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 издание

Учебное пособие по дисциплине
Машинно-зависимые языки и основы компиляции

Москва

УДК 004.4'41

Рецензенты:

доцент, к.т.н., Наталья Владимировна Новик

Иванова Г.С., Ничушкина Т.Н.

Основы конструирования компиляторов: Учебное пособие по дисциплине «Машинно-зависимые языки и основы компиляции». Учебное пособие. – М.: МГТУ имени Н.Э. Баумана, 2020. – 87 с.

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 классификации 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 2 курса кафедры Компьютерные системы и сети направлений 09.03.01 Информатика и вычислительная техника и 09.93.03 Прикладная информатика, изучающих дисциплину Машинно-зависимые языки и основы компиляции.

Рекомендовано учебно-методической комиссией факультета «Информатика и системы управления» МГТУ им. Н.Э. Баумана

Учебное пособие

Иванова Галина Сергеевна

Ничушкина Татьяна Николаевна

Основы конструирования компиляторов.

Оглавление

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 4 дисциплины «Машинно-зависимые языки и основы компиляции» и подготовки к выполнению домашнего задания указанного модуля.

Цель учебного пособия – закрепление теоретических знаний и получение практических навыков, необходимых для построения лексических и синтаксических анализаторов формальных языков программирования, формирование умений построения распознающих конечных автоматов и автоматов с магазинной памятью, а также знакомство с методами грамматического разбора.

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 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.

В процессе изучения материала пособия необходимо внимательно разбирать предлагаемые примеры, также целесообразно синтезировать и прорабатывать 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.

Оглавление

Введение

Построение транслирующих и компилирующих программ базируется на глубоком понимании теории формальных языков. Однако соответствующий раздел математики сложен для усвоения, и, в первую очередь, это связано с большим количеством 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 методами рекурсивного спуска и стековым;
- польская запись и алгоритм Бауэра-Замельзона.

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

1 Структура компилятора

1.1 Основные понятия и определения

Любой язык (формальный или естественный) обязательно подчиняется определенным правилам, которые определяют его синтаксис и семантику. *Синтаксис* – это совокупность правил, определяющих 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 специалисту 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.

1. ***Лексический анализ*** – процесс преобразования исходного текста в строку однородных символов. Каждый символ результирующей строки – *токен* соответствует языковой единице (символу или «слову» языка) – *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 2 типа лексем:

а) лексемы, соответствующие символам алфавита языка, таким как «Служебные слова» и «Служебные символы»;

б) лексемы, соответствующие базовым понятиям языка, таким как «Идентификатор» и «Литерал».

Пример. Выполнить лексический анализ предложения языка Паскаль:

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

Отдельные части – «слова» предложения разделяются:

1) пробелами;

2) 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 (таблица 1) и, возможно, проверить таблицы переменных (таблица 2) и литералов (таблица 3).

Таблица 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 2 – Таблица идентификаторов переменных

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

Таблица 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 варианте анализ лексем встроен в синтаксический анализ. Единой программы сканера при этом не создают, 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).

2. ***Синтаксический анализ*** – процесс распознавания конструкций языка в строке токенов, полученной после работы сканера, или формируемой по мере разбора предложения. Главным результатом является отождествление строки или подстроки с некоторой конструкцией и, при необходимости, информация об обнаруженных при этом ошибках.

Пример. Выполнить синтаксический анализ строки токенов, описанной таблицей 1.

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

3. ***Семантический анализ*** – процесс распознавания/проверки смысла конструкции. По результатам распознавания строится последовательность, приближенная к последовательности операторов будущей программы и выполняются предусмотренные проверки правильности программы.

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

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

5. ***Генерация и оптимизация объектного кода*** – процесс формирования программы на выходном языке, которая семантически эквивалентна исходной программе. На этом этапе также обычно выполняется оптимизация генерируемого кода.

***

Лексический и синтаксический анализ предполагают выполнение грамматического разбора. 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 © E_R, где E_L, E_R – левый и правый операнды, © – операция.

Основной проблемой при этом является необходимость учета *приоритетов* операций. Например, для выражения:

`a+b*c` должны быть построены тройки: T_1 = `b*c`, `d` = `a+`T_1.

Исторически первым для решения этой задачи 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]≠'_'

Если S[J] = '(' или S[J] = <операнд>

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. Получить тройки для выражения: (((a+b)*c)+d)/k.

Процесс построения троек выглядит следующим образом:

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

Основной недостаток метода – требование полной скобочной записи. Как правило, для получения полной скобочной записи 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 практически не применим для разбора не связанных с выражениями операторов языков программирования. Решение проблем компиляции долгое время было не простым. 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.

С тех пор была создана математическая теория формальных языков, предложен соответствующий инструментарий, в результате чего небольшой компилятор может стать темой курсовой работы студента.

Контрольные вопросы

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

Ответ.

2. Назовите основные этапы процесса компиляции. Уточните задачи каждого этапа.

Ответ

3. В чем заключается метод Рутисхаузера?

Ответ

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

Ответ

2 Основные положения теории формальных языков

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: α, β, γ, … Строка, не содержащая символов, называется *пустой* и далее будет обозначаться латинской буквой *e*. В литературе с этой целью также используют греческие буквы ε или λ.

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: $A^* = A^+ \cup \{e\}$.

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

Задать язык $L(A)$ значит:

- 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 буквам латинского алфавита, а в правилах конкретного языка – записывают их имена в угловых скобках, например **<Целое>**.

Полное описание грамматики представляет собой набор правил, который определяет 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 – вспомогательный алфавит, символы которого обозначают допустимые понятия языка, $V_T \cap V_N = \varnothing$; $V = V_T \cup V_N$ – словарь грамматики;

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

$S \in V_N$ – 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. Определим грамматику записи десятичных чисел G_0:

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

V_N = {<Целое>, <ЦелоеБезЗзнака>, <Цифра>, <Знак>};

P = {<Целое> → <Знак><Целое без знака>,

<Целое> → <Целое без знака>,

<ЦелоеБезЗнака> → <Цифра><ЦелоеБезЗнака>,

<ЦелоеБезЗнака> → <Цифра>,

<Цифра> → 0, <Цифра> → 1, <Цифра> → 2,

<Цифра> → 3, <Цифра> → 4, <Цифра> → 5,

<Цифра> → 6, <Цифра> → 7,<Цифра> → 8,

<Цифра> → 9,

<Знак> → +, <Знак> → - };

S = <Целое>.

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

<ЦелоеБезЗнака> → <Цифра>**<ЦелоеБезЗнака>.**

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:

<Целое>::=<Знак><ЦелоеБезЗнака>|<ЦелоеБезЗнака>,

<ЦелоеБезЗнака>::=<Цифра><ЦелоеБезЗнака>|<Цифра>,

<Цифра>::=0|1|2|3|4|5|6|7|8|9,

<Знак>::=+|-.

Расширенная форма Бэкуса-Наура. РФБН – 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:

« = » – «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 возможет повтор ноль раз);

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

«(…)» – 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, если вспомнить определение графа, это очевидно не граф. Синтаксическая диаграмма – наглядное графическое представление правил продукции. На ней терминальные символы рисуют в круглых или овальных блоках, нетерминальные – в прямоугольных или квадратных. Стрелками показывают пути формирования допустимых строк из терминальных и нетерминальных символов (см. далее рисунки 5-9).

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

2.2 Понятие грамматического разбора

Распознавание принадлежности строки конкретному языку осуществляется его выводом из аксиомы. *Вывод* представляет собой последовательность

подстановок, при выполнении которых левая часть правила заменяется правой.

Аксиому грамматики и строки, получаемые 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 «⇒», справа от которого указывают номер подстановки.

Пример. 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 (рисунок 1).

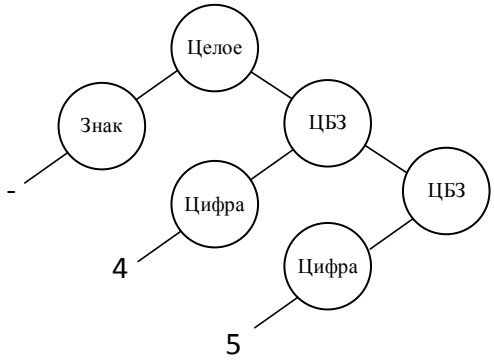


Рисунок 1 – Синтаксическое дерево

Неоднозначные грамматики. Существуют грамматики, в которых для одной строки можно построить несколько деревьев. Такие грамматики называются *неоднозначными*. Разбор неоднозначных грамматик затруднен, по-

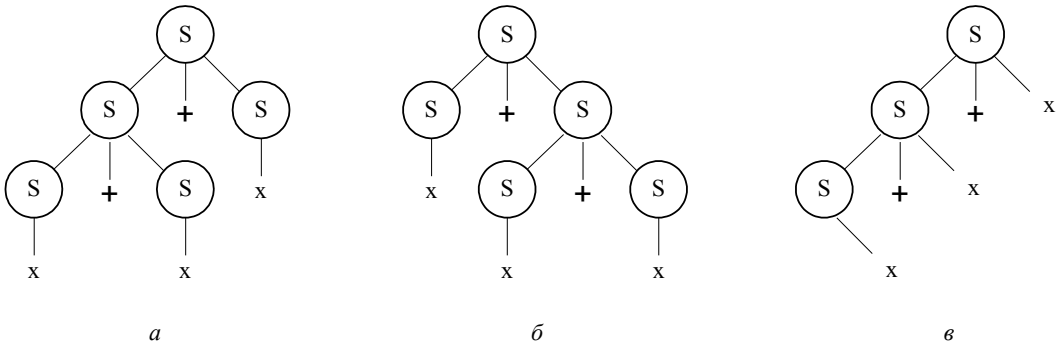
этому их, если возможно, преобразуют в однозначные или ограничивают правилами.

***Пример* 1.** Строка x+x+x, порождаемая грамматикой с правилами**:**

$S \rightarrow S + S$**,**

$S \rightarrow x$,

имеет два дерева разбора (рисунок 2, *а*–*б*).



а *б* *в*

Рисунок 2 – Деревья разбора неоднозначной грамматики (*а*, *б*) и единственное дерево соответствующей однозначной грамматики (*в*)

Описывающая тот же язык однозначная грамматика (рисунок 2, *в*) использует правила:

$S \rightarrow S + x$**,**

$S \rightarrow x$**.**

***Пример* 2.** 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* – процедура построения синтаксического дерева для конкретного предложения языка. Построение такого дерева позволяет однозначно доказать, что анализируемая

строка языка является *допустимой*, т.е. принадлежит рассматриваемому языку.

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 (разбор «сверху-вниз»)

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

Исходно строка правил содержит 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 осуществляют возврат к ближайшему шагу замены левого нетерминала, на котором существует возможность выбора другого правила для подстановки.

Разбор завершается успешно, если все символы строки оказываются распознанными и удаленными из разбираемой строки и строка правил также оказывается пустой.

Если все альтернативные замены правил проверены, но часть символов строки остались нераспознанными, то разбираемая строка не соответствует аксиоме, т.е. содержит ошибку. Аналогично, 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. Разобрать строку «-45».

Правила грамматики:

а) **<Целое> ::= <Знак><ЦелоеБезЗнака>|<ЦелоеБезЗнака>**,

б) **< ЦелоеБезЗнака> ::= <Цифра>< ЦелоеБезЗнака>|<Цифра>**,

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

г) **<Знак> ::= +|- .**

Для уменьшения длины строки правил заменим имя нетерминала <ЦелоеБезЗнака> сокращением <ЦБЗ>. Тогда последовательность разбора будет выглядеть следующим образом (полужирным выделен заменяемый на следующем шаге нетерминал):

№ 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
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

Оглавление

	-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 продолжение разбора 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 приходилось возвращаться и выбирать альтернативные правила. В таблице часть шагов, приводящих к возвратам, в очевидных случаях сгруппированы, что обозначено двумя точками, например 0..4 означает, что подстановки цифр 1..4, приводят к ошибкам и возвратам.

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 осуществлении грамматического разбора 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. Выполнить грамматический разбор строки «-45».

Правила грамматики:

а) **<Целое> ::= <Знак><ЦелоеБезЗнака>|<ЦелоеБезЗнака>,**

б) **<ЦелоеБезЗнака> ::= <ЦелоеБезЗнака><Цифра>|<Цифра>,**

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

г) **<Знак> ::= +|- .**

Будем использовать сокращение <ЦБЗ> для имени нетерминала <ЦелоеБезЗнака>, как в предыдущем разделе. Тогда последовательность сентенциальных форм, полученных рассматриваемым методом, будет выглядеть следующим образом (полужирным на каждом шаге выделена основа):

№ шага	Распознаваемая строка	Основа	Операция
1	-45	-	Свертка по правилу г1
2	<Знак> **45**	4	Свертка по правилу в5
3	<Знак> <**Цифра**> 5	<Цифра>	Свертка по правилу б2
4	**<Знак> <Цбз>** 5	<Знак> <Цбз>	Свертка по правилу а1
5	<Целое> 5	Тупик!	**Возврат к шагу 4**
6	<Знак> **<Цбз>** 5	<Цбз>	Свертка по правилу а2
7	<Знак><Целое>5	Тупик!	**Возврат к шагу 6**
8	<Знак> <Цбз> **5**	5	Свертка по правилу г6
9	**<Знак> <Цбз>**<Цифра>	<Знак> <Цбз>	Свертка по правилу а1
10	<Целое> <Цифра>	Тупик!	**Возврат к шагу 9**
11	<Знак> **<Цбз>**<Цифра>	<Цбз>	Свертка по правилу а2
12	<Знак> <Целое> <Цифра>	Тупик!	**Возврат к шагу 11**
13	<Знак> **<Цбз> <Цифра>**	<Цбз> <Цифра>	Свертка по правилу б1
14	**<Знак> <Цбз>**	<Знак> <Цбз>	Свертка по правилу а1
15	<Целое>	Конец	-

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

Таким образом реализация обоих рассмотренных методов в общем случае 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 осуществления грамматического разбора предложений соответствующих языков.

2.3 Расширенная классификация грамматик Хомского

В зависимости от вида правил различают грамматики следующих типов:

тип 0 – грамматики фразовой структуры или грамматики «без ограничений», грамматики используют правила вида:

$\alpha \to \beta$, 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 грамматики используют правила вида:

$\alpha \boldsymbol{X} \beta \to \alpha \boldsymbol{x} \beta$, где $X \in V_N$, $x \in V_T$, $\alpha, \beta \in V^*$, $V = V_T \cup V_N$, причем α, β одновременно не являются пустыми, а значит возможность подстановки x вместо символа X определяется присутствием хотя бы одной из подстрок α и β, т. е. *контекста*, откуда и происходит название типа грамматик;

тип 2 – контекстно-свободные грамматики (КС-грамматики) используют правила вида:

$A \to \beta$, 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 грамматики используют правила вида:

$A \to \alpha$, $A \to \alpha B$ или $A \to B\alpha$, 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 описываю правила синтаксиса самых простых языков.

Классификация построена по правилам иерархии, т.е. грамматики третьего типа являются частным случаем грамматик второго типа и т.д. (рисунок 3).

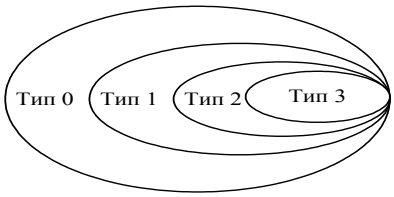


Рисунок 3 – Классификация типов грамматик

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

Грамматики второго типа отличают от грамматик третьего типа по наличию в правилах явной или косвенной *вложенной* рекурсии, т.е. правил вида $A \to \alpha\, A\, \beta$, где α и $\beta \in V^+$, т.е. подстроки α и β 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 $\alpha\, \boldsymbol{X}\, \beta \to \alpha\, \boldsymbol{x}\, \beta$, где $\alpha, \beta \in V^*$, в которых только одна из подстрок α или β может быть пустой.

Пример. Язык скобок, описываемый правилами:

а) $\boldsymbol{S \to (S)}$,

б) $\boldsymbol{S \to SS}$,

в) $\boldsymbol{S \to e}$.

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*. Смысл леммы заключается в следующем: в строке регулярного языка всегда можно найти непустую подстроку, повторение которой произвольное количество раз порождает новые строки того же языка.

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

Распознаватели. Проверка соответствия строк языку осуществляется специальной программой – *распознавателем*. Распознаватели могут использоваться для определения языка также, 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

1. Определите, что такое алфавит языка.
   Ответ.
2. Дайте определение формального языка. Почему формальные языки обычно не определяют перечислением допустимых предложений?
   Ответ.
3. Дайте определение формальной грамматики.
   Ответ.
4. Что такое «Форма Бэкуса-Наура»? Для чего она используется? Чем отличается от «Расширенной формы Бэкуса-Наура»?
   Ответ.
5. Определите цель грамматического разбора предложений языка.
   Ответ.
6. В чем заключается левосторонний восходящий грамматический разбор? Назовите, в каких случаях он не применим и определите его основной недостаток.
   Ответ.

7. В чем заключается правосторонний нисходящий грамматический разбор? Назовите, в каких случаях он не применим и определите его основной недостаток.

   Ответ.

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

   Ответ.

3 Распознавание регулярных грамматик

3.1 Конечный автомат и его программная реализация

Как уже было указано в разделе 2.3, распознаватели регулярных грамматик строят на конечных автоматах. *Конечный автомат* – это математическая модель, свойства и поведение которой полностью определяются пятеркой:

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

где Q – конечное множество состояний,

Σ – конечное множество входных символов,

$\delta(q_i, c_k)$ – функция переходов, где $q_i \in Q$ – текущее состояние, $c_k \in \Sigma$ – очередной символ,

$q_0 \in Q$ – начальное состояние,

$F \subset Q$ – подмножество допускающих состояний.

Конечный автомат – одна из базовых математических моделей, используемых при проектировании программного и аппаратного обеспечения средств вычислительной техники.

***Пример*.** Математически автомат «Чет-нечет» описывается следующим образом:

Q = {Чет, Нечет};

Σ = {0, 1};

δ(Чет, 0) = Чет, δ(Нечет, 0) = Нечет, δ(Чет, 1) = Нечет,

δ(Нечет, 1) = Чет;

q_0 = Чет;

F = {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 (таблица 3), граф переходов состояний (рисунок 4) и синтаксическая диаграмма (рисунок 5).

Таблица 3 – Функция переходов

q \ σ	0	1
Чет	Чет	Нечет
Нечет	Нечет	Чет

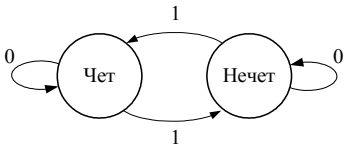


Рисунок 4 – Граф переходов

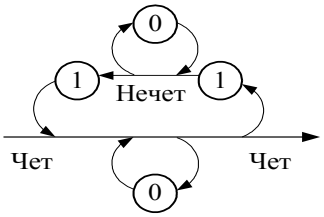


Рисунок 5 – Синтаксическая диаграмма

Разработчик автомата обычно использует представление автомата 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 (таблица 4).

Таблица 4 – Дополненная таблица конечного автомата

c / *q*	0	1	**Символы завершения**	**Другие символы**
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* ≠ «Ошибка» и *q* ≠ «Конец»

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

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

Все-цикл

Если *q* = «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, C или C++), 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 1. 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 рисунке 6.

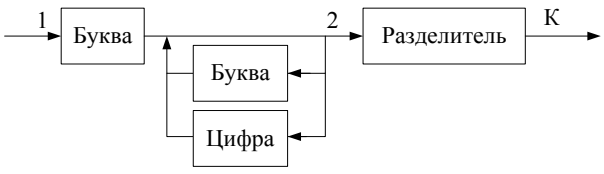


Рисунок 6 – Синтаксическая диаграмма лексемы Идентификатор

Таблица автомата (таблица 5), построенная по синтаксической диаграмме, учитывает возможное присутствие неразрешенных (других) символов.

Таблица 5 – Таблица автомата, распознающего идентификаторы

Состояние	Буква	Цифра	Разделитель	Другой символ
1	2	Error	Error	Error
2	2	2	10	Error

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

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

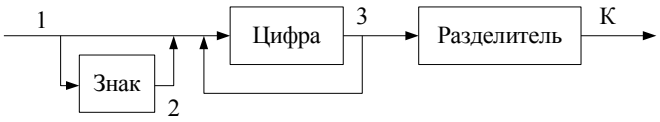
Текст функции, реализующей автомат, выглядит следующим образом:

```
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
    Probel(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;
```

Пример 2. 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 (рисунок 7).



Состояния автомата:

1 – начало разбора; 2 – распознан знак; 3 – целое; *К* – завершение разбора

Рисунок 7 – Синтаксическая диаграмма

В таблице 6 цифрами обозначены разрешенные переходы синтаксической диаграммы. Запрещенные переходы (ошибки) обозначены символом «Е».

Таблица 6 – Таблица переходов

	Знак	Цифра	Разделитель	Другие
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 укажем подпрограммы обработки, которые должны быть выполнены при осуществлении указанного перехода:

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 таблице 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*** **≠ «*E*» и** ***q*** **≠ «*K*»**

Если ***S*****[*Ind*] = «+» или** ***S*****[*Ind*] = «-»,**

то j := 1

иначе Если ***S*****[*Ind*] ≥ «0» и** ***S*****[*Ind*] ≤ «9»,**

то ***j*** **:= 2,**

иначе ***j*** **:= 3**

Все-если

Все-если

Выполнить A_i **:=** ***Table*** **[*q*, *j*].** ***A*****()**

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

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

Все-цикл

Если ***q*** **= «К»**

то Выполнить А3

Вывести сообщение «Это число»

иначе Вывести сообщение Ді

Все-если

Пример 3. 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 1-2-х символов, завершающиеся символами операций:

```
I+   I*   (    I+   I) .
```

Анализ строк токенов показал, что существует единственный случай, когда в строке токенов 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);
```

и функция распознавателя идентификаторов

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

Тела обеих подпрограмм рассмотрены в примере 1 настоящего раздела.

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];
            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)
```

Ниже рассмотрен еще один, более сложный, вариант сканирующей программы.

Пример 4. 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 разрешаем использовать только простые сравнения и служебные слова можно вводить только строчными символами.

На выходе сканер должен формировать строку токенов длиной 2 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_	)	;_

где 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 предыдущем примере:

```
Procedure Probel(Var St:shortstring);  // удаление пробелов
Function Id(Var St:shortstring;Razd:setofChar):boolean;
                                  //распознаватель идентификаторов
```

Текст функции-сканера:

```
Function Scaner(
     St:shortstring;        // исходная строка
     Razd:setofChar;        // множество разделителей
     Var StS:shortstring    // строка токенов
             ):boolean;
  Const SlSl:array[1..3] of shortString=
                  ('if','then','else');   // служебные слова
  Var R:boolean; i,k:byte; Stl: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
             Stl:=Copy(St,1,k-1);
             i:=1;
             while (Stl<>SlSl[i]) and (i<4) do i:=i+1;
             if i<>4 then  // служебное слово
               begin
                  WriteLn('Slugeb slovo ',Stl);
                  Delete(St,1,k-1);
                  if Stl[1]='i' then
                          StS:=StS+'@@';
                  StS:=StS+Copy(Stl,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
      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 – процесс распознавания конструкций языка в строке токенов. Важным результатом, помимо распознавания заданной кон-

струкции, является информация об ошибках в выражениях, операторах и описаниях программы.

Способ построения синтаксического анализатора определяется типом грамматики языка:

- для регулярных грамматик используют конечные автоматы;
- для КС-грамматик – автоматы с магазинной памятью.

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

Пример. 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 (рисунок 8).

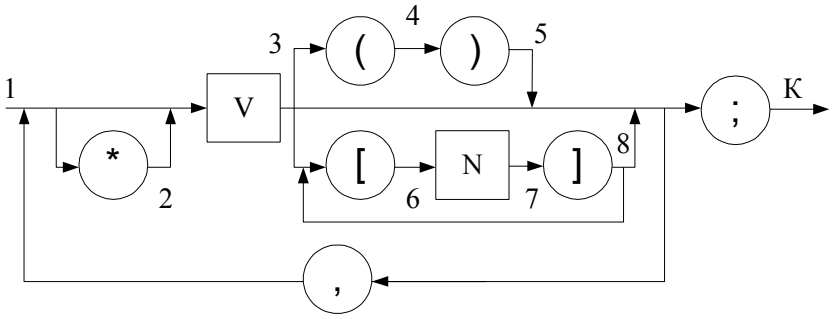


Рисунок 8 – Синтаксическая диаграмма конструкции

По диаграмме построим таблицу автомата (таблица 7).

Таблица 7 –Таблица переходов

	V	N	*	(	)	[	]	,	;	Другие
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**	**К**	**E**
4	**E**	**E**	**E**	**E**	**5**	**E**	**E**	**E**	**E**	**E**
5	**E**	**E**	**E**	**E**	**E**	**E**	**E**	**1**	**К**	**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**	**К**	**E**

Алгоритм распознавателя:

***Ind* := 1**

***q* := 1**

Цикл-пока *q* ≠ «E» и *q* ≠ «К»

***q* := *Table* [*q*, *Pos*(*S*[*Ind*], «VN*()[];»)]**

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

Все-цикл

Если *q* = «К»

то Выполнить А3

Вывести сообщение «Это число»

иначе Вывести сообщение Ді

Все-если

В описании алгоритма функция ***Pos*(*S*[*Ind*], «VN*()[];»)** 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

1. Определите конечный автомат. В чем особенность его использования при грамматическом разборе.
   Ответ.
2. Какие способы описания конечных автоматов существуют? Какой способ используется при программной реализации и почему?
   Ответ.
3. Как построить конечный автомат для лексического анализа? В чем заключается его особенность?
   Ответ.
4. Как построить конечный автомат для синтаксического анализа? Что подается на вход синтаксического анализатора.
   Ответ.

4 Распознавание контекстно-свободных грамматик

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;

Σ – конечный входной алфавит;

Γ – конечное множество магазинных символов;

$\delta\ (q, c_k, z_j)$ – функция переходов;

$q_0 \in Q$ – начальное состояние автомата;

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

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

Пример. 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>).

Грамматику 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 рисунке 9.

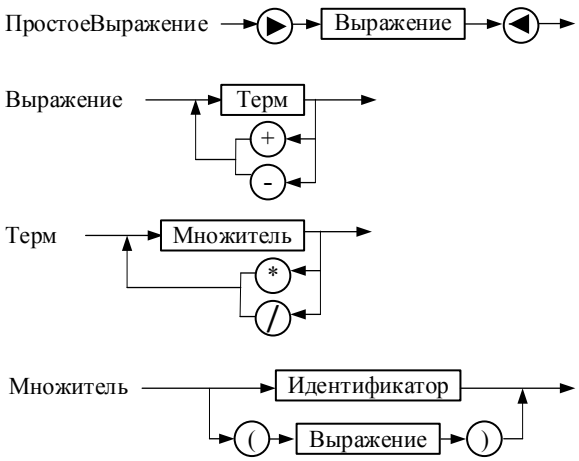


Рисунок 9 – Синтаксические диаграммы грамматики выражений

Теперь попробуем подставить диаграммы одна в другую, исключая промежуточные конструкции Терм и Множитель. В результате получим полную синтаксическую диаграмму (рисунок 10), которая состоит из двух частей: основной и рекурсивной. Это связано с наличием рекурсивного вложения в правилах продукции грамматики, описывающей синтаксис арифметического выражения.

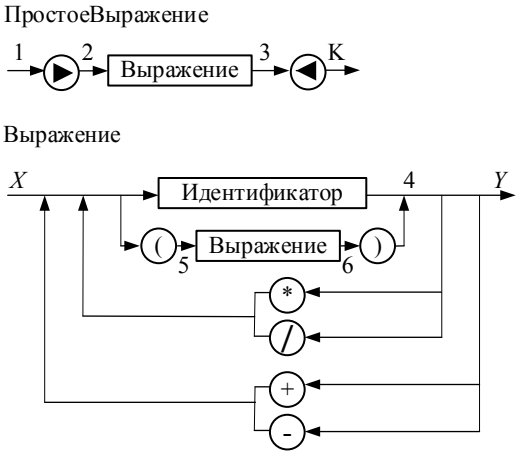


Рисунок 10 – Объединенная синтаксическая диаграмма

По диаграмме построим таблицу автомата, которая также будет состоять из двух частей (таблица 9).

Таблица 9 – Таблица переходов автомата

	<Ид>	+	-	*	/	(	)	▶	◀
1								2	
2						↓ *Y*=3			

3									К

	<Ид>	+	-	*	/	(	)	►	◄
X	4					5			
4		*X*	*X*	*X*	*X*		↑ *Y*		↑ *S*
5						↓ *Y*=6			
6							4		

В таблице использованы следующие условные обозначения:

К – конец разбора;

Е – состояние ошибки – все пустые ячейки таблицы должны содержать значение *Е*;

↓ *Y* = <состояние> – рекурсивный вызов распознавателя, справа указан номер состояния после возврата из данного вызова;

↑*Y* – возврат из рекурсии, следующее состояние определяется значением *Y*.

Необходимость запоминания состояния возврата при рекурсивном вызове подтверждает, что описываемый таблицей 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; ↓ – запись в стек, ↑ – чтение из стека;

q – текущее состояние;

Table (<Текущее состояние>, <Анализируемый символ>) – элемент таблицы (матрицы) переходов, состоящий из следующих полей: *q* – следующее состояние (в том числе «↓» – рекурсивный вызов, «↑» – возврат из рекурсии), *S* – состояние после возврата из рекурсии, если необходим рекурсивный вызов, или ∅;

String – исходная строка;

Ind – номер символа исходной строки.

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

***q*:=1**
***Ind* := 1**
***Mag* :=∅**
Цикл-пока *q* ≠ «Е» и *q* ≠ «К»
q* := *Table* [*q*, *String*[*Ind*]].*q
Если *q* = '↓'
то *Mag* ↓ *Table* [*q*, *String*[*Ind*]].*S*
q* := *X
иначе Если *q* = '↑'
то *Mag* ↑*q*
иначе *Ind* := *Ind* +1
Все-если
Все-если
Если *q* = «К»
то «Строка принята»
иначе «Строка отвергнута»
Все-если

Построение автомата с магазинной памятью для сложного языка – задача не простая, поскольку таблица получается длинной – сотни строк. Поэто-

му 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. Осуществить левосторонний восходящий грамматический разбор строки токенов:

►<**Ид**>+(<**Ид**>-<**Ид**>)*<**Ид**>◄,

где <Ид> = <Идентификатор>, ►◄ – начало и конец выражения.

Грамматика записи выражений (вариант с левосторонней рекурсией):

а) **<ПростоеВыражение> ::= ►<Выражение>◄**,

б) **<Выражение> ::= <Терминал> + <Выражение> |**
<Терминал> - <Выражение> |
<Терминал>,

в) **<Терминал> ::= <Множитель> * <Терминал> |**
<Множитель> / <Терминал> |
<Множитель>,

г) **<Множитель> ::= (<Выражение>) | <Идентификатор>**.

При анализе, чтобы сократить запись в таблице, будем использовать следующие сокращения:

<Выр> = <Выражение>, **<Терм> = <**Терминал**>**, <Множ> = <Множитель>, <Ид> = <Идентификатор>.

Последовательность разбора (начальный фрагмент):

№ шага	Исходная строка	Строка правил	Действие
1	►<Ид>+(<Ид>-Ид>)*<Ид>◄	**<ПростоеВыражение>**	Подстановка правила а1
	►<Ид>+(<Ид>-Ид>)*<Ид>◄	►<**Выр**>◄	Символ распознан и удален
2	<Ид>+(<Ид>-Ид>)*<Ид>◄	<**Выр**>◄	Подстановка правила б1
3	<Ид>+(<Ид>-Ид>)*<Ид>◄	<**Терм**>+<Выр>◄	Подстановка правила в1
4	<Ид>+(<Ид>-Ид>)*<Ид>◄	<**Множ**>*<Терм>+<Выр>◄	Подстановка правила г1
	<Ид>+(<Ид>-Ид>)*<Ид>◄	(<Выр>)*<Терм>+<Выр>◄	Ошибка, возврат к шагу 4
5	<Ид>+(<Ид>-Ид>)*<Ид>◄	<**Множ**>*<Терм>+<Выр>◄	Подстановка правила г2
	<Ид>+(<Ид>-Ид>)*<Ид>◄	<Ид>*<Терм>+<Выр>◄	Символ распознан и удален
	+(<Ид>-<Ид>)*<Ид>◄	*<Терм>+<Выр>◄	Ошибка, возврат к шагу 3
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, возврат к шагу 7
8	<Ид>+(<Ид>-<Ид>)*<Ид>◄	<**Множ**>/<Терм>-<Выр>◄	Подстановка правила г2
	<Ид>+(<Ид>-<Ид>)*<Ид>◄	<Ид>/<Терм>-<Выр>◄	Символ распознан и удален
	+(<Ид>-<Ид>)*<Ид>◄	/<Терм>-<Выр>◄	Ошибка, возврат 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, возврат к шагу 10
11	<Ид>+(<Ид>-<Ид>)*<Ид>◄	<**Множ**>+<Выр>◄	Подстановка правила г2
	<Ид>+(<Ид>-<Ид>)*<Ид>◄	<Ид>+<Выр>◄	Символ распознан и удален
	+(<Ид>-<Ид>)*<Ид>◄	+<Выр>◄	Символ распознан и удален
12	(<Ид>-<Ид>)*<Ид>◄	<**Выр**>◄	Подстановка правила б1
13	(<Ид>-<Ид>)*<Ид>◄	<**Терм**>+<Выр>◄	Подстановка правила в1

Оглавление

14	(<Ид>-<Ид>)*<Ид>◄	<**Множ**>*<Терм>+<Выр>◄	Подстановка правила г1
	(<Ид>-<Ид>)*<Ид>◄	(<Выр>)*<Терм>+<Выр>◄	Символ распознан 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, возврат к шагу 17
18	<Ид>-<Ид>)*<Ид>◄	<**Множ**>*<Терм>+<Выр>)*<Терм>+<Выр>◄	Подстановка правила г2
	<Ид>-<Ид>)*<Ид>◄	<Ид>*<Терм>+<Выр>)*<Терм>+<Выр>◄	Символ распознан и удален
	-<Ид>)*<Ид>◄	*<Терм>+<Выр>)*<Терм>+<Выр>◄	Ошибка, возврат 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
	<Ид>-<Ид>)*<Ид>◄	(<Выр>)/<Терм>+<Выр>)*<Терм>+<Выр>◄	Ошибка, возврат к шагу 20а
21	<Ид>-<Ид>)*<Ид>◄	<**Множ**>/<Терм>+<Выр>)*<Терм>+<Выр>◄	Подстановка правила г2
	<Ид>-<Ид>)*<Ид>◄	<Ид>/<Терм>+<Выр>)*<Терм>+<Выр>◄	Символ распознан и удален
	-<Ид>)*<Ид>◄	/<Терм>+<Выр>)*<Терм>+<Выр>◄	Ошибка, возврат к шагу 19
22	<Ид>-<Ид>)*<Ид>◄	<**Терм**>+<Выр>)*<Терм>+<Выр>◄	Подстановка правила в3
23	<Ид>-<Ид>)*<Ид>◄	<**Множ**>+<Выр>)*<Терм>+<Выр>◄	…

Анализ процесса разбора показывает, что большинство подстановок лишние, так 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 будет зафиксирована ошибка и не будет выполнен возврат.

Разработать же алгоритм выбора правильной подстановки правил для используемой в 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 выше, по виду правил *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:

а) **<ПростоеВыражение> ::= ►<Выражение>◄**,

б) **<Выражение> ::= <Терминал> <Сложение>,**

в) **<Сложение> ::= + <Терминал> <Сложение>** |
- <Терминал><Сложение> |
***e*,**

г) **<Терминал> ::= <Множитель><Умножение>,**

д) **<Умножение> ::= *<Множитель><Умножение>** |
/ <Множитель><Умножение> |
***e*,**

е) **<Множитель> ::= (<Выражение>) | <Идентификатор>,**

где *e* – пустая строка.

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

При разборе используем следующие сокращения:

<Выр> = <Выражение>, <Терм> = <Терминал>, <Слож> = <Сложение>,

<Умн> = <Умножение>, <Множ> = <Множитель>,

<Ид> = <Идентификатор>.

Тогда последовательность разбора будет записываться следующим образом:

№	Распознаваемая строка	Строка правил	Действие
1	<Ид>*(<Ид>+<Ид>)+<Ид>	<**Выр**>	Подстановка правила а
2	<Ид>*(<Ид>+<Ид>)+<Ид>	**<Терм>**<Слож>	Подстановка 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 осуществление разбора без «возвратов», что существенно упрощает составление соответствующей программы.

Однако наиболее удобную для программирования реализацию разбора *LL*(*k*)-грамматик обеспечивает *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 рекурсивного спуска основывается на синтаксических диаграммах, описывающих конструкции языка. Согласно этому методу для каждого нетерминала разрабатывают рекурсивную процедуру. Основная программа вызывает процедуру аксиомы, которая вызывает процедуры нетерминалов, упомянутые в правой части аксиомы и т. д. В эти же процедуры встраивают семантическую обработку распознанных конструкций.

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

Синтаксис языка описания выражений можно описать синтаксическими диаграммами (рисунок 10).

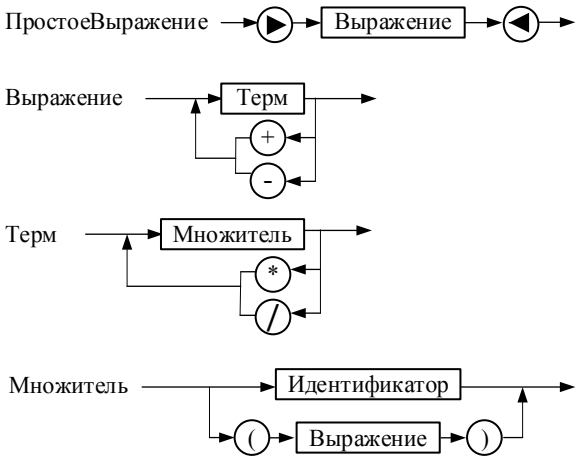


Рисунок 6 – Синтаксические диаграммы языка описания выражений

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 на рисунке 11.

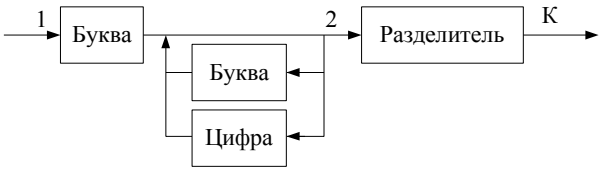


Рисунок 7 – Синтаксическая диаграмма нетерминала <Идентификатор>

```
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 Probel(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 1 из раздела 3.2}

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

```
  Begin
    Probel(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;
  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. Осуществить разбор выражения

►<Ид>+(<Ид>-<Ид>)*<Ид>◄,

где <Ид> – <Идентификатор>, ►◄ – начало и конец выражения соответственно.

Грамматика записи выражений (вариант с правосторонней рекурсией):

а) **<ПростоеВыражение> ::= ►<Выражение>◄**,

б) **<Выражение> ::= <Выражение> + <Терминал> |**

<Выражение> - <Терминал> |

<Терминал>

в) **<Терминал> ::= <Терминал> * <Множитель> |**

<Терминал> / <Множитель> |

<Множитель>

г) **<Множитель> ::= (<Выражение>) | <Идентификатор>**

При записи процесса разбора используем следующие обозначения: <Выр> – <Выражение>, **<Терм> – <**Терминал**>**, <Множ> – <Множитель>, <Ид> – <Идентификатор>.

Последовательность разбора (полужирным выделена основа, свертываемая на следующем шаге):

№ шага	Распознаваемая строка	Основа	Операция
1	<**Ид**>+(<Ид>-<Ид>)*<Ид>	<Ид>	Свертка по правилу г2
2	**<Множ>**+(<Ид>-<Ид>)*<Ид>	<Множ>	Свертка по правилу в3
3	**<Терм>**+(<Ид>-<Ид>)*<Ид>	<Терм>	Свертка по правилу б3
4	<Выр>+(<**Ид**>-<Ид>)*<Ид>	<Ид>	Свертка по правилу г2
5	<Выр>+(<**Множ**>-<Ид>)*<Ид>	<Множ>	Свертка по правилу в3
6	<Выр>+(<**Терм**>-<Ид>)*<Ид>	<Терм>	Свертка по правилу б3
7	<Выр>+(<Выр>-<**Ид**>)*<Ид>	<Ид>	Свертка по правилу г2
8	<Выр>+(<Выр>-<**Множ**>)*<Ид>	<Множ>	Свертка по правилу в3
9	<Выр>+(**<Выр>-<Терм>**)*<Ид>	<Выр>-<Терм>	Свертка по правилу б2
10	<Выр>+**(<Выр>)***<Ид>	(<Выр>)	Свертка по правилу г1
11	<Выр>+**<Множ>***<Ид>	<Множ>	Свертка по правилу в3
12	**<Выр>+<Терм>***<Ид>	<Выр>+<Терм>	Свертка по правилу б1
13	<Выр>*<Ид>	Тупик!	Возврат к шагу 12.
14	<Выр>+<Терм>*<**Ид**>	<Ид>	Свертка по правилу г2
15	**<Выр>+<Терм>***<Множ>	<Выр>+<Терм>	Свертка по правилу б1
16	<Выр>*< Множ>	Тупик!	Возврат к шагу 16.
17	<Выр>+<**Терм>*<Множ**>	<Терм>*<Множ>	Свертка по правилу в1
18	**<Выр>+<Терм>**	<Выр>+<Терм>	Свертка по правилу б1
19	<Выр>	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. Сверт-

ка выполняется, 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 метода 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) α принадлежит основе, а β – нет, т. е. α – конец основы: $\alpha \cdot> \beta$;
2) β принадлежит основе, а α – нет, т. е. β – начало основы: $\alpha <\cdot \beta$;
3) α и β принадлежит одной основе, т. е. $\alpha =\cdot \beta$;
4) α и β 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.

Пример. 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 таблицу 9.

Таблица 5 – Таблица предшествования

	+	*	(	)	◄
►	<·	<·	<·	?	Выход
+	·>	<·	<·	·>	·>
*	·>	·>	<·	·>	·>
(	<·	<·	<·	=	?
)	·>	·>	?	·>	·>

Обозначения:
? – ошибка;
<· - начало основы;
·> - конец основы;
= - принадлежат одной основе;
►- начало выражения;
◄ - конец выражения.

Тогда при разборе выражения:

►d + c * (a + b) ◄,

где a, b, c, d – обозначения токенов вида <Ид>, соответствующих идентификаторам переменных, констант или литералов чисел, содержимое стека будет выглядеть следующим образом:

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◄	·>	Свертка	R_2:= c* R_1	<Терм>
►d+	R_2◄	·>	Свертка	R_3:= d+ R_2	<Выражение>
►	R_3◄	Конец			

Текст программы, реализующей данный метод:

```
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 раздела 3.2}

Procedure Probel(Var St:shortstring);  { Подпрограмма
              удаления пробелов, текст программы приведен в примере 1
              раздела 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 1 раздела 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 Odna_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;
          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;
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:

1) K_I, где I – идентификатор операнда – выбрать число по имени I и заслать его в стек операндов;

2) K_ξ, где ξ – операция – выбрать два верхних числа из стека операндов, произвести над ними операцию ξ 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 используется два стека: стек операций – на первом этапе и стек операндов – на втором этапе.

Построение польской записи выполняется следующим образом: транслятор читает выражение слева направо и вырабатывает последовательность команд по следующему правилу:

а) если символ – операнд, то вырабатывается команда $\mathbf{К_I}$,

б) если символ – операция, то выполняются действия согласно таблице 10:

Таблица 6 – Таблица генератора записи

η\ξ	+	*	(	)	◄
►	I	I	I	?	Выход
+	II	I	I	IV	IV
*	IV	II	I	IV	IV
(	I	I	I	III	?

Обозначения:

? – ошибка;

η – верхний символ в стеке операций;

ξ – текущий символ.

Операции:

I – заслать ξ в стек операций и читать следующий символ;

II – генерировать $\mathbf{K_\eta}$, заслать ξ в стек операций и читать следующий символ;

III – удалить верхний символ из стека операций и читать следующий символ;

IV – генерировать $\mathbf{K_\eta}$ и повторить с тем же входным символом.

На этапе выполнения польская запись читается слева направо и выполняется.

***Пример*.** Построить тройки для выражения **► (a+b*c)/d ◄.**

Этап 1. Построение польской записи:

Стек операций	Символ	Действие	Команда
►	(	I	

► (	a		$\mathbf{K_a}$
► (	+	I	
► (+	b		$\mathbf{K_b}$
► (+	*	I	
► (+ *	c		$\mathbf{K_c}$
► (+ *	)	IV	$\mathbf{K_*}$
► (+	)	IV	$\mathbf{K_+}$
► (	)	III	
►	/	I	
► /	d		$\mathbf{K_d}$
► /	◄	IV	$\mathbf{K_/}$
►	◄	Конец	

В результате получаем:

K_a K_b K_c K_* K_+ K_d $K_/$ или **a b c * + d /.**

Теперь необходимо выполнить польскую запись в соответствии с ее определением.

Этап 2. Выполнение польской записи:

Стек операндов	Команда	Тройка
∅	$\mathbf{K_a}$	
a	$\mathbf{K_b}$	
a b	$\mathbf{K_c}$	
a b c	$\mathbf{K_*}$	$\mathbf{T_1= b*c}$
a T_1	$\mathbf{K_+}$	$\mathbf{T_2= a+T_1}$
$\mathbf{T_2}$	$\mathbf{K_d}$	
$\mathbf{T_2}$ **d**	$\mathbf{K_/}$	$\mathbf{T_3= T_2/d}$
$\mathbf{T_3}$		

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

1. Определите, автомат с магазинной памятью. В чем особенность его использования при грамматическом разборе?
   Ответ.
2. Как построить автомат с магазинной памятью по синтаксическим диаграммам? В чем заключается особенность полученной программы?
   Ответ.
3. Определите LL(k) грамматики. Какими особенностями они обладают? В чем заключается метод рекурсивного спуска?
   Ответ.
4. Определите LR(k) грамматики. Какими особенностями они обладают? В чем заключается стековый метод?
   Ответ.
5. Какой вид имеет польская запись? Для чего она используется?
   Ответ.
6. В чем заключается алгоритм Бауэра-Замельзона?
   Ответ.

5 Распределение памяти под программы и данные

После распознавания конструкций и определения таблиц переменных, именованных констант и временных переменных осуществляют распределение памяти под эти данные и программы.

Для универсальных языков программирования различают несколько классов памяти переменных и разные типы распределения памяти (см. таблицу 11).

Таблица 11 – 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 вызы-	От момента вызова подпрограммы до ее завер-	Автоматическое

				ваемых из нее под-программ	шения	
4	Статиче-ские static	-	В ос-новном сегменте данных про-граммы	Из подпро-граммы, в которой объявлены, и вызы-ваемых из нее под-программ	От запуска програм-мы до ее заверше-ния	Статиче-ское

Статическое распределение памяти выполняется:

а) во время компиляции – для подпрограмм и для инициализированных переменных;

б) во время загрузки программы на выполнение – для неинициализированных переменных.

Автоматическое распределение памяти выполняется 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 выполняется во время выполнения программы специальными подпрограммами, например, new и delete.

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

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

Контрольные вопросы

1. Какие виды распределения памяти Вы знаете? Чем они различаются и для чего используются?

   Ответ.
2. Чем различаются локальная и глобальная виды памяти программы?

   Ответ.

6 Генерация и оптимизация кодов

6.1 Генерация кодов

Генерация машинного кода выполняется заменой распознанной конструкции соответствующими машинными командами, например, тройка сложения целых чисел может заменяться последовательностью команд:

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

В такой последовательности команд выполняется подстановка адресов операндов и результата.

Очевидно, что код, полученный таким образом, является не оптимальным. Поэтому при генерации кода выполняется оптимизация.

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 разного типа таблицы.

6.2 Машинно-независимая оптимизация

6.2.1 Удаление недостижимого кода

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

Пример. В следующем фрагменте выполняется только оператор S1:

```
if (1) S1; else S2;  ⇒   S1;
```

6.2.2 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 выполняется:

а) удаление бесполезных присваиваний:

```
a = b*c; d = b+c; a = d*c;  ⇒  d = b+c;  a = d*c;
```

Примечание. В следующем примере эта операция уже не бесполезна:

```
p = &a;  a = b*c;  d = *p+c;  a = d*c;
```

б) исключение избыточных вычислений:

```
d = d+b*c;  a = d+b*c;  c = d+b*c;          ⇒
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:

```
i = 2+1; j = 6*i+i;  ⇒  i = 3; j = 21;
```

г) перестановка операций для дальнейшей свертки или оптимизации вычислений:

```
a = 2*b*3*c; ⇒   a = (2*3)*(b*c);

a = (b+c)+(d+e); ⇒   a = (b+(c+(d+e)));
```

В первом примере 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:

```
a = b*c+b*d; ⇒   a = b*(c+d);

a*1  ⇒  a,    a*0  ⇒  0,    a+0 ⇒  a;
```

е) оптимизация вычисления логических выражений:

`a||b||c||d;` ⇒ `a,` если `a` = **`true.`**

Но! `a||f(b)||g(c)` не всегда `a` при **`a = true`**, так как возможен побочный эффект из-за функций.

6.2.3 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.

6.2.4 Оптимизация вычислений в циклах

Выполняется анализ параметров циклов и циклически выполняемых операций и осуществляется:

а) вынесение инвариантных вычислений из циклов:

```
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:

```
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; }
```

// избавились от одной индуктивной переменной

в) слияние циклов:

```
 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) развертывание циклов:

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

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

6.3 Машинно-зависимая оптимизация

Машинно-зависимые преобразования результирующей объектной программы зависят от архитектуры вычислительной системы, на которой будет выполняться результирующая программа. При этом может учитываться объ-

ем кэш-памяти, методы организации работы процессора и другие особенности системы.

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

6.3.1 Распределение регистров процессора

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

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

6.3.2 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*, т. к. методы передачи парамет-

ров 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 (*register*), какие параметры и локальные переменные желательно разместить в регистрах.

6.3.3 Оптимизация кода для процессоров, допускающих распараллеливание 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 этого надо найти оптимальный порядок выполнения операций для каждого оператора (переставить их).

```
a + b + c + d + e + f;
```
⇒

для одного потока обработки данных:
```
(((a + b) + c) + d) + e) + f;
```

для двух потоков обработки данных:
```
((a + b) + c) + ((d + e) + f);
```

для трех потоков обработки данных:
```
(a + b) + (c + d) + (e + f);
```

Контрольные вопросы

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

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

   Ответ.

Литература

1. Д. Грис. Конструирование компиляторов для цифровых вычислительных машин – М.: Мир, 1975. – 544 с.
2. Ахо А., Сети Р., Ульман Д. Компиляторы: принципы, технологии и инструменты. Пер. с англ. – М.: «Вильямс», 2003.
3. В. Дж. Рейуорд–Смит. Теория формальных языков. Вводный курс. – М.: Радио и связь, 1988. – 128 с.