебный символ	Код «++»	—
)	Служебный символ	Код «)»	—
if	Служебное слово	Код «if»	—
(	Служебный символ	Код «(»	—
i1	Идентификатор	—	Адрес в таблице идентификаторов
%	Служебный символ	Код «%»	—
3	Литерал	—	Адрес в таблице литералов
==	Служебный символ	Код «==»	—
0	Литерал	—	Адрес в таблице литералов
)	Служебный символ	Код «)»	—
s	Идентификатор	—	Адрес в таблице идентификаторов

[Оглавление](#)

+=	Служебный символ	Код «+=»	—
abc	Идентификатор	—	Адрес в таблице идентификаторов
[	Служебный символ	Код «[»	—
i1	Идентификатор	—	Адрес в таблице идентификаторов
]	Служебный символ	Код «]»	—
;	Служебный символ	Код «;»	—

**Таблица П.5** Таблица  
идентификаторов переменных

Имя	Тип	Адрес
i1	int	Ø (не распределена)
s	int	Ø (не распределена)
abc	int *	Ø (не распределена)

**Таблица П.3** Таблица  
констант

Константа	Тип	Значение
0	int	0x 00000000
5	int	0x 00000005
3	int	0x 00000003

Результат синтаксического анализа представлен на рис. П.1.

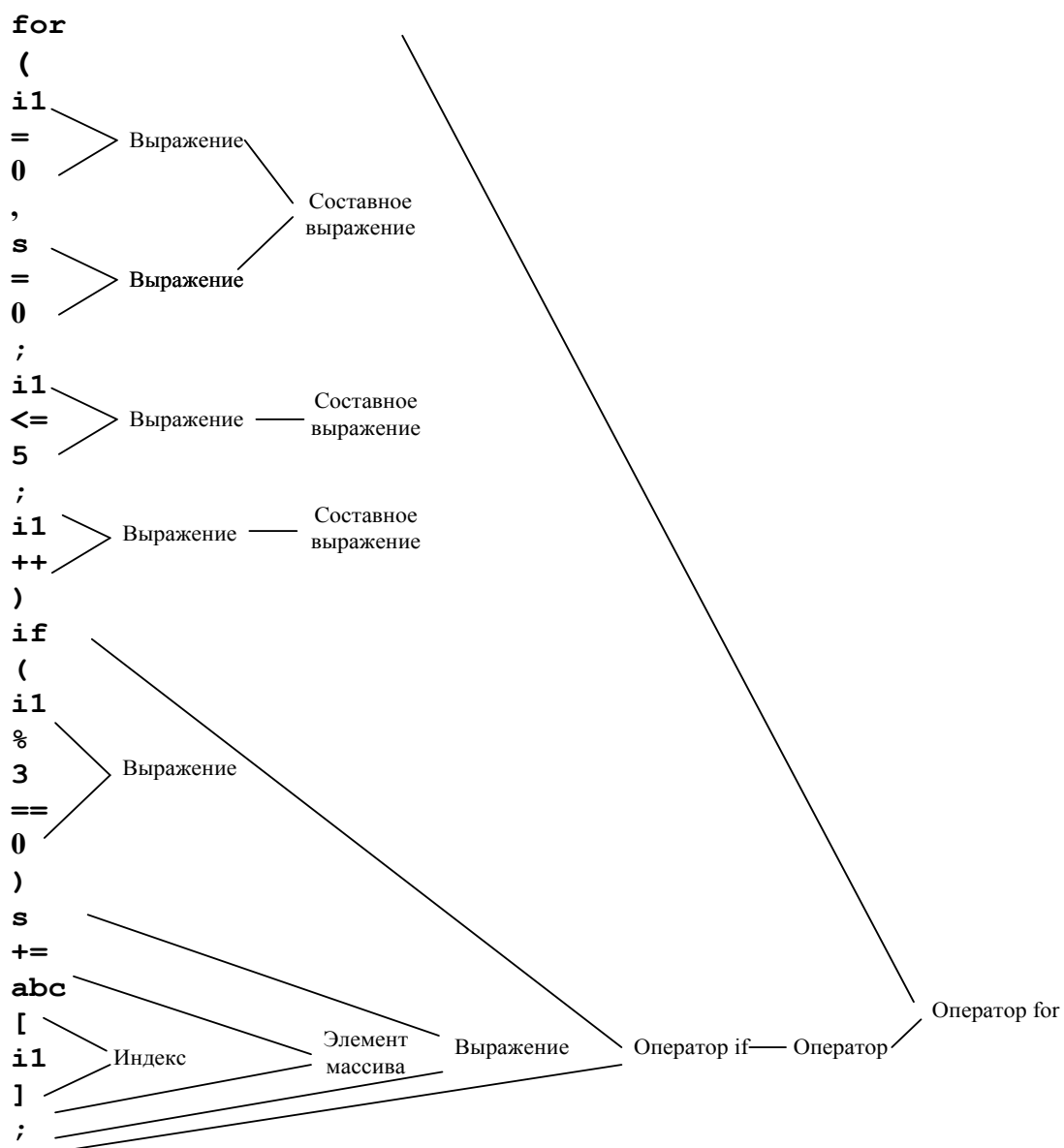


Рис. П.1. Последовательность синтаксического анализа оператора

[Оглавление](#)

**Задание 1.6.** Выполните построение троек методом Рутисхаузера для выражения  $r12 * (h+gn) + f * tt1$ .

**Решение.**

Сначала переписываем выражение в полной скобочной записи

$(r12 * (h+gn)) + (f * tt1)$ .

Построение троек методом Рутисхаузера показано в табл. П.4.

**Таблица П.4** Построение троек из выражения методом Рутисхаузера

№ шага	Строка / Индексы	Тройка
1	S: $( r12 * ( h + gn ) ) + ( f * tt1 )$ N: 0 1 2 <u>1 2 3 2 3</u> 2 1 2 1 2 1 0 1 0	$T_1 = h+gn$
2	S: $( r12 * T_1 ) + ( f * tt1 )$ N: 0 <u>1 2 1 2 1</u> 0 1 2 1 2 1 0	$T_2 = r12 * T_1$
3	S: $T_2 + ( f * tt1 )$ N: 0 <u>1 0 1 2 1</u> 2 <u>1</u> 0	$T_3 = f * tt1$
4	S: $T_2 + T_3$ N: <u>0 1 0 1 0</u>	$T_4 = T_2 + T_3$

**Задание 2.9.** Разработайте грамматику, позволяющую описывать строки, содержащие запись вещественных чисел в форме с фиксированной точкой с обязательным присутствием знака, например,

$-0.1, +25.356, +0.0$

К какому типу по Хомскому относится данная грамматика? Почему?

**Решение.**

Грамматика вещественных чисел может быть описана следующим образом:

- а)  $\langle \text{Вещественное} \rangle ::= \langle \text{Знак} \rangle \langle \text{ЦелоеБезЗнака} \rangle . \langle \text{ЦелоеБезЗнака} \rangle$ ,
- б)  $\langle \text{ЦелоеБезЗнака} \rangle ::= \langle \text{Цифра} \rangle \langle \text{ЦелоеБезЗнака} \rangle | \langle \text{Цифра} \rangle$ ,
- в)  $\langle \text{Цифра} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$ ,
- г)  $\langle \text{Знак} \rangle ::= + | -$ .

Грамматика относится к третьему типу по Хомскому, поскольку самое сложное рекурсивное правило содержит правостороннюю рекурсию.

**Задание 2.10.** Выполните разбор строки, содержащей запись числа  $-4.5$ , по левостороннему нисходящему методу, используя правила, сформулированные в задании 2.9.

**Решение.**

Последовательность разбора показана в табл. П.5.

**Таблица П.5** Последовательность разбора левосторонним нисходящим методом

№ шага	Распознано	Распознаваемая строка	Строка правил	Действие
1		$-4.5$	<Вещественное>	Подстановка правила а1
2		$-4.5$	<Знак><ЦБЗ>. <ЦБЗ>	Подстановка правила г1
		$-4.5$	+<ЦБЗ>.<ЦБЗ>	Ошибка, возврат к шагу 2
3		$-4.5$	<Знак><ЦБЗ>. <ЦБЗ>	Подстановка правила г2
		$-4.5$	-<ЦБЗ>.<ЦБЗ>	Символ распознан и удален
4	—	$4.5$	<ЦБЗ>.<ЦБЗ>	Подстановка правила б1
5	—	$4.5$	<Цифра><ЦБЗ>. <ЦБЗ>	Подстановка правила в1
	—	$4.5$	0 <ЦБЗ>.<ЦБЗ>	Ошибка, возврат к шагу 5
6	—	$4.5$	<Цифра><ЦБЗ>. <ЦБЗ>	Подстановка правила в2
	—	$4.5$	1 <ЦБЗ>.<ЦБЗ>	Ошибка, возврат к шагу 6
7	—	$4.5$	<Цифра><ЦБЗ>. <ЦБЗ>	Подстановка правила в3
	—	$4.5$	2 <ЦБЗ>.<ЦБЗ>	Ошибка, возврат к шагу 7
8	—	$4.5$	<Цифра><ЦБЗ>. <ЦБЗ>	Подстановка правила в4
	—	$4.5$	3 <ЦБЗ>.<ЦБЗ>	Ошибка, возврат к шагу 8

9	—	4.5	<Цифра>.<ЦБЗ>. <ЦБЗ>	Подстановка пра- вила в5
	—	4.5	4.<ЦБЗ>.<ЦБЗ>	Символ распознан
10	—4	.5	<ЦБЗ>.<ЦБЗ>	Подстановка пра- вила б1
11..20	—4	.5	<Цифра>.<ЦБЗ>. <ЦБЗ>	Подстановка пра- вила в1—в10
	—4	.5	0..9.<ЦБЗ>.<ЦБЗ>	Ошибки и возвраты к шагу 11, потом к 10
21	—4	.5	<ЦБЗ>.<ЦБЗ>	Подстановка пра- вила б2
22..31	—4	.5	<Цифра>.<ЦБЗ>	Подстановка пра- вил в1—в10
	—4	.5	0..9.<ЦБЗ>.<ЦБЗ>	Ошибки и возвраты к шагу 23
32	—4	.5	<Цифра>.<ЦБЗ>	Ошибка и возврат к шагу 4
33	—	4.5	<ЦБЗ>.<ЦБЗ>	Подстановка пра- вила б2
34-37	—	4.5	<Цифра>.<ЦБЗ>	Подстановка пра- вил в1—в4
	—	4.5	0..3.<ЦБЗ>	Ошибки и возвраты к шагу 35
38	—	4.5	<Цифра>.<ЦБЗ>	Подстановка пра- вила в5
		4.5	4.<ЦБЗ>	Символы распознаны
39	—4.	5	<ЦБЗ>	Подстановка пра- вила б1
<i>Примечание.</i> Далее разбор выполняется аналогично разбору в примере 2.5, на- чиная с шага 10				

**Задание 2.11.** Выполните разбор строки, содержащей запись числа —4.5, по левостороннему восходящему методу, используя соответствующим образом скорректированные правила, сформулированные в задании 2.9.

**Решение.**

Необходимость корректировки правил вызвана тем, что предлагаемый метод разбора не позволяет использовать правила с правосторонней рекурсией. После корректировки правила будут выглядеть следующим образом:

- а) <Вещественное>::=<Знак><ЦелоеБезЗнака>.<ЦелоеБезЗнака>,
- б) <ЦелоеБезЗнака>::=<ЦелоеБезЗнака><Цифра>|<Цифра>,



в) **<Цифра> ::= 0|1|2|3|4|5|6|7|8|9,**

г) **<Знак> ::= +| - .**

Последовательность разбора числа показана в табл. П.6.

**Таблица П.6** Последовательность разбора числа левосторонним восходящим методом

№ шага	Распознаваемая строка	Основа	Операция
1	−4.5	−	Свертка по правилу г2
2	<b>&lt;Знак&gt; 4.5</b>	<b>&lt;Знак&gt;</b>	Нет правила для свертки. Формирование следующей основы
3	<b>&lt;Знак&gt; 4.5</b>	<b>&lt;Знак&gt; 4</b>	То же
4	<b>&lt;Знак&gt; 4.5</b>	<b>&lt;Знак&gt; 4.</b>	»
5	<b>&lt;Знак&gt; 4.5</b>	<b>&lt;Знак&gt; 4.5</b>	»
6	<b>&lt;Знак&gt; 4.5</b>	<b>4</b>	Свертка по правилу в5
7	<b>&lt;Знак&gt; &lt;Цифра&gt;.5</b>	<b>&lt;Знак&gt;</b>	Нет правила для свертки. Формирование следующей основы
8	<b>&lt;Знак&gt; &lt;Цифра&gt;.5</b>	<b>&lt;Знак&gt; &lt;Цифра&gt;</b>	То же
9	<b>&lt;Знак&gt; &lt;Цифра&gt;.5</b>	<b>&lt;Знак&gt; &lt;Цифра&gt;.</b>	»
10	<b>&lt;Знак&gt; &lt;Цифра&gt;.5</b>	<b>&lt;Знак&gt; &lt;Цифра&gt;.5</b>	»
11	<b>&lt;Знак&gt; &lt;Цифра&gt;. 5</b>	<b>&lt;Цифра&gt;</b>	Свертка по правилу б2
12	<b>&lt;Знак&gt; &lt;ЦБЗ&gt;.5</b>	<b>&lt;Знак&gt;</b>	Нет правила для свертки. Формирование следующей основы
13	<b>&lt;Знак&gt; &lt;ЦБЗ&gt;.5</b>	<b>&lt;Знак&gt; &lt;ЦБЗ&gt;</b>	То же
14	<b>&lt;Знак&gt; &lt;ЦБЗ&gt;.5</b>	<b>&lt;Знак&gt; &lt;ЦБЗ&gt;.</b>	»
15	<b>&lt;Знак&gt; &lt;ЦБЗ&gt;.5</b>	<b>&lt;Знак&gt; &lt;ЦБЗ&gt;.5</b>	»
16	<b>&lt;Знак&gt; &lt;ЦБЗ&gt;.5</b>	<b>&lt;ЦБЗ&gt;</b>	»
17	<b>&lt;Знак&gt; &lt;ЦБЗ&gt;.5</b>	<b>&lt;ЦБЗ&gt;.</b>	»
18	<b>&lt;Знак&gt; &lt;ЦБЗ&gt;.5</b>	<b>&lt;ЦБЗ&gt;.5</b>	»
19	<b>&lt;Знак&gt; &lt;ЦБЗ&gt;.5</b>	<b>.</b>	»
20	<b>&lt;Знак&gt; &lt;ЦБЗ&gt;.5</b>	<b>.5</b>	»
21	<b>&lt;Знак&gt; &lt;ЦБЗ&gt;.5</b>	<b>5</b>	Свертка по правилу в6
22	<b>&lt;Знак&gt; &lt;ЦБЗ&gt;.&lt;Цифра&gt;</b>	<b>&lt;Знак&gt;</b>	Нет правила для свертки. Формирование следующей основы
23	<b>&lt;Знак&gt; &lt;ЦБЗ&gt;.&lt;Цифра&gt;</b>	<b>&lt;Знак&gt; &lt;ЦБЗ&gt;</b>	То же
24	<b>&lt;Знак&gt; &lt;ЦБЗ&gt;.&lt;Цифра&gt;</b>	<b>&lt;Знак&gt; &lt;ЦБЗ&gt;.</b>	»

25	<Знак> <ЦБЗ>.<Цифра>	<Знак> <ЦБЗ>. <Цифра>	»
26	<Знак> <ЦБЗ>.<Цифра>	<ЦБЗ>	»
27	<Знак> <ЦБЗ>.<Цифра>	<ЦБЗ>.	»
28	<Знак> <ЦБЗ>.<Цифр>	<Цбз>.<Цифра>	»
29	<Знак> <ЦБЗ>.<Цифра>	.	»
30	<Знак> <ЦБЗ>.<Цифра>	. <Цифра>	»
31	<Знак> <ЦБЗ>.<Цифра>	<Цифра>	Свертка по правилу б2
32	<Знак> <ЦБЗ>.<ЦБЗ>	<Знак>	Нет правила для свертки. Формирование следующей основы
33	<Знак> <ЦБЗ>.<ЦБЗ>	<Знак> <ЦБЗ>	То же
34	<Знак> <ЦБЗ>.<ЦБЗ>	<Знак> <ЦБЗ>.	»
35	<Знак> <ЦБЗ>.<ЦБЗ>	<Знак> <ЦБЗ>. <Цифра>	Свертка по правилу а1
36	<Вещественное>		Конец

**Задание 3.5.** Постройте синтаксическую диаграмму и таблицу конечного автомата. На вход автомата поступает последовательность непустых строк, содержащих группы букв и цифр, при этом цифры есть обязательно. Каждая строка завершается символом «#». Автомат должен распознавать и допускать строки, в которых встречается только четное количество цифр в группах, например, допустимы строки:

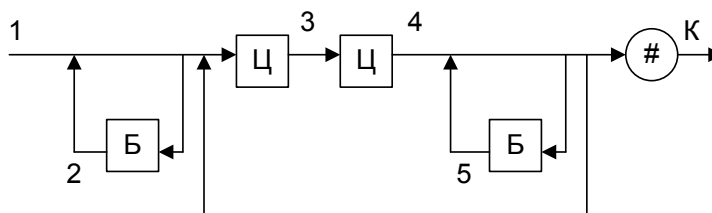
а) fhgfhfj f56787979tuytuuyu45cvbcbv#

б) 56плорл354356#

в) 6789#

**Решение.**

Синтаксическая диаграмма приведена на рис. П.2, а таблица распознающего автомата – в виде табл. П.7.



**Рис. П.2.** Синтаксическая диаграмма распознающего автомата:

Б – буква; Ц – цифра; 1 – исходное состояние; 2 – распознана буква в начале строки; 3 – распознана первая цифра; 4 – распознана вторая цифра; 5 – распознана буква после цифр;  
К – конец строки

**Таблица П.7** Таблица переходов распознающего автомата

Состояние $q$	Входной символ $s$			
	Б	Ц	#	Другие символы
1	2	3	E	E
2	2	3	E	E
3	E	4	E	E
4	5	3	К	E
5	5	3	К	E

Анализ таблицы показывает, что полученный автомат не минимален, так как есть совпадающие строки (1 и 2, 4 и 5). Удаляем 2-ю и 5-ю строки и заменяем состояния: 2 – на 1, 5 – на 4. В результате получаем минимальный автомат (табл. П.8)

**Таблица П.8** Таблица переходов минимального распознающего автомата

Состояние $q$	Входной символ $s$			
	Б	Ц	#	Другие символы
1	1	3	E	E
3	E	4	E	E
4	4	3	К	E

**Задание 4.7.** Дана грамматика, описывающая упрощенный вариант сложных выражений C++:

**<Сложное выражение> ::= <Сложное выражение>, <Выражение> | <Выражение>**

**<Выражение> ::= <Выражение>+<Ид>|<Выражение>-<Ид>|<Ид>**

Приведите последовательность грамматического разбора левосторонним восходящим методом для сентенциальной формы

**<Ид>+<Ид>-<Ид> , <Ид>-<Ид>, <Ид>-<Ид>**

левосторонним восходящим методом.

**Решение:**

Введем обозначения:

$C$  – <Сложное выражение>;

$V$  – <Выражение>;

Ид – <Идентификатор>.

Тогда правила будут выглядеть так:

**$C ::= C, V \mid V$**

**$V ::= V + \text{Ид} \mid V - \text{Ид} \mid \text{Ид}$**

Грамматика принадлежит к 3-му типу по Хомскому, так как:

- правила в левой части содержат единственный символ – нетерминал, соответственно это не грамматики 0 и 1 типов;
- в правой части правил отсутствует вложенная рекурсия, соответственно это не грамматика 2 типа.

Основу в сентенциальной форме ищем слева направо, правила подбираем по правой части снизу–вверх и слева–направо (основа отмечена красным цветом). Тогда последовательность разбора будет выглядеть следующим образом:

- 1) **Ид** + Ид – Ид, Ид – Ид, Ид – Ид
- 2) **V + Ид** – Ид, Ид – Ид, Ид – Ид
- 3) **V – Ид**, Ид – Ид, Ид – Ид
- 4) **V**, Ид – Ид, Ид – Ид
- 5)  $C$ , **Ид** – Ид, Ид – Ид
- 6) **C, V** – Ид, Ид – Ид
- 7)  $C - \text{Ид}$ , Ид – Ид
- 8)  $C - V$ , Ид – Ид
- 9)  $C - C$ , **Ид** – Ид
- 10)  $C - C, V$  – Ид
- 11)  $C - C - \text{Ид}$
- 12)  $C - C - V$

## тупик... возврат к 10

## ТУПИК... ВОЗВРАТ К 6

## ТУПИК... ВОЗВРАТ К 6

## ТУПИК... ВОЗВРАТ К 6

## ТУПИК... ВОЗВРАТ К 6

## ТУПИК... ВОЗВРАТ К 6

## ТУПИК... ВОЗВРАТ К 6

## ТУПИК... ВОЗВРАТ К 6

## ТУПИК... ВОЗВРАТ К 6

## ТУПИК... ВОЗВРАТ К 6

ТУПИК... ВОЗВРАТ К 20

ТУПИК... ВОЗВРАТ К 20

ТУПИК... ВОЗВРАТ К 20

ТУПИК... ВОЗВРАТ К 20

$$\mathbf{I_d} \rangle + \langle \mathbf{I_d} \rangle - \langle \mathbf{I_d} \rangle, \langle \mathbf{I_d} \rangle - \langle \mathbf{I_d} \rangle, \langle \mathbf{I_d} \rangle - \langle \mathbf{I_d} \rangle.$$

**Решение.**

**Решение.**

**Решение.**

**Решение.**

**Решение.**

Последовательность разбора заданного выражения стековым методом представлена в табл. П.10.

**Таблица П.10** Разбор выражения стековым методом

Содержимое стека	Анализируемые символы	Отношение	Операция	Тройка	Результат свертки
►	Ид +	<.	Перенос	—	—
► Ид +	Ид –	<.	Свертка	R1 = Ид + Ид	<Выражение>
►	R1 –	<.	Перенос	—	—
► R1 –	Ид,	>.	Свертка	R2 = R1 – Ид	<Выражение>
►	R2,	<.	Перенос	—	—
► R2,	Ид –	<.	Перенос	—	—
► R2, Ид –	Ид,	>.	Свертка	R3 = Ид – Ид	<Выражение>
► R2,	R3,	>.	Свертка	R4 = R2, R3	<Сложное выражение>
►	R4,	<.	Перенос	—	—
► R4,	Ид –	<.	Перенос	—	—
► R4, Ид –	Ид,	>.	Свертка	R5 = Ид – Ид	<Выражение>
► R4,	R5 ◀	>.	Свертка	R6 = R4, R5	<Сложное выражение>
►	R6 ◀	Выход	—	—	—

**Задание 4.9.** Для грамматики, определенной в задании 4.7, приведите последовательность выполнения грамматического разбора с использованием польской записи для сентенциальной формы

$\langle \text{Ид} \rangle + \langle \text{Ид} \rangle - \langle \text{Ид} \rangle$  ,  $\langle \text{Ид} \rangle - \langle \text{Ид} \rangle$  ,  $\langle \text{Ид} \rangle - \langle \text{Ид} \rangle$  .

**Решение:**

Для получения польской записи на основе грамматики получим таблицу генератора польской записи (табл. П.11).

**Таблица П.11** Таблица генератора записи

Верхний символ в стеке операций $\eta$	Текущий символ $\xi$		
	+	,	◀
▶	I	I	Выход
+	II	IV	.>
,	I	II	.>

1-й этап – построение польской записи показано в табл. П.12.

**Таблица П.12** Построение польской записи

Стек операций	Символ	Действие	Команда
▶	Ид	—	$K_{Ид}$
▶	+	I	—
▶ +	Ид	—	$K_{Ид}$
▶ +	—	II	$K_+$
▶ —	Ид	—	$K_{Ид}$
▶ —	,	IV	$K_-$
▶ ,	,	I	—
▶ ,	Ид	—	$K_{Ид}$
▶ ,	—	I	—
▶ , —	Ид	—	$K_{Ид}$
▶ , —	,	IV	$K_{-,}$
▶ , ,	,	II	$K_{,,}$
▶ , ,	Ид	—	$K_{Ид}$
▶ , ,	—	I	—
▶ , , —	Ид	—	$K_{Ид}$
▶ , , —	◀	IV	$K_{-,}$
▶ , , ,	◀	IV	$K_{,,}$
▶ , , ,	◀	Выход	—

В результате получена польская запись выражения

$K_{Ид} K_{Ид} K_+ K_{Ид} K_- K_{Ид} K_{Ид} K_- K_{Ид} K_{Ид} K_- K_{Ид} K_{Ид} K_-$ , или

$Ид Ид + Ид - Ид Ид - , Ид Ид - ,$

2-й этап – выполнение польской записи показано в табл. П.13.

**Таблица П.13** Выполнение польской записи

Стек операндов	Команда	Тройка
$\emptyset$	$K_{Ид}$	—
$K_{Ид}$	$K_{Ид}$	—
$K_{Ид} K_{Ид}$	$K_+$	$T_1 = K_{Ид} + K_{Ид}$
$T_1$	$K_{Ид}$	—
$T_1 K_{Ид}$	$K_-$	$T_2 = T_1 - K_{Ид}$
$T_2$	$K_{Ид}$	—
$T_2 K_{Ид}$	$K_{Ид}$	—
$T_2 K_{Ид} K_{Ид}$	$K_-$	$T_3 = K_{Ид} - K_{Ид}$
$T_2 T_3$	$K_,$	$T_4 = T_2, T_3$
$T_4$	$K_{Ид}$	—
$T_4 K_{Ид}$	$K_{Ид}$	—
$T_4 K_{Ид} K_{Ид}$	$K_-$	$T_5 = K_{Ид} - K_{Ид}$
$T_4 T_5$	$K_,$	$T_6 = T_4, T_5$
$T_6$	Конец	—

**Задание 5.5.** Выполните ручную оптимизацию следующих фрагментов программы:

- а) `c=-b*b*b+(g-1)*b;`
- б) `a=b*128;`
- в) `a=b+c; b=a-5; a=b-d;`
- г) `a=0;c=5; while (b&&(!b))a+=c;`
- д) `for(s=0,i=0;i<=10;i++)a[i]=k*5;`
- е) `a=5;b=(a+1)*j;`



**Решение.**

а)  $c = -b * b * b + (g - 1) * b; \Rightarrow c = b * (g - b * b - 1);$  (выносим  $b$  за скобки);

б)  $a = b * 128; \Rightarrow a = b \ll 7;$  (если  $b$  – целочисленная переменная, то заменяем операцию умножения на степень двойки операцией сдвига);

в)  $a = b + c; b = a - 5; a = b - d; \Rightarrow b += b + c - 5; a = b - d;$  (подставляем первое выражение во второе, тогда первое можно удалить);

г)  $a = 0; c = 5; \text{while } (b \& \& (!b)) a += c; \Rightarrow a = 0; c = 5;$  (цикл можно удалить, так как он не выполнится ни разу);

д)  $\text{for } (s = 0, i = 0; i \leq 16; i++) s += k * 4; \Rightarrow$

$s = 16 * 4 * k; \Rightarrow s = 64 * k \Rightarrow s = k \ll 6;$  (сначала суммирование одинаковых элементов заменяем умножением, потом умножаем константы и в завершение, если  $k$  – целое число, заменяем умножение на степень двойки операцией сдвига);

е)  $a = 5; b = (a + 1) * j; \Rightarrow a = 5; b = 6 * j;$  (выполняем операцию над константами).