Л. Г. Гагарина, Е. В. Кокорева

ВВЕДЕНИЕ В ТЕОРИЮ АЛГОРИТМИЧЕСКИХ ЯЗЫКОВ И КОМПИЛЯТОРОВ

Допущено Учебно-методическим объединением вузов по университетскому политехническому образованию в качестве учебного пособия для студентов высших учебных заведений, обучающихся но направлению 09.03.01 «Информатика и вычислительная техника»

Москва
ИД «ФОРУМ» — ИНФРА-М
2022

УДК 004.2(075.32)
ББК 32.973-02я723
Г12

ФЗ № 436-ФЗ	Издание не подлежит маркировке в соответствии с п. 1 ч. 4 ст. 11

Рецензенты:
кандидат технических наук, генеральный директор ОАО «ОТИК-групп» (Общероссийский технический информационный канал) *Д.Б. Ломоносов*;
кандидат технических наук, доцент *А.А. Петров* (кафедра «Информационные технологии» филиала Санкт-Петербургского гуманитарного университета профсоюзов)

Гагарина Л. Г.

Г12 Введение в теорию алгоритмических языков и компиляторов : учебное пособие / Л.Г. Гагарина, Е.В. Кокорева ; под ред. Л.Г. Гагариной. — Москва : ИД «ФОРУМ», 2022. — 176 с: ил. — (Высшее образование).

ISBN 978-5-8199-0404-6

Учебник написан в соответствии с Федеральным государственным образовательным стандартом 3-го поколения. Приведен систематизированный курс освоения теории формальных языков и грамматик — как регулярных, так и контекстно-свободных. Рассмотрены современные задачи лексического, синтаксического и семантического анализа, известные принципы их использования для решения практических задач создания программного обеспечения. Строгий стиль изложения сопровождается многочисленными примерами, а также задачами для самостоятельного решения в составе практических заданий, необходимых для глубокого усвоения материала.

Для студентов, аспирантов, научных сотрудников, преподавателей высших учебных заведений, а также для тех, кто интересуется математическими основами программирования.

УДК 004.2(075.32)
ББК 32.973-02я723

ISBN 978-5-8199-0404-6

Предисловие

Предлагаемое учебное пособие посвящено технологии преобразования программ для выполнения их на компьютере с помощью трансляторов — специальных средств для перевода программ с языка программирования на машинный язык. Со времени появления первого такого средства в мире разработки компиляторов произошли значительные изменения: эволюционировали языки программирования, архитектура компьютеров; появилось множество различных вычислительных ресурсов, грамотное использование которых необходимо для получения лучших результатов. Приведенные в пособии методы — грамматики, регулярные выражения, синтаксические анализаторы, а также методы оптимизации — сегодня находят применение не только в компиляторах, но и в инструментарии для поиска ошибок в программном обеспечении и, что особенно важно, при проверке безопасности существующего кода.

По всей видимости, далеко не все читатели книги будут заниматься разработкой или хотя бы поддержкой компилятора для какого-либо языка программирования, однако изложенный далее систематизированный материал может применяться для решения широкого диапазона задач проектирования и разработки программного обеспечения.

Основной текст пособия состоит из пяти взаимосвязанных глав, в каждой из которых содержатся примеры, поясняющие излагаемый теоретический материал, предусмотрены контрольные вопросы. Глава 1 содержит терминологический и понятийный аппарат теории формальных языков, классификации грамматик, языков, распознавателей. Глава 2 посвящена лексическому анализу, регулярным выражениям, конечным автоматам и инструментарию для создания детерминированных конечных автоматов. Материал главы может быть использован при разработке текстовых редакторов. В гл. 3 приведены различные методы синтаксического анализа, нисходящие (рекурсивного спуска, *LL*)

и восходящие (*LR* и его варианты); подробно рассмотрен пример разработки синтаксического анализатора заданных конструкций языка СИ+. Глава 4 знакомит с принципиальными идеями семантического анализа, которые лежат в основе генерации промежуточного кода в типичном языке программирования. Глава 5 посвящена проблемам, возникающим в процессе разработки трансляторов на практике, подробно рассмотрена вся технология программирования.

Структура пособия подчинена определенной внутренней логике, т. е. материал излагается последовательно, в соответствии с приобретением профессиональных знаний. Особенностью данного пособия являются практические задания, необходимые для закрепления изученного теоретического материала. Только выполнив практические задания (фактически проверочные задачи для самостоятельной работы студентов к гл. 1 и 2), можно перейти к лабораторным работам, представляющим собой своеобразный тренинг программиста-разработчика трансляторов на базе теории гл. 3—5.

Данное учебное пособие предназначено, в первую очередь, для студентов высших учебных заведений, обучающихся по специальности 230105.65 «Программное обеспечение вычислительной техники и автоматизированных систем» по направлению подготовки специалистов 654600 «Информатика и вычислительная техника» и направлению подготовки бакалавров и магистров 552800 «Информатика и вычислительная техника». Может быть полезно аспирантам, преподавателям вузов, курсов повышения квалификации и центров сертификации.

Труд авторов распределился следующим образом: предисловие, гл. 1, практические задания к гл. 5 — профессор, доктор технических наук Л. Г. Гагарина, гл. 2—5, практические задания к гл. 1—3 — доцент Е. В. Кокорева.

Авторский коллектив выражает глубокую признательность ведущему специалисту Государственного унитарного предприятия Научно-технический центр «Спурт» В. Г. Дорогову за конструктивную поддержку, оказанную при написании пособия, а также аспиранту кафедры «Информатика и программное обеспечение вычислительных систем» Московского института электронной техники А. В. Солякову за верификацию практических заданий.

Глава 1
ОСНОВЫ ТЕОРИИ ФОРМАЛЬНЫХ ЯЗЫКОВ

Для построения трансляторов необходимо однозначное и точное задание входного и выходного языков. Такое задание предполагает определение правил построения допустимых конструкций выражений языка.

Существует целый ряд математических формализмов для задания языков. Наиболее распространенным механизмом являются порождающие грамматики, которые задают все цепочки языка с помощью некоторых правил. Одно из достоинств грамматик заключается в том, что существует множество систем, которые по заданной грамматике генерируют программу, проверяющую соответствие входной цепочки определяемому языку.

Грамматика представляет собой математическую систему, порождающую язык. Строки языка строятся способом, точно определенным правилами грамматики [2—5].

1.1. Терминология предметной области

По мнению ученых, развитие разума на нашей планете происходило под влиянием языка, что позволило не только выражать и сохранять информацию, но и обмениваться ею друг с другом.

С созданием и развитием компьютерной техники возникла необходимость в появлении специальных языков для обмена информацией с устройством, понятных и удобных для человека и в то же время воспринимаемых компьютером. Совмещение обоих этих свойств в одном языке оказалось невозможным, поэтому были разработаны специальные средства для перевода текстов с языка, понятного программисту, на язык, понятный устройству, называемые *трансляторами*. Перевод программы с языка программирования на машинный язык называется *трансляцией*.

Технология преобразования программ для выполнения их на компьютере сформировалась в начале 1960-х гг. и с тех пор не претерпела существенных изменений.

Подготовка программы начинается с редактирования файла, содержащего текст этой программы, который имеет стандартное расширение для данного языка. Затем выполняется его трансляция, которая включает несколько этапов [1, 2]:

- препроцессор;
- анализ, который включает в себя три фазы:
  - — лексический анализ;
  - — синтаксический анализ;
  - — семантический анализ;
- синтез (генерация машинного представления кода программы), включающий все или некоторые из нижеперечисленных фаз:
  - — генерация машинно-независимого кода;
  - — оптимизация машинно-независимого кода;
  - — распределение памяти;
  - — генерация машинного кода;
  - — оптимизация машинного кода.

В результате трансляции получается *объектный модуль*. Файл объектного модуля имеет стандартное расширение .obj. *Компоновка* (сборка) программы заключается в объединении одного или нескольких объектных модулей программы и объектных модулей, взятых из библиотечных файлов и содержащих стандартные функции. В результате получается исполняемая программа в виде отдельного файла (*загрузочный модуль*, *программный файл*) со стандартным расширением .exe, который затем загружается в память и выполняется.

Фазы трансляции

Препроцессор — это предварительная фаза трансляции, которая выполняет замену одних частей текста на другие.

В языке Си директивы препроцессора оформлены отдельными строками программы, которые начинаются с символа #. Рассмотрим некоторые наиболее известные:

```
#define идентификатор строка_текста
```

Директива обеспечивает замену идентификатора, встречающегося в тексте программы, на соответствующую строку текста. Наиболее часто она применяется для символического обозначе-

ния константы, которая встречается многократно в различных частях программы. Например, размерность массива:

```
#define SIZE 100
int A[SIZE];
for (i=0; i<SIZE; i++) {...}
```

В данном примере вместо имени SIZE в текст программы будет подставлена строка, содержащая константу 100. В том случае, если нас не устраивает размерность массива, нам достаточно увеличить это значение в директиве define и повторно оттранслировать программу.

```
#include <имя_файла>
#include "имя_файла"
```

В текст программы вместо указанной директивы включается текст файла, находящегося в системном или соответственно в текущем (явно указанном) каталоге. При этом в программу включаются тексты заголовочных файлов, содержащие информацию о функциях, находящихся в других объектных модулях и библиотеках. Например, текст

```
#include <stdio.h>
```

включает в программу текст заголовочного файла, содержащего объявления внешних функций из библиотеки стандартного ввода-вывода.

Работу транслятора можно представить состоящей из нескольких фаз. Первая фаза собственно трансляции называется *лексическим анализом*, а программа, выполняющая ее, — *лексическим анализатором* (иногда *сканером*).

Лексика языка программирования представляет собой правила составления слов программы (идентификаторы, константы, служебные слова, комментарии) из символов языка. Особенность любой лексики заключается в том, что ее элементами являются регулярные линейные последовательности символов. Например, ***идентификатор*** — это произвольная последовательность букв, цифр и символа подчеркивания, начинающаяся с буквы или символа подчеркивания.

На вход лексического анализатора подается последовательность символов входного языка, в которой анализатор выделяет простейшие конструкции языка (перечисленные выше), назы-

ваемые лексическими единицами, и заменяет их внутренним представлением — *лексемами*.

Вторую фазу работы транслятора называют *синтаксическим анализом*, а соответствующую ей программу — *синтаксическим анализатором*.

Синтаксис языка программирования — это правила составления предложений языка из отдельных слов. Такими предложениями являются операции, операторы, определения функций и переменных. К особенностям синтаксиса относится принцип вложенности (рекурсивность) правил построения предложений. Это значит, что элемент синтаксиса языка в своем определении прямо или косвенно в одной из его частей содержит сам себя. Например, в определении оператора цикла телом цикла является оператор, частный случай которого — все тот же оператор цикла.

Синтаксический анализатор получает на вход набор лексем и преобразует их в *промежуточный код*, представляющий собой последовательность символов действия или *атомов*. Каждый атом включает описание операции, которую нужно выполнить, с указанием используемых операндов. При этом последовательность расположения атомов в отличие от лексем соответствует порядку выполнения операций, необходимому для получения результата.

На следующей фазе проводится *семантический анализ*.

Семантика языка программирования — это смысл, который закладывается в каждую конструкцию языка. Семантический анализ — это проверка смысловой правильности конструкции. Например, в языке C++ переменная, используемая в выражении, должна быть определена выше. Исходя из этого определения решается вопрос о допустимости применения данной переменной в данном выражении.

Заключительная фаза трансляции — генерация кода и оптимизация.

Генерация кода — это преобразование элементарных действий, полученных в результате лексического, синтаксического и семантического анализа программы, в результирующую объектную программу. Каждому символу действия, поступающему на вход генератора, он ставит в соответствие одну или несколько команд выходного языка. В качестве выходного языка могут быть использованы команды устройства, команды ассемблера либо операторы какого-либо другого языка. В процессе генерации кода производится и его оптимизация.

При проектировании трансляторов выделяются два подхода к процессу трансляции: компилирующий и интерпретирующий.

Компиляция — преобразование объектов (данных и операций над ними) с входного языка в объекты на другом языке для всей программы в целом с последующим выполнением полученной программы в виде отдельного шага.

Интерпретация — анализ отдельного объекта на входном языке с одновременным выполнением, т. е. при компиляции фазы преобразования и выполнения действий разнесены во времени, но зато каждая из них выполняется над всеми объектами программы одновременно. При интерпретации, наоборот, преобразование и выполнение действий объединены во времени, но для каждого объекта программы в отдельности.

Реальный транслятор, как правило, совмещает функции компилятора и интерпретатора, причем граница между ними может перемещаться от входного языка (тогда мы имеем чистый интерпретатор) до машинного кода (тогда речь идет о чистом компиляторе).

Кроме того, для выполнения программы, написанной на определенном формальном языке, после ее компиляции необходим интерпретатор, выполняющий эту программу, но уже записанную на выходном языке компилятора. Таким образом, процессор, память любого компьютера и вся программная среда, создаваемая операционной системой, является ***интерпретатором*** машинного кода.

Структура транслятора

Процесс трансляции не является линейным. Анализ фрагментов программы, формирование внутреннего промежуточного представления и синтез выходного представления программы не являются последовательными процедурами. Эти процессы протекают в несколько этапов, оказывают влияние друг на друга и могут осуществляться параллельно.

Таким образом, обобщенную структуру транслятора можно описать следующим образом:

- каждая фаза транслятора получает файл данных от предыдущей фазы, обрабатывает его, создает внутренние таблицы данных и по ним формирует выходной файл с данными для следующей фазы;
- фазы трансляции вызывают одна другую в процессе обработки соответствующих языковых единиц;

Обработка исходного текста начинается, как правило, с синтаксического анализа — центрального элемента такой структуры, т. е. основной программой транслятора является синтаксический анализатор, который при разборе структурной единицы языка, называемой предложением (выражение, оператор, определение типа или переменной), вызывает лексический анализатор, для чтения очередной лексической компоненты (идентификатора, константы), а по завершении разбора — семантическую процедуру, процедуры генерации кода или интерпретации.

Из этой схемы выпадает только препроцессор, который обычно представляет собой независимую предварительную фазу трансляции.

1.2. Основные понятия и определения

Формальная грамматика, или просто грамматика, в теории формальных языков — это способ описания формального языка, т. е. выделения некоторого подмножества из множества всех слов некоторого алфавита [4].

Определение 1.1. *Алфавит* — это конечное множество символов [5].

Пример 1.1. Алфавит $X = \{a, b, c, +, -, *, /\}$ содержит семь букв, а алфавит $Y = \{00, 01, 10, 11\}$ — четыре буквы, каждая из которых состоит из двух символов.

Определение 1.2. *Цепочкой символов в алфавите V* называется любая конечная последовательность символов этого алфавита.

В качестве соглашения примем, что для обозначения отдельных символов будем использовать буквы латинского алфавита: *a*, *b*, *c*, *d* и т. д., а для обозначения цепочек символов — буквы греческого алфавита: α, β, γ и т. д.

Определение 1.3. Цепочка, которая не содержит ни одного символа, называется *пустой цепочкой*. Для ее обозначения будем использовать символ λ.

Определение 1.4. *Длина цепочки* — это число составляющих ее символов.

Пример 1.2. Если $\alpha = abcdefg$, то длина α равна 7.

Длину цепочки α будем обозначать $|\alpha|$. Длина λ равна 0, или $|\lambda| = 0$.

Определение 1.5. Если α и β — цепочки, то цепочка $\alpha\beta$ называется *конкатенацией* (или *сцеплением*) цепочек α и β.

Пример 1.3. Если $\alpha = ab$ и $\beta = cd$, то $\alpha\beta = abcd$.

Для любой цепочки α всегда $\alpha\lambda = \lambda\alpha = \alpha$.

Более формально цепочка символов в алфавите V определяется следующим образом.

1) λ — цепочка в алфавите V;

2) если α — цепочка в алфавите V и a — символ этого алфавита, то αa или $a\alpha$ — цепочка в алфавите V;

3) β — цепочка в алфавите V тогда и только тогда, когда она является таковой в силу положений пп. 1 и 2.

Определение 1.6. n-й степенью цепочки α (будем обозначать α^n) называется конкатенация n цепочек α. Тогда $\alpha^0 = \lambda$; $\alpha^n = \alpha\alpha^{n-1} = \alpha^{n-1}\alpha$.

Определение 1.7. *Обращением* (или *реверсом*) цепочки α называется цепочка, символы которой записаны в обратном порядке. Обращение цепочки α будем обозначать α^R.

Пример 1.4. Если $\alpha = abcdef$, то $\alpha^R = fedcba$.

Для пустой цепочки: $\lambda = \lambda^R$.

Определение 1.8. *Язык* в алфавите V — это подмножество цепочек конечной длины в этом алфавите.

Определение 1.9. V^* — множество, содержащее все цепочки в алфавите V, включая пустую цепочку λ.

Пример 1.5. Если $V = \{a, b\}$, то $V^* = \{\lambda, a, b, ab, aa, ba, bb, aba, aab, abb, \ldots\}$.

Определение 1.10. Обозначим через V^+ множество, содержащее все цепочки в алфавите V, исключая пустую цепочку λ. Следовательно, $V^* = V^+ \cup \{\lambda\}$.

Таким образом, каждый язык в алфавите V является подмножеством множества V^*.

Определение 1.11. *Порождающая грамматика* — это четверка объектов $G = (T, N, P, S)$, где:

- T — алфавит терминальных символов (терминалов);
- N — алфавит нетерминальных символов (нетерминалов);
- P — множество *правил вывода (продукций)* грамматики вида

$$\alpha \to \beta, \text{ где } \alpha \in (N \cup T)^+, \ \beta \in (N \cup T)^*;$$

- S — *начальный символ* (*цель*) грамматики, $S \in N$.

Терминалами называют символы грамматики, имеющие конкретное неизменяемое значение, из которых образуются слова языка. В формальных языках, используемых на компьютере, в качестве терминалов применяют все или часть стандартных символов ASCII — буквы, цифры, специальные символы и т. д. Далее будем употреблять для обозначения терминальных символов строчные буквы латинского алфавита, цифры, знаки препинания, знаки арифметических действий и другие символы, кроме прописных букв латинского алфавита.

Нетерминалами называют объекты, обозначающие какие-либо *сущности* языка (формулы, арифметические выражения, команды) и не имеющие конкретного символьного значения. Для обозначения нетерминальных символов будем использовать прописные буквы латинского алфавита.

При этом $N \cup T = V$ — словарь грамматики G. Дополнительное условие: $N \cap T = \varnothing$ означает, что ни один символ не может быть одновременно терминальным и нетерминальным.

Для записи правил вывода с одинаковыми левыми частями

$$\alpha \to \beta_1, \ \alpha \to \beta_2, \ ..., \ \alpha \to \beta_n$$

можно использовать сокращенную запись:

$$\alpha \to \beta_1 \mid \beta_2 \mid ... \mid \beta_n.$$

Каждую цепочку β_i ($i = 1, 2, ..., n$) будем называть *альтернативой* правила вывода из цепочки α.

Пример 1.6. Грамматика $G_1 = (\{a, b\}, \{S, A, B\}, P, S)$, где P состоит из правил:

$S \to A + B \mid B + A;$

$A \to a;$

$B \to b,$

порождает цепочки $a + b$ и $b + a$. Альтернативами являются $A + B$ и $B + A$.

Определение 1.12. Цепочка $\beta \in (T \cup N)^*$ *непосредственно выводима* из цепочки $\alpha \in (T \cup N)^+$ в грамматике $G = (T, N, P, S)$, если $\alpha = \mu\gamma\tau$, $\beta = \mu\delta\tau$, где $\mu, \tau, \delta \in (T \cup N)^*$, $\gamma \in (T \cup N)^+$ и правило вывода $\gamma \to \delta$ содержится в P. Непосредственный вывод цепочки β из цепочки α обозначается $\alpha \Rightarrow \beta$.

Пример 1.7. Цепочка $00A11$ непосредственно выводима из $0A1$ в грамматике $G_2 = (\{0, 1\}, \{A, S\}, P, S)$ с правилами:

P: $S \to 0A1$;

$0A \to 00A1$;

$A \to \lambda$.

Определение 1.13. Цепочка $\beta \in (T \cup N)^*$ *выводима* из цепочки $\alpha \in (T \cup N)^+$ в грамматике $G = (T, N, P, S)$, если существуют цепочки $\gamma_0, \gamma_1, \ldots, \gamma_n$ $(n \geq 0)$, такие, что $\alpha \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \Rightarrow \gamma_n \Rightarrow \beta$.

Определение 1.14. Последовательность $\gamma_0, \gamma_1, \ldots, \gamma_n$ называется *выводом длины n*.

Вывод цепочки β из цепочки α обозначается $\alpha \Rightarrow^* \beta$. Вывод цепочки β из цепочки α за один или более шагов обозначается $\alpha \Rightarrow^+ \beta$.

Пример 1.8. Цепочка $000A111$ выводима из S ($S \Rightarrow^* 000A111$) в грамматике G_2 (см. пример 1.7), так как существует вывод $S \Rightarrow \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111$. Длина вывода равна 3.

Определение 1.15. *Языком, порождаемым грамматикой* $G = (T, N, P, S)$, называется множество всех цепочек из алфавита терминальных символов T^*, выводимых из S с помощью правил данной грамматики: $L(G) = \{\alpha \in T^* \mid S \Rightarrow^* \alpha\}$.

Пример 1.9. $L(G_1) = \{a + b, b + a\}$, G_1 из примера 1.6.

Пример 1.10. $L(G_2) = \{0^n1^n \mid n > 0\}$, G_2 из примера 1.7.

Определение 1.16. Если цепочка $\alpha \in (T \cup N)^*$ является выводимой из начального символа S ($S \Rightarrow^* \alpha$) в грамматике $G = = (T, N, P, S)$, то ее называют *сентенциальной формой*.

Таким образом, язык, порождаемый грамматикой, можно определить как множество терминальных сентенциальных форм.

Определение 1.17. Грамматики G_1 и G_2 называются *эквивалентными*, если они порождают один и тот же язык: $L(G_1) = L(G_2)$.

Пример 1.11. Грамматики G_3 и G_4 с правилами

P_3: $S \to AB$;
$A \to a \mid Aa$;
$B \to b \mid Bb$

P_4: $S \to AS \mid SB \mid AB$;
$A \to a$;
$B \to b$

являются *эквивалентными*, так как обе порождают язык:
$L(G_3) = L(G_4) = \{a^n b^m \mid n, m > 0\}$.

Определение 1.18. Грамматики G_1 и G_2 *почти эквивалентны*, если языки, ими порождаемые, отличаются не более, чем на λ $L(G_1) = L(G_2) \cup \{\lambda\}$.

Пример 1.12. Грамматики $G_5 = (\{0, 1\}, \{A, S\}, P_5, S)$ и $G_6 = (\{0, 1\}, \{S\}, P_6, S)$, где

P_5: $S \to 0A1$;
$0A \to 00A1$;
$A \to \lambda$;

P_6: $S \to 0S1 \mid \lambda$,

почти эквивалентны, так как $L(G_5) = \{0^n 1^n \mid n > 0\}$, а $L(G_6) = \{0^n 1^n \mid n \geq 0\}$, т. е. $L(G_6) = L(G_5) \cup \{\lambda\}$.

1.3. Способы задания схем грамматик

Схема грамматики содержит правила вывода, определяющие структуру цепочек порождаемого языка. Для задания правил используются различные формы описания: *символическая*, *форма Бэкуса — Наура*, *итерационная форма* и *синтаксические диаграммы* [2].

При рассмотрении общих свойств грамматик обычно применяют символическую форму задания правил (см. подразд. 1.1). В практических применениях вместо символического представления схем грамматик обычно используют форму Бэкуса — Наура, или синтаксические диаграммы.

1.3.1. Форма Бэкуса — Наура

При описании синтаксиса конкретных языков программирования приходится вводить большое число нетерминальных символов, и символическая форма записи теряет свою наглядность. Для того чтобы этого избежать, применяют форму Бэкуса — Наура (БНФ), в которой нетерминальные символы представляют собой слова естественного языка, заключенные в угловые скобки, а разделителем служит специальный знак — два двоеточия и равенство (::=).

БНФ-конструкция будет содержать несколько предложений следующего вида:

<символ> ::= <цепочка1> | <цепочка2> | ... | <цепочка$_n$>.

Пример 1.13. Заданные в символической форме правила грамматики ($L \to EL$; $L \to E$, где символу L соответствует понятие «список», а символу E — понятие «элемент списка») можно заменить следующей БНФ-конструкцией:

<список> ::= <элемент списка><список>
<список> ::= <элемент списка>

или сокращенной БНФ-конструкцией:

<список> ::= <элемент списка><список> | <элемент списка>.

Пример 1.14. Следующая БНФ-конструкция служит для описания целого числа:

<целое> ::= <целое б/зн.> | + <целое б/зн.> | – <целое б/зн.>
<целое б/зн.> ::= <цифра> | <целое б/зн.><цифра>
<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.

1.3.2. Итерационная форма

Для получения более компактных описаний синтаксиса применяют итерационную форму. Она предполагает введение специальной операции, которая называется итерацией и обозначается парой фигурных скобок со звездочкой. Итерация вида $\{a\}^*$ определяется как множество, включающее цепочки всевозмож-

ной длины, построенные с использованием символа a, и пустую цепочку

$\{a\}^* = \{\lambda, a, aa, aaa, aaaa, ...\}$.

Пример 1.15. Используя итерацию в символической форме описания списков, получим:

$L \to E \{ E \}^*$.

Пример 1.16. Описание множества цепочек, каждая из которых должна начинаться с буквы a и может состоять из произвольного числа букв b и c, может быть представлено в итерационной форме так:

$I \to a \{b \mid c\}^*$.

Кроме того, в итерационных формах наряду с фигурными скобками применяют квадратные скобки, чтобы показать, что цепочка, заключенная в них, может быть опущена. С помощью таких скобок правила

$A \to xAyBz$;

$A \to xBz$

могут быть записаны следующим образом:

$A \to x\ [Ay]\ Bz$.

1.3.3. Синтаксические диаграммы

Представление правил грамматики в графической форме в виде синтаксических диаграмм служит для облегчения понимания сложных синтаксических конструкций посредством зрительного восприятия.

Правила построения синтаксических диаграмм можно сформулировать следующим образом.

1. Правилу вида $A \to \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_k$ ставится в соответствие диаграмма, структура которой определяется правой частью правила.

2. Каждое появление терминального символа x в цепочке α_i изображается на диаграмме дугой, помеченной этим символом x, заключенным в кружок (рис. 1.1).

3. Каждому появлению нетерминального символа <A> в цепочке α_i ставится в соответствие на диаграмме дуга, помеченная символом, заключенным в квадрат (рис. 1.2).

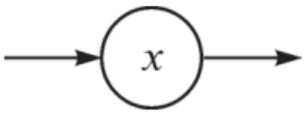


Рис. 1.1. Обозначение терминала на диаграмме

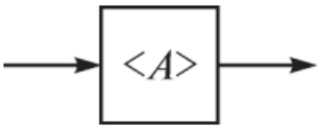


Рис. 1.2. Обозначение нетерминала на диаграмме

4. Правило грамматики вида $A \to \alpha_1\alpha_2...\alpha_n$ изображается на диаграмме в соответствии с рис. 1.3.

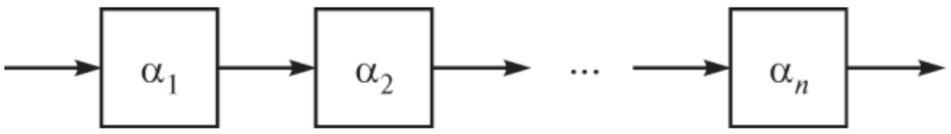


Рис. 1.3. Обозначение последовательности цепочек на диаграмме

5. Правило грамматики, имеющее вид $A \to \alpha 1 \mid \alpha 2 \mid ... \mid \alpha_n$, изображается на диаграмме в соответствии с рис. 1.4.

6. Если порождающее правило задано в виде итерации $A \to \{\alpha\}^*$, то ему соответствует синтаксическая диаграмма (рис. 1.5).

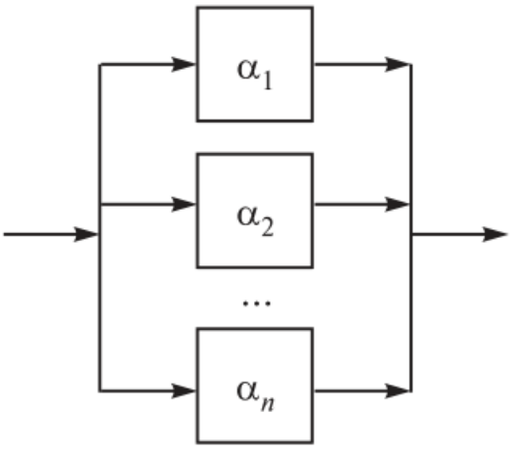


Рис. 1.4. Обозначение альтернативных цепочек на диаграмме

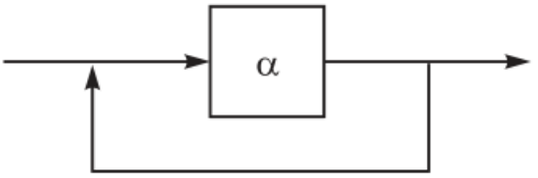


Рис. 1.5. Обозначение итерации на диаграмме

Количество правил порождающей грамматики определяет число построенных для нее синтаксических диаграмм. Для того чтобы сократить это число, диаграммы объединяют, заменяя нетерминальные символы построенными для них диаграммами.

Пример 1.17. На рис. 1.6 приведена синтаксическая диаграмма правила грамматики для описания идентификаторов; БНФ-форма представлена ниже.

<идентификатор> ::= <буква><продолжить> | <_><продолжить>
<продолжить> ::= <буква><продолжить> | <цифра><продолжить> |
<_><продолжить> | <пусто>.

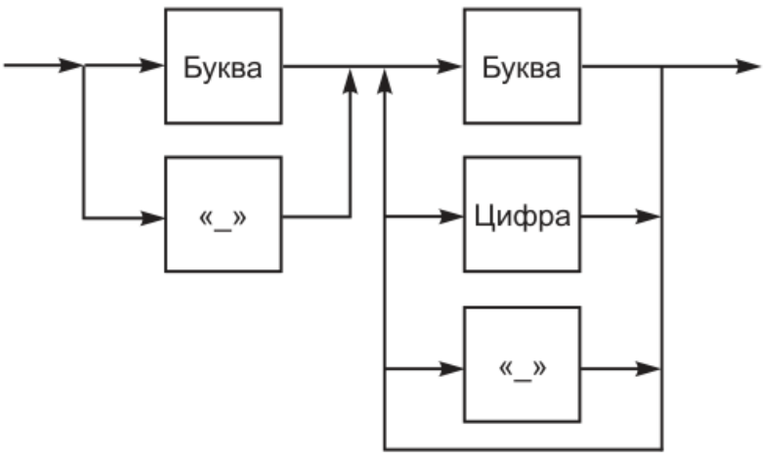


Рис. 1.6. Пример синтаксической диаграммы для описания идентификатора

1.4. Классификация грамматик и языков по Хомскому

1.4.1. Иерархия Хомского

Иерархия Хомского — классификация формальных языков и формальных грамматик, согласно которой они делятся на четыре типа в соответствии с их условной сложностью [2, 5—7].

Тип 0:

Грамматика $G = (T, N, P, S)$ называется *неограниченной грамматикой*, если на правила вывода не накладывается никаких ограничений (кроме тех, которые указаны в определении грамматики).

Тип 1:

Грамматика $G = (T, N, P, S)$ называется *контекстно-зависимой* (КЗ), или *неукорачивающей грамматикой*, если каждое правило из P имеет вид $\alpha \to \beta$, где $\alpha \in (T \cup N)^+$, $\beta \in (T \cup N)^+$ и $|\alpha| \le |\beta|$.

Тип 2:

Грамматика $G = (T, N, P, S)$ называется *контекстно-свободной* (КС), если каждое правило из P имеет вид $A \to \alpha$, где $A \in N$ — нетерминал, а $\alpha \in (T \cup N)^*$ — строка из нетерминалов и терминалов или пустая строка.

Тип 3:

Грамматика $G = (T, N, P, S)$ называется *праволинейной*, если каждое правило из P имеет вид $A \to aB$ либо $A \to a$, где $A \in N$, $B \in N$, $a \in T$.

Грамматика $G = (T, N, P, S)$ называется *леволинейной*, если каждое правило из P имеет вид $A \to Ba$ либо $A \to a$, где $A \in N$, $B \in N$, $a \in T$.

Леволинейные и праволинейные грамматики являются *регулярными* грамматиками.

1.4.2. Соотношения между типами грамматик и языков

Формальные языки классифицируются по типу порождающих их грамматик. Заметим, что один и тот же язык может порождаться грамматиками различного типа. Поэтому, когда говорят о типе языка, имеют в виду максимально возможный его номер [2].

Между типами формальных грамматик существуют следующие соотношения.

1. Любая регулярная грамматика является КС-грамматикой.
2. Любая КС-грамматика, кроме *укорачивающей* (в правой части правил которой есть пустая цепочка), является КЗ-грамматикой (неукорачивающей грамматикой);
3. Любая КЗ-грамматика является грамматикой типа 0.

Между типами формальных языков существуют следующие соотношения:

1. Каждый регулярный язык является КС-языком, но существуют КС-языки, которые не являются регулярными (например, $L = \{a^n b^n \mid n > 0\}$).
2. Каждый КС-язык, кроме *укорачивающего* (содержащего пустую цепочку), является КЗ-языком, но существуют КЗ-языки, которые не являются КС-языками (например, $L = \{a^n b^n c^n \mid n > 0\}$).
3. Каждый КЗ-язык является языком типа 0.

Пример 1.18. Рассмотрим две грамматики, порождающие идентификаторы языка программирования, состоящие из первой буквы, за которой следует целое число. Грамматика $G_8 = (\{a, 0, 1, ..., 9\}, \{D, S\}, P, S)$ является праволинейной и содержит следующие продукции:

$S \to a \mid aD;$

$D \to 0 \mid 1 \mid 2 \mid 3 \mid ... \mid 9 \mid 0D \mid 1D \mid 2D \mid 3D \mid ... \mid 9D.$

Грамматика $G_9 = (\{a, 0, 1, ..., 9\}, \{D, F, S\}, P, S)$ определяет то же самое множество идентификаторов, но представляет собой контекстно-свободную грамматику с множеством продукций:

$S \to a \mid aD;$

$F \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9;$

$D \to F \mid DF.$

В эту грамматику дополнительно включен нетерминальный символ F для обозначения цифр. Язык, порождаемый обеими грамматиками, представляет собой множество предложений ax, где $x \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^*$.

1.4.3. Примеры грамматик и порождаемых ими языков

Язык типа 0:

$L = \{a^2 \mid n \geq 1\}$.

P: $S \to aaCFD$;

$F \to AFB \mid AB$;

$AB \to bBA$;

$Ab \to bA$;

$AD \to D$;

$Cb \to bC$;

$CB \to C$;

$bCD \to \lambda$.

Язык типа 1:

L = {цепочки из 0 и 1 с одинаковым числом 0 и 1}.

P: $S \to ASB \mid AB$;

$AB \to BA$;

$A \to 0$;

$B \to 1$.

Язык типа 2:

$L = \{(ac)^n(cb)^n \mid n > 0\}$.

P: $S \to aQb \mid accb$;

$Q \to cSc$.

Язык типа 3:

$L = \{\omega \perp \mid \omega \in \{a, b\}^+$, где нет рядом стоящих $a\}$.

P: $S \to A \perp \mid B \perp$;

$A \to a \mid Ba$;

$B \to b \mid Bb \mid Ab$.

Здесь символ $\perp$ обозначает конец цепочки [5].

1.5. Разбор цепочек

Одной из основных задач анализа является решение вопроса о принадлежности цепочки данному языку [2]. Если цепочка принадлежит языку, порождаемому грамматикой, то существует ее вывод из целевого нетерминала этой грамматики. Процесс построения такого вывода (а, следовательно, и определения принадлежности цепочки языку) называется *разбором*.

С практической точки зрения наибольший интерес представляют контекстно-свободные грамматики, которые используются в качестве порождающих грамматик для большинства синтаксических структур языков программирования. Существует множество хорошо известных и отработанных способов решения задачи разбора для КС-грамматик [5].

В дальнейшем методы разбора будут рассматриваться для различных подклассов КС-грамматик.

1.5.1. Виды разбора

Рассмотрим некоторые понятия, связанные с разбором по КС-грамматике [2, 5].

Определение 1.19. Вывод цепочки $\alpha \in (T)^*$ из $S \in N$ в КС-грамматике $G = (T, N, P, S)$ называется *левым* (*левосторонним*), если в этом выводе каждая очередная сентенциальная форма получается из предыдущей заменой самого левого нетерминала.

Определение 1.20. Вывод цепочки $\alpha \in (T)^*$ из $S \in N$ в КС-грамматике $G = (T, N, P, S)$ называется *правым* (*правосторонним*), если в этом выводе каждая очередная сентенциальная форма получается из предыдущей заменой самого правого нетерминала.

Для одной и той же цепочки может существовать несколько выводов. Выводы являются эквивалентными, если в них в одних и тех же местах применяются одни и те же правила вывода в различном порядке.

Пример 1.19. Для цепочки $a + b + a$ в грамматике

$G = (\{a, b\}, \{S, T\}, \{S \to T \mid T + S;\ T \to a \mid b\}, S)$

можно построить выводы:

(1) $S \Rightarrow T+S \Rightarrow a+S \Rightarrow a+T+S \Rightarrow a+b+S \Rightarrow a+b+T \Rightarrow a+b+a$;

(2) $S \Rightarrow T+S \Rightarrow T+T+S \Rightarrow T+T+T \Rightarrow T+T+a \Rightarrow T+b+a \Rightarrow a+b+a$;

(3) $S \Rightarrow T+S \Rightarrow T+T+S \Rightarrow T+T+T \Rightarrow a+T+T \Rightarrow a+b+T \Rightarrow a+b+a$.

Здесь (1) — левосторонний вывод, (2) — правосторонний, а (3) не является ни левосторонним, ни правосторонним, но все эти выводы — эквивалентные в указанном выше смысле.

1.5.2. Дерево вывода. Нисходящий и восходящий разбор

Для КС-грамматик можно ввести удобное графическое представление вывода, называемое *деревом вывода*, причем для всех эквивалентных выводов деревья вывода совпадают [4, 5, 8].

Определение 1.21. Дерево есть дерево вывода для контекстно-свободной грамматики $G = (N, T, P, S)$, если оно удовлетворяет следующим условиям [5]:

1) каждый узел имеет метку — символ из алфавита $V = N \cup T$;

2) метка корня — S;

3) если узел имеет по крайней мере одного потомка, то его метка должна быть нетерминалом;

4) если узлы $n_1, n_2, ..., n_k$ — прямые потомки узла n, перечисленные слева направо, с метками $A_1, A_2, ..., A_k$ соответственно, а метка узла n есть A, то грамматика G должна содержать правило:

$A \to A_1A_2...A_k \in P$.

Пример 1.20. Рассмотрим КС-грамматику $G = (\{a, b\}, \{S, A\}, P, S)$, где $P = \{S \to aAS, S \to a, A \to SbA, A \to ba, A \to SS\}$.

На рис. 1.7 изображено дерево, представляющее вывод:

$S \Rightarrow aAS \Rightarrow aSbAS \Rightarrow aabAS \Rightarrow aabbaS \Rightarrow aabbaa$.

Дерево вывода может быть построено нисходящим или восходящим способом разбора. При нисходящем разборе дерево вывода строится от корня к листьям. На каждом шаге метка вершины — нетерминал из левой части правила — заменяется правой частью правила, которое позволяет получить символы исходной цепочки. Восходящий разбор заключается в том, что исходную цепочку пытаются «свернуть» к начальному символу S.

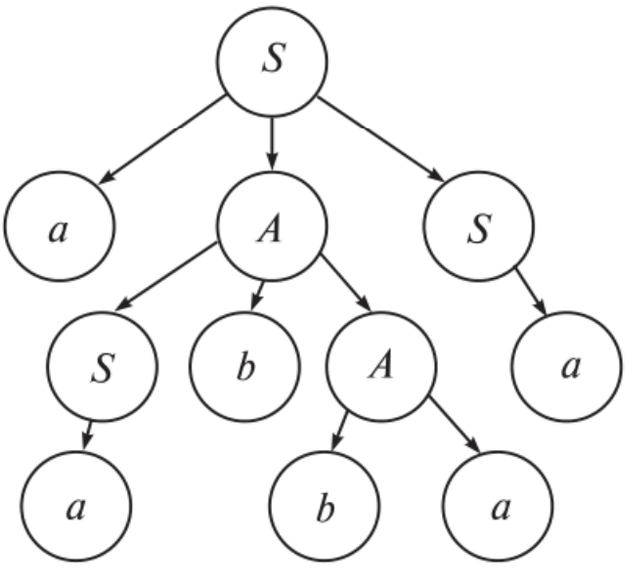


Рис. 1.7. Дерево вывода

На каждом шаге ищут подцепочку, совпадающую с правой частью какого-либо правила вывода и заменяют ее нетерминалом из левой части этого правила.

1.5.3. Неоднозначность грамматик

Может возникнуть ситуация, когда в одной и той же грамматике для цепочки можно сформировать не одно, а два и более различных деревьев вывода [4, 8].

Определение 1.22. Контекстно-свободная грамматика G называется *неоднозначной*, если в языке $L(G)$, существует хотя бы одна цепочка, для которой может быть построено два или более различных деревьев вывода. В противном случае грамматика называется *однозначной*.

Это также означает, что в однозначной грамматике существует не более одного левостороннего (правостороннего) вывода цепочки.

Определение 1.23. Язык, порождаемый грамматикой, называется *неоднозначным*, если он не может быть порожден никакой однозначной грамматикой.

С точки зрения формального языка, заданного грамматикой, не имеет значения, какая цепочка вывода и какое дерево вывода из возможных вариантов будут построены. Однако для языков программирования, которые не являются чисто формальными языками и несут некоторую смысловую нагрузку, это важно.

Пример 1.21. Формальная грамматика

$G = (\{+, -, *, /, (,), a, b\}, \{S\}, P, S)$;

P: $S \to S+S \mid S-S \mid S*S \mid S/S \mid (S) \mid a \mid b$

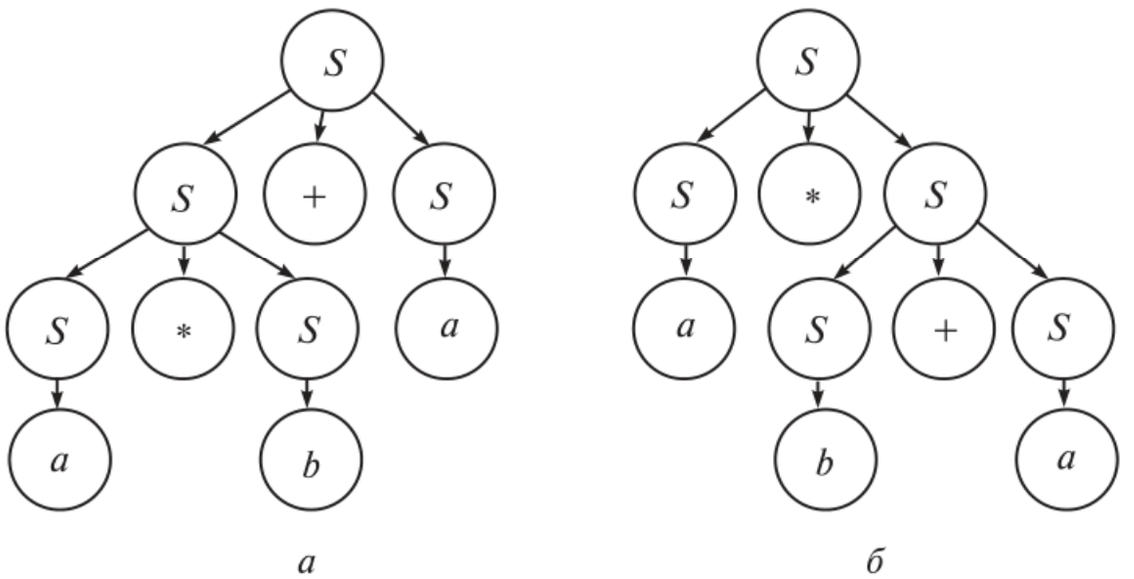
определяет язык арифметических операций. Примерами цепочек языка, заданного такой грамматикой, являются $a * b - a$, $a * (a + b)$, $a * b + a * a$ и т. д.

Построим левосторонний вывод для цепочки $a * b + a$. Получится два варианта:

а) $S \Rightarrow S+S \Rightarrow S*S+S \Rightarrow a*S+S \Rightarrow a*b+S \Rightarrow a*b+a$;

б) $S \Rightarrow S*S \Rightarrow a*S \Rightarrow a*S+S \Rightarrow a*b+S \Rightarrow a*b+a$.

Каждому из этих вариантов будет соответствовать свое дерево вывода (рис. 1.8). Рассмотренная грамматика явно неоднозначна.



а *б*

Рис. 1.8. Дерево вывода цепочки $a*b+a$ в неоднозначной грамматике: *а* — вариант 1; *б* — вариант 2

Поскольку рассмотренная выше грамматика определяет язык арифметических выражений, то порядок построения дерева в ней соответствует порядку выполнения арифметических операций. Но в данной грамматике этот порядок не определен и поэтому все операции равноправны, что неправильно с точки зрения арифметики, где операции умножения и деления обладают более высоким приоритетом по сравнению с операциями сложения и вычитания. И хотя синтаксическая структура полученных выражений правильная, не определен порядок выполнения операций слева направо. Такая ситуация связана с неоднозначностью грамматики.

Неоднозначность — это свойство грамматики, а не языка. Для некоторых языков, заданных неоднозначными грамматиками, иногда удается построить эквивалентную однозначную грамматику (задающую тот же язык). Для того чтобы убедиться в том, что некоторая грамматика является неоднозначной, согласно определению достаточно найти в заданном ею языке хотя бы одну цепочку, которая допускала бы более чем один левосторонний или правосторонний вывод. Однако не всегда удается легко обнаружить такую цепочку символов, поскольку перебрать все цепочки языка, как правило, невозможно — их может быть бесконечно много, так же как невозможно доказать и однозначность грамматики. Данная проблема в общем случае алгоритмически неразрешима.

Можно указать некоторые виды правил вывода, которые приводят к неоднозначности:

- $A \to AA \mid \alpha$;
- $A \to A\alpha A \mid \beta$;
- $A \to \alpha A \mid A\beta \mid \gamma$;
- $A \to \alpha A \mid \alpha A \beta A \mid \gamma$.

Алгоритмически неразрешимой является в общем случае проблема проверки эквивалентности двух грамматик так же, как и проблема преобразования произвольной неоднозначной грамматики в эквивалентную ей однозначную. Однако в большинстве случаев эта задача разрешима [2].

Пример 1.22. Для рассмотренной в примере 1.21 неоднозначной грамматики арифметических выражений над операндами a и b существует эквивалентная ей однозначная грамматика следующего вида:

$$G' = (\{+, -, *, /, (,), a, b\}, \{S, T, E\}, P', S);$$

$$P': \quad S \to S + T \mid S - T \mid T;$$

$$T \to T * E \mid T/E \mid E;$$

$$E \to (S) \mid a \mid b.$$

В этой грамматике для цепочки символов языка $a*b+a$ возможен только один левосторонний вывод:

$$S \Rightarrow S + T \Rightarrow T + T \Rightarrow T * E + T \Rightarrow E * E + T \Rightarrow a * E + T \Rightarrow$$

$$\Rightarrow a * b + T \Rightarrow a * b + E \Rightarrow a * b + a.$$

Дерево вывода приведено на рис. 1.9.

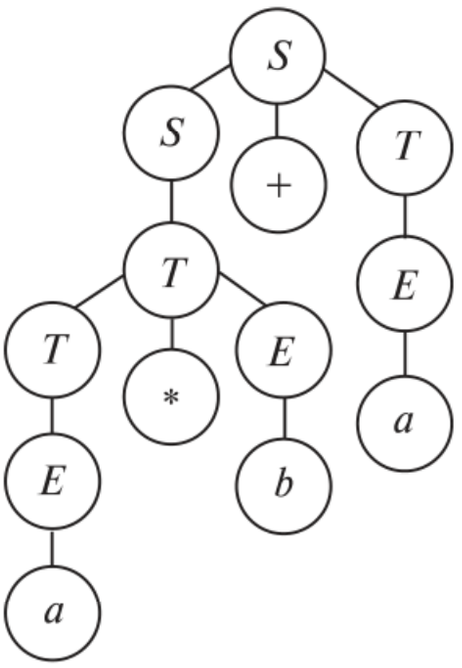


Рис. 1.9. Дерево вывода цепочки *a*b+a* в однозначной грамматике

Приоритет арифметических операций соответствует порядку построения дерева разбора в однозначной грамматике. Таким образом, для построения трансляторов и языков программирования, которые не являются чисто формальными, а несут некоторую смысловую нагрузку, необходимо использовать только однозначные грамматики.

1.6. Распознаватели. Некоторые подходы к классификации

Один из методов описания языков базируется на использовании распознавателей.

Определение 1.24. Распознаватель — это специальный алгоритм, который позволяет определить принадлежность цепочки символов некоторому языку [3].

Таким образом, задача разбора в общем виде заключается в следующем: на основе имеющейся грамматики некоторого языка построить распознаватель для этого языка.

1.6.1. Общая схема распознавателя

Компилятор должен распознать исходную программу и построить эквивалентную ей результирующую программу. В отношении исходной программы человек, написавший ее, выступает

в качестве *генератора цепочек*, а компилятор — в качестве распознавателя.

Задача распознавателя заключается в том, чтобы на основе исходной цепочки дать ответ, принадлежит она заданному языку или нет. Распознаватели, как было сказано выше, представляют собой один из способов определения языка.

В общем виде распознаватель можно отобразить в виде условной схемы (автомата), представленной на рис. 1.10 [3]. Условной эта схема является потому, что распознаватель, входящий в состав компилятора, представляет собой часть программного обеспечения, а не часть устройства компьютера.

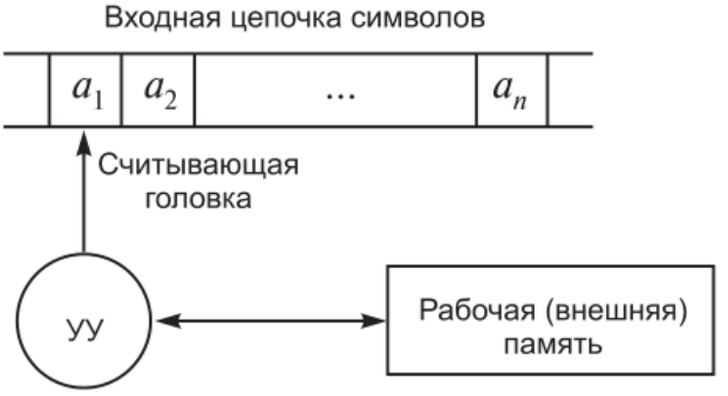


Рис. 1.10. Условная схема распознавателя [3]

Этот автомат состоит из трех частей: входной ленты, устройства управления с конечной памятью и внешней (рабочей) памяти.

Входная лента — линейная последовательность ячеек, каждая из которых содержит один входной символ из конечного входного алфавита. Могут присутствовать левый и правый концевые маркеры, может присутствовать только один концевой маркер (левый или правый), могут отсутствовать оба маркера.

Считывающая головка в каждый момент читает одну входную ячейку. За один шаг входная головка может сдвинуться на одну ячейку влево, вправо и остаться неподвижной.

Внешняя память хранит информацию, построенную только из символов конечного алфавита памяти. Может иметь различную структуру: очередь, стек (магазин) и т. д. Можно читать из так называемой вспомогательной памяти и писать в нее. Для стека и очереди используются специфические операции (вталкивание, выталкивание).

Устройство управления с конечной памятью (УУ) — программа, управляющая поведением распознавателя. Может являться аналогом конечного автомата. Определяет перемещение входной

головки и работу с памятью на каждом шаге (такте). Переходит за шаг из одного состояния в другое (состояние сохраняется в памяти УУ).

Распознаватель работает с символами своего алфавита, который является конечным. Алфавит распознавателя включает все допустимые символы входных цепочек, а также некоторый дополнительный алфавит символов, которые могут обрабатываться УУ и храниться в рабочей памяти распознавателя.

Вся работа распознавателя состоит из *последовательности тактов*. В начале каждого такта состояние распознавателя определяется его конфигурацией. В процессе работы конфигурация распознавателя меняется.

Конфигурация распознавателя определяется следующими параметрами:

- содержимым входной цепочки символов и положением считывающей головки в ней;
- состоянием УУ;
- содержимым внешней памяти.

Начальная конфигурация — устройство управления находится в заданном начальном состоянии, входная головка читает самый левый символ на входной ленте, память имеет заранее установленное начальное содержимое.

Заключительная конфигурация — устройство управления находится в одном из состояний, принадлежащем заранее выделенному множеству заключительных состояний, входная головка обозревает правый концевой маркер или, если маркер отсутствует, сошла с конца входной ленты. Иногда требуется, чтобы заключительная конфигурация памяти удовлетворяла некоторым условиям.

Распознаватель *допускает входную цепочку символов* α, если, находясь в начальной конфигурации и получив на вход эту цепочку, он может проделать последовательность шагов, заканчивающуюся одной из его конечных конфигураций.

Язык, определяемый распознавателем, — это множество всех цепочек, которые допускает распознаватель.

1.6.2. Виды распознавателей

Распознаватели различаются в зависимости от совокупности их компонентов: считывающего устройства, устройства управления (УУ) и внешней памяти [2].

По видам считывающего устройства распознаватели могут быть *двусторонние* и *односторонние*.

У односторонних распознавателей считывающая головка перемещается по ленте входных символов только в одном направлении — слева направо, так же, как читается входная программа. Когда говорят об односторонних распознавателях, то имеют в виду левосторонние, которые читают исходную цепочку слева направо без возвращения к уже прочитанной части цепочки. Двусторонние распознаватели допускают перемещение считывающей головки в обоих направлениях: как вперед, от начала ленты к концу, так и назад, возвращаясь к уже прочитанным символам.

По видам устройства управления распознаватели бывают *детерминированные* и *недетерминированные*.

Детерминированным называется распознаватель, у которого для каждой допустимой на некотором шаге работы существует единственно возможная конфигурация, в которую распознаватель перейдет на следующем шаге. У недетерминированного распознавателя существует хотя бы одна допустимая на некотором шаге конфигурация, из которой возможен переход во множество возможных на следующем шаге конфигураций.

По видам внешней памяти распознаватели бывают следующих типов:

- без внешней памяти;
- с ограниченной внешней памятью;
- с неограниченной внешней памятью.

У распознавателей без внешней памяти используется только конечная память устройства управления.

Для распознавателей с ограниченной внешней памятью размер внешней памяти ограничен в зависимости от длины исходной цепочки символов. Эти ограничения могут налагаться некоторой зависимостью объема памяти от длины цепочки — линейной, полиномиальной, экспоненциальной и т. д. Кроме того, для таких распознавателей может быть указан способ организации внешней памяти — стек, очередь, список и т. п.

Распознавателям с неограниченной внешней памятью для работы может потребоваться внешняя память неограниченного объема (как правило, вне зависимости от длины входной цепочки). Для означенных распознавателей характерна память с произвольным доступом.

Чем выше в данной классификации распознаватель, тем сложнее создавать алгоритм, обеспечивающий его работу. Разра-

батывать двусторонние распознаватели сложнее, чем односторонние. Можно заметить, что недетерминированные распознаватели по сложности выше детерминированных. Зависимость затрат на создание алгоритма от типа внешней памяти также очевидна.

1.6.3. Классификация распознавателей по типам языков

Сложность распознавателя также напрямую связана с типом языка, входные цепочки которого может принимать (допускать) распознаватель [2, 3].

Выше было определено четыре основных типа языков. Доказано, что для каждого из этих типов языков существует свой тип распознавателя с определенным составом компонентов и, следовательно, с заданной сложностью алгоритма работы.

Для *языков с фразовой структурой* (тип 0) необходим распознаватель, равномощный машине Тьюринга — *недетерминированный двусторонний автомат, имеющий неограниченную внешнюю память.* Поэтому для языков данного типа нельзя гарантировать, что за ограниченное время на ограниченных вычислительных ресурсах распознаватель завершит работу и примет решение о том, принадлежит или нет входная цепочка заданному языку. Отсюда можно заключить, что практического применения языки с фразовой структурой не имеют, а потому далее они не рассматриваются.

Для *контекстно-зависимых языков* (тип 1) распознавателями являются *двусторонние недетерминированные автоматы с линейно ограниченной внешней памятью.* Алгоритм работы такого автомата в общем случае имеет экспоненциальную сложность — количество шагов (тактов), необходимых автомату для распознавания входной цепочки, экспоненциально зависит от длины этой цепочки. Следовательно, и время, необходимое на разбор входной цепочки по заданному алгоритму, экспоненциально зависит от длины входной цепочки символов.

Как правило, такие распознаватели применяются для автоматизированного перевода и анализа текстов на естественных языках, когда временные ограничения на разбор текста несущественны. В компиляторах контекстно-зависимые распознаватели не применяются, поскольку скорость работы компилятора имеет существенное значение, а синтаксический разбор текста про-

граммы можно выполнять в рамках более простого, контекстно-свободного типа языков.

Для *контекстно-свободных языков* (тип 2) распознавателями являются *односторонние недетерминированные автоматы с магазинной (стековой) внешней памятью — МП-автоматы.* При простейшей реализации алгоритма работы такого автомата он имеет экспоненциальную сложность, однако путем некоторых усовершенствований алгоритма можно добиться полиномиальной (кубической) зависимости времени, необходимого на разбор входной цепочки, от длины этой цепочки. Следовательно, можно говорить о полиномиальной сложности распознавателя для КС-языков.

Среди всех КС-языков можно выделить класс *детерминированных КС-языков*, распознавателями для которых являются *детерминированные автоматы с магазинной (стековой) внешней памятью — ДМП-автоматы.* Эти языки обладают свойством однозначности — доказано, что для любого детерминированного КС-языка всегда можно построить однозначную грамматику. Кроме того, для таких языков существует алгоритм работы распознавателя с квадратичной сложностью. Поскольку эти языки являются однозначными, именно они представляют наибольший интерес для построения компиляторов.

Более того, среди всех детерминированных КС-языков существуют такие классы языков, для которых можно построить *линейный распознаватель* — распознаватель, у которого время принятия решения о принадлежности цепочки языку имеет линейную зависимость от длины цепочки. Синтаксические конструкции практически всех существующих языков программирования могут быть отнесены к одному из таких классов языков. Это обстоятельство очень важно для разработки современных быстродействующих компиляторов.

Для *регулярных языков* (тип 3) распознавателями являются *детерминированные автоматы без внешней памяти — конечные автоматы* (КА). Это очень простой тип распознавателя, который всегда предполагает линейную зависимость времени на разбор входной цепочки от ее длины. Кроме того, конечные автоматы имеют важную особенность: любой недетерминированный КА всегда может быть преобразован в детерминированный. Это обстоятельство существенно упрощает разработку программного обеспечения, обеспечивающего функционирование распознавателя.

Простота и высокая скорость работы распознавателей определяют широкую область применения регулярных языков.

В компиляторах распознаватели на основе регулярных языков используются для лексического анализа текста исходной программы — выделения в нем простейших конструкций языка, таких как идентификаторы, строки, константы и т. п. Это позволяет существенно сократить объем исходной информации и упростить синтаксический разбор программы.

Кроме компиляторов регулярные языки находят применение еще во многих областях, связанных с разработкой программного обеспечения вычислительных систем. На их основе функционируют многие командные процессоры как в системном, так и в прикладном программном обеспечении. Для регулярных языков существуют развитые, математически обоснованные механизмы, которые позволяют облегчить создание распознавателей. Они положены в основу существующих разнообразных программных средств, которые позволяют автоматизировать этот процесс.

Контрольные вопросы

1. Что такое алфавит и цепочка (строка) символов в алфавите?
2. Дайте определение формальной грамматики и формального языка.
3. Объясните различие между терминальными и нетерминальными символами формальной грамматики.
4. Что такое правила вывода и вывод цепочки?
5. В каком случае цепочка β выводима из цепочки α?
6. Дайте понятие *сентенциальной формы* грамматики.
7. Опишите способы задания схем грамматик.
8. Приведите классификацию формальных языков и грамматик по Хомскому.
9. Как построить дерево вывода?
10. Какие существуют виды разбора цепочек?
11. В чем заключается неоднозначность грамматик и языков?
12. Дайте определение и общую схему распознавателя.
13. Приведите классификацию распознавателей по составляющим их компонентам.
14. Приведите классификацию распознавателей по типам языков.

Глава 2
ЛЕКСИЧЕСКИЙ АНАЛИЗ

Лексический анализ — первая наиболее простая фаза трансляции, выполняемая программой — лексическим анализатором (сканером). При этом сканер считывает поток входных символов и выделяет элементарные конструкции языка — лексемы, которые затем передает процедуре синтаксического разбора. Как правило, для лексического анализа служат конечные автоматы и регулярные грамматики [1, 3, 4, 9].

2.1. Лексический анализатор как конечный автомат

В компьютерной технике широко применяется такой математический аппарат, как конечные автоматы (КА). Например, КА используются при разработке управляющей схемы, реализующей заданный алгоритм, для реализации аппаратных устройств. Программные КА являются формальной основой лексического анализатора [1].

Конечный автомат — это один из самых простых механизмов, используемых при работе с языками. У этого распознавателя есть только фиксированный набор ячеек памяти, а управляющее

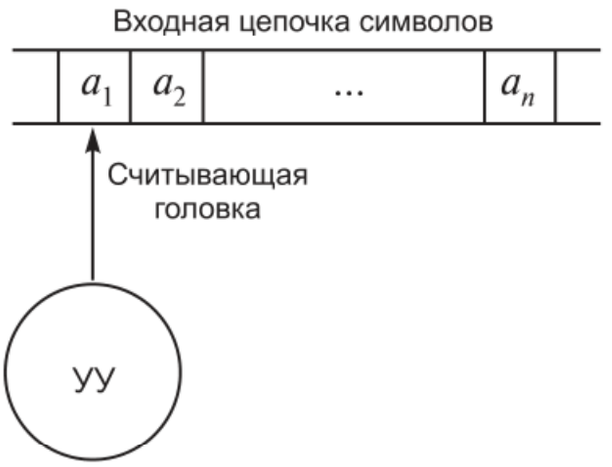


Рис. 2.1. Условная схема распознавателя типа конечного автомата

устройство может только сдвигаться вправо по входной ленте и изменять состояния, хранящиеся в его памяти. Условная схема распознавателя в виде конечного автомата приведена на рис. 2.1.

2.1.1. Диаграмма состояний-переходов конечного автомата

Конечный автомат представляет собой систему, которая в каждый момент времени может находиться в одном из заданных состояний конечного множества. При нахождении КА в определенном состоянии и поступлении на вход одного из множества входных сигналов автомат переходит в однозначно определенное состояние и вырабатывает определенный выходной сигнал. Иными словами, находясь в состоянии *A*, при получении сигнала *a* автомат переходит в состояние *B* и выполняет действие *d*. Каждый такой переход называют *шагом* работы конечного автомата.

Обычно поведение конечного автомата представляют в виде его диаграммы состояний-переходов. На диаграмме каждое состояние автомата изображается в виде кружка с подписью — «именем» состояния, каждый переход обозначается дугой со стрелкой, надпись над которой соответствует входному воздействию. На рис. 2.2 показан пример диаграммы состояний-перехо-

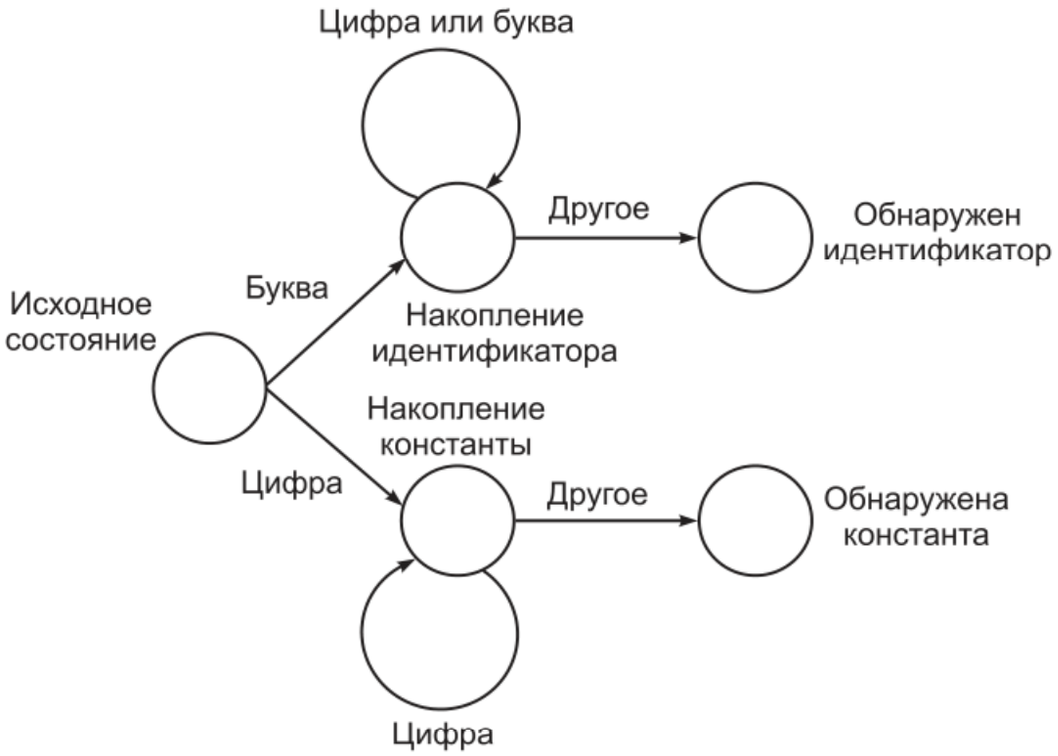


Рис. 2.2. Пример диаграммы состояний-переходов конечного автомата

дов конечного автомата, распознающего идентификатор на входе, или константа.

Для того чтобы конечный автомат мог использоваться в качестве лексического анализатора, должны выполняться несколько условий:

- автомат должен иметь некоторое начальное состояние, из которого он начинает работу;
- автомат должен иметь конечное число состояний;
- в каждом состоянии не могут осуществляться переходы в разные состояния по одному и тому же условию (входному сигналу), т. е. автомат должен быть *детерминированным*.

Пример 2.1. Рассмотрим программу, которая распознает в строке комментарии типа /* ... */, реализованную в виде конечного автомата. Для начала определим состояния КА:

- состояние 0 — распознавание обычного текста;
- состояние 1 — обнаружен символ «/»;
- состояние 2 — обнаружено начало комментария «/*»;
- состояние 3 — в комментарии обнаружен символ «*».

Программа, представляющая собой КА, выглядит следующим образом:

```
void f(char in[],char out[])
{int i, state, j;
for(i=0,state=0,j=0; in[i]!=0; i++)
switch(state)
  {
case 0: if (in[i])!='/' out[j++] = in[i];
        else state=1;
        break;
case 1: if (in[i]!='*')
        {out[j++]=in[i-1]; out[j++]=in[i]; state=0;}
        else state=2;
        break;
case 2: if (in[i]=='*') state=3;
        break;
case 3: if (in[i]=='/') state=0;
        break;
  }
}
```

В данной программе представлен цикл, завершающийся окончанием входной строки символов. На каждом шаге цикла анализируется текущий символ строки, являющийся входным воздействием, по которому происходит переход из состояния в состояние. Текущее состояние автомата хранится в переменной `state`.

2.1.2. Формальное определение конечного автомата

Определение 2.1. Конечный автомат — это пятерка $A = (Q, T, \delta, q_0, F)$, где [2, 3, 7, 9]:

Q — конечное множество состояний автомата;

T — непустое конечное множество допустимых входных символов;

δ — отображение множества $Q \times T \Rightarrow R(Q)$, определяющее поведение управляющего устройства (эту функцию часто называют функцией переходов). $\delta(A, a) = B$ означает, что из состояния A по входному символу a происходит переход в состояние B. Другой вид записи $\delta = (q_1, a, q_2)$ также обозначает переход из состояния q_1 в состояние q_2 по входному символу a;

$q_0 \in Q$ — начальное состояние управляющего устройства;

$F \in Q$ — непустое множество заключительных состояний автомата.

Пример 2.2. Пусть $Q = \{1, 2\}$, $T = \{a, b\}$, $q_0 = \{2\}$, $F = \{2\}$, $\delta = \{(1, aaa, 1), (1, ab, 2), (1, b, 2), (2, \lambda, 1)\}$. Тогда (Q, T, δ, q_0, F) — конечный автомат.

Диаграмма состояний-переходов определяет конечный автомат, построенный по регулярной грамматике, который допускает множество цепочек языка, определяемого этой грамматикой [10]. Состояния и дуги диаграммы — это графическое изображение функции переходов конечного автомата из состояния в состояние при условии, что очередной анализируемый символ совпадает с символом-меткой дуги. Среди всех состояний выделяется начальное (считается, что в начальный момент работы автомат находится в этом состоянии) и конечное (если работа завершается переходом в это состояние, то анализируемая цепочка автоматом допускается).

Определение 2.2. Конечный автомат *допускает цепочку* $a_1a_2...a_n$, если $\delta(q_0,a_1) = A_1$; $\delta(A_1, a_2) = A_2$; ...; $\delta(A_{n-2}, a_{n-1}) = A_{n-1}$; $\delta(A_{n-1}, a_n) = S$, где $a_i \to T$, $A_j \to Q$, $j = 1, 2, ..., n-1$; $i = 1, 2, ..., n$; q_0 — начальное состояние, $S \in F$ — одно из заключительных состояний.

Определение 2.3. Множество цепочек, допускаемых конечным автоматом, составляет определяемый им *язык*.

2.1.3. Алгоритм разбора по диаграмме состояний конечного автомата

Одной из задач лексического анализа является разбор цепочек, т. е. определение того, принадлежит ли входная цепочка данному языку $L(G)$. Разбор осуществляется следующим образом [2].

1. Объявляем текущим начальное состояние;

2. Далее многократно (до тех пор, пока не считаем признак конца цепочки) выполняем следующие шаги: считываем очередной символ исходной цепочки и переходим из текущего состояния в другое по дуге, помеченной этим символом. Состояние, в которое мы при этом попадаем, становится текущим.

При работе алгоритма возможны следующие ситуации.

1. Прочитана вся цепочка; на каждом шаге находилась единственная дуга, помеченная очередным символом анализируемой цепочки; в результате последнего перехода оказались в состоянии S. Это означает, что исходная цепочка принадлежит $L(G)$.

2. Прочитана вся цепочка; на каждом шаге находилась единственная «нужная» дуга; в результате последнего шага автомат перешел в состояние, отличное от S. Это означает, что исходная цепочка не принадлежит $L(G)$.

3. На некотором шаге не нашлось дуги, выходящей из текущего состояния и помеченной очередным анализируемым символом. Это означает, что исходная цепочка не принадлежит $L(G)$.

4. На некотором шаге работы алгоритма оказалось, что есть несколько дуг, выходящих из текущего состояния, помеченных очередным анализируемым символом, но ведущих в разные состояния. Это говорит о *недетерминированности* разбора, что будет подробно описано в подразд. 2.3.1.

2.2. Регулярные грамматики и языки

В дальнейшем, говоря о регулярных грамматиках, будем иметь в виду леволинейные грамматики (см. разд. 1.3.1), которые используются в методах восходящего разбора. Это обусловлено тем, что все языки программирования определяют нотацию записи «слева направо». В той же нотации работают и компиляторы [2—4].

На основе леволинейных грамматик строятся КА, которые служат для лексического анализа.

Напомним, что в леволинейной грамматике $G = (T, N, P, S)$ все правила имеют вид $A \to Ba$ либо $A \to a$, где $A \in N$, $B \in N$, $a \in T$.

2.2.1. Алгоритм разбора цепочки для леволинейной грамматики

Для удобства предположим, что анализируемая цепочка заканчивается специальным символом $\perp$ — признаком конца цепочки.

Алгоритм разбора будет следующим:

1. Первый символ исходной цепочки $a_1a_2...a_n\perp$ заменяем нетерминалом A, для которого в грамматике есть правило вывода $A \to a_1$ (другими словами, производим свертку терминала a_1 к нетерминалу A).

2. Затем многократно (пока не считан признак конца цепочки) выполняем следующие шаги: полученный на предыдущем шаге нетерминал A и расположенный непосредственно справа от него очередной терминал a_i исходной цепочки заменяем нетерминалом B (производим свертку), для которого в грамматике действует правило вывода $B \to Aa_i$ ($i = 2, 3, ..., n$).

Таким образом дерево разбора строится снизу вверх, от листьев к корню, пока не достигнем целевого нетерминала грамматики S.

При работе этого алгоритма возможны следующие ситуации.

1. Прочитана вся цепочка; на каждом шаге находилась единственная нужная свертка; на последнем шаге свертка произошла к символу S. Это означает, что исходная цепочка $a_1a_2...a_n\perp$ принадлежит $L(G)$.

2. Прочитана вся цепочка; на каждом шаге находилась единственная нужная свертка; на последнем шаге свертка произошла к символу, отличному от S. Это означает, что исходная цепочка $a_1a_2...a_n\perp$ не принадлежит $L(G)$.

3. На некотором шаге не нашлось нужной свертки, т. е. для полученного на предыдущем шаге нетерминала A и расположенного непосредственно справа от него очередного терминала a_i исходной цепочки не нашлось нетерминала B, для которого в грамматике действовало бы правило вывода $B \to Aa_i$. Это означает, что исходная цепочка $a_1a_2...a_n\perp$ не принадлежит $L(G)$.

4. На некотором шаге работы алгоритма оказалось, что имеется более одной подходящей свертки, т. е. в грамматике разные нетерминалы имеют правила вывода с одинаковыми правыми частями, и поэтому непонятно, к какому из них производить свертку. Это говорит о недетерминированности разбора. Анализ ситуации рассмотрен в подразд. 2.3.1.

Для детерминированного разбора все возможные свертки удобно представить в виде таблицы, строки которой помечены нетерминальными символами, а столбцы — терминальными [4]. Значение в ячейке таблицы представляет собой нетерминальный символ, к которому сворачивается пара нетерминал—терминал в соответствующих строке и столбце. Если такой свертки не существует, в таблице ставится «–».

Пример 2.3. Дана грамматика $G = (\{a, b, \perp\}, \{S, A, B, C\}, P, S)$ с правилами:

P: $S \to C\perp$;

$C \to Ab \mid Ba$;

$A \to a \mid Ca$;

$B \to b \mid Cb$.

Этой грамматике будет соответствовать следующая табл. 2.1.

Таблица 2.1. **Таблица детерминированного разбора**

Нетерминал	Терминал		
	a	b	$\perp$
S	—	—	—
A	—	C	—
B	C	—	—
C	A	B	S
	A	B	—

2.2.2. Построение конечного автомата на основе регулярной грамматики

Для того чтобы на основе леволинейной грамматики $G = (T, N, P, S)$ построить конечный автомат $A = (Q, T, \delta, q_0, F)$, необходимо выполнить алгоритм.

1. Определить множество состояний автомата Q. При этом каждому состоянию КА будет соответствовать один нетерминальный символ из множества N грамматики G. Кроме того, к множеству состояний автомата добавляется еще одно дополнительное состояние — H, т. е $Q = N \cup \{H\}$.

2. Входным алфавитом автомата A является множество терминальных символов T грамматики G.

3. Сформировать функцию переходов КА. Для этого просмотреть все множество правил P исходной грамматики:

- если встречается правило вида $A \to a$, где $A \in N$, $a \in T$, то добавить функцию переходов $\delta(H, a) = A$;
- если встречается правило вида $A \to Bb$, где $A, B \in N$, $b \in T$, то добавить функцию переходов $\delta(B, b) = A$.

4. Начальным состоянием автомата A является состояние H: $q_0 = H$.

5. Множество конечных состояний автомата A представляет собой одно состояние, соответствующее целевому символу грамматики G: $F = \{S\}$.

Процесс создания диаграммы состояний-переходов КА.

1. Строим вершины графа, соответствующие нетерминалам грамматики, и добавляем еще одну вершину, помеченную символом H. Вершины графа являются состояниями конечного автомата, а символ H — начальное состояние.

2. Соединяем эти состояния дугами по следующим правилам:

- для каждого правила грамматики вида $N \to t$ соединяем направленной дугой состояния H и N и помечаем дугу символом t;
- для каждого правила $N \to Mt$ соединяем дугой состояния N и M, стрелка направлена от M к N; помечаем дугу символом t.

Пример 2.4. Для грамматики из примера 2.2 диаграмма состояний-переходов конечного автомата представлена на рис. 2.3.

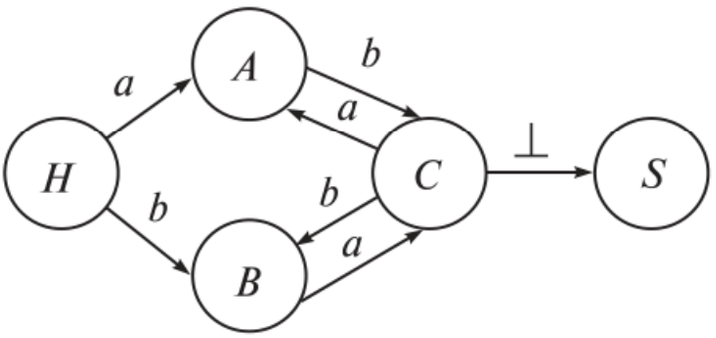


Рис. 2.3. Пример диаграммы состояний-переходов конечного автомата

2.2.3. Построение леволинейной грамматики на основе конечного автомата

Для того чтобы построить на основе конечного автомата $A = (Q, T, \delta, q_0, F)$ регулярную грамматику $G = (T, N, P, S)$, необходимо выполнить алгоритм [2, 3].

1. Множество терминальных символов грамматики G строится из алфавита входных символов T автомата A.

2. Множество нетерминальных символов грамматики G строится из множества состояний автомата A, исключая начальное состояние q_0: $N = Q \setminus \{q_0\}$. Причем, каждому нетерминалу соответствует одно состояние автомата.

3. Для построения правил грамматики просматривается функция переходов автомата A для всех возможных состояний из множества Q и всех возможных входных символов из множества T. При этом для каждого состояния A и входного символа t справедливо:

- если $\delta(A, t) = \varnothing$, то ничего не выполняем;
- если $\delta(A, t) = \{B_1, B_2, \ldots, B_n\}$, $n > 0$, $A, B_i \in Q$, $i = 1, 2, \ldots, n$, $t \in T$, тогда для всех состояний B_i выполняем следующее:
  - — во множество P правил грамматики G добавляем правило $B_i \to t$, если $A = q_0$;
  - — во множество P правил грамматики G добавляем правило $B_i \to At$, если $A \neq q_0$.

4. Если множество конечных состояний F автомата A содержит только одно состояние $F = \{F_0\}$, то целевым символом S грамматики G становится символ множества N, соответствующий этому состоянию: $S = F_0$; иначе, если множество конечных состояний F автомата A содержит более одного состояния $F = \{F_1, F_2, \ldots, F_n\}$, $n > 1$, тогда во множество нетерминальных символов N грамматики G добавляется новый нетерминальный сим-

вол S: $N = N \cup \{S\}$, а во множество правил P грамматики G добавляются правила: $S \to F_1 \mid F_2 \mid \ldots \mid F_n$.

На этом построение грамматики заканчивается.

2.3. Преобразование конечных автоматов

Для лексического анализа можно использовать только детерминированные конечные автоматы, поэтому нас интересует возможность преобразования недетерминированного конечного автомата (НКА) в детерминированный (ДКА) [3, 7, 9].

Один из наиболее важных результатов теории конечных автоматов состоит в том, что классы языков, определяемых недетерминированными и детерминированными конечными автоматами, совпадают. Это означает, что для любого НКА всегда можно построить ДКА, определяющий тот же язык.

2.3.1. О недетерминированном разборе

Некоторые регулярные грамматики содержат правила с одинаковыми правыми частями для разных нетерминалов в левой части. В этом случае непонятно, к какому нетерминалу делать свертку, т. е. в диаграмме состояний-переходов из одного состояния выходит несколько дуг, ведущих в разные состояния, но помеченных одним и тем же входным символом.

Функция перехода такого КА $\delta(A, t) = \{B_1, B_2, \ldots, B_n\}$ означает, что из состояния A по входному символу t можно осуществить переход в любое из состояний B_i, $i = 1, 2, \ldots, n$.

Пример 2.5. Дана грамматика $G = (\{0, 1\}, \{U, V, Q, Z, S\}, P, S)$ с правилами:

P: $S \to Z\perp$;

$Z \to U1 \mid V0 \mid Z0 \mid Z1$;

$U \to Q1 \mid 1$;

$V \to Q0 \mid 0$;

$Q \to Q0 \mid Q1 \mid 0 \mid 1$,

которой соответствует КА — $A = (\{H, U, V, Q, Z, S\}, \{0, 1\}, \delta, H, \{S\})$, диаграмма состояний которого представлена на рис. 2.4.

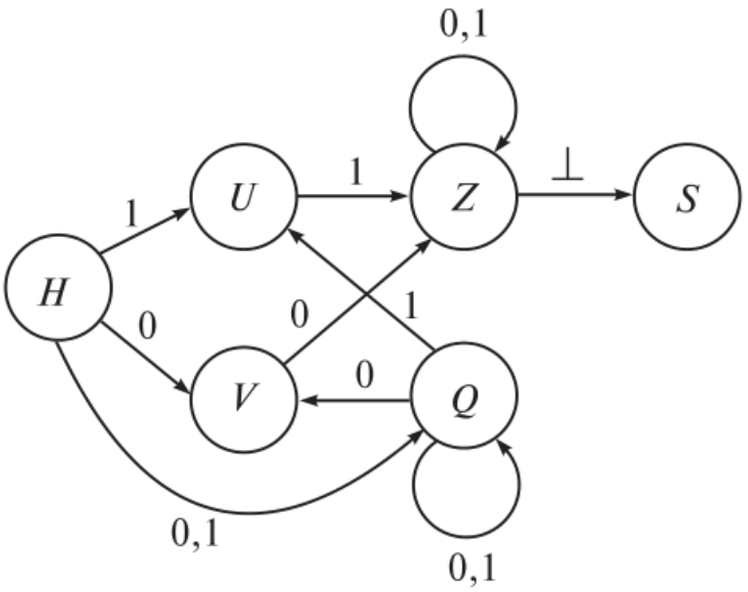


Рис. 2.4. Диаграмма состояний-переходов недетерминированного конечного автомата

В данном случае функции переходов $\delta(H, 0) = \{Q, V\}$, $\delta(H, 1) = \{Q, U\}$, $\delta(Q, 0) = \{Q, V\}$, $\delta(Q, 1) = \{Q, U\}$ предполагают переход из одного состояния по одному входному символу в два разных состояния. Реализовать разбор цепочки с помощью такого КА практически невозможно, поскольку придется перебирать множество различных путей для нахождения правильного, поэтому встает задача преобразовать НКА в ДКА.

2.3.2. Преобразование недетерминированного конечного автомата в детерминированный

Процесс преобразования недетерминированного конечного автомата $A = (Q, T, \delta, H, S)$ в детерминированный конечный автомат $A = (Q', T, \delta', H', S')$ с тем же самым языком заключается в следующем [2, 5].

1. Формируем множество состояний Q', которое содержит все подмножества множества Q. Каждое состояние из Q' будем обозначать $[A_1, A_2, ..., A_n]$, где $A_i \in Q$.

2. Функцию δ' определим как $\delta'([A_1, A_2, ..., A_n], t) = [B_1, B_2, ..., B_m]$, где для каждого $1 \le j \le m$ $\delta(A_i, t) = B_j$ для каких-либо $1 \le i \le n$.

3. Если $H = \{H_1, H_2, ..., H_k\}$, тогда $H' = [H_1, H_2, ..., H_k]$.

4. Если $S = \{S_1, S_2,..., S_p\}$, тогда S' — все состояния из Q', имеющие вид $[...S_i...]$, $S_i \in S$ для какого-либо $1 \le i \le p$.

Состояния из Q', не достижимые из начального состояния, можно исключить.

Пример 2.6. Пусть задан НКА: $A = (\{H, A, B, S\}, \{0, 1\}, \delta, \{H\}, \{S\})$, где

$\delta(H, 1) = B$; $\delta(B, 0) = A$;

$\delta(A, 1) = B$; $\delta(A, 1) = S$,

тогда соответствующий детерминированный конечный автомат имеет вид :

$Q' = \{[H], [A], [B], [S], [HA], [HB], [HS], [AB], [AS], [BS], [HAB], [HAS], [ABS], [HBS], [HABS]\}$

$\delta'([A], 1) = [BS]$; $\delta'([H], 1) = [B]$;

$\delta'([B], 0) = [A]$; $\delta'([HA], 1) = [BS]$;

$\delta'([HB], 1) = [B]$; $\delta'([HB], 0) = [A]$;

$\delta'([HS], 1) = [B]$; $\delta'([AB], 1) = [BS]$;

$\delta'([AB], 0) = [A]$; $\delta'([AS], 1) = [BS]$;

$\delta'([BS], 0) = [A]$; $\delta'([HAB], 0) = [A]$;

$\delta'([HAB], 1) = [BS]$; $\delta'([HAS], 1) = [BS]$;

$\delta'([ABS], 1) = [BS]$; $\delta'([ABS], 0) = [A]$;

$\delta'([HBS], 1) = [B]$; $\delta'([HBS], 0) = [A]$;

$\delta'([HABS], 1) = [BS]$; $\delta'([HABS], 0) = [A]$;

$S' = \{[S], [HS], [AS], [BS], [HAS], [ABS], [HBS], [HABS]\}$.

Достижимыми состояниями в получившемся КА являются $[H]$, $[B]$, $[A]$ и $[BS]$, поэтому остальные состояния удаляются.

Таким образом, $A' = (\{[H], [B], [A], [BS]\}, \{0, 1\}, \delta', H, \{[BS]\})$, где (рис. 2.5)

$\delta'([A], 1) = [BS]$; $\delta'([H], 1) = [B]$;

$\delta'([B], 0) = [A]$; $\delta'([BS], 0) = [A]$.

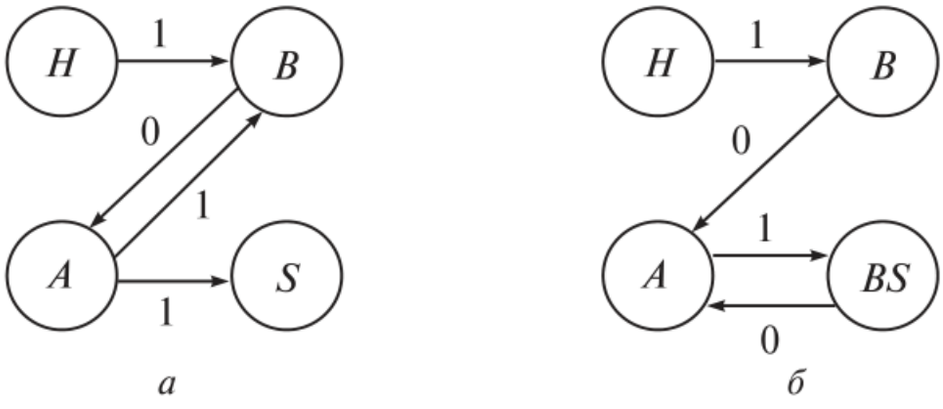


Рис. 2.5. Диаграммы состояний-переходов:
а — НКА; *б* — ДКА

2.3.3. Построение детерминированного конечного автомата с минимальным числом состояний

Одной из задач преобразования КА является удаление из него избыточных состояний или минимизация [3, 9], которая производится с помощью нижеследующего алгоритма минимизации.

1. Исключим из автомата все состояния, не достижимые из начального.

2. Разобьем все состояния КА на классы эквивалентности следующим способом:

- 0 — эквивалентное разбиение: в первый класс отнесем все конечные состояния, а во второй — все остальные;
- 1 — эквивалентное разбиение: выделим состояния, из которых по одинаковым символам происходит переход в 0 — эквивалентные состояния;
- последовательно строим n+1-эквивалентные разбиения по n-эквивалентным, при этом увеличивая число классов эквивалентности до тех пор, пока n+1-эквивалентное разбиение не совпадет с n-эквивалентным;
- полученные классы эквивалентности представляют собой состояния минимизированного КА, эквивалентного исходному.

Пример 2.7. Рассмотрим конечный автомат, представленный на рис. 2.6, *а*. Состояния *F* и *G* недостижимы из начального состояния *A*, исключим их из КА. Строим классы эквивалентности (табл. 2.2).

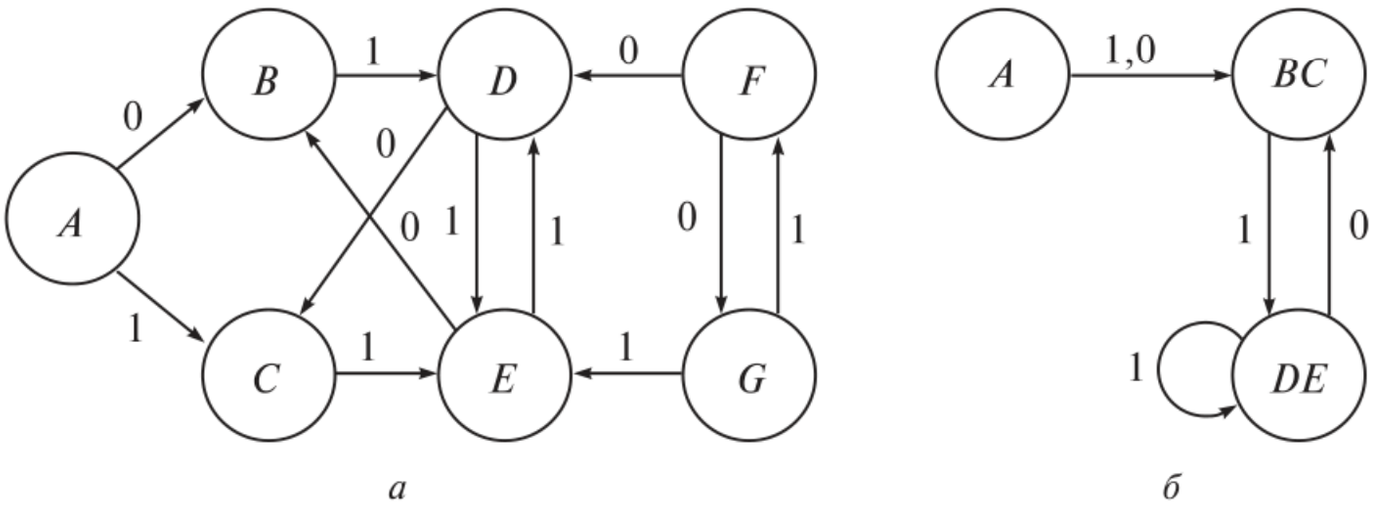


Рис. 2.6. Диаграммы состояний-переходов конечных автоматов: *а* — избыточное; *б* — минимизированное

Таблица 2.2. **Таблица эквивалентных разбиений**

Эквивалентное разбиение	Класс эквивалентности
0	(ABC) (DE)
1	(A) (BC) (DE)
2	(A) (BC) (DE)

Как видно из таблицы, 2-й класс совпал с 1-м. В результате получится КА, представленный на рис. 2.6, *б*.

2.4. Построение лексических анализаторов

Компилятор — это программа, которая считывает текст программы на одном языке — *исходном*, и транслирует (переводит) его в эквивалентный текст на другом языке — *целевом*.

Компиляция состоит из двух частей: анализа и синтеза. *Анализ* — это разбиение исходной программы на составные части и создание ее промежуточного представления [11, 12]. *Синтез* — конструирование требуемой целевой программы из промежуточного представления. При компиляции анализ состоит из трех фаз.

1. *Линейный анализ*, при котором поток символов исходной программы считывается слева направо и группируется в *токены* (*token*), представляющие собой последовательности символов с определенным совокупным значением.

2. *Иерархический анализ*, при котором символы или токены иерархически группируются во вложенные конструкции с совокупным значением.

3. *Семантический анализ*, позволяющий проверить, насколько корректно совместное размещение компонентов программы.

Остановимся подробнее на первой фазе анализа — линейном анализе. В компиляторах линейный анализ называется *лексическим*, или *сканированием*. Например, при лексическом анализе символы в инструкции присвоения

```
pos = init + rate * 66.6
```

будут сгруппированы в следующие токены:

- идентификатор `pos`;
- символ присвоения =;

- идентификатор `init`;
- знак сложения `+`;
- идентификатор `rate`;
- знак умножения `*`;
- число `66.6`.

Пробелы, разделяющие символы этих токенов, при лексическом анализе обычно отбрасываются.

2.4.1. Роль лексических анализаторов

Основная задача лексического анализатора состоит в чтении новых символов и выдаче последовательности токенов, используемых синтаксическим анализатором [3, 12]. На рис. 2.7 схематично показано взаимодействие лексического и синтаксического анализаторов, которое обычно реализуется путем создания лексического анализатора в качестве подпрограммы синтаксического анализатора (или подпрограммы, вызываемой им). При получении запроса на очередной токен лексический анализатор считывает входной поток символов до точной идентификации следующего токена.

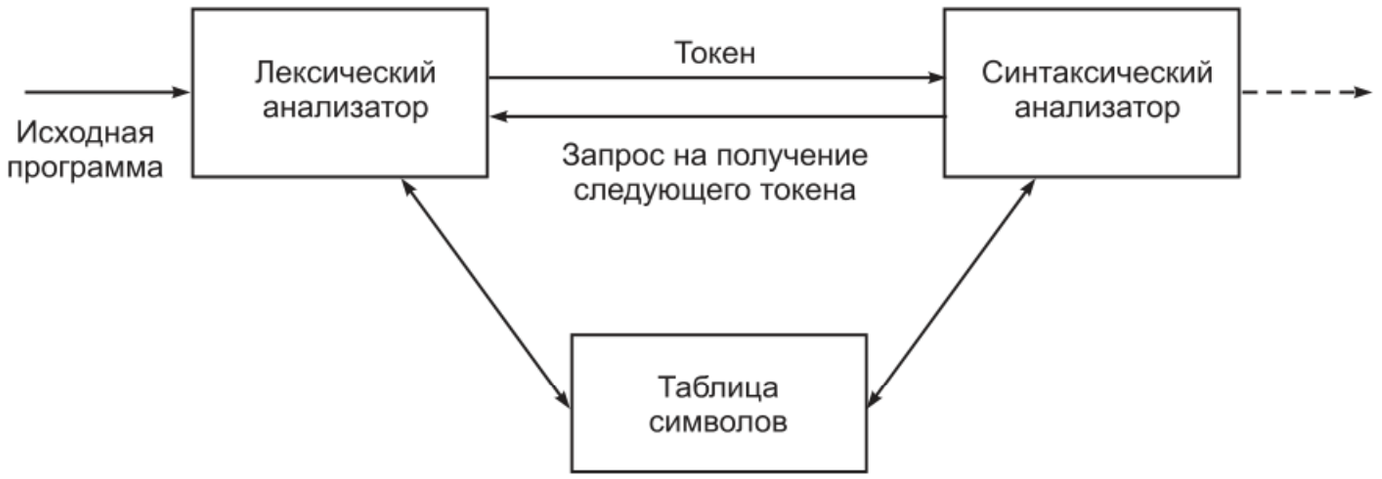


Рис. 2.7. Схема взаимодействия лексического и синтаксического анализаторов

Поскольку лексический анализатор является частью, которая считывает исходный текст, он может также выполнять некоторые вторичные задачи, например, удаление из текста исходной программы комментариев и не несущих смысловой нагрузки пробелов (а также символов табуляции и новой строки). Еще одна задача состоит в согласовании сообщений об ошибках ком-

пиляции и текста исходной программы. Например, лексический анализатор может подсчитать количество считанных строк и указать строку, вызвавшую ошибку.

2.4.2. Задачи лексического анализа

Имеется ряд причин, по которым фаза анализа компиляции разделяется на лексический и синтаксический анализы.

Во-первых, наиболее важной причиной является упрощение разработки. Отделение лексического анализатора от синтаксического часто позволяет упростить одну из фаз анализа.

Во-вторых, увеличивается эффективность компилятора. Отдельный лексический анализатор позволяет создать специализированный и потенциально более эффективный процессор для решения поставленной задачи. Поскольку на чтение исходной программы и разбор ее на токены тратится много времени, специализированные технологии буферизации и обработки токенов могут существенно повысить производительность компилятора.

В-третьих, увеличивается переносимость компилятора. Особенности входного алфавита и другие специфичные характеристики используемых устройств могут ограничивать возможности лексического анализатора.

2.4.3. Токены, шаблоны, лексемы

В лексическом анализе употребляются термины «токен», «шаблон» и «лексема», примеры использования которых приведены в табл. 2.3. При этом во входном потоке могут встречаться строки, для которых в качестве выхода получается один и тот же токен. Означенные строки описываются правилом, именуемым *шаблоном* (*pattern*), связанным с токеном. О шаблоне говорят, что он *соответствует* (*match*) каждой строке в наборе. Лексема, представляя собой последовательность символов исходной программы, в свою очередь, соответствует шаблону токена [12].

В большинстве языков программирования в качестве токенов выступают ключевые слова, операторы, идентификаторы, константы, строки литералов и символы пунктуации, например скобки, запятые и точки с запятыми. Возврат токена чаще всего реализуется путем передачи целого числа, соответствующего токену.

Таблица 2.3. **Терминология лексического анализа**

Токен	Пример лексем	Неформальное описание шаблона
`const`	`const`	`Const`
`if`	`if`	`If`
`relation`	`<, <=, ==, !=, >, >=`	< или <= или == или != или > или >=
`identifier`	`count, i, D2, pi`	Буква, за которой следуют буквы и цифры
`number`	`3.1415, 0, 6e-4`	Любая числовая константа
`literal`	`«core dumped»`	Любые символы между двумя парными кавычками, исключая сами кавычки

Шаблон — правило, описывающее набор лексем, которые могут представлять определенный токен в исходной программе. Так, шаблон токена `const` представляет собой просто строку const, являющуюся ключевым словом. Шаблон токена `relation` — набор всех шести операторов сравнения в языке C.

Во многих языках ряд строк является *зарезервированным*, т. е. их смысл предопределен и не может быть изменен пользователем. Если ключевые слова не зарезервированы, лексический анализатор должен уметь определить, с чем он имеет дело — ключевым словом или идентификатором, введенным пользователем.

2.4.4. Атрибуты токенов

Если шаблону соответствует несколько лексем, лексический анализатор должен обеспечить дополнительную информацию о лексемах для последующих фаз компиляции. Например, шаблон `number` соответствует как строке `0`, так и строке `77.7`, и при генерации кода крайне важно знать, какая именно строка соответствует токену.

Лексический анализатор хранит информацию о токенах в связанных с ними атрибутах. Токены определяют работу синтаксического анализатора; атрибуты — трансляцию токенов. На практике токены обычно имеют единственный атрибут — указатель на запись в таблице символов, в которой хранится информация о токене. Для диагностических целей могут понадобиться как лексемы идентификаторов, так и номера строк, в которых они впервые встретились в программе. Вся эта и другая информация может храниться в записях в таблице симво-

лов. Следует отметить, что для некоторых токенов (например, для токена ***присваивание***) нет необходимости хранить какие-либо атрибуты.

2.4.5. Лексические ошибки

На уровне лексического анализатора определяются только некоторые ошибки, поскольку лексический анализатор рассматривает исходный текст программы в ограниченном контексте. Если в программе на языке C строка `fi` впервые встретится в контексте `fi (a == f(x))`, лексический анализатор не сможет определить, что именно представляет собой `fi` — неверно записанное слово `if` или необъявленный идентификатор функции. Поскольку буквосочетание `fi` является корректным идентификатором, лексический анализатор должен вернуть токен для идентификатора и предоставить обработку ошибки другой части компилятора.

Представим теперь, что лексический анализатор не способен продолжать работу, поскольку ни один из шаблонов не соответствует оставшейся части входного потока. Простейшим в этой ситуации будет восстановление в «режиме паники». Подробное описание см. в подразд. 3.7.3.

При восстановлении корректного входного потока могут выполняться различные преобразования. Самая простая стратегия состоит в проверке, не может ли начало оставшейся части входного потока быть заменено корректной лексемой путем единственного преобразования.

2.4.6. Определение токенов

В определении токенов важную роль играют регулярные выражения. Каждый шаблон соответствует множеству строк, так что регулярные выражения можно рассматривать как имена таких множеств.

В языке Pascal идентификатор представляет собой букву, за которой следует нуль или несколько букв или цифр. С помощью регулярных выражений можно определить идентификатор, например, так:

```
letter (letter | digit)*
```

Вертикальная черта означает союз «или», скобки используются для группировки подвыражений, а непосредственное соседство символа `letter` с оставшейся частью выражения означает конкатенацию.

Регулярное выражение строится из более простых регулярых выражений с использованием набора правил. Предположим, что `r` и `s` — регулярные выражения. Тогда:

- `(r) | (s)` означает либо `r`, либо `s`;
- `(r) (s)` означает конкатенацию;
- `(r)*` — `0` или несколько конкатенаций выражения `r`;
- `(r)` эквивалентна `r`.

Пример 2.8. Пусть дан алфавит символов {*a*, *b*}. Тогда:

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

2. Регулярное выражение (*a* | *b*) (*a* | *b*) обозначает множество {*aa*, *ab*, *ba*, *bb*}. Другое регулярное выражение для того же множества — *aa* | *ab* | *ba* | *bb*.

3. Регулярное выражение *a** — множество всех строк из 0 и более *a*.

4. Регулярное выражение (*a* | *b*)* обозначает множество всех строк, содержащих 0 или несколько экземпляров *a* и *b*, т. е. множество всех строк из *a* и *b*. Другое регулярное выражение для этого множества — (*a***b**)*.

5. Регулярное выражение *a* | *a***b* — множество, содержащее строку a и все строки, состоящие из 0 или нескольких *a*, за которыми следует *b*.

Для удобства записи регулярным выражениям можно давать имена и определять регулярные выражения с использованием этих имен так, как если бы это были символы.

Некоторые конструкции встречаются в регулярных выражениях настолько часто, что для них введены специальные сокращения:

1. Для обозначения одного или нескольких экземпляров используется унарный постфиксный оператор +. Например, `(r)+`.

2. Для обозначения нуля или одного экземпляра используется унарный постфиксный оператор `?`. Например, `(r)?`.

3. Запись [*abc*], где *a*, *b* и *c* — символы алфавита, означает регулярное выражение *a* | *b* | *c*. Сокращенный класс символов типа [*a*-*z*] означает регулярное выражение *a* | *b* | ... | *z*. С использованием классов символов можно описать идентификаторы как строки, заданные регулярным выражением [*A*-*Za*-*z*][*A*-*Za*-*z*0-9]*.

2.4.7. Распознавание токенов

В качестве примера вышесказанного рассмотрим язык, порождаемый следующей грамматикой:

```
stmt → if expr then stmt | if expr then stmt else stmt | EPS
expr → term relop term | term
term → id | num,
```

где нетерминалы `if`, `then`, `else`, `relop`, `id` и `num` порождают множества строк, задаваемых следующими регулярными определениями:

```
if → if
then → then
else → else
relop → < | <= | = | <> | > | >=
id → letter (letter | digit)*
num → digit+ (.digit+)? (E(+ | -)? digit+)?
letter → [A-Za-z]
digit → [0-9]
```

Для этого фрагмента языка лексический анализатор будет распознавать ключевые слова `if`, `then`, `else`, а также лексемы `relop`, `id` и `num`. Для простоты будем считать, что ключевые слова зарезервированы и, таким образом, не могут использоваться в качестве идентификаторов. Предположим также, что лексемы разделены «пустым пространством», состоящим из непустых последовательностей пробелов, символов табуляций и новой строки. Лексический анализатор будет отбрасывать эти символы путем сравнения строки с регулярным определением `ws`, имеющим нижеследующее регулярное определение:

```
delim → space | tab | newline
ws → delim+
```

Если находится соответствие нетерминалу `ws`, лексический анализатор не возвращает токен синтаксическому анализатору. Вместо этого происходит поиск нового токена, следующего за пробелами, и передача его синтаксическому анализатору.

Цель — построить лексический анализатор, который будет выделять из входного буфера лексему для следующего токена и,

используя табл. 2.4, выдавать пары, состоящие из токена и атрибута-значения. Атрибут-значение для операторов отношения определяется символическими константами `LT`, `LE`, `EQ`, `NE`, `GT`, `GE`.

Таблица 2.4. **Пары атрибут—значение**

Регулярное выражение	Токен	Атрибут—значение
`ws`	—	—
`if`	`if`	—
`then`	`then`	—
`else`	`else`	—
`id`	`id`	Указатель на запись в таблице
`num`	`num`	Указатель на запись в таблице
`<`	`relop`	`LT`
`<=`	`relop`	`LE`
`=`	`relop`	`EQ`
`<>`	`relop`	`NE`
`>`	`relop`	`GT`
`>=`	`relop`	`GE`

Поскольку ключевые слова есть не что иное как последовательности букв, они представляют собой исключение из правила, гласящего, что последовательность букв и цифр, начинающаяся с буквы, является идентификатором. Но вместо кодирования исключений можно рассматривать ключевые слова как специальные идентификаторы. Тогда после выделения токена из потока символов выполняется некоторый код, который определяет, чем является лексема — ключевым словом или идентификатором. Таким образом, при добавлении нового ключевого слова оно просто заносится в таблицу ключевых слов, но код при этом не меняется.

Построение распознавателя беззнаковых чисел, задаваемых регулярным определением

```
num → digit+ (.digit+)? (E(+ | -)? digit+)?
```

имеет ряд особенностей.

Данное определение имеет вид `digits fraction? exponent?`, где `fraction` и `exponent` — необязательные части.

Лексема для данного токена должна быть максимально возможной длины. Например, лексический анализатор не должен останавливаться после того, как встретит сторку (подстроку) `12` или даже строку `12.3`, если введено число `12.3E4`. Вообще говоря, правило поиска лексемы наибольшей длины — общепринятое правило работы лексического анализатора, позволяющее избежать множества неприятностей.

Контрольные вопросы

1. Для чего предназначен лексический анализатор в компиляторах?
2. Охарактеризуйте лексический анализатор как конечный автомат.
3. Дайте определение регулярных грамматик.
4. Как построить конечный автомат на основе леволинейной грамматики?
5. Что такое диаграмма состояний конечного автомата?
6. Как построить регулярную грамматику на основе конечного автомата?
7. Приведите пример разбора цепочки символов по диаграмме состояний конечного автомата.
8. Чем отличается недетерминированный автомат от детерминированного?
9. Как преобразовать недетерминированный конечный автомат к детерминированному?
10. Приведите пример преобразования недетерминированного автомата.
11. Что такое минимизация конечного автомата? Приведите пример.

Глава 3
СИНТАКСИЧЕСКИЙ АНАЛИЗ

Синтаксический анализ — более сложная по сравнению с лексическим анализом фаза трансляции, которая служит для построения внутреннего представления программы (*промежуточного кода*) [1].

Синтаксический анализатор получает строку токенов от лексического анализатора (см. рис. 2.7) и проверяет, может ли эта строка порождаться грамматикой исходного языка. Он также сообщает обо всех выявленных ошибках, причем достаточно внятно и полно. Кроме того, он должен уметь обрабатывать обычные, часто встречающиеся ошибки и продолжать работу с оставшейся частью программы [12].

Имеются три основных типа синтаксических анализаторов грамматик:

- нисходящие;
- восходящие;
- универсальные.

Универсальные методы разбора, такие как алгоритмы Кока — Янгера — Касами или Эрли, могут работать с любой грамматикой. Однако эти методы слишком неэффективны для использования в промышленных компиляторах. Методы, обычно применяемые в компиляторах, классифицируются как нисходящие (сверху вниз — top-down) или восходящие (снизу вверх — bottom-up). Нисходящие синтаксические анализаторы строят дерево разбора сверху (от корня) вниз (к листьям), тогда как восходящие начинают с листьев и идут к корню. В обоих случаях входной поток синтаксического анализатора сканируется посимвольно слева направо.

Синтаксические анализаторы всегда строятся на основе контекстно-свободных грамматик.

Наиболее эффективные нисходящие и восходящие методы синтаксического анализа работают только с подклассами грамматик, однако некоторые из этих подклассов, такие как *LL*-

и LR-грамматики, достаточно выразительны для описания большинства синтаксических конструкций языков программирования (более подробно в подразд. 3.3). Реализованные вручную синтаксические анализаторы чаще работают с LL-грамматиками.

Для описания синтаксиса языка программирования используют однозначные (см. подразд. 1.4.3) *приведенные* КС-грамматики.

3.1. Приведенные грамматики

Приведенные грамматики — это КС-грамматики, которые не содержат недостижимых и бесплодных символов, циклов и λ-правил («пустых» правил). Приведенные грамматики называют также КС-грамматиками в каноническом виде [2, 3, 12].

Для того чтобы преобразовать произвольную КС-грамматику к приведенному виду, необходимо удалить:

- все бесплодные символы;
- все недостижимые символы;
- λ-правила;
- цепные правила.

Следует подчеркнуть, что шаги преобразования должны выполняться строго в указанном порядке.

3.1.1. Удаление бесплодных символов

Бесплодные нетерминальные символы не участвуют в порождении цепочек языка, поэтому могут быть удалены из грамматики [9, 13].

Определение 3.1. Символ $A \in$ называется *бесплодным* в грамматике $G = (T, N, P, S)$, если множество $\{\alpha \mid \alpha \in T^*, A \Rightarrow^* \alpha\} = \varnothing$.

Рассмотрим метод удаления бесплодных символов.

Рекурсивно строим множества N_0, N_1, ... , N_i, ... следующим образом:

1. $N_0 = \varnothing$, $i = 1$.
2. $N_i = \{A \mid (A \to \alpha) \in P \text{ и } \alpha \in (N_{i-1} \cup T)^*\} \cup N_{i-1}$.

3. Если $N_i \neq N_{i-1}$, то присвоим $i = i+1$ и переходим к п. 2, иначе $N' = N_i$; P' состоит из правил множества P, содержащих только символы из $N' \cup T$; $G' = (T, N', P', S)$.

3.1.2. Удаление недостижимых символов

Определение 3.2. Символ $X \in (T \cup N)$ называется *недостижимым* в грамматике $G = (T, N, P, S)$, если он не появляется ни в одной сентенциальной форме этой грамматики.

Метод удаления недостижимых символов заключается в следующем [9, 13]:

строим множества V_0, V_1, ... , V_i, ... таким образом:

1. $V_0 = \{S\}$; $i = 1$.

2. $V_i = \{X \mid X \in (T \cup N), (A \to \alpha x \beta) \in P \text{ и } A \in V_{i-1}\} \cup V_{i-1}$.

3. Если $V_i \neq V_{i-1}$, то $i = i+1$ и переходим к п. 2, иначе $N' = V_i \cap N$; $T' = V_i \cap T$; P' состоит из правил множества P, содержащих только символы из V_i; $G' = (T', N', P', S)$.

Пример 3.1. Удалить бесплодные и недостижимые символы из грамматики G с правилами: $P = \{I \to aIa \mid bAd \mid c; A \to cBd \mid aAd; C \to d\}$.

Здесь бесплодные символы — A, B;

Недостижимые символы — C.

3.1.3. Исключение цепных правил

Определение 3.3. Правило грамматики вида $A \to B$, где $A, B \in N$, называется *цепным*.

Для КС-грамматики G, содержащей цепные правила, всегда можно построить эквивалентную ей грамматику G' без цепных правил.

Правила грамматики G вида $A \to B$, $B \to C$, $C \to aX$ могут быть заменены одним правилом $A \to aX$, потому что вывод $A \Rightarrow B \Rightarrow C \Rightarrow aX$ цепочки aX в грамматике G может быть получен в грамматике G' с помощью правила $A \to aX$.

Метод исключения цепных правил заключается в следующем [3, 13].

1. Разобьем множество правил P грамматики G на два подмножества P_1 и P_2, включая в P_1 все правила вида $A \to B$.

2. Построим множество правил $R(A_i)$ на множестве P_1 по следующему принципу: если существует вывод символа A_j из A_i и в P_2 есть правило $A_j \to \alpha$, где α — цепочка словаря $(N \cup T)^*$, то правило $A_i \to \alpha$ помещается в множество $R(A_i)$.

3. Построим новое множество правил P' путем объединения правил P_2 и всех построенных множеств $R(A_i)$.

В результате получим грамматику $G' = \{T, N, P', S\}$, которая эквивалентна заданной и не содержит правил вида $A \to B$ (цепных).

Пример 3.2. Дана грамматика G:

$G = (\{+,*,(,),a\}, \{E, T, F\}, P = \{E \to E+T \mid T, T \to T*F \mid F,$

$F \to (E) \mid a\}, E)$.

Необходимо исключить правила $E \to T, T \to F$.

Разбиваем правила грамматики на два подмножества:

$P_1 = \{E \to T, T \to F\}$;

$P_2 = \{E \to E+T, T \to T*F, F \to (E) \mid a\}$.

Для каждого правила из P_1 построим соответствующее подмножество.

$R(E) = \{E \to T*F, E \to (E) \mid a\}$;

$R(T) = \{T \to (E) \mid a\}$.

В результате получаем искомое множество правил грамматики без цепных правил в виде

$P' = P_2 \cup R(E) \cup R(T) = \{E \to T+T \mid T*F \mid (E) \mid a;$

$T \to T*F \mid (E) \mid a, F \to (E) \mid a\}$.

3.1.4. Исключение аннулирующих правил

Грамматика, не содержащая *пустых*, или *аннулирующих*, правил называется неукорачивающей. Неукорачивающей является также грамматика, которая содержит только одно правило вида $S \to \lambda$, где S — начальный символ грамматики и он не встречается в правых частях остальных правил грамматики.

Определение 3.4. Правило вида $A \to \lambda$ называется пустым (аннулирующим) правилом.

Для каждой КС-грамматики G', содержащей аннулирующие правила, можно построить эквивалентную ей неукорачивающую грамматику G, такую, что $L(G') = L(G)$.

Исключение из грамматики пустых правил приведет к построению дополнительных правил и к «разрастанию» грамматики. Для того чтобы построить дополнительные правила, необходимо выполнить все возможные подстановки пустой цепочки вместо аннулирующего нетерминала во все правила грамматики [3, 13].

Если же в грамматике есть правило вида $S \to \lambda$, где S — начальный символ грамматики, и символ S входит в правые части других правил грамматики, то следует ввести новый начальный символ I и заменить правило $S \to \lambda$ двумя новыми правилами: $I \to \lambda$ и $I \to S$.

Пример 3.3. Дана грамматика $G_1 = (\{a, b\}, \{I, A, B, C\}, \{I \to ABC, A \to BB \mid \lambda, B \to CC \mid a, C \to AA \mid b\}, I)$. Построить эквивалентную ей неукорачивающую грамматику.

Заменим аннулирующий нетерминал A пустой цепочкой в тех правилах грамматики G, где он встречается в правой части. Получаем:

$I \to ABC \mid BC,\ C \to AA \mid A \mid b.$

Остальные правила добавим без изменений. Получим:

$G_1' = (\{a, b\}, \{I, A, B, C\}, \{I \to ABC \mid BC, A \to BB, B \to CC \mid a,$

$C \to AA \mid A \mid b\}, I).$

Пример 3.4. Исключить аннулирующие правила из грамматики

$G_2 = (\{a, b, c\}, \{S, D\}, \{S \to SS \mid aDb \mid \lambda, D \to Sc\}, S).$

Произведем замену:

$S \to SS \mid S \mid aDb,$

$D \to Sc \mid c.$

Введем новый нетерминал I и два правила $I \to S$ и $I \to \lambda$. Получим:

$G_2' = (\{a, b, c\}, \{S, D\}, \{I \to S \mid \lambda, S \to SS \mid S \mid aDb \mid, D \to Sc \mid c\}, S).$

Все приведенные выше преобразования грамматик могут быть использованы при построении как конечных, так и магазинных автоматов.

3.2. КС-грамматики в нормальной форме

Для построения некоторых видов распознавателей необходимо использовать КС-грамматики в нормальной форме. К самым распространенным из них относятся *нормальная форма Хомского* (или *бинарная нормальная форма*) для построения бинарного дерева разбора и *нормальная форма Грейбах* для построения нисходящего левостороннего распознавателя [11].

3.2.1. Грамматики в нормальной форме Хомского

Любая приведенная КС-грамматика может быть преобразована в нормальную форму Хомского [2, 3].

Определение 3.5. КС-грамматика $G = (T, N, P, S)$ называется грамматикой в нормальной форме Хомского, если в ее множестве правил P присутствуют только правила следующего вида:

1. $A \to BC$, где $A, B, C \in N$.
2. $A \to a$, где $A \in N$ и $a \in T$.
3. $S \to \lambda$, если $\lambda \in L(G)$, причем начальный символ грамматики S не должен встречаться в правых частях других правил.

Название *бинарная нормальная форма* происходит от того, что на каждом шаге вывода в такой грамматике один нетерминальный символ может быть заменен только на два других нетерминальных символа. Поэтому в дереве вывода грамматики в нормальной форме Хомского каждая вершина либо распадается на две другие вершины (в соответствии с первым видом правил), либо содержит один последующий лист с терминальным символом (в соответствии со вторым видом правил). Третий вид правил введен для того, чтобы к нормальной форме Хомского можно было преобразовывать грамматики КС-языков, содержащих пустые цепочки символов.

Ниже приведен алгоритм преобразования грамматики G в эквивалентную грамматику G' ($L(G) = L(G')$) в нормальной форме Хомского [2].

1. Преобразовать исходную грамматику G к приведенному виду, т. е. исключить из нее бесплодные и недостижимые символы, цепные и λ-правила.

2. Множество нетерминальных символов N' результирующей грамматики G' строится на основе множества нетерминальных символов N исходной грамматики G: $N' = N$.

3. Далее просматриваются правила из множества P исходной грамматики G и в зависимости от вида каждого правила строится множество правил P' результирующей грамматики G' и дополняется множество нетерминальных символов этой грамматики N':

- если встречается правило вида $A \to a$, где $A \in N$ и $a \in T$, то оно переносится во множество P' без изменений;
- если встречается правило вида $A \to BC$, где A, B, $C \in N$, то оно переносится во множество P' без изменений;
- если встречается правило вида $S \to \lambda$, где S — целевой символ грамматики G, то оно переносится во множество P' без изменений;
- если встречается правило вида $A \to aB$, где A, $B \in N$ и $a \in T$, то во множество правил P' включаются правила $A \to <D>B$ и $<D> \to a$ и новый символ $<D>$ добавляется во множество нетерминальных символов N' грамматики G';
- если встречается правило вида $A \to Ba$, где A ,$B \in N$ и $a \in T$, то во множество правил P' включаются правила $A \to B<D>$ и $<D> \to a$, и новый символ $<D>$ добавляется во множество нетерминальных символов N' грамматики G'.
- если встречается правило вида $A \to ab$, где $A \in N$ и a, $b \in T$, то во множество правил P' включаются правила $A \to <D><E>$, $<D> \to a$ и $<E> \to b$, новые символы $<D>$ и $<E>$ добавляются во множество нетерминальных символов N' грамматики G';
- если встречается правило вида $A \to X_1...X_k$, $k > 2$, где $A \in N$ и $X_i \in T \cup N$, $i = 1, 2, ..., k$, то во множество правил P' включается цепочка правил:

  $A \to <X_1'><X_2...X_k>$;

  $<X_2...X_k> \to <X_2'><X_3...X_k>$;

  $<X_{k-1}X_k> \to <X_{k-1}'><X_k>$,

  новые нетерминальные символы $<X_2...X_k>$, $<X_3...X_k>$, ..., $<X_{k-1}, X_k>$ включаются во множество нетерминальных символов N' грамматики G', кроме того, если $X_i \in N$, то $<X_i'> \equiv$ $\equiv X_i$ иначе (если $X_i \in T$) $<X_i'>$ — это новый нетерминальный символ, он добавляется во множество N', а во множество

правил P' грамматики G' добавляется правило $<X_i'> \to X_i$, $i = 1, 2, ..., k$.

Целевым символом результирующей грамматики G' является целевой символ исходной грамматики G.

Пример 3.5. Дана грамматика $G = (\{a, b\}, \{S, A, B\}, P, S)$, где $P = \{S \to bA \mid aB, A \to a \mid aS \mid bAA, B \to b \mid bS \mid aBB\}$. Требуется построить эквивалентную ей грамматику G' в нормальной форме Хомского.

Грамматика имеет приведенный вид (в ней нет бесполезных символов, аннулирующих и цепных правил).

Правила $A \to a$ и $B \to b$ переносятся в грамматику G' без изменений.

Правило $S \to bA$ заменяем на $S \to <C>A$, $<C> \to b$.

Правило $S \to aB$ заменяем на $S \to <D>B$, $<D> \to a$.

Правило $A \to aS$ заменяем на $A \to <E>S$, $<E> \to a$.

Правило $A \to bAA$ заменяем на $A \to <F><G>$, $<F> \to b$, $<G> \to AA$.

Правило $B \to bS$ заменяем на $B \to <H>S$, $<H> \to b$.

Правило $B \to aBB$ заменяем на $B \to <I><J>$, $<I> \to a$, $<J> \to BB$.

Получаем эквивалентную G грамматику $G' = (\{a, b\}, \{S, A, B, <C>, <D>, <E>, <F>, <H>, <I>, <J>\}, P', S)$, где $P' = \{S \to <C>A \mid <D>B, A \to a \mid <E>S \mid <F><G>, B \to b \mid <HS>S \mid <I><J>, <C> \to b, <D> \to a, <E> \to a, <F> \to b, <H> \to b, <I> \to a, <G> \to AA, <J> \to BB\}$.

Как видно из примера, в результате преобразований количество правил грамматики увеличилось, но следует помнить, что целью построения грамматики в бинарной нормальной форме является не упрощение самой грамматики, а упрощение построения распознавателя на ее основе.

3.2.2. Грамматики в нормальной форме Грейбах

КС-грамматика $G = (T, N, P, S)$ называется грамматикой в нормальной форме Грейбах, если она не является *леворекурсивной* и в ее множестве правил P присутствуют только правила следующего вида [2, 3].

1. $A \to a\alpha$, где $a \in T$ и $\alpha \in N^*$.

2. $S \to \lambda$, если $\lambda \in L(G)$, причем символ S не должен встречаться в правых частях других правил.

Определение 3.6. Правило вида $A \to \alpha A$, где $A \in N$, $\alpha \in (T \cup N)^*$, называется праворекурсивным, а правило вида $A \to A\alpha$ — леворекурсивным.

Для каждой КС-грамматики G, содержащей леворекурсивные правила, можно построить эквивалентную грамматику G', не содержащую леворекурсивных правил.

Алгоритм устранения левой рекурсии из правил грамматики G заключается в следующем:

допустим, что исходная грамматика G содержит правила

$$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n,$$

где ни одна цепочка β не начинается с правила A и α_i, $\beta_i \in (T \cup N)^*$.

Для того чтобы исключить левую рекурсию, необходимо ввести новый нетерминал $<B>$ и преобразовать правила так:

$$A \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \mid \beta_1 <B> \mid \beta_2 <B> \mid \ldots \mid \beta_n <B>,$$

$$<B> \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m \mid \alpha_1 <B> \mid \alpha_2 <B> \mid \ldots \mid \alpha_m <B>.$$

Пример 3.6. Требуется преобразовать грамматику G с правилами:

$$P: \; E \to E + T \mid T;$$
$$T \to T * F \mid F;$$
$$F \to (E) \mid a.$$

Следуя описанному способу, правила $E \to E + T \mid T$ преобразуем в правила $E \to T \mid TE'$ и $E' \to +T \mid +TE'$, а правила $T \to T * F \mid F$ преобразуем в правила $T \to F \mid FT'$ и $T' \to F \mid *FT'$.

В результате получаем грамматику G' с правилами:

$$P': \; E \to T \mid TE';$$
$$E' \to +T \mid +TE';$$
$$T \to F \mid FT';$$
$$T' \to *F \mid *FT';$$
$$F \to a \mid (E),$$

не содержащую леворекурсивных правил.

Нормальная форма Грейбах является удобной формой представления грамматик для построения нисходящих левосторонних распознавателей, в которых недопустима левая рекурсия.

3.3. Виды контекстно-свободных грамматик

Накладывая ограничения на правые части продукций контекстно-свободных грамматик, можно выделить ряд их типов, которые нашли наиболее широкое практическое применение [14].

Для того чтобы дать определение некоторых типов КС-грамматик, рассмотрим отношения, которые устанавливаются между символами грамматик.

3.3.1. Отношения между символами формальной грамматики

В дальнейшем будут использоваться следующие функции, которые определяют множества символов грамматики, на основе которых строятся некоторые виды распознавателей [1, 3, 12].

Определение 3.7. Функция FIRST(U) определяет множество терминальных символов, с которых может начинаться цепочка, выводимая из нетерминального символа U;

Определение 3.8. Функция FIRST(α) определяет множество терминальных символов, с которых может начинаться цепочка, выводимая из цепочки α;

Определение 3.9. Функция FOLLOW(U) определяет множество терминалов a, которые могут появиться непосредственно справа от нетерминала U в сентенциальных формах грамматики, т. е. существует вывод $S \Rightarrow^* \alpha U a \beta$, где $\alpha, \beta \in (N \cup T)$;

Определение 3.10. Функция LAST(U) определяет множество символов, завершающих все возможные цепочки, выводимые из заданного нетерминала.

Рассмотрим алгоритм вычисления множества FIRST для нетерминалов грамматики G:

1. Положить значение функции FIRST(U) = $\varnothing$.

2. Добавить к множеству FIRST(U) все символы, с которых начинаются цепочки, непосредственно выводимые из символа U:

$$\text{FIRST}(U) = \text{FIRST}(U) \cup \{X \mid U \to X\alpha \in P,\ X \in (T \cup N),\ \alpha \in (T \cup N)^*\}.$$

3. Добавить к множеству FIRST(U) все символы, с которых начинаются цепочки, непосредственно выводимые из нетерминалов, вошедших в FIRST(U):

FIRST(U) = FIRST(U) $\cup$ FIRST(V), $V \in$ (FIRST(U) $\cap$ N).

4. Исключить из множества FIRST(U) все нетерминальные символы:

FIRST(U) = FIRST(U) $\setminus$ N.

Пример 3.7. Рассмотрим грамматику $G = (T, N, P, S)$ с правилами:

$S \to BA$;

$A \to +BA \mid \lambda$;

$B \to DC$;

$C \to *DC \mid \lambda$;

$D \to (S) \mid a$.

Для этой грамматики множества FIRST определяются следующим образом:

FIRST(D) = {(, a};

FIRST(C) = {*, λ};

FIRST(B) = FIRST(D) = {(, a};

FIRST(A) = {+, λ};

FIRST(S) = FIRST(B) = {(, a}.

Построение множества FIRST(α) для цепочек вида $\alpha = X_1X_2... ...X_n$ выполняется следующим образом.

1. Вычислить значения функций FIRST(X_i) для $i = 1, 2, ..., n$.

2. Вычислить значение функции FIRST(α), включив в него все элементы множеств FIRST(X_i), кроме пустых цепочек:

FIRST(α) = (FIRST(X_1)\{λ}) $\cup$ (FIRST(X_2)\{λ}) $\cup$ ...

... $\cup$ (FIRST(X_n)\{λ}).

3. Включить в множество FIRST(α) пустую цепочку, если она входит во все множества FIRST(X_i):

FIRST(α) = FIRST(X_1) $\cup$ {λ}, если $\lambda \in$ FIRST(X_i), для всех i.

Множество LAST(U) строится аналогично множеству FIRST(U), только для завершающих символов цепочки.

Множество FOLLOW(U) связано с использованием в формальных грамматиках аннулирующих правил, которые заменяют нетерминал левой части на пустую цепочку, завершая тем самым цикл повторения, либо исключая необязательные элементы. В этом случае принципиально важным является ответ на вопрос, в каком окружении может находиться такой нетерминал, точнее, каков последующий за нетерминалом символ.

Приведем алгоритм вычисления множества FOLLOW для нетерминалов грамматики G:

1. Положить множество FOLLOW(U) = $\varnothing$;

2. Добавить к множеству FOLLOW(U) все символы, которые в правых частях правил грамматики встречаются непосредственно за U:

$$\text{FOLLOW}(U) = \text{FOLLOW}(U) \cup \{X \mid V \to \alpha U X \beta \in P,\ V \in N,$$

$$X \in (T \cup N),\ \alpha,\ \beta \in (T \cup N)^*\}.$$

3. Добавить к множеству FOLLOW(S) пустую цепочку, это означает, что в конце разбора за целевым символом S цепочка кончается:

$$\text{FOLLOW}(S) = \text{FOLLOW}(S) \cup \{\lambda\}.$$

4. Добавить к множеству FOLLOW(U) все символы, с которых начинаются цепочки, непосредственно выводимые из нетерминалов, вошедших в FOLLOW(U):

$$\text{FOLLOW}(U) = \text{FOLLOW}(U) \cup \text{FIRST}(V),$$

$$V \in (\text{FOLLOW}(U) \cap N).$$

5. Добавить к множеству FOLLOW(U) все символы, которые следуют за нетерминалами, вошедшими в FOLLOW(U), для которых есть аннулирующие правила:

$$\text{FOLLOW}(U) = \text{FOLLOW}(U) \cup \text{FOLLOW}(V),$$
$V \in (\text{FOLLOW}(U) \cap N)$ и существует $V \to \lambda$.

6. Добавить к множеству FOLLOW(U) все символы, которые следуют за нетерминалами, для которых есть правила с нетерминалом U в правой части:

$$\text{FOLLOW}(U) = \text{FOLLOW}(U) \cup \text{FOLLOW}(V),$$

если существует $V \to \alpha U$, $\alpha \in (T \cup N)^*$.

7. Исключить из FOLLOW(U) все нетерминальные символы:

$$\text{FOLLOW}(U) = \text{FOLLOW}(U) \setminus N.$$

Пример 3.8. Рассмотрим грамматику G из примера 3.7. Для нее функция FOLLOW определятся следующим образом:

FOLLOW(S) = {), λ};

FOLLOW(A) = FOLLOW(S) = {), λ};

FOLLOW(B) = {+, λ};

FOLLOW(C) = FOLLOW(B) = {+, λ};

FOLLOW(D) = {*, λ}.

Кроме описанных выше функций, для построения распознавателей полезно определить функцию SEL(p) для продукций грамматики G $p \in P$ [13].

Определение 3.11. функция SEL(p) определяет множество *выбирающих* символов для правил грамматики.

Если продукция грамматики имеет вид $U \to a\alpha \in P$, где $a \in T$, $\alpha \in (T \cup N)^*$, то SEL($U \to a\alpha$) = a. Если продукция имеет вид $U \to \lambda$, то SEL($U \to \lambda$) = FOLLOW(U).

Множество выбирающих символов определяет, какое из правил грамматики будет использоваться на следующем шаге разбора цепочки.

3.3.2. S-грамматики

Для S-грамматик должны выполняться следующие условия [14]:

1) правая часть каждого правила грамматики начинается с терминального символа;

2) для двух правил, имеющих одинаковые нетерминалы в левой части, правые части правил должны начинаться с различных терминалов;

3) не допускаются аннулирующие правила.

Пример 3.9. Следующая грамматика является S-грамматикой:

$G = (\{a, b\},\{S, T\}, P, S)$;

P: $S \to aT \mid bSb$;

$T \to aT \mid b$.

Данная грамматика порождает цепочки вида *aa...aaab* и *b..baaa..aaabb..b*.

Дерево нисходящего разбора цепочки *bbaabbb* изображено на рис. 3.1.

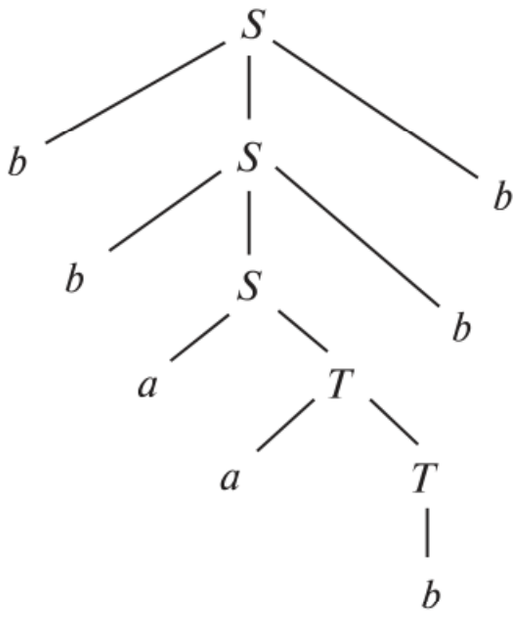


Рис. 3.1. Дерево разбора *S*-грамматики

Рассмотрим разбор цепочек, принадлежащих языку, порожденному *S*-грамматикой.

1. На каждом шаге в полученной цепочке единственный нетерминальный символ должен быть заменен на правую часть одного из правил, в котором он присутствует в левой части.

2. Для замены выбирается правило, в котором первый терминальный символ в правой части совпадает с очередным «незакрытым» символом во входной цепочке (выбирающий символ).

Для рассмотренного выше примера вывод будет следующим:

$S \Rightarrow bSb \Rightarrow bbSbb \Rightarrow bbaTbb \Rightarrow bbaaTbb \Rightarrow bbaabbb.$

3.3.3. Q-грамматики

Данная грамматика отличается от *S*-грамматики наличием дополнительных аннулирующих правил вида $A \to \lambda$ [14]. При разборе цепочек языка, порожденного *Q*-грамматикой, необходимо руководствоваться выбирающими символами правил.

Пример 3.10. Следующая грамматика является *Q*-грамматикой:

$G = (\{a, b\}, \{S, T\}, P, S)$;

P: $S \to aT \mid bSb$;

$T \to aT \mid \lambda$;

$\mathrm{SEL}(S \to aT) = a$;

$\mathrm{SEL}(S \to bSb) = b$;

$\mathrm{SEL}(T \to aT) = a$;

$\mathrm{SEL}(T \to \lambda) = \mathrm{FOLLOW}(T) = b$.

Тогда разбор цепочки *bbaabb* имеет вид:

$S \Rightarrow bSb \Rightarrow bbSbb \Rightarrow bbaTbb \Rightarrow bbaaTbb \Rightarrow bbaabb.$

3.3.4. LL-грамматики

Рассмотрим дерево вывода в процессе получения левого вывода цепочки α (рис. 3.2). Промежуточная цепочка в процессе вывода состоит из цепочки из терминалов w, самого левого нетерминала A, неразобранной части входной цепочки x.

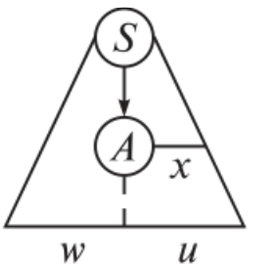


Рис. 3.2. Схема построения дерева вывода для *LL*-грамматики

Для продолжения разбора необходимо заменить нетерминал A по одному из правил вида $A \to y$, причем для детерминированного разбора правило следует выбирать специальным способом. Говорят, что грамматика имеет свойство $LL(k)$, если для выбора правила достаточно рассмотреть только цепочку wAx и первые k символов непросмотренной цепочки u. Формально данное свойство можно описать следующим образом [3, 13, 14].

Определение 3.12. Грамматика G имеет свойство $LL(k)$, если из существования двух цепочек левых выводов:

$S \Rightarrow^* wAx \Rightarrow wzx \Rightarrow^* wu$;

$S \Rightarrow^* wAx \Rightarrow wtx \Rightarrow^* wv$

и из условия $\mathrm{FIRST}(u) = \mathrm{FIRST}(v)$ следует $z = t$.

Пример 3.11. Рассмотрим грамматику $G_1 = (\{a, b\}, \{S, T\}, P, S)$

P: $S \to aTS \mid b$;

$T \to bST \mid a$

и два вывода:

(1) $S \Rightarrow^* wSv \Rightarrow wzv \Rightarrow^* wx$;

(2) $S \Rightarrow^* wSv \Rightarrow wtv \Rightarrow^* wy$.

Пусть цепочки x и y начинаются с символа a, при этом $\mathrm{FIRST}(x) = \mathrm{FIRST}(y) = a$. Это означает, что в выводе участвовало

правило $S \to aAS$. Следовательно, $z = t = aAS$. Пусть цепочки x и y начинаются с символа b, при этом FIRST(x) = FIRST(y) = b. Это означает, что в выводе участвовало правило $S \to b$. Следовательно, $z = t = b$.

Для выводов

(3) $S \Rightarrow^* wAv \Rightarrow wzv \Rightarrow^* wx$;

(4) $S \Rightarrow^* wAv \Rightarrow wtv \Rightarrow^* wy$

рассуждение аналогично. Таким образом, грамматика обладает свойством $LL(1)$.

Определение 3.13. Грамматика называется *LL(k)-грамматикой*, если она обладает свойством $LL(k)$ для некоторого $k > 0$.

В названии грамматики первая литера L происходит от слова left и означает, что входная цепочка символов читается слева направо. Вторая литера L также происходит от слова left и означает, что при работе распознавателя используется левосторонний вывод. Вместо символа k в названии класса грамматики стоит некоторое число, которое показывает, сколько символов надо рассмотреть, чтобы сделать однозначный вывод. Так, существуют $LL(1)$-грамматики, $LL(2)$-грамматики и другие классы LL-грамматик. Очевидно, что $LL(0)$-грамматика бессмысленна, так как этом случае разбор вообще не зависит от входной цепочки.

С использованием описанного выше множества выбирающих символов можно дать следующее определение $LL(1)$-грамматик [11].

Определение 3.14. КС-грамматика называется $LL(1)$-грамматикой тогда и только тогда, когда множества выбирающих символов для правил с одинаковой левой частью не пересекаются.

Пример 3.12. Дана грамматика $G_2 = (\{a, +, *, (,)\}, \{S, E, F, A, B\}, P, S)$, где

P: $S \to AE$;

$E \to +AE \mid \lambda$;

$A \to FB$;

$B \to *FB \mid \lambda$;

$F \to a \mid (S)$.

Множество выбирающих символов для правил с одинаковой левой частью:

$\mathrm{SEL}(E \to +AE) = \{+\}$;

$\mathrm{SEL}(E \to +AE) = \mathrm{FOLLOW}(E) = \{)\}$.

Множества $\{+\} \cap \{)\} = \varnothing$ не пересекаются.

$\mathrm{SEL}(B \to *FB) = \{*\}$;

$\mathrm{SEL}(B \to +AE) = \mathrm{FOLLOW}(B) = \{+,)\}$.

Множества $\{*\} \cap \{+,)\} = \varnothing$ не пересекаются.

$\mathrm{SEL}(F \to a) = \{a\}$;

$\mathrm{SEL}(F \to (S)) = \{(\}$.

Множества $\{a\} \cap \{(\} = \varnothing$ не пересекаются.

Можно сделать вывод, что G_2 является $LL(1)$-грамматикой.

В случае $k = 1$ для выбора правила для замены нетерминала A достаточно знать только нетерминал A и a — первый символ цепочки u:

- следует выбрать правило $A \to x$, если a входит в $\mathrm{FIRST}(x)$;
- следует выбрать правило $A \to e$, если a входит в $\mathrm{FOLLOW}(A)$.

Для $LL(k)$-грамматик известны следующие полезные свойства:

- всякая $LL(k)$-грамматика для любого $k > 0$ является однозначной;
- существует алгоритм, позволяющий проверить, является ли заданная грамматика $LL(k)$-грамматикой для строго определенного числа k.

Есть, однако, неразрешимые проблемы для произвольных КС-грамматик:

- не существует алгоритма, который мог бы проверить, является ли заданная КС-грамматика $LL(k)$-грамматикой для некоторого произвольного числа k;
- не существует алгоритма, который мог бы преобразовать произвольную КС-грамматику к виду $LL(k)$-грамматики для некоторого k (или доказать, что преобразование невозможно) [2].

$LL(k)$-грамматики служат для построения нисходящих распознавателей без возвратов.

3.3.5. LR-грамматики

Более широкий класс языков по сравнению с $LL(k)$-грамматиками определяется так называемыми *LR(k)-грамматиками*. На их основе строятся восходящие распознаватели без возвратов. Если в процессе LR-разбора (рис. 3.3) принять детерминированное решение о замене удается, рассматривая только цепочку x и первые k символов непросмотренной части входной цепочки u (эти k символов называют *авангардной цепочкой*), говорят, что грамматика обладает $LR(k)$-свойством [3, 13, 14].

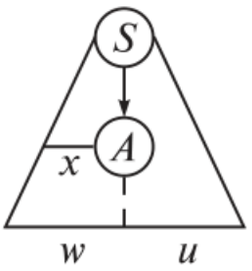


Рис. 3.3. Схема построения дерева разбора для LR-грамматики

Для определения $LR(k)$-грамматики придется ввести понятие *пополненной грамматики*. Грамматика G' является пополненной грамматикой, построенной на основании исходной грамматики $G = (T, N, P, S)$, если выполняются следующие условия:

- грамматика G' совпадает с грамматикой G, если целевой символ S не встречается нигде в правых частях правил грамматики G;
- грамматика G' строится как грамматика $G' = (T, N \cup \{S'\}, P \cup \{S' \to S\}, S')$, если целевой символ S встречается в правой части хотя бы одного правила из множества P в исходной грамматике G.

Определение 3.15. Грамматика G' имеет свойство $LR(k)$, если из существования двух выводов:

$$S \Rightarrow^* \alpha A \omega \Rightarrow \alpha z \omega;$$

$$S \Rightarrow^* \gamma B x \Rightarrow \alpha z y$$

и из условия $\mathrm{FIRST}(\omega) = \mathrm{FIRST}(y)$ следует

$$\alpha A y = \gamma B x \ (\alpha = \gamma, A = B, y = x).$$

Определение 3.16. Грамматика называется $LR(k)$-грамматикой, если она обладает свойством $LR(k)$ для некоторого $k \geq 0$.

В названии $LR(k)$-грамматики первая литера L — left — означает порядок чтения входной цепочки символов слева направо. Вторая литера R происходит от слова right и означает, что в ре-

зультате работы распознавателя получается правосторонний вывод. Вместо символа k в названии грамматики стоит число, которое показывает, сколько символов входной цепочки надо рассмотреть, чтобы принять решение о действии на каждом шаге работы распознавателя.

Пример 3.13. Грамматика $G_3 = (\{a, t, ,\}, \{S, A, B\}, P, S)$, где

P: $S \to A, S \mid A$;

$A \to aB$;

$B \to t \mid t, B$

является $LR(2)$-грамматикой.

Доказано, что класс LR-грамматик является более широким, чем класс LL-грамматик, т. е. есть для каждого КС-языка, заданного LL-грамматикой, может быть построена LR-грамматика, задающая тот же язык, но не наоборот.

Для $LR(k)$-грамматик известны следующие полезные свойства:

- всякая $LR(k)$-грамматика для любого $k \geq 0$ является однозначной;
- существует алгоритм, позволяющий проверить, является ли заданная грамматика $LR(k)$-грамматикой для строго определенного числа k [11].

Есть, однако, неразрешимые проблемы для произвольных КС-грамматик:

- не существует алгоритма, который мог бы проверить, является ли заданная КС-грамматика $LR(k)$-грамматикой для некоторого произвольного числа k;
- не существует алгоритма, который мог бы преобразовать (или доказать, что преобразование невозможно) произвольную КС-грамматику к виду $LR(k)$-грамматики для некоторого k [2].

3.3.6. Грамматики предшествования

Грамматики предшествования характеризуются тем, что для каждой упорядоченной пары символов в грамматике устанавливается некоторое отношение, называемое *отношением предшествования*.

Существует несколько видов грамматик предшествования, которые различаются в зависимости от того, какие отношения предшествования в них определены и между какими типами символов (терминальными или нетерминальными) установлены эти отношения [2, 13].

Выделяют следующие виды грамматик предшествования:

- простого;
- расширенного;
- слабого;
- смешанной стратегии;
- операторного.

Далее рассмотрим наиболее простой и распространенный тип — грамматики простого предшествования.

Определение 3.17. *Грамматикой простого предшествования* называется такая приведенная КС-грамматика $G = (T, N, P, S)$, для которой выполняются следующие условия:

1. Для каждой упорядоченной пары терминальных и нетерминальных символов выполняется не более одного из трех отношений предшествования:
   - $X =\bullet Y$ для всех $X, Y \in (T \cup N)$, если и только если существует правило $A \to \alpha X Y \beta \in P$, где $A \in N$, $\alpha, \beta \in (T \cup N)^*$;
   - $X <\bullet Y$ для всех $X, X \in (T \cup N)$, если и только если существуют правило $A \to \alpha X D \beta \in P$ и вывод $D \Rightarrow^* Y\gamma$, где $A, D \in N$, $\alpha, \beta, \gamma \in (T \cup N)^*$;
   - $X \bullet> Y$ для всех $X, Y \in (T \cup N)$, если и только если существуют правило $A \to \alpha C Y \beta \in P$ и вывод $C \Rightarrow^* \gamma X$ или существуют правило $A \to \alpha C D \beta \in P$ и выводы $C \Rightarrow^* \gamma X$ и $D \Rightarrow^* Y\omega$, где $A, C, D \in N$, $\alpha, \beta, \gamma, \omega \in (T \cup N)^*$.
2. Разные правила в грамматике не должны иметь одинаковые правые части.

Отношения $=\bullet$, $<\bullet$ и $\bullet>$ называют отношениями предшествования для символов. Отношение предшествования однозначно определено для каждой упорядоченной пары символов. Если два символа не могут находиться рядом ни в одном элементе разбора синтаксически правильной цепочки, то отношение предшествования между ними отсутствует.

Отношения предшествования зависят от порядка, в котором стоят символы, и в этом смысле их нельзя путать со знаками математических операций — они не обладают ни свойством коммутативности, ни свойством ассоциативности. Например, если

известно, что $X \bullet> Y$, то не обязательно выполняется отношение $Y <\bullet X$.

Для грамматик простого предшествования характерны следующие полезные свойства:

- каждая из них является однозначной;
- чтобы проверить, является ли произвольная КС-грамматика грамматикой простого предшествования, достаточно проверить, удовлетворяет ли она описанным выше условиям [2].

Однако для грамматик простого предшествования не существует алгоритма, который мог бы преобразовать произвольную КС-грамматику в грамматику простого предшествования (или доказать, что преобразование невозможно).

Пример 3.14. Дана грамматика $G = (\{a, b, c\}, \{S\}, P, S)$ с правилами $P = \{S \to aSSb \mid c\}$. Определить отношения предшествования между символами грамматики.

Получили:

$a =\bullet S$; $S =\bullet S$; $S =\bullet b$;

$a <\bullet a$; $a <\bullet c$; $S <\bullet a$; $S <\bullet c$;

$b \bullet> a$; $b \bullet> b$; $b \bullet> c$; $b \bullet> S$; $c \bullet> a$; $c \bullet> b$; $c \bullet> c$; $c \bullet> S$.

Для построения распознавателя на основе грамматик предшествования к входной цепочке добавляют два символа: $\perp_{\text{Н}}$ — символ начала цепочки и $\perp_{\text{К}}$ — символ конца цепочки. Эти дополнительные символы вступают в следующие отношения с символами грамматики:

- $\perp_{\text{Н}} <\bullet X$ для всех $X \in (T \cup N)$, если существует вывод $S \Rightarrow^* \Rightarrow^* Xy$, где $S \in N$, $y \in (T \cup N)^*$;
- $\perp_{\text{К}} \bullet> X$ для всех $X \in (T \cup N)$, если существует вывод $S \Rightarrow^* \Rightarrow^* yX$, где $S \in N$, $y \in (T \cup N)^*$.

3.4. Недетерминированные и детерминированные магазинные автоматы

Если конечные автоматы, рассмотренные в гл. 2, дают возможность построить распознаватели на основе регулярных грамматик, применяемых для лексического анализа, то для распознавателей КС-языков используются *магазинные автоматы*.

3.4.1. Распознаватели на основе автоматов с магазинной памятью

Модель магазинного автомата (рис. 3.4) состоит из:
- входной ленты;
- устройства управления;
- вспомогательной ленты, называемой магазином или стеком [3, 4, 9].

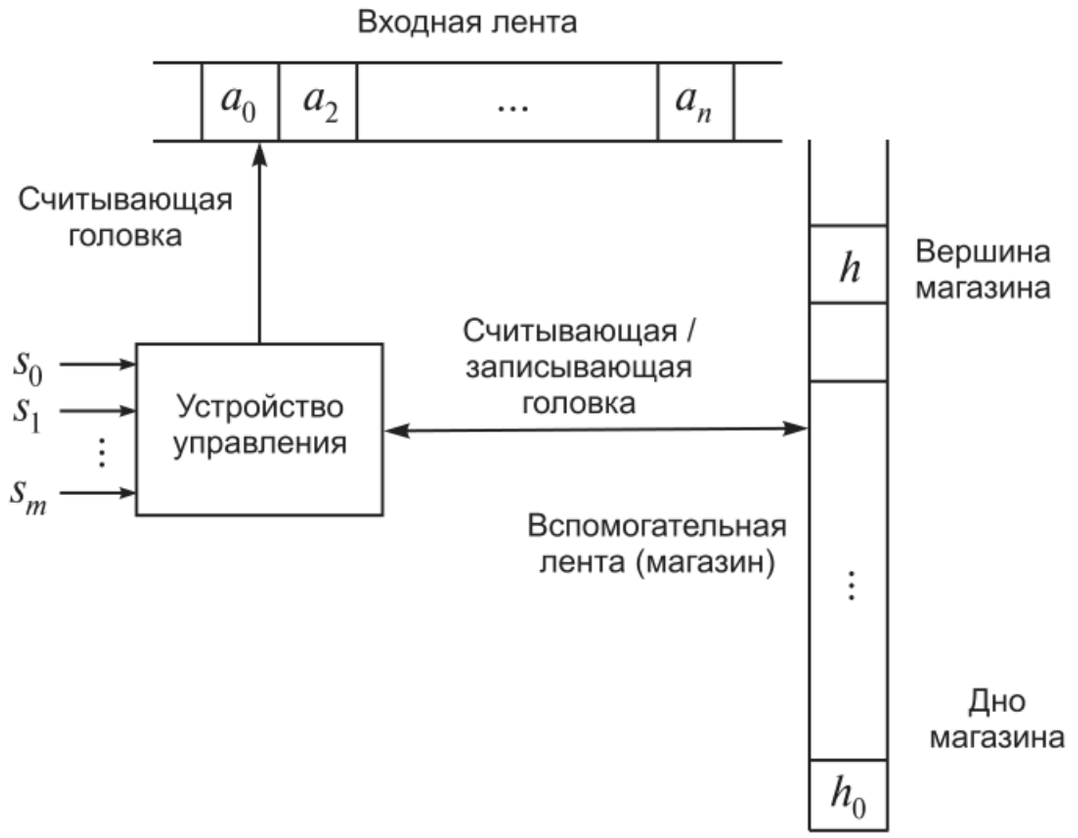


Рис. 3.4. Модель магазинного автомата

Входная лента содержит символы входного алфавита. При этом предполагается, что в неиспользуемых клетках входной ленты расположены пустые символы λ.

Вспомогательная лента разделена на клетки, в которых могут располагаться символы магазинного алфавита. Начало вспомогательной ленты называется дном магазина. Устройство управления перемещает вдоль ленты две головки.

Головка входной ленты является считывающей и может перемещаться только в одну сторону (вправо) или оставаться на месте. Головка вспомогательной ленты может выполнять как чтение, так и запись, но эти операции связаны с перемещением головки определенным образом:
- при записи головка предварительно сдвигается на одну позицию вверх, а затем символ заносится на ленту;

- при чтении символ, находящийся под головкой, считывается с ленты, а затем головка сдвигается на одну позицию вниз, таким образом головка всегда установлена против последнего записанного символа. Позицию, находящуюся в рассматриваемый момент времени под головкой, называют вершиной магазина.

Определение 3.18. Магазинный автомат M определяется следующей совокупностью семи объектов: $M = \{S, T, H, \delta, s_0, h_0, F\}$, где [3] S — алфавит состояний; T — входной алфавит; H — алфавит магазинных символов, записываемых на вспомогательную ленту; δ — функция переходов, δ: $\{S \times T \cup \{\lambda\} \times H\} \to S \times H^*$, если M — автомат детерминированный, и δ: $\{S \times T \cup \{\lambda\} \times H\} \to 2^{S \times H^*}$, если M — автомат недетерминированный; s_0 — начальное состояние, $s_0 \in S$; h_0 — маркер дна, он всегда записывается на дно магазина , $h_0 \in H$; F — множество конечных состояний, являющееся подмножеством мкожества S.

Функция δ отображает тройки (p_i, s_j, h_k) в пары (s_r, υ) для детерминированного автомата или в множество таких пар для недетерминированного автомата, где $\upsilon \in H^*$, а h_k — символ в вершине магазина.

Эта функция описывает изменение состояния магазинного автомата, происходящее при чтении символа с входной ленты и перемещении входной головки. В дальнейшем при построении магазинных автоматов потребуются две разновидности функций переходов:

1) функция переходов с пустым символом в качестве входного символа

$$\delta^0(s, \lambda, h) = (s', \beta),$$

предписывающая прочитать символ h из вершины магазина, изменить состояние автомата на s' и записать цепочку β в магазин независимо от того, какой символ находится под читающей головкой входной ленты;

2) функция переходов с определенным входным символом

$$\delta(s, a, h) = (s', \beta),$$

предписывающая прочитать входной символ a, изменить состояние автомата на s' и заменить верхний символ магазина h цепочкой β.

3.4.2. Работа магазинного автомата

Для того чтобы описать, как работает автомат, введем понятие *конфигурации*.

Определение 3.19. Конфигурацией автомата M называют тройку $(s, \alpha, \gamma) \in S \times T^* \times H^*$, где [3, 9] s — текущее состояние управляющего устройства; α — неиспользованная часть входной цепочки $\alpha \in T^*$, самый левый символ этой цепочки находится под головкой. Если $a = \lambda$, то считается, что вся входная цепочка прочитана; γ — цепочка, записанная в магазине, $\gamma \in H^*$, самый правый символ цепочки считается вершиной магазина. Если $\gamma = \lambda$, то магазин пуст.

Работа автомата может быть представлена как смена конфигураций. Один такт работы автомата заключается в определении новой конфигурации исходя из текущей, что символизирует запись:

$(s, a\alpha, \gamma h) \vdash (s', \alpha, \gamma\beta)$.

Такая смена конфигураций возможна, если функция $\delta(s, a, h)$ или $\delta(s, \lambda, h)$ определена и имеет значение (s', β). При этом предполагается, что автомат:

- читает символ a, находящийся под головкой, или не читает ничего в случае входного символа λ;
- определяет новое состояние s';
- читает символ h, находящийся в вершине магазина и записывает цепочку β в магазин вместо символа h. Если $\beta = \lambda$, то верхний символ оказывается удаленным из магазина.

Таким образом, возможны три случая работы автомата:

- функция $\delta(s, a, h)$ определена и выполняется такт работы;
- функция $\delta(s, a, h)$ не определена, но определена функция $\delta(s, \lambda, h)$ и выполняется пустой такт (без чтения входной информации);
- функции $\delta(s, a, h)$ и $\delta(s, \lambda, h)$ не определены, в этом случае дальнейшая работа автомата невозможна.

Начальной конфигурацией называется конфигурация (s_0, α, h_0), где s_0 — начальное состояние, α — исходная цепочка, h_0 — маркер дна магазина.

Заключительная конфигурация — (s, λ, λ), где $s \in F$.

Для обозначения последовательности сменяющих друг друга конфигураций будем использовать знак $\vdash^*$. Таким образом последовательность

$$(s_1, \alpha_1, \beta_1) \vdash (s_2, \alpha_2, \beta_2) \vdash \ldots \vdash (s_n, \alpha_n, \beta_n)$$

записывается в сокращенном виде как

$$(s_1, \alpha_1, \beta_1) \vdash^* (s_n, \alpha_n, \beta_n).$$

3.4.3. Язык, допускаемый магазинным автоматом

Определение 3.20. Цепочка α называется *допустимой* для автомата M, если существует последовательность конфигураций, в которой первая конфигурация является начальной с цепочкой α, а последняя — заключительной, т. е. $(s_0, \alpha, h_0) \vdash^* (s_1, \lambda, \lambda)$, где $s_1 \in F$.

Определение 3.21. Множество цепочек, допустимых для автомата M, называется языком, *допускаемым* или *определяемым* автоматом M, и обозначается $L(M)$

$$L(M) = \{\alpha \mid (s_0, \alpha, h_o) \vdash^* (s', \lambda, \lambda) \text{ и } s' \in F\}$$

Пример 3.15. Пусть магазинный автомат M_1 имеет вид

M_1: $T = \{a, b\}$; $S = \{s_0, s_1, s_2\}$; $H = \{h_0, a\}$; $F = \{s_0\}$;

$\delta(s_0, a, h_0) = (s1, h_0a)$;

$\delta(s_1, a, a) = (s_1, aa)$;

$\delta(s_1, b, a) = (s_2, \lambda)$;

$\delta(s_2, b, a) = (s_2, \lambda)$;

$\delta(s_2, \lambda, h_0) = (s_0, \lambda)$.

Этот автомат является *детерминированным*, поскольку каждому набору аргументов соответствует единственное значение функции. Работу автомата при распознавании входной цепочки *aabb* можно представить в виде последовательности конфигураций:

$$(s_0, aabb, h_0) \vdash (s_1, abb, h_0a) \vdash (s_1, bb, h_0aa) \vdash (s_2, b, h_0a) \vdash$$

$$\vdash (s_2, \lambda, h_0) \vdash (s_0, \lambda, \lambda).$$

Нетрудно проверить, что при задании входной цепочки *aabbb* автомат не сможет закончить работу. Следовательно, эта цепочка не принадлежит языку, допускаемому автоматом M_1.

Пример 3.16. Магазинный автомат M_2, заданный следующим описанием:

M_2: $T = \{a, b\}$; $S = \{s_0, s_1, s_2\}$; $Z = \{h_0, a, b\}$; $F = \{s_2\}$;

1) $\delta(s_0, a, h_0) = (s_0, h_0a)$;

2) $\delta(s_0, b, h_0) = (s_0, h_0b)$;

3) $\delta(s_0, a, a) = \{(s_0, aa), (s_1, \lambda)\}$;

4) $\delta(s_0, b, a) = (s_0, ab)$;

5) $\delta(s_0, a, b) = (s_0, ba)$;

6) $\delta(s_0, b, b) = \{(s_0, bb), (s_1, \lambda)\}$;

7) $\delta(s_1, a, a) = (s_1, \lambda)$;

8) $\delta(s_1, b, b) = (s_1, \lambda)$;

9) $\delta(s_2, \varepsilon, h_0) = (s_2, \lambda)$,

является *недетерминированным* автоматом, поскольку одному и тому же набору аргументов, например (s_0, a, a), соответствуют два значения функции. Рассмотрим работу автомата для входной цепочки *abba*. Если использовать последовательность команд (1), (4), (6.1), (5), то получим последовательность конфигураций:

$(s_0, abba, h_0) \vdash (s_0, bba, h_0a), \vdash (s_0, ba, h_0ab), \vdash (s_0, a, h_0abb), \vdash$

$\vdash (s_0, \lambda, h_0abba)$,

которая показывает, что дальнейшая работа невозможна, так как входная цепочка прочитана и переход (s_0, λ, h_0abba) не определен. Если же использовать последовательность команд (1), (4), (6.2), (3), (9), то получим заключительную конфигурацию:

$(s_0, abba, h_0) \vdash (s_0, bba, h_0a), \vdash (s_0, ba, h_0ab), \vdash (s1, a, h_0a), \vdash$

$\vdash (s_1, \lambda, h_0) \vdash (s2, \lambda, \lambda)$.

Таким образом, цепочка *abba* допускается автоматом M_2.

3.4.4. Построение магазинного автомата

Пусть задана контекстно-свободная грамматика $G = \{N, T, P, S\}$. Определим компоненты автомата M следующим образом:

$S = \{s_0\}$, $T = T$, $H = N \cup T \cup \{h_0\}$, $F = \{s_0\}$,

в качестве начального состояния автомата примем символ s_0 и построим функцию переходов так:

1. Для всех $A \in N$ таких, что встречаются в левой части правил $A \to \alpha$, построим команды вида:

   $\delta^0(s_0, \lambda, A) = (s_0, \alpha^R)$, где α^R — реверс цепочки α.

2. Для всех $a \in T$ построим команды вида:

   $\delta(s_0, a, a) = (s_0, \lambda)$.

3. Для перехода в конечное состояние построим команду

   $\delta(s_0, \lambda, h_0) = (s_0, \lambda)$.

4. Начальную конфигурацию автомата определим в виде:

   $(s_0, \omega, h_0\ I)$,

где ω — исходная цепочка, записанная на входной ленте.

Автомат, построенный по приведенным выше правилам, работает следующим образом. Если в вершине магазина находится терминал, и символ, читаемый с входной ленты, совпадает с ним, то по команде типа (2) терминал удаляется из магазина, а входная головка сдвигается. Если в вершине магазина находится нетерминал, то выполняется команда типа (1), которая вместо терминала записывает в магазин цепочку, представляющую собой правую часть правила грамматики. Таким образом, автомат, последовательно заменяя нетерминалы, появляющиеся в вершине магазина, строит в магазине левый вывод входной цепочки, удаляя полученные терминальные символы, совпадающие с символами входной цепочки. Это означает, что каждая цепочка, которая может быть получена с помощью левого вывода в грамматике G, допускается построенным автоматом M.

Пример 3.17. Дана грамматика с правилами:

P: $E \to E+T \mid T$;

$T \to T*F \mid F$;

$F \to (E) \mid a$.

Для искомого автомата имеем:

$T = \{a, +, *,), (\}$, $H = \{E, T, F, a, +, *,), h_0, (\}$,

$S = \{s_0\}$, $F = \{s_0\}$.

Для всех правил грамматики строим команды типа (1):

1) $\delta^0(s_0, \varepsilon, E) = \{(s_0, T+E); (s_0, T)\}$;

2) $\delta^0(s_0, \lambda, T) = \{(s_0, F*T); (s_0, F)\}$;

3) $\delta^0(s_0, \lambda, F) = \{(s_0, (E)); (s_0, a)\}$.

Для всех терминальных символов строим команды типа (2):

4) $\delta(s_0, a, a) = (s_0, \lambda)$;

5) $\delta(s_0, +, +) = (s_0, \lambda)$;

6) $\delta(s_0, *, *) = (s_0, \lambda)$;

7) $\delta(s_0, (, () = (s_0, \lambda)$;

8) $\delta(s_0,),)) = (s_0, \lambda)$.

Для перехода в конечное состояние построим команду:

9) $\delta(s_0, \lambda, h_0) = (s_0, \lambda)$.

Построенный автомат является недетерминированным.

Начальную конфигурацию с цепочкой $a+a*a$ запишем так:

$(s_0, a+a*a, h_0E)$.

Последовательность тактов работы построенного автомата, показывающая, что заданная цепочка допустима, имеет вид:

$(s_0, a+a*a, h_0E) \vdash (s_0, a+a*a, h_0T+E) \vdash (s_0, a+a*a, h_0T+T) \vdash$

$\vdash (s_0, a+a*a, h_0T+F) \vdash (s_0, a+a*a, h_0T+a) \vdash (s_0, +a*a, h_0T+) \vdash$

$\vdash (s_0, a*a, h_0T) \vdash (s_0, a*a, h_0F*T) \vdash (s_0, a*a, h_0F*F) \vdash$

$\vdash (s_0, a*a, h_0F*a) \vdash (s_0, *a, h_0F*) \vdash (s_0, a, h_0F) \vdash (s_0, a, h_0a) \vdash$

$\vdash (s_0, \lambda, h_0) \vdash (s_0, \lambda, \lambda)$.

Отметим, что последовательность правил, используемая построенным автоматом, соответствует левому выводу входной цепочки:

$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+T*F \Rightarrow a+F*F \Rightarrow$

$\Rightarrow a+a*F \Rightarrow a+a*a$.

Такой вывод является нисходящим разбором; построение дерева разбора проводится сверху вниз — от корня к листьям (рис. 3.5).

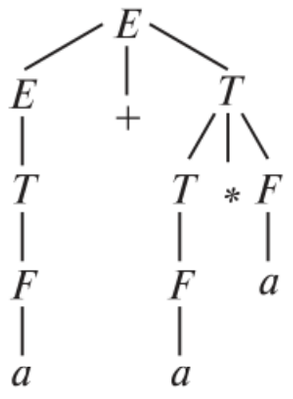


Рис. 3.5. Дерево нисходящего разбора

Магазинные автоматы часто называют распознавателями, поскольку они определяют, является ли цепочка, подаваемая на вход автомата, допустимой, и, следовательно, отвечают на вопрос, принадлежит ли эта цепочка языку, порождаемому грамматикой, использованной для построения автомата.

Учитывая характер построения вывода в магазине, автоматы рассмотренного типа называют нисходящими распознавателями.

3.5. Распознаватели КС-грамматик

Ниже будут рассмотрены наиболее распространенные распознаватели КС-грамматик.

3.5.1. Распознаватели с возвратом

Наиболее простой тип распознавателей для КС-языков — это *распознаватели с возвратом*, которые создаются на основе недетерминированных МП-автоматов (в общем случае не преобразуемых в детерминированные) [9, 13].

На некотором шаге работы недетерминированного автомата может возникнуть неоднозначность — наличие нескольких допустимых конфигураций автомата, следующих за текущей. Возможны два выхода из данной ситуации: либо реализация нескольких параллельных процессов, пока один из них не достигнет заключительной конфигурации (все остальные при этом завершаются), либо последовательный перебор возможных конфигураций в месте неоднозначности с сохранением этого состояния автомата в памяти и возвратом в него, пока какой-либо из путей не приведет к заключительной конфигурации. Если ни один из вариантов не привел к заключительной конфигурации, алгоритм завершается с ошибкой.

Реализация первого алгоритма связана с управлением параллельными процессами и является достаточно сложной, поэтому большее распространение получил второй вариант — разбор с возвратами. Однако время выполнения второго алгоритма экспоненциально зависит от длины входной цепочки и поэтому его использование возможно только для простых КС-языков с малой длиной входных предложений. Для многих классов КС-языков существуют более эффективные методы разбора.

Нисходящий распознаватель с возвратом

Нисходящий распознаватель с возвратом представляет собой МП-автомат с одним состоянием s: $M = (\{s\}, T, H, \delta, s, h_0, \{s\})$. Автомат распознает цепочки КС-языка, заданного КС-грамматикой $G = (T, N, P, S)$. Входной алфавит автомата содержит терминальные символы грамматики: $T = T$, а алфавит магазинных символов строится из терминальных и нетерминальных символов грамматики: $H = T \cup N$ [2].

Начальной конфигурацией автомата является (s, α, S) — автомат пребывает в единственном состоянии s, считывающая головка находится в начале входной цепочки символов $\alpha \in T$, в стеке находится символ, соответствующий целевому символу грамматики S.

Заключительной конфигурацией автомата является набор (s, λ, λ) — автомат пребывает в своем единственном состоянии s, считывающая головка находится за концом входной цепочки символов, стек пуст.

Функция переходов МП-автомата δ строится на основе правил грамматики, как описано в подразд. 3.3.4:

- $\delta(s, \lambda, A) = (s, \alpha^R)$, $A \in N$, $\alpha \in (T \cup N)^*$, если правило $A \to \alpha$ содержится во множестве правил P-грамматики;
- $\delta(s, a, a) = (s, \lambda)$ для всех $a \in T$.

Работа данного МП-автомата происходит следующим образом:

- если на вершине стека автомата находится нетерминальный символ A, то он заменяется на цепочку символов α, если в грамматике языка есть правило $A \to \alpha$ без сдвига считывающей головки автомата;
- если на верхушке стека находится терминальный символ a, который совпадает с текущим символом входной цепочки, то этот символ выбрасывается из стека со сдвигом считывающей головки на одну позицию вправо.

Если грамматика содержит более одного правила вида $A \to \alpha$, то МП-автомат будет недетерминированным.

Данный МП-автомат строит левосторонние выводы для грамматики $G = (T, N, P, S)$ и читает цепочку входных символов слева направо, поэтому естественным для него является построение дерева вывода сверху вниз. Грамматика G при этом не должна содержать левую рекурсию, иначе автомат может войти в бесконечный цикл.

Алгоритм «сдвиг—свертка»

Данный распознаватель строится на основе МП-автомата с одним состоянием s: $M = (\{s\}, T, H, \delta, s, h_0, \{s\})$. Автомат распознает цепочки КС-языка, заданного грамматикой $G = (T, N, P, S)$. Входной алфавит автомата содержит терминальные символы грамматики: $T = T$, а алфавит магазинных символов состоит из терминальных и нетерминальных символов грамматики: $H = T \cup \cup N$ [2, 3, 13].

Начальной конфигурацией автомата является набор (s, α, λ) — автомат пребывает в своем единственном состоянии s, считывающая головка находится в начале входной цепочки символов $\alpha \in T$, стек пуст.

Заключительной конфигурацией автомата является набор (s, λ, S) — автомат пребывает в своем единственном состоянии s, считывающая головка находится за концом входной цепочки символов, в стеке находится целевой символ грамматики S.

Функция переходов МП-автомата δ строится на основе правил грамматики:

- $\delta(s, \lambda, \beta) = (s, B)$, $B \in N$, $\beta \in (T \cup N)^*$, если правило $B \to \beta$ содержится во множестве правил P грамматики G;
- $\delta(s, b, \lambda) = (s, b)$, для всех $b \in T$.

Работа данного МП-автомата происходит следующим образом:

- если на вершине стека автомата находится цепочка символов β, то проводится ее свертка к нетерминальному символу B, если в грамматике языка существует правило вида $B \to \beta$, без сдвига считывающей головки автомата;
- если под считывающей головкой находится некоторый символ входной цепочки a, то он помещается в стек со сдвигом головки на одну позицию вправо.

Такой алгоритм получил название «сдвиг—свертка» по основным действиям алгоритма.

Данный МП-автомат строит правосторонние выводы для грамматики $G = (T, N, P, S)$ и читает цепочку входных символов слева направо, т. е. дерево вывода строится снизу вверх. Такой распознаватель называется *восходящим*. Для моделирования этого автомата необходимо, чтобы грамматика G не содержала λ-правил и цепных правил (см. подразд. 3.1), иначе алгоритм может войти в бесконечный цикл из сверток.

Рассмотренные алгоритмы обладают двумя преимуществами: 1) универсальностью — возможностью распознавать цепочки

языков, заданных любой приведенной КС-грамматикой; 2) простотой реализации.

Тем не менее они не получили применения в реальных трансляторах; из-за низкой эффективности их используют только при разборе коротких цепочек, принадлежащих простым КС-языкам.

3.5.2. Табличные распознаватели для КС-языков

В отличие от распознавателей на основе МП-автоматов табличные распознаватели, получая на вход цепочку символов, сначала создают для нее промежуточное хранилище информации в виде таблицы, а затем с ее использованием производят разбор данной цепочки. В связи с этим эффективность таких распознавателей определяется длиной входной цепочки.

Табличные распознаватели являются универсальными, т. е. они могут использоваться для разбора цепочек, принадлежащих любому КС-языку, и наиболее эффективны с точки зрения вычислительных ресурсов по сравнению с другими универсальными распознавателями [13].

В реальных компиляторах такие распознаватели, как правило, не используются из-за полиномиальной зависимости от длины цепочки.

Алгоритм Кока — Янгера — Касами

Язык, допускаемый распознавателем, задается грамматикой $G = (T, N, P, S)$. Предположим, что на входе цепочка символов $\alpha \in T^*$, $\alpha = a_1 a_2 ... a_n$, $|\alpha| = n$. В соответствии с алгоритмом Кока — Янгера — Касами для данной цепочки строится таблица R размерностью $n \times n$, такая, что $A \in R_{ij}$, если $A \Rightarrow^+ a_i ... a_{i+j-1}$, $A \in N$. В результате каждая ячейка таблицы содержит множество нетерминалов грамматики G [2, 3].

Тогда то, что $S \in R_{1n}$, означает, что цепочка α принадлежит данному языку. В этом случае по таблице R можно проследить разбор цепочки $S \Rightarrow^* \alpha$.

Для построения вывода по алгоритму Кока — Янгера — Касами грамматика G должна быть в нормальной форме Хомского.

Алгоритм Кока — Янгера — Касами фактически состоит из трех вложенных циклов, поэтому время его выполнения имеет кубическую зависимость от длины входной цепочки символов.

Требуемый для хранения таблицы $R_{n\times n}$ объем памяти имеет квадратичную зависимость от длины входной цепочки.

Алгоритм Эрли

Алгоритм Эрли предусматривает для заданной КС-грамматики $G = (T, N, P, S)$ и входной цепочки $\omega = a_1a_2...a_n$, $\omega \in T$, $|\omega| = n$ построение последовательности списков ситуаций I_0, I_1, ..., I_n. Каждая ситуация, входящая в список I_j для входной цепочки ω, представляет собой структуру вида $[A \to X_1X_2 \ldots X_k \bullet X_{k+1} \ldots X_m, i]$, $X_k \in (N \cup T)$, где правило $A \to X_i \ldots X_m$ принадлежит множеству правил P грамматики G, и $0 \le i \le n$, $0 \le k \le m$ [2, 3]. Символ $\bullet$ (точка) — это метасимвол, который не входит ни во множество терминальных (N), ни во множество нетерминальных (T) символов грамматики. В ситуации I_j этот символ может стоять в любой позиции, в том числе в начале ($\bullet X_1 \ldots X_m$,) или в конце ($X_1 \ldots X_m \bullet$) всей цепочки символов правила $A \to X_1 \ldots X_m$. Если цепочка символов правила пустая ($A \to \lambda$), то ситуация будет выглядеть так: $[A \to \bullet, i]$.

Список ситуаций строится таким образом, что для j, $0 \le j \le n$: $[A \to \alpha \bullet \beta, i] \in I_j$ тогда и только тогда, если существует $S \Rightarrow^* \gamma A \delta$, $\gamma \Rightarrow^* a_1 \ldots a_i$, и $\alpha \Rightarrow^* a_{i+1} \ldots a_j$. Иначе говоря, между вторым компонентом ситуации и номером списка, в котором он появляется, заключена часть входной цепочки, выводимая из правила A. Условия ситуации I_j гарантируют возможность применения правила $A \to \alpha\beta$ в выводе некоторой входной цепочки, совпадающей с заданной цепочкой ω до позиции j.

Вывод заданной входной цепочки ω в грамматике G после завершения алгоритма Эрли возможен при условии $[S \to \omega \bullet, 0] \in I_n$. На основе выявленного в результате выполнения алгоритма списка ситуаций можно с помощью специальной процедуры построить всю цепочку вывода и получить номера применяемых правил.

Как и все табличные алгоритмы, алгоритм Эрли обладает полиномиальными характеристиками зависимости времени выполнения и объема памяти от длины входной цепочки. Для произвольной КС-грамматики время выполнения данного алгоритма будет иметь кубическую зависимость от длины входной цепочки, а необходимый объем памяти — квадратичную зависимость от длины входной цепочки. Но для однозначных КС-грамматик алгоритм Эрли имеет лучшие характеристики — время его выполнения в этом случае квадратично зависит от длины входной це-

почки. Кроме того, для некоторых классов КС-грамматик время выполнения алгоритма линейно зависит от длины входной цепочки (правда, для этих классов, как правило, существуют более простые алгоритмы распознавания).

В целом алгоритм Эрли имеет наилучшие характеристики среди всех универсальных алгоритмов распознавания входных цепочек для произвольных КС-грамматик. По ряду параметров он превосходит алгоритм Кока — Янгера — Касами для однозначных грамматик, хотя и является более сложным в реализации.

3.5.3. Метод рекурсивного спуска

Метод рекурсивного спуска — легко реализуемый детерминированный метод разбора сверху вниз. С его помощью на основе соответствующей грамматики можно написать синтаксический анализатор [2—4, 9, 13].

Синтаксический анализатор содержит по одной рекурсивной процедуре для каждого нетерминала *N*. Каждая такая процедура осуществляет разбор фраз, выводимых из *N*. Процедуре сообщается, с какого места данной программы следует начинать поиск фразы, выводимой из *N*. Процедура ищет эту фразу, сравнивая программу в указанном месте с правыми частями правил для *N* и вызывая по мере необходимости другие процедуры для распознавателя промежуточных целей.

Для разбора предложения языка может потребоваться много рекурсивных вызовов процедур, составляющих нетерминалы в грамматике.

Преимущества написания рекурсивного нисходящего анализатора очевидны. Основное из них — это скорость написания анализатора на основе соответствующей грамматики. Другое преимущество заключается в соответствии между грамматикой и анализатором, благодаря чему увеличивается вероятность того, что анализатор окажется правильным, или по крайней мере, что ошибки будут простыми. Эффективность алгоритма определяется линейной зависимостью времени его работы от длины входной цепочки.

Недостатки метода, хотя и менее очевидны, но весьма реальны. Из-за большого числа вызовов процедур во время синтаксического анализа анализатор становится относительно медленным. Кроме того, он может быть довольно большим по сравнению с анализаторами, основанными на табличных методах

разбора (методы предшествования, *LL*- и *LR*-методы). Метод рекурсивного спуска способствует включению в анализатор действий по генерированию кода, что неизбежно приводит к смешению различных фаз компиляции. Последнее снижает надежность компиляторов или усложняет обращение с ними и как бы привносит в анализатор зависимость от машины.

Пример 3.18. Дана грамматика $G = (\{a, b, c, \perp\}, \{S, A, B\}, P, S)$, где

P: $S \to AB\perp$;

$A \to a \mid cA$;

$B \to bA$.

Требуется определить, принадлежит ли цепочка *caba* языку $L(G)$.

Построим вывод этой цепочки:

$S \Rightarrow AB\perp \Rightarrow cAB\perp \Rightarrow caB\perp \Rightarrow cabA\perp \Rightarrow caba\perp$.

На рис. 3.6 проиллюстрирован процесс построения нисходящего дерева разбора.

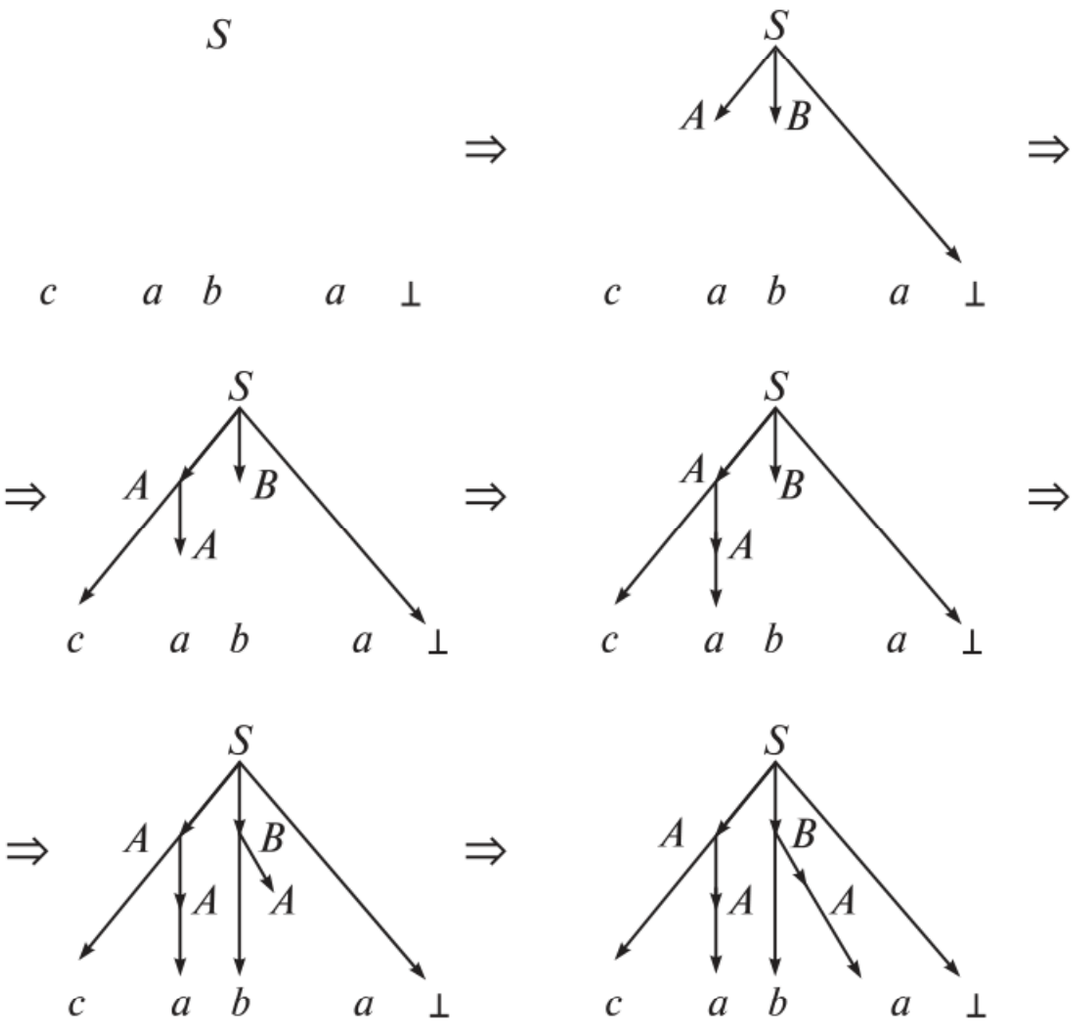


Рис. 3.6. Построение нисходящего дерева разбора

Ниже приведен набор процедур рекурсивного спуска для заданной грамматики. Функции *A*(), *B*(), *S*() соответствуют нетерминальным символам грамматики *A*, *B*, *S*.

```
#include <stdio.h>
int c;
FILE *fp;     /* указатель на файл, который содержит */
              /* исходную цепочку */
void A();
void B();
void ERROR(); /* функция обработки ошибок */
void S() {A(); B();
     if (c != '⊥') ERROR();
     }
void A() {if (c=='a') c = fgetc(fp);
     else if (c == 'c') {c = fgetc(fp); A();}
       else ERROR();
     }
void B() {if (c == 'b') {c = fgetc(fp); A();}
     else ERROR();
     }
void ERROR() {printf("ERROR !!!"); exit(1);}
main() {fp = fopen("data","r");
     c = fgetc(fp);
     S();
     printf("SUCCESS !!!"); exit(0);
    }
```

Описанный метод можно применять в том случае, если каждое правило грамматики имеет вид:

- $A \to \alpha$, где $\alpha \in (T \cup N)^*$ является единственным правилом вывода для этого нетерминала;
- $A \to a_1\alpha_1 \mid a_2\alpha_2 \mid \ldots \mid a_n\alpha_n$, где $a_i \in T$ для всех $i = 1, 2, \ldots, n$; $a_i \neq a_j$ для $i \neq j$; $\alpha_i \in (T \cup N)^*$, т. е. если для нетерминала A правил вывода несколько, то они должны начинаться с терминалов, причем все эти терминалы должны быть различными.

Если грамматика не удовлетворяет вышеприведенным условиям, то для реализации метода рекурсивного спуска необходимо преобразовать ее к требуемому виду, однако не существует алгоритма, который даст ответ на вопрос: можно ли построить

эквивалентную произвольной КС-грамматике грамматику, к которой применим метод рекурсивного спуска.

Известно также, что если для некоторых нетерминальных символов грамматики $\mathrm{FIRST}(A) \cap \mathrm{FOLLOW}(A) \neq \varnothing$, то метод рекурсивного спуска к данной грамматике не применим.

Метод рекурсивного спуска можно применять к достаточно узкому подклассу КС-грамматик. Известны более широкие подклассы КС-грамматик, для которых существуют эффективные анализаторы, обладающие тем же свойством, что и анализатор, написанный методом рекурсивного спуска, — входная цепочка считывается один раз слева направо и процесс разбора полностью детерминирован, т. е. время, затраченное на обработку, линейно зависит от длины цепочки. К таким грамматикам относятся $LL(k)$- и $LR(k)$-грамматики, грамматики предшествования и некоторые другие виды.

3.5.4. Нисходящий распознаватель без возвратов на основе LL(k)-грамматик

Класс LL-грамматик широк, но недостаточен для того, чтобы охватить все возможные синтаксические конструкции в языках программирования (к ним относятся все детерминированные КС-языки). Известно, что существуют детерминированные КС-языки, которые не могут быть заданы $LL(k)$-грамматикой ни для каких k. Однако LL-грамматики удобны для использования, поскольку позволяют построить распознаватели с линейными характеристиками (линейной зависимостью требуемых для работы алгоритма распознавания вычислительных ресурсов от длины входной цепочки символов) [1—5, 9—14].

Распознаватель $LL(k)$-грамматик представляет собой МП-автомат с одним состоянием s: $R = (\{s\}, T, H, \delta, s, h_0, \{s\})$, где $H = T \cup N$, а S — целевой символ грамматики G. Функция переходов автомата строится на основе управляющей таблицы M, которая отображает множество $(H \cup \{\lambda\}) \times T^*$ на множество, состоящее из следующих элементов:

- пар вида $[\beta, i]$, где β — цепочка символов, помещаемая автоматом на вершину стека, а i — номер правила вида $A \to \beta$, $A \in N$, $\beta \in H^*$;
- «выброса»;
- «допуска»;
- «ошибки».

Для построения распознавателя используются два важных множества, определяемые следующим образом:

- FIRST(k, α) — множество терминальных цепочек, выводимых из строк $\alpha \in (T \cup N)^*$, укороченных до k символов. Очевидно, что если имеется цепочка терминальных символов $\alpha \in T^*$, то FIRST(k, α) — это первые k символов цепочки α;
- FOLLOW(k, A) — множество укороченных до k символов терминальных цепочек, которые могут следовать непосредственно за нетерминалом $A \in N$ в цепочках вывода.

Конфигурацию распознавателя можно отобразить в виде конфигурации МП-автомата с дополнением цепочки ω, в которую помещаются номера примененных правил. Если считать, что X — это символ на вершине стека автомата, α — непрочитанная автоматом часть входной цепочки символов, а υ = FIRST(k, α), то работу алгоритма распознавателя можно представить следующим образом:

- $(\alpha, X\gamma, \omega) \vdash (\alpha, \beta\gamma, \omega i)$, $\gamma \in H^*$, если $M[X, \upsilon] = [\beta, i]$. Верхний символ магазина X заменяется цепочкой $\beta \in H$, к выходу добавляется i — номер правила. Считывающая головка не сдвигается;
- $(\alpha, X\gamma, \omega) \vdash (\sigma, \gamma, \omega)$, если $X = a \in T$ и $\alpha = a\sigma$, $M[a, \upsilon]$ = «выброс». Когда верхний символ магазина совпадает с текущим входным символом, он выбрасывается из магазина и считывающая головка сдвигается на один символ вправо;
- $(\lambda, \lambda, \omega)$ — завершение работы, при этом $M[\lambda, \lambda]$ — «допуск»;
- остальные случаи — «ошибка», разбор прекращается.

Цепочка υ = FIRST(k, α) при работе автомата называется «аванцепочкой».

Таким образом, для создания алгоритма распознавателя языка, заданного произвольной $LL(k)$-грамматикой, достаточно построить управляющую таблицу M на основе правил исходной грамматики. Алгоритм построения управляющей таблицы приведен в [3, 11].

Алгоритм разбора для LL(1)-грамматик

Алгоритм работы распознавателя для $LL(1)$-грамматик заключается всего в двух условиях, проверяемых на шаге выбора альтернативы. Исходными данными для этих условий являются символ $a \in T$, обозреваемый считывающей головкой МП-автомата (текущий символ входной цепочки), и символ $A \in N$, находящийся на вершине стека автомата [3, 11].

Эти условия можно сформулировать так:

- в качестве альтернативы выбирается правило $A \to x$, если символ $a \in \text{FIRST}(x)$;
- в качестве альтернативы выбирается правило $A \to \lambda$, если символ $a \in \text{FOLLOW}(A)$.

Если ни одно из этих условий не выполняется, это значит, что цепочка не принадлежит заданному языку и МП-автомат должен выдать сообщение об ошибке.

Пример 3.19. Рассмотрим грамматику $G = (\{a, b\}, \{S, T\}, P, S)$,

P: (1) $S \to aTS$; (2) $S \to b$;

(3) $T \to a$; (4) $T \to bST$.

Проанализируем цепочку *abbab*.

Таблица M для этой грамматики будет следующей (табл. 3.1).

Таблица 3.1. **Управляющая таблица разбора**

Символ	a	b	λ
S	aAS, 1	B, 2	«Ошибка»
A	a, 3	bSA, 4	«Ошибка»
a	«Выброс»	«Ошибка»	«Ошибка»
b	«Ошибка»	«Выброс»	«Ошибка»
h_0	«Ошибка»	«Ошибка»	«Допуск»

Работа автомата на основе данной таблицы будет выглядеть так:

$(abbab, Sh_0, \lambda) \vdash (abbab, aASh_0, 1) \vdash (bbab, ASh_0, 1) \vdash (bbab, bSASh_0, 1\ 4) \vdash (bab, SASh_0, 1\ 4) \vdash (bab, bASh_0, 1\ 4\ 2) \vdash (ab, ASh_0, 1\ 4\ 2) \vdash (ab, aSh_0, 1\ 4\ 2\ 3) \vdash (b, Sh_0, 1\ 4\ 2\ 3) \vdash (b, bh_0, 1\ 4\ 2\ 3\ 2) \vdash (\lambda, h_0, 1\ 4\ 2\ 3\ 2)$.

Цепочка разобрана, 1 4 2 3 2 — номера правил грамматики для разбора.

3.5.5. Восходящий распознаватель без возвратов на основе LR(k)-грамматик

Восходящий детерминированный распознаватель строится согласно алгоритму «сдвиг—свертка», рассмотренному в подразд. 3.5.1.

Необходимо модифицировать этот алгоритм таким образом, чтобы на каждом шаге его работы можно было однозначно ответить на вопросы [1—4, 9—13]:

- что следует выполнять: «сдвиг» (перенос) или «свертку»;
- какую цепочку символов α выбрать из стека для выполнения «свертки»;
- какое правило выбрать для выполнения «свертки» (в том случае, если существует несколько правил вида $A_1 \to \alpha$, $A_2 \to \alpha$, ..., $A_n \to \alpha$).

Для модификации алгоритма служат рассмотренные в подразд. 3.3.5 $LR(k)$-грамматики.

Распознаватель для $LR(k)$-грамматик функционирует под управлением некоторой таблицы T, которая состоит из столбцов — «действие» и «переход». В строках таблицы содержатся все цепочки символов, находящихся на вершине стека, которые влияют на работу распознавателя.

В столбце «действие» распределены все входные цепочки символов длиной не более k (аванцепочки), которые могут следовать за считывающей головкой автомата в процессе выполнения разбора; а в столбце «переход» — все терминальные и нетерминальные символы грамматики, которые могут появляться на вершине стека автомата при выполнении действий (сдвигов или сверток).

Ячейки таблицы «действие» содержат следующие элементы:

- «сдвиг» — если в данной ситуации требуется выполнить сдвиг (перенос текущего символа из входной цепочки в стек);
- «успех» — если возможна свертка к целевому символу грамматики S и разбор входной цепочки завершен;
- «свертка» (целое число) — если возможно выполнение свертки (число обозначает номер правила грамматики, по которому должна выполняться свертка);
- «ошибка» — в остальных ситуациях.

Ячейки «переход» управляющей таблицы T служат для выяснения номера строки таблицы, которая будет использована для определения выполняемого действия на очередном шаге. Эти клетки содержат следующие данные:

- целое число — номер строки таблицы T;
- «ошибка» — в остальных ситуациях.

Для удобства работы с распознавателем будем считать, что входная строка начинается с $\perp_{\text{Н}}$ — символа начала цепочки и заканчивается $\perp_{\text{К}}$ — символом конца цепочки.

В начальном состоянии символ $\perp_{\text{Н}}$ находится на вершине стека, а считывающая головка обозревает первый символ входной цепочки. В конечном состоянии в стеке должны находиться символы S (целевой символ грамматики) и $\perp_{\text{Н}}$, а считывающая головка автомата должна обозревать символ $\perp_{\text{К}}$.

Тогда алгоритм разбора будет следующим.

Шаг 1. Поместить в стек символ $\perp_{\text{Н}}$ и начальную (нулевую) строку управляющей таблицы T. В конец входной цепочки поместить символ $\perp_{\text{К}}$. Перейти к выполнению шага 2.

Шаг 2. Прочитать с вершины стека строку управляющей таблицы T. Выбрать из этой строки часть «действие» в соответствии с аванцепочкой, обозреваемой считывающей головкой автомата. Перейти к выполнению шага 3.

Шаг 3. В соответствии с типом действия выполнить выбор из четырех вариантов:

1) «сдвиг» — если входная цепочка не прочитана до конца, прочитать и запомнить как «*новый символ*» очередной символ из входной цепочки, сдвинуть считывающую головку на одну позицию вправо; в противном случае прервать выполнение алгоритма и сообщить об ошибке;

2) целое число («свертка») — выбрать правило в соответствии с номером, удалить из стека цепочку символов, составляющую правую часть выбранного правила, взять символ из левой части правила и запомнить его как «*новый символ*»;

3) «ошибка» — прервать выполнение алгоритма, сообщить об ошибке;

4) «успех» — произвести свертку к целевому символу S, прервать выполнение алгоритма, сообщить об успешном разборе входной цепочки символов, если входная цепочка прочитана до конца; в противном случае сообщить об ошибке.

Конец выбора. Перейти к выполнению шага 4.

Шаг 4. Прочитать с вершины стека строку управляющей таблицы T. Выбрать из этой строки часть «переход» в соответствии с символом, который был занесен в память как «*новый символ*» на шаге 3. Перейти к выполнению шага 5.

Шаг 5. Если часть «переход» содержит вариант «ошибка», прервать выполнение алгоритма и сообщить об ошибке, в противном случае (если там содержится номер строки управляющей таблицы T) — положить в стек «*новый символ*» и строку таблицы T с выбранным номером. Перейти к выполнению шага 2.

Здесь «*новый символ*» — промежуточная переменная для хранения терминалов и нетерминалов, полученных на некоторых шагах работы МП-автомата.

Распознаватель для LR(0)-грамматики

Для данного типа грамматик решение о сдвиге или свертке принимается только на основе содержимого стека. Текущий символ входной цепочки в расчет не принимается [2, 3, 11].

Рассмотрим работу распознавателя на примере.

Пример 3.20. Имеется грамматика $G = (\{a, b\}, \{S\}, P, S)$,

P: $S \to aSS \mid b$.

Пополненная грамматика для грамматики G будет иметь вид:

$G' = (\{a, b\}, \{S', S\}, P, S)$,

P: (1) $S' \to S$;

(2) $S \to aSS$;

(3) $S \to b$.

Управляющая таблица для данной грамматики — табл. 3.2.

Таблица 3.2. **Управляющая таблица *T***

Стек	Действие	Переход		
		S	*a*	*b*
$\perp_{\text{Н}}$	Сдвиг	1	2	3
S	Успех, 1			
a	Сдвиг	4	2	3
b	Свертка, 3			
aS	Сдвиг	5	2	3
aSS	Свертка, 2			

Разбор цепочки *abababb* приведен ниже.

Каждая строка разбора содержит конфигурацию МП-автомата в виде трех компонентов: непрочитанной части входной цепочки символов, содержимого стека МП-автомата, последовательности номеров примененных правил грамматики.

1. $(abababb\perp_{\text{К}}, \{\perp_{\text{Н}},0\}, \lambda)$;

2. $(bababb\perp_{\text{К}}, \{\perp_{\text{Н}},0\}\{a,2\}, \lambda)$;

3. $(ababb\perp_K, \{\perp_H,0\}\{a,2\}\{b,3\}, \lambda)$;

4. $(ababb\perp_K, \{\perp_H,0\}\{a,2\}\{S,4\}, 3)$;

5. $(babb\perp_K, \{\perp_H,0\}\{a,2\}\{S,4\}\{a,2\}, 3)$;

6. $(abb\perp_K, \{\perp_H,0\}\{a,2\}\{S,4\}\{a,2\}\{b,3\}, 3)$;

7. $(abb\perp_K, \{\perp_H,0\}\{a,2\}\{S,4\}\{a,2\}(S,4\}, 3,3)$;

8. $(bb\perp_K, \{\perp_H,0\}\{a,2\}\{S,4\}\{a,2\}\{S,4\}\{a,2\}, 3,3)$;

9. $(b\perp_K, \{\perp_H,0\}\{a,2\}\{S,4\}\{a,2\}\{S,4\}\{a,2\}\{b,3\}, 3,3)$;

10. $(b\perp_K, \{\perp_H,0\}\{a,2\}\{S,4\}\{a,2\}\{S,4\}\{a,2\}\{S,4\}, 3,3,3)$;

11. $(\perp_K\perp_K, \{\perp_H,0\}\{a,2\}\{S,4\}\{a,2\}\{S,4\}\{a,2\}\{S,4\}\{b,3\}, 3,3,3)$;

12. $(\perp_K, \{\perp_H,0\}\{a,2\}\{S,4\}\{a,2\}\{S,4\}\{a,2\}\{S,4\}\{S,5\}, 3,3,3,3)$;

13. $(\perp_K, \{\perp_H,0\}\{a,2\}\{S,4\}\{a,2\}\{S,4\}\{S,5\}, 3,3,3,3,2)$;

14. $(\perp_K, \{\perp_H,0\}\{a,2\}\{S,4\}\{S,5\}, 3,3,3,3,2,2)$;

15. $(\perp_K, \{\perp_H,0\}\{S,1\}, 3,3,3,3,2,2,2)$;

16. $(\perp_K, \{\perp_H,0\}\{S',*\}, 3,3,3,3,2,2,2,1)$.

Разбор завершен.

Вывод цепочки представлен следующим образом:

$S' \Rightarrow S \Rightarrow aSS \Rightarrow aSaSS \Rightarrow aSaSaSS \Rightarrow aSaSaSb \Rightarrow aSaSabb \Rightarrow$
$\Rightarrow aSababb \Rightarrow abababb.$

3.5.6. Метод предшествования

Для разбора цепочек, принадлежащих языку, порожденному рассмотренными в подразд. 3.3.6 грамматиками предшествования, используется алгоритм «сдвиг—свертка», адаптированный к данным грамматикам [2, 3].

Этот алгоритм выполняется расширенным МП-автоматом с одним состоянием s: $M = (\{s\}, T, H, \delta, s, h_0, \{s\})$. Отношения предшествования определяют в процессе работы автомата: какое действие — сдвиг или свертка — должно выполняться на каждом шаге алгоритма, и служат для того, чтобы выбрать цепочку для свертки. Правило при свертке выбирается однозначно за счет различия правых частей правил грамматики.

В начальной конфигурации автомата считывающая головка обозревает первый символ входной цепочки на входной ленте, в

магазине (стеке) МП-автомата находится символ $\perp_{\text{Н}}$, в конец цепочки помещен символ $\perp_{\text{К}}$.

Работу автомата можно описать следующим образом.

Шаг 1. Поместить на вершине стека символ $\perp_{\text{Н}}$. Считывающая головка должна находиться в начале входной цепочки символов.

Шаг 2. Сравнить с помощью отношения предшествования символ, находящийся на вершине стека (левый символ отношения), с текущим символом входной цепочки (правый символ отношения).

Шаг 3. Если имеет место отношение $<\bullet$ или $=\bullet$, то произвести сдвиг (перенос текущего символа из входной цепочки в стек и сдвиг считывающей головки на один шаг вправо) и выполнить шаг 2. В противном случае перейти к выполнению шага 4.

Шаг 4. Если имеет место отношение $\bullet>$, то произвести свертку, для этого надо найти на вершине стека все символы, связанные отношением $=\bullet$ («основу»), и удалить эти символы из стека. Затем выбрать из грамматики правило, имеющее правую часть, совпадающую с «основой», и поместить в стек левую часть выбранного правила. Если символов, связанных отношением $=\bullet$, на верхушке стека нет, то в качестве «основы» используется один, самый верхний символ стека. Затем, если разбор не закончен (считывающая головка не дошла до конца цепочки), вернуться к выполнению шага 2. Если правило, совпадающее с «основой», найти не удалось, то необходимо прервать выполнение алгоритма и сообщить об ошибке.

Шаг 5. Если не установлено ни одно отношение предшествования между текущим символом входной цепочки и символом на вершине стека, необходимо прервать выполнение алгоритма и сообщить об ошибке.

Разбор завершается, если считывающая головка автомата обозревает символ конца цепочки и при этом больше не может быть выполнена свертка. Решение о принадлежности цепочки языку зависит от содержимого магазина. Цепочка символов считается принадлежащей языку, если в результате завершения алгоритма в магазине находятся начальный символ грамматики S и символ $\perp_{\text{Н}}$.

Если на каком-либо этапе выполнения алгоритма возникает ошибка (невозможно перейти к очередному этапу), он прерывается, и цепочка считается не принадлежащей языку.

Для удобства отношения предшествования предварительно заносят в таблицу, строки и столбцы которой помечены терминалами и нетерминалами грамматики, включая символы начала и конца цепочки.

3.6. Пример простейшего синтаксического анализатора выражений

Синтаксический анализатор предназначен для распознавания синтаксической структуры строки, составленной из лексем (т. е. потока лексем в том порядке, в каком они поступают на вход синтаксического анализатора), т. е. если говорить о синтаксисе языка C++, синтаксический анализатор должен принять последовательность лексем `int a = 5;` и отказаться принимать такую последовательность тех же самых лексем `a = 5 int;`.

Приведем один из множества алгоритмов синтаксического анализа арифметических выражений, позволяющий вычислить их значение [12].

Вычисление значений выражений с помощью обратной польской нотации

Среди алгоритмов вычисления значений арифметических выражений в настоящее время наиболее популярен метод трансляции с помощью обратной польской записи, предложенный польским математиком Я. Лукашевичем. Рассмотрим данный метод на простом примере [15].

Пусть задано арифметическое выражение вида

$$(A + B) * (C + D) - E. \qquad (3.1)$$

Представим выражение (3.1) в виде дерева, в котором узлам соответствуют операции, а листьям — операнды. Построение начнем с корня, в качестве которого выбирается операция, выполняющаяся последней. Левой ветви соответствует левый операнд операции, а правой ветви — правый (рис. 3.7).

Совершим обход дерева, под которым будем понимать формирование строки символов из символов узлов и листьев дерева, от самой левой ветви вправо, и переписывать узел в выходную строку только после рассмотрения всех его ветвей. Результат обхода дерева имеет вид

$$AB + CD + * E -. \qquad (3.2)$$

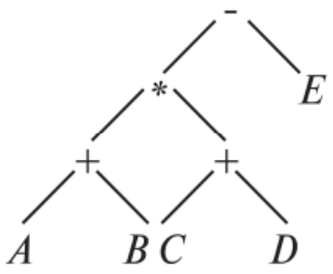


Рис. 3.7. Дерево арифметического выражения

Характерные особенности выражения (3.2) состоят в следовании символов операций за символами операндов и в отсутствии скобок. Такая запись называется *обратной польской записью.* Она обладает рядом замечательных свойств, которые превращают ее в идеальный промежуточный язык при трансляции.

Во-первых, вычисление выражения, записанного в обратной польской записи, может проводиться путем однократного просмотра, что весьма удобно при генерации объектного кода программ. Например, вычисление выражения (3.2) может быть произведено, как показано в табл. 3.3.

Таблица 3.3. **Вычисление арифметического выражения, представленного обратной польской записью**

№ п/п	Анализируемая строка	Действие
0	$A\ B+\ C\ D+*\ E-$	$r1 = A+B$
1	$r1\ C\ D+*\ E-$	$r2 = C+D$
2	$r1\ r2*\ E-$	$r1 = r1*r2$
3	$r1\ E-$	$r1 = r1-E$
4	$r1$	Вычисление окончено
Здесь $r1$, $r2$ — вспомогательные переменные.		

Во-вторых, получение обратной польской записи из исходного выражения может осуществляться на основе алгоритма, предложенного Дейкстрой. Для этого вводится понятие *стекового приоритета операций* (табл. 3.4).

Получение обратной польской записи происходит следующим образом.

Исходная строка символов просматривается слева направо, операнды записываются в выходную строку, а знаки операций заносятся в стек на основе следующих соображений:

а) если стек пуст, то операция из входной строки переписывается в стек;

Таблица 3.4. **Приоритеты операций**

Операция	Приоритет
(	1
)	1
+ | –	2
* | /	3
^	4

б) операция выталкивает из стека все операции с большим или равным приоритетом в выходную строку;

в) если очередным символом из исходной строки является открывающая скобка, то он помещается в стек;

г) закрывающая скобка выталкивает все операции из стека до ближайшей открывающей скобки, сами скобки в выходную строку не переписываются, а уничтожают друг друга.

Процесс получения обратной польской записи выражения (3.1) схематично представлен на рис. 3.8.

Просматриваемый символ	1	2	3	4	5	6	7	8	9	10	11	12	13	
Входная строка	(	*A*		*B*	)	*	(	*C*		*D*	)	–	*E*	
Состояние стека	(	(	+ (	+ (		*	(*	(*	+ (*	+ (*	*	–	–	
Выходная строка		*A*		*B*				*C*		*D*		*	*E*	–

Рис. 3.8. Процесс получения обратной польской записи

3.7. Пример разработки синтаксического анализатора заданных конструкций языка СИ++

Каждый язык программирования имеет правила, которые предписывают синтаксическую структуру корректных программ. В языке Pascal, например, программа создается из блоков, блок — из инструкций, инструкции — из выражений, выражения — из токенов и т. п. Синтаксис конструкций языка программирования может быть описан контекстно-свободной грамматикой с помощью нотации БНФ [3, 12].

3.7.1. Обобщенная модель синтаксического анализатора

Синтаксический анализатор получает строку токенов от лексического анализатора, как показано на рис. 3.9, и проверяет, может ли эта строка порождаться грамматикой исходного языка. Он также сообщает обо всех выявленных ошибках.

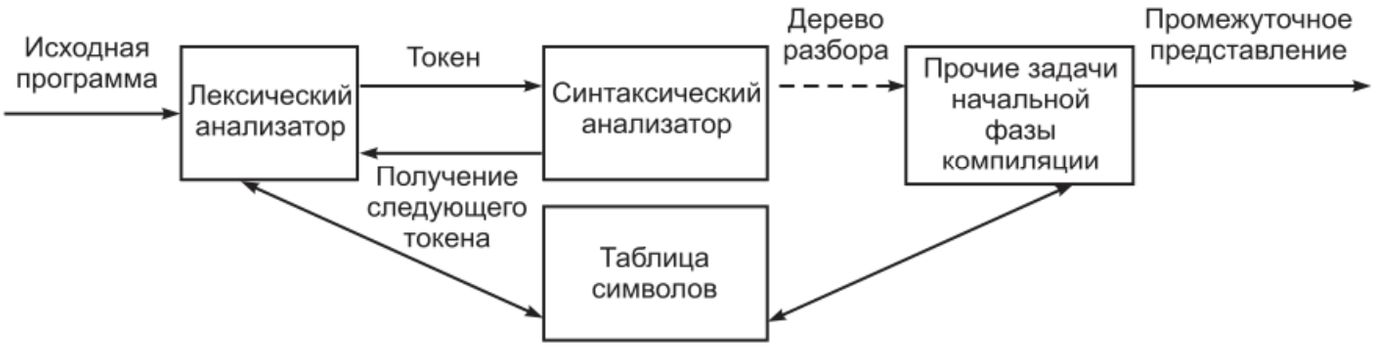


Рис. 3.9. Синтаксический анализатор в модели компилятора

Построение синтаксического анализатора основывается на распознавателях, виды которых подробно рассмотрены в подразд. 3.5.

Таким образом, выход синтаксического анализатора является некоторым представлением дерева разбора входного потока токенов, выданного лексическим анализатором. На практике имеется множество задач, которые могут сопровождать процесс разбора, например, сбор информации о различных токенах в таблице символов, выполнение проверки типов и других видов семантического анализа, а также создание промежуточного кода.

3.7.2. Обработка синтаксических ошибок

Любая программа потенциально содержит множество ошибок самого разного уровня. Например, ошибки бывают:

- лексическими, такими, как неверно записанные идентификаторы, ключевые слова или операторы;
- синтаксическими, например арифметические выражения с несбалансированными скобками;
- семантическими, такими, как операторы, применяемые к несовместимым с ними операндам;
- логическими, например бесконечная рекурсия.

Основные действия по выявлению ошибок и восстановлению состояния анализатора концентрируются в фазе синтаксического анализа. Одна из причин этого состоит в том, что многие ошибки по своей природе являются синтаксическими или проявляются, когда поток токенов, идущий от лексического анализатора, нарушает определяющие язык программирования грамматические правила.

Обработчик ошибок синтаксического анализатора имеет просто формулируемые цели:

- ясно и однозначно сообщать о наличии ошибок;
- обеспечивать быстрое восстановление после обнаружения ошибки, чтобы продолжить поиск следующих ошибок;
- существенно не замедлять обработку корректной программы.

Эффективная реализация этих целей представляет собой весьма сложную задачу.

К счастью, типичные ошибки достаточно просты, и для их обработки зачастую реализуются относительно простые механизмы обработки ошибок. В некоторых случаях, однако, ошибка может произойти задолго до момента обнаружения и определить ее природу весьма непросто. Бывают ситуации, когда обработчик ошибок должен догадаться, что имел в виду программист при написании программы.

Каким образом обработчик ошибок сообщит о наличии ошибки? Как минимум — указать место в исходной программе, где была выявлена ошибка; как правило, ошибка находится в пределах нескольких предыдущих токенов. Общая стратегия, используемая многими компиляторами, — вывод ошибочной строки с указанием позиции, в которой обнаружена ошибка. Если имеется обоснованное предположение о природе ошибки, вывод включает диагностическое пояснение типа «отсутствие точки с запятой в данной позиции».

В чем состоит восстановление синтаксического анализатора после того, как ошибка выявлена? В большинстве случаев окончание анализа после выявления первой ошибки не является корректным решением, так как восстановление и продолжение анализа могли бы выявить ряд последующих ошибок. Обычно после обнаружения ошибки синтаксический анализатор пытается восстановить некоторое свое состояние, в котором продолжается обработка входного потока так, как если бы он был корректным.

Неадекватное восстановление после обнаружения ошибки может вызвать настоящую лавину ложных ошибок, которые не

были допущены программистом, а появились вследствие изменений состояния синтаксического анализатора в процессе восстановления после ошибки. Аналогично восстановление после синтаксической ошибки может привести к ложным семантическим ошибкам, которые выявятся позже, на стадии семантического анализа или генерации кода. Например, при восстановлении после ошибки синтаксический анализатор может пропустить объявление некоторой переменной.

Консервативная стратегия компилятора состоит в запрете сообщений об ошибках, которые возникают во входном потоке слишком близко к друг другу. После обнаружения одной синтаксической ошибки компилятор должен проанализировать несколько входных токенов и только после этого разрешить вывод других сообщений об ошибках.

3.7.3. Стратегии восстановления состояния анализатора после ошибок

Существуют различные стратегии, которые синтаксический анализатор может использовать для восстановления состояния после синтаксической ошибки. Перечислим некоторые из них [3, 12].

Режим паники. Эта стратегия реализуется проще всего и может использоваться большинством методов синтаксического анализа. При обнаружении ошибки синтаксический анализатор поочередно пропускает входные символы, пока не будет найден один из специально определенного множества *синхронизирующих* токенов. Обычно такими токенами являются разделители, например *end* или *точка с запятой*, роль которых в исходной программе совершенно очевидна. Разработчик компилятора, естественно, должен выбрать синхронизирующие токены исходя из особенностей исходного языка. Несмотря на то, что при коррекции часто пропускается значительное количество символов без проверки на наличие ошибок, метод достаточно прост и в отличие от других гарантирует отсутствие зацикливания. В случаях, когда несколько ошибок в одной инструкции встречается очень редко, этот метод работает вполне адекватно.

Уровень фразы. При обнаружении ошибки синтаксический анализатор может выполнить локальную коррекцию оставшегося входного потока, т. е. заменить префикс остальной части потока

некоторой строкой, которая позволит синтаксическому анализатору продолжить работу. Типичная локальная коррекция может состоять в удалении лишней точки с запятой или вставке недостающей. Выбор способа локальной коррекции — прерогатива разработчика компилятора. Естественно, здесь нужно быть предельно осторожным, чтобы не полусить, например, бесконечный цикл.

Этот тип замещения может скорректировать любую входную строку и используется в ряде компиляторов, исправляющих ошибки. Основной его недостаток заключается в сложности обработки ситуации, когда ошибка реально располагается до точки обнаружения.

Продукции ошибок. Если известны наиболее распространенные ошибки, можно расширить грамматику языка продукциями, порождающими ошибочные конструкции. Затем на основе такой расширенной грамматики построить синтаксический анализатор. Если последний использует одну из «ошибочных» продукций, можно сгенерировать корректное диагностическое сообщение для распознанной ошибки.

3.7.4. Нерекурсивный предиктивный анализ

Нерекурсивный предиктивный (предсказывающий) анализ является одним из вариантов синтаксического анализа. Анализатор такого вида строится с помощью явного использования стека [9]. Ключевая проблема предиктивного анализа заключается в определении продукции, которую следует применить к нетерминалу. Нерекурсивный синтаксический анализатор, представленный на рис. 3.10, ищет необходимую для анализа продукцию в таблице разбора (далее будет показано, как строится эта таблица на основе грамматики).

Предиктивный синтаксический анализатор под управлением таблицы разбора имеет входной буфер, стек, таблицу разбора и выходной поток. Входной буфер содержит анализируемую строку с символом ее правого конца — \$; в стеке находится последовательность символов грамматики с символом \$ на дне. В начальный момент стек содержит стартовый символ грамматики непосредственно над символом \$. Таблица разбора представляет собой матрицу $\mathbf{M}_{i,j}$, строки которой помечены $A_i \in N$ — нетерминалами, а столбцы $a_j \in T \cup \$$ — терминалами грамматики или символом \$. Записью таблицы является продукция грамматики

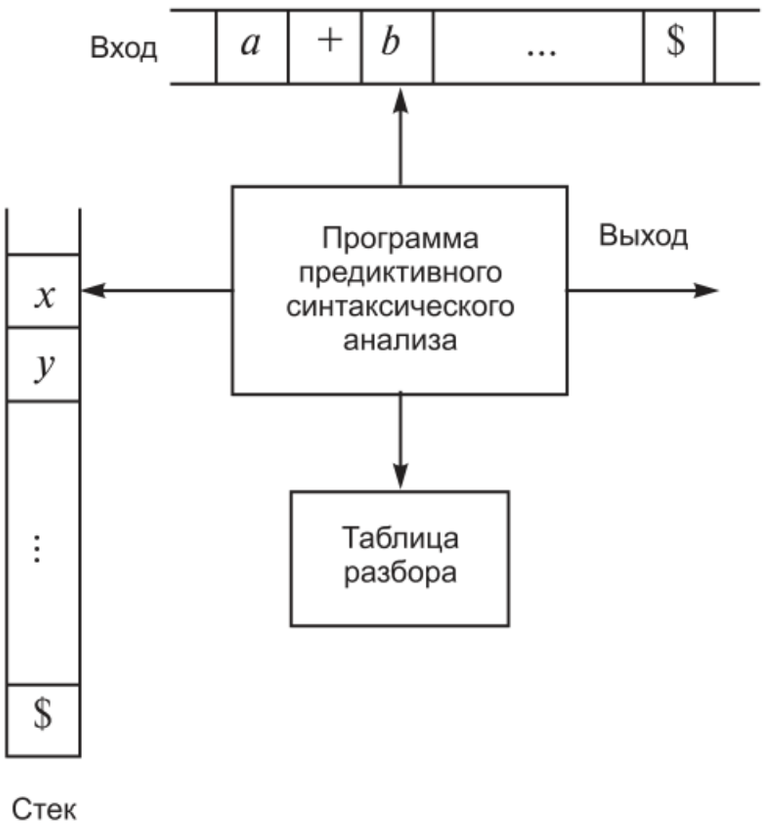


Рис. 3.10. Модель нерекурсивного предиктивного синтаксического анализатора (специальный символом $ — конец строки и дно стека)

$A_i \to \alpha \in P$ или сообщения об ошибке. Обозначим ячейки таблицы $\mathbf{M}[A_i, a_j]$.

Программа синтаксического анализа работает следующим образом. Рассматривается X — символ на вершине стека и a — текущий входной символ, которые определяют действия синтаксического анализатора.

Возможны следующие варианты.

1. Если символ $X = a = \$$, то синтаксический анализатор прекращает работу и сообщает об успешном завершении разбора.

2. Если символ $X = a \neq \$$, то синтаксический анализатор выталкивает X из стека и перемещает указатель входного потока к следующему символу.

3. Если символ X представляет собой нетерминал $X \in N$, то программа рассматривает запись $\mathbf{M}[X, a]$ из таблицы разбора $\mathbf{M}$:

- при $\mathbf{M}[X, a] = \{X \to Y\beta\}$, где $Y, \beta \in (T \cup N)^*$, синтаксический анализатор замещает символ X на вершине стека на βY (Y будет располагаться на вершине стека). Полагаем, что в качестве выхода синтаксический анализатор просто выводит использованную продукцию, на самом деле здесь может выполняться любой необходимый код;
- при $\mathbf{M}[X, a] =$ `error`, синтаксический анализатор вызывает программу восстановления после ошибки.

Предположим, что задана грамматика *G*, для нее имеется таблица разбора **M** и на вход поступает цепочка символов ω.

Алгоритм нерекурсивного предиктивного анализа будет выглядеть следующим образом:

```
Установить указатель ip на первый символ ω$;
repeate
  обозначим через X символ на вершине стека,
  а через a — символ, на который указывает ip.
if X — терминал или $ then
   if X = a then
      снять со стека символ X и переместить ip
   else error()
else /*X — нетерминал*/
   if M[X, a] = X → Y1Y2...Yk then begin
     снять X со стека;
     Поместить в стек Y1, Y2, ..., Yk с Y1 на вершине
     стека;
     Вывести продукцию X → Y1Y2...Yk
   end
   else error()
until X = $ /* Стек пуст */
```

3.7.5. Построение таблиц предиктивного анализа

Заполнение таблицы разбора производится с помощью множеств FIRST и FOLLOW (см. подразд. 3.3.1).

Приведем алгоритм построения таблицы. Предположим, что грамматика $A \to \alpha$ представляет собой продукцию, у которой символ $a \in \text{FIRST}(\alpha)$. Тогда синтаксический анализатор заменит грамматику A строкой α при текущем входном символе a. В случае $\alpha = \lambda$ или $\alpha \Rightarrow \lambda$ необходимо снова заменить A на α, если текущий входной символ имеется в множестве FOLLOW(A) или из входного потока получен символ \$, который входит в множество FOLLOW(A).

Рассмотрим теперь метод построения таблицы разбора.

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

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

3. Если в множество FIRST(α) входит λ, для каждого терминала *b* из множества FOLLOW(*A*) добавляем $A \to \alpha$ в ячейку **M**[*A*, *b*]. Если λ входит в FIRST(α), символ \$ — в множество FOLLOW(*A*), добавляем $A \to \alpha$ в ячейку **M**[*A*, \$].

4. В остальные ячейки добавляем сообщение об ошибке «error».

Данный алгоритм может быть применен к любой грамматике *G* для построения таблицы разбора *M*. Однако, если грамматика *G* неоднозначная или леворекурсивная, в одной ячейке таблицы может оказаться несколько записей. Устранив левую рекурсию и неоднозначность (это не всегда возможно), можно построить эквивалентную *G*-грамматику, для которой применим метод нерекурсивного предиктивного анализа.

Проиллюстрируем вышесказанное примером:

Пример 3.21. Рассмотрим грамматику $G = (T, N, P, S)$ со следующими продукциями:

P: $E \to TA$;

$A \to +TA \mid \$$;

$T \to FB$;

$B \to *FB \mid \$$;

$F \to (E) \mid w$.

Для построения таблицы разбора данной грамматики необходимо определить множества FIRST и FOLLOW.

FIRST(E) = FIRST(T) = FIRST(F) = {(, w};

FIRST(A) = {+, λ};

FIRST(B) = {*, λ};

FOLLOW(E) = FOLLOW(A) = {), \$};

FOLLOW(T) = FOLLOW(B) = {+,), \$};

FOLLOW(F) = {+, *,), \$}.

Таблица 3.5 содержит матрицу предиктивного анализа этой грамматики. Пустые ячейки означают ошибки; непустые указывают продукции, с помощью которых заменяются нетерминалы на вершине стека.

При входном потоке $w+w*w$ предиктивный синтаксический анализатор осуществляет последовательность перемещений, показанную в табл. 3.6. Входной указатель при перемещении указывает на крайний слева символ в столбце «Вход».

Таблица 3.5. **Таблица предиктивного разбора**

Нетерминал	Входной символ					
	w	+	*	(	)	$
E	$E \to TA$	Ошибка	Ошибка	$E \to TA$	Ошибка	Ошибка
A	Ошибка	$A \to +TA$	Ошибка	Ошибка	$A \to \lambda$	$A \to \lambda$
T	$T \to FB$	Ошибка	Ошибка	$T \to FB$	Ошибка	Ошибка
B	Ошибка	$B \to \lambda$	$B \to *FB$	Ошибка	$B \to \lambda$	$B \to \lambda$
F	$F \to w$	Ошибка	Ошибка	$F \to (E)$	Ошибка	Ошибка

Таблица 3.6. **Перемещения предиктивного синтаксического анализатора**

Стек	Вход	Выход
$E	$w+w*w$	
$AT	$w+w*w$	$E \to TA$
$ABF	$w+w*w$	$T \to FB$
$ABw, w+w*w$	$F \to w$	
$AB	$+w*w$	
$A	$+w*w$	$B \to \lambda$
$AT+	$+w*w$	$A \to +TA$
$AT	$w*w$	
$ABF	$w*w$	$T \to FB$
$ABw, w*w$	$F \to w$	
$AB	$*w$	
$ABF*	$*w$	$B \to *FB$
$ABF	w	
ABw, w	$F \to w$	
$AB	$	
$A	$	$B \to \lambda$
$	$	$A \to \lambda$

3.7.6. Восстановление после ошибок в предиктивном анализе

Синтаксический анализатор находит ошибку в тот момент, когда терминал на вершине стека не соответствует очередному входному символу или на вершине стека находится нетерминал A, очередной входной символ a, а ячейка таблицы синтаксического анализа $\mathbf{M}[A, a]$ пуста.

Восстановление после ошибки в «режиме паники» заключается в том, что символы из входного потока пропускаются до тех пор, пока не будет обнаружен токен из выбранного множества синхронизирующих токенов. Эффективность этого метода зависит от выбора синхронизирующего множества, которое должно быть таким, чтобы синтаксический анализатор быстро восстанавливался после часто встречающихся ошибок. Приведем некоторые эвристические правила.

- В качестве первого приближения можно поместить в синхронизирующее множество для нетерминала A все символы из множества FOLLOW(A).
- Множества FOLLOW(A) недостаточно. Например, если инструкция завершается точкой с запятой, то ключевое слово, с которого начинается инструкция, может не оказаться во множестве FOLLOW нетерминалов, порождающих выражения. Отсутствие точки с запятой после присваивания может, таким образом, привести к пропуску ключевого слова, с которого начинается новая инструкция. Зачастую в языке имеется иерархическая структура конструкций — например, выражения появляются в инструкциях, которые находятся в блоках, и т. д. В таком случае к синхронизирующему множеству конструкции нижнего уровня можно добавить символы, с которых начинаются конструкции более высокого уровня, например, добавить ключевые слова в самое начало инструкций в синхронизирующих множествах нетерминалов, порождающих выражения.
- Если добавить символы из множества FIRST(A) в синхронизирующее множество для нетерминала A, то станет возможным продолжать анализ в соответствии с A, когда во входном потоке появится символ из множества FIRST(A).
- Если нетерминал может порождать пустую строку, то в качестве продукции по умолчанию используется продукция, порождающая пустую строку λ. Это может отсрочить обна-

ружение ошибки, зато произойдет ее пропуск и сократится число нетерминалов, которые следует рассмотреть в процессе восстановления после ошибки.

- Если терминал на вершине стека невозможно сопоставить с входным символом, то данный терминал удаляется из стека, генерируется сообщение о вставке терминала в программу и синтаксический анализ продолжается. По сути, при этом подходе синхронизирующее множество токена состоит из всех остальных токенов.

Пример 3.22. Рассмотрим грамматику G из примера 3.10.

Таблица 3.7 синтаксического анализа для этой грамматики приведена ниже. «Synch» указывает синхронизирующие токены, полученные из множества FOLLOW соответствующего терминала.

Таблица 3.7. **Таблица предиктивного разбора с синхронизирующими токенами**

Нетерминал	Входной символ					
	w	++	*	(	)	$
E	$E \to TA$	—	—	$E \to TA$	Synch	Synch
A	—	$A \to +TA$	—		$A \to \lambda$	$A \to \lambda$
T	$T \to FB$	Synch	—	$T \to FB$	Synch	Synch
B	—	$B \to \lambda$	$B \to *FB$	—	$B \to \lambda$	$B \to \lambda$
F	$F \to w$	Synch	Synch	$F \to (E)$	Synch	Synch

Метод восстановления в «режиме паники» на основе данной таблицы работает следующим образом. Если синтаксический анализатор просматривает ячейку $M[A, a]$ и обнаруживает, что она пуста, то входной символ a пропускается. Если в этой ячейке находится запись Synch, то нетерминал снимается с вершины стека в попытке продолжить синтаксический анализ. Если токен на вершине стека не соответствует входному символу, то он удаляется из стека.

В случае ошибочного ввода строки $)w*+w$ синтаксический анализатор и механизм восстановления после ошибок поведут себя, как показано в табл. 3.8.

В общем случае компилятор должен не только обнаруживать ошибки, восстанавливаться после них и продолжать разбор, но и выдавать информационные сообщения об обнаруженных ошиб-

Таблица 3.8. **Перемещения предиктивного синтаксического анализатора при разборе ошибочной цепочки**

Стек	Вход	Примечание
$\$E$	$)w^*+w\$$	Ошибка, пропускаем символ «)»
$\$E$	$w^*+w\$$	$w \in \mathrm{FIRST}(E)$
$\$AT$	$w^*+w\$$	—
$\$ABF$	$w^*+w\$$	—
$\$ABw$	$w^*+w\$$	—
$\$AB$	$^*+w\$$	—
$\$ABF^*$	$^*+w\$$	
$\$ABF$	$+w\$$	Ошибка, $M[F, +]$ = **Synch**
$\$AB$	$+w\$$	F снимается со стека
$\$A$	$+w\$$	—
$\$AT+$	$+w\$$	—
$\$AT$	$w\$$	—
$\$ABF$	$w\$$	—
$\$ABw$	$w\$$	—
$\$AB$	$\$$	—
$\$A$	$\$$	—
$\$$	$\$$	—

ках. Разработчик компилятора должен обеспечить эту возможность.

Восстановление на уровне фразы. Реализуется заполнением пустых ячеек в таблице предиктивного синтаксического анализа указателями на подпрограммы обработки ошибок. Эти подпрограммы могут изменять, вставлять или удалять символы входного потока и выводить соответствующие сообщения об ошибках. Кроме того, они могут удалять элементы из стека. При таком способе восстановления возникает вопрос, можно ли разрешить изменение символов в стеке или разрешить только размещение новых символов, поскольку после этого синтаксический анализатор не обязательно приведет к порождению хотя бы одного слова языка. В любом случае необходимо предусмотреть отсутствие зацикливания программы. Надежная защита от зацикliва-

ния — проверка того, что каждое действие по восстановлению после ошибки в конечном счете приводит к обработке очередного входного символа (или к удалению элементов из стека, если достигнут конец входного потока).

Контрольные вопросы

1. Укажите место синтаксического анализатора в процессе трансляции.
2. Какие грамматики называются контекстно-свободными?
3. Что такое недостижимые и бесплодные символы грамматики? Как их удалить?
4. Дайте определение приведенных грамматик.
5. Как избавить грамматику от цепных и аннулирующих правил?
6. Перечислите особенности контекстно-свободных грамматик на примерах.
7. Как привести грамматику к нормальной форме Хомского?
8. Что такое грамматика в нормальной форме Грейбах?
9. Приведите примеры *S*- и *Q*-грамматик.
10. Дайте определение *LL*(*k*)-грамматики. Приведите примеры.
11. Для чего предназначены *LR*(*k*)-грамматики? Приведите примеры.
12. Опишите грамматики предшествования.
13. Как устроены и функционируют автоматы с магазинной памятью?
14. Приведите пример разбора цепочки символов магазинным автоматом.
15. Как построить МП-автомат по заданной контекстно-свободной грамматике?
16. Охарактеризуйте разницу между недетерминированными и детерминированными магазинными автоматами.
17. Какие виды распознавателей КС-грамматик вы знаете?
18. Приведите примеры распознавателей с возвратами.
19. Какие существуют табличные распознаватели? В чем заключаются их преимущества и недостатки?
20. Как работает метод рекурсивного спуска? В чем его недостаток?
21. Охарактеризуйте нисходящие детерминированные распознаватели.
22. Какой вид грамматик лежит в основе восходящих детерминированных распознавателей? Как работают эти распознаватели?
23. В чем заключаются преимущества и недостатки распознавателей на основе грамматик предшествования?

Глава 4
СЕМАНТИЧЕСКИЙ АНАЛИЗ, ПРОМЕЖУТОЧНОЕ ПРЕДСТАВЛЕНИЕ ПРОГРАММЫ И ГЕНЕРАЦИЯ КОДА

Семантический анализ — наименее формализуемая часть процесса трансляции, поскольку он определяет смысл конструкций языка, которые анализируются на этапах лексического и синтаксического анализа. В процессе трансляции в памяти строится определенная структура данных, содержащая все смысловые (семантические) взаимоотношения между объектами программы. Эта структура получила название *семантических таблиц* (на самом деле структура может быть любой) [1, 3, 12].

4.1. Семантический анализ

Семантика программы — внутренняя модель системы именованных объектов, с которыми работает программа, с описанием их свойств и характеристик [1]. ***Семантический анализ*** — это проверка смысловой правильности языковой конструкции [1].

4.1.1. Место семантического анализатора в общем процессе трансляции

Обычно объектами программы являются ***типы***, ***переменные*** и ***функции*** (***процедуры***), которые обозначаются именами (идентификаторами). Для них уже на этапе лексического анализа может быть составлена таблица, дополняемая на каждом шаге трансляции данными о семантике объектов. Тогда общая схема взаимодействия фаз транслятора имеет вид, приведенный на рис. 4.1.

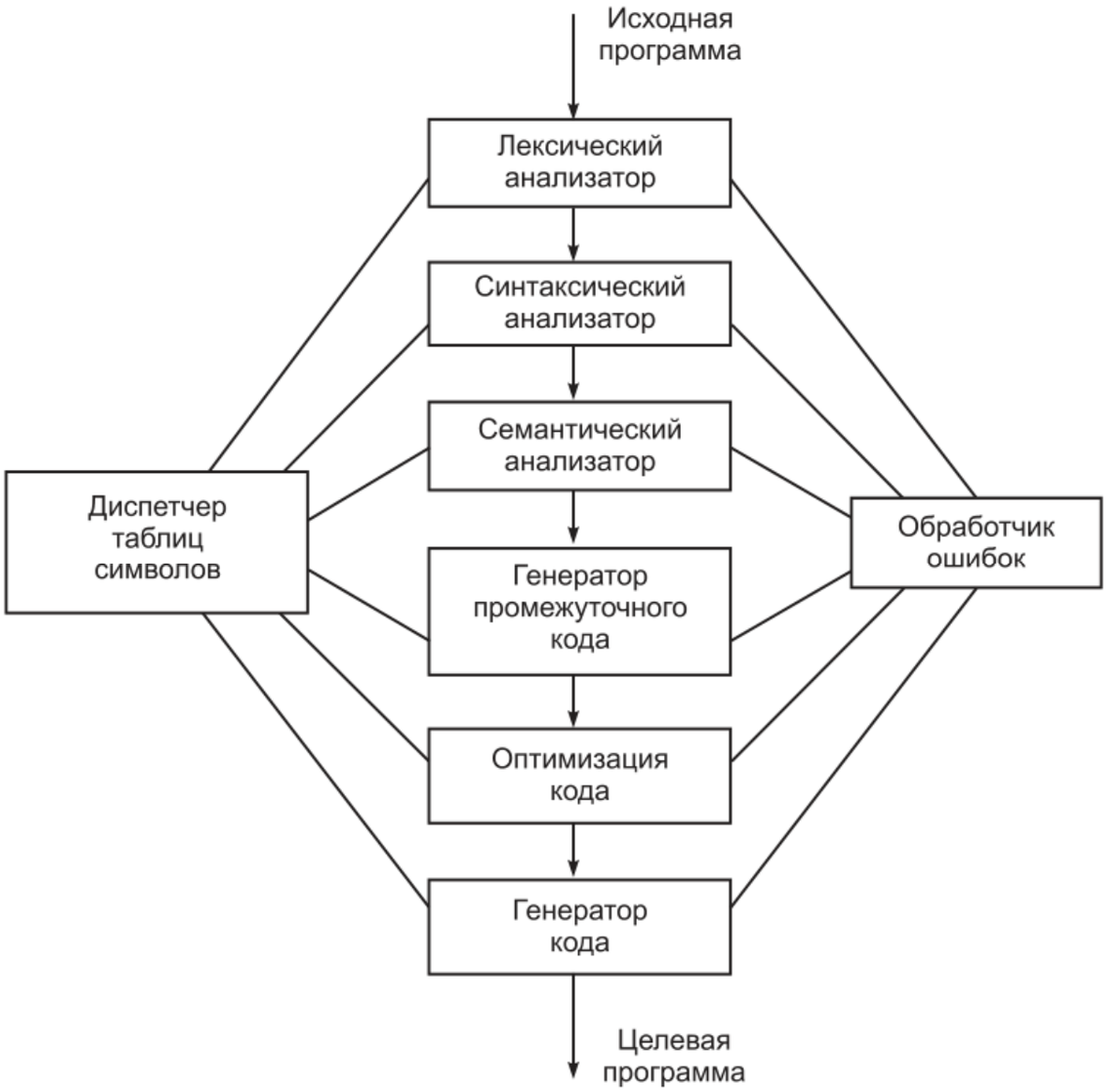


Рис. 4.1. Обобщенная схема компилятора

1. Как правило, в процессе трансляции создается несколько семантических таблиц по видам используемых в языке объектов с различным содержанием. Например, в Си-компиляторе имеет смысл заводить таблицы типов, локальных и глобальных переменных, функций. Они заполняются семантическими процедурами при разборе соответствующих правил. В частности, при разборе определения переменной ее имя должно заноситься в таблицу локальных или глобальных имен.

2. Лексический анализатор определяет значение очередной лексемы (разобранную цепочку символов), которое помещается в семантические таблицы.

3. Синтаксический анализатор при разборе правил, в которых в качестве терминалов фигурируют имена, вызывает семантическую процедуру, соответствующую каждому из этих правил. Эта процедура получает ссылки на имена из соответствующих лексем.

4. Семантическая процедура по ссылкам на имена определяет, является ли операция, заданная правилом, допустимой для семантики этих имен. Кроме того, семантическая процедура может создать новые записи в таблицах, изменить их содержание, провести поиск и т. д. путем возвращения ссылок на соответствующие элементы семантических таблиц.

5. Элементы семантических таблиц в свою очередь могут быть связаны с некоторыми динамическими структурами данных, которые отражают внутреннюю семантику хранимых объектов — списков, массивов, деревьев. В целом образуется некоторая семантическая сеть, по структуре которой семантические процедуры и производят анализ тех или иных частей программы.

6. Фаза генерации кода получает на вход некоторое промежуточное представление программы, оптимизирует его и превращает в объектный код [12].

4.1.2. Семантические таблицы

Семантическая таблица для переменной должна отражать ее основные свойства в языке и включать следующее содержание:

- имя переменной;
- указатель на описание типа в таблице типов;
- смещение (адрес), который получает эта переменная при трансляции в том сегменте данных, где она размещается компилятором;
- указатель на область памяти, где размещаются ее значение, — для интерпретатора.

Семантический анализ переменных при таком подходе характеризуется рядом особенностей:

- для правил определений и объявлений переменных семантическая процедура параллельно строит семантическую сеть и заполняет таблицу типов, в описание переменной в таблице переменных включается указатель на ее тип;
- для правил построения выражений, включающих заданную переменную, семантические процедуры параллельно проверяют соответствие текущей операции текущему типу данных в семантической сети. Результат операции также получает указатель на элемент семантической сети, таким образом он связывается со своим типом данных для следующей операции.

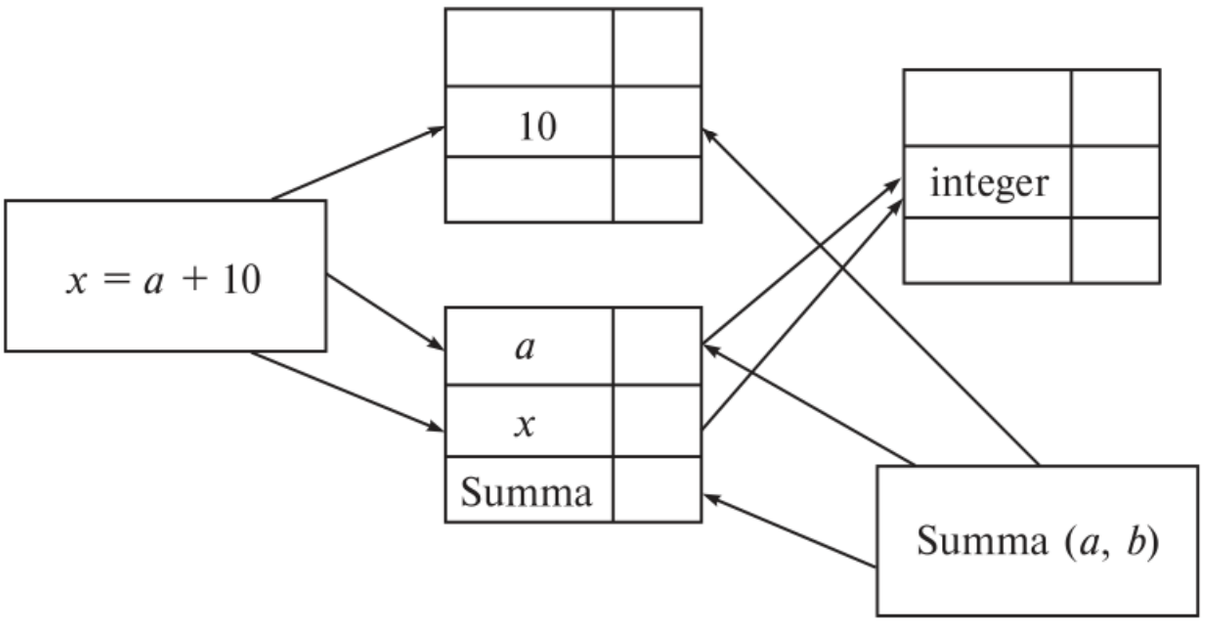


Рис. 4.2. Схема организации семантических таблиц

Схема организации семантических таблиц в упрощенном виде представлена на рис. 4.2.

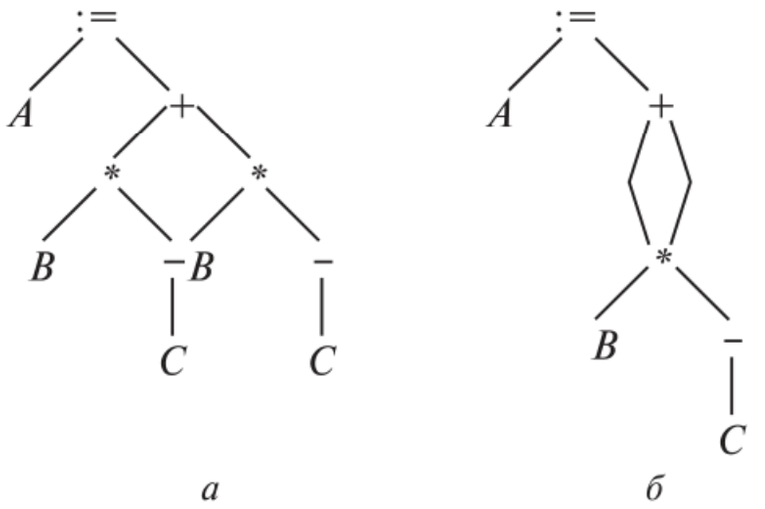
4.2. Промежуточное представление программы

Прежде чем перейти к заключительной фазе — генерации кода, — транслятор, как правило, создает некое промежуточное представление программы, которое может принимать следующие формы [9, 12]:

- ориентированный граф или абстрактное дерево;
- тетрады операций;
- триады операций;
- линеаризованное представление;
- собственно команды языка Ассемблера.

4.2.1. Формы промежуточного представления

Простейшей формой промежуточного представления является *синтаксическое дерево разбора* (рис. 4.3, *а*). Более полную информацию о входной программе дает *ориентированный ациклический граф* (рис. 4.3, *б*), в котором в одну объединены вершины синтаксического дерева, представляющие одни и те же константы, переменные и общие подвыражения. На рис. 4.3 рассмотрены описанные выше формы промежуточного представления выражения $A := B* - C + B* - C$:



а *б*

Рис. 4.3. Промежуточное представление выражения в виде: *а* — дерева разбора; *б* — ориентированного графа

Тетрады представляют собой запись операций из четырех составляющих:

<операция>(<операнд1>,<операнд2>,<результат>).

Триады представляют собой запись операций из трех составляющих:

<операция>(<операнд1>,<операнд2>),

при этом один или оба операнда могут быть ссылками на другую триаду. Триады при записи последовательно номеруют для удобства указания ссылок одних триад на другие. Например, выражение $A := B * C + D - B * 10$, записанное в виде триад, будет иметь вид:

1. * (B, C).
2. + (^1, D).
3. * (B, 10).
4. – (^2, ^3).
5. := (A, ^4).

Знак ^ означает ссылку операнда одной триады на результат другой.

Линеаризованное представление позволяет достаточно легко хранить промежуточное представление в памяти и обрабатывать его в порядке чтения. Наиболее распространенные виды линеаризованного представления:

- *постфиксная*, или *обратная польская запись*, которая представляет собой запись дерева в порядке его обхода снизу вверх;

- *префиксная*, или *прямая польская запись*, которая представляет собой запись дерева в порядке обхода его сверху вниз.

Преимуществом обратной польской записи является то, что все операции записываются непосредственно в порядке их выполнения. Она чрезвычайно эффективна в тех случаях, когда для вычислений используется стек (подробно рассмотрена в подразд. 3.5).

Дерево на рис. 4.3, *а* в постфиксной записи может быть представлено следующим образом:

*a b c – * b c * + :=.*

В префиксной записи сначала указывается операция, а затем ее операнды. Например, для приведенного выше выражения имеем:

*:= a + * b – c * b - c.*

Команды Ассемблера удобны тем, что при их использовании внутреннее представление программы полностью соответствует объектному коду и сложные преобразования не требуются. Однако использование команд Ассемблера требует дополнительных структур для отображения их взаимосвязи.

4.2.2. Уровень промежуточного представления

Промежуточное представление программы может в различной степени соответствовать либо исходной программе, либо машинному представлению. Например, если промежуточное представление содержит адреса переменных, то его следует обработать перед переносом на другую машину. С другой стороны, промежуточное представление может содержать раздел описаний программы, и тогда информацию об адресах извлекают из обработки описаний. Ясно, что первое представление более эффективно, чем второе [3, 12].

Операторы управления в промежуточном представлении могут быть представлены в исходном виде (в виде операторов языка `if`, `for`, `while` и т. д.), а могут содержаться в виде переходов. В первом случае некоторая информация извлекается из самой структуры (например, для оператора `for` — информация о переменной цикла, которую, возможно, следует хранить на регистре, для оператора `case` — информация о таблице меток и т. д.). Во втором случае можно представить информацию проще и более унифицированной.

Некоторые формы промежуточного представления больше подходят для различного рода оптимизаций (в частности, ориентированные графы), другие, например, префиксная запись, не годятся для этой цели.

4.3. Генерация кода

Генерация объектного кода — это перевод компилятором внутреннего представления исходной программы в результирующую объектную программу на языке Ассемблер или непосредственно на машинном языке.

Генерация объектного кода выполняется после синтаксического анализа программы и всех необходимых действий по подготовке к генерации кода: распределения адресного пространства под функции и переменные, проверки соответствия имен и типов переменных, констант и функций в синтаксических конструкциях исходной программы и т. д.

Выходом генератора кода является целевая программа, которую можно представить в виде: абсолютного машинного языка, перемещаемого машинного языка, или языка Ассемблер.

Преимуществом генерации абсолютной *программы на машинном языке* является то, что такой код помещается в фиксированное место в памяти и немедленно выполняется; небольшие программы при этом быстро компилируются и выполняются.

Генерация *перемещаемой программы на машинном языке* (объектного модуля) обеспечивает возможность раздельной компиляции подпрограмм.

Получение программы на *языке Ассемблер* позволяет создавать символьные инструкции и использовать возможности макросов ассемблера. Недостаток заключается в дополнительном шаге обработки ассемблерной программы после генерации кода [11, 12].

4.4. Проектирование компилятора

Два компилятора, реализующие один и тот же язык на одной и той же машине, отличаются друг от друга, если в процессе разработки ставились различные цели. К таким целям относятся:

1) получение эффективного объектного кода;

2) разработка и получение объектных программ;

3) минимизация времени компиляции;

4) разработка компилятора минимального размера;

5) создание компилятора, обладающего широкими возможностями обнаружения и исправления ошибок;

6) обеспечение надежности компилятора.

Перечисленные цели в некоторой степени противоречивы. Компилятор, который должен выдавать эффективный код, будет медленней. Для осуществления некоторых стандартных методов оптимизации, таких, как устранение избыточности кода, исключение последовательностей кода из циклов и т. д., может потребоваться много времени. Компилятор, предназначенный для сложных программ оптимизации, также велик по объему. Размер компилятора влияет на время компиляции программы. В процессе работы часто невозможно оптимизировать время и объем памяти одновременно. Необходимо заранее решить, что следует оптимизировать с учетом того, к чему это может привести. Кроме того, минимизация времени на написание компилятора сама по себе является препятствием к включению некоторых сложных методов оптимизации.

Обычно от компилятора ожидают вспомогательного сообщения об ошибке и, кроме того, продолжения анализа программы после обнаружения ошибки. Однако компиляторы существенно отличаются по своим возможностям обнаружения и исправления ошибок в силу того, что проблема исправления ошибок (особенно синтаксических) довольно сложна и не может быть полностью решена из-за принципиальной неоднозначности процедуры исправления. Важно помнить, что исправление ошибок осуществляется исключительно для того, чтобы компилятор мог продолжить анализ программы, а не останавливаться на первой же найденной ошибке. Компилятор, выдающий полезные сообщения и элегантно выходящий из ситуации ошибки, имеет, как правило, больший размер, что вполне компенсируется повышением эффективности работы программы.

Обеспечение надежности должно быть первостепенной задачей при создании любого компилятора. Компиляторы часто представляют собой очень большие программы, а современное состояние науки и техники в этой области не позволяет дать формальное подтверждение правильности компилятора. Наибольшее значение здесь имеет хороший общий проект. Если различные фазы процесса сделать относительно различимыми и каждую фазу построить как можно проще, то вероятность создания надежного компилятора возрастает. Надежность компилятора

повышается и в том случае, если он базируется на ясном и однозначном формальном определении языка и если используются такие автоматические средства, как генератор синтаксических анализаторов.

Цели проектирования компилятора часто зависят от той среды, в которой он должен использоваться. Если он предназначается для обучения студентов, эффективность кода будет иметь меньшее значение, чем скорость компиляции и возможность обнаружения ошибок. В производственной среде — наоборот.

При построении компиляторов большую роль играет решение о числе проходов, которые ему придется выполнить. Создание многопроходного компилятора связано с проектированием промежуточных языков для версий исходного текста, существующих между проходами. Строятся также таблицы какого-либо прохода, которые могут понадобиться в дальнейшем.

Контрольные вопросы

1. Что такое семантика программы и семантический анализ?
2. Что представляют собой семантические таблицы?
3. Как связана семантическая процедура с другими фазами процесса трансляции?
4. Какие формы промежуточного представления программы вы знаете?
5. Приведите пример преобразования арифметического выражения в префиксную (постфиксную) запись.
6. Чем отличается представление в виде ориентированного ациклического графа от синтаксического дерева?
7. Какие существуют виды линеаризованного представления?
8. Для чего нужна генерация кода?
9. Какую информацию принимает на вход генератор кода, что он выдает на выходе?
10. Приведите обобщенную структуру транслятора.
11. В чем различие компиляторов и интерпретаторов?

Глава 5
ПРИМЕРЫ РАЗРАБОТКИ ТРАНСЛЯТОРОВ

В данной главе приведены теоретические основы различного вида трансляторов для условных компьютерных языков с целью реализации их в процессе лабораторных работ с использованием информации, полученной из глав 1—4.

5.1. Разработка интерпретатора

Наличие интерпретаторов имеет большое значение в силу нескольких причин. Во-первых, они обеспечивают удобную интерактивную среду (что зачастую удобнее, чем просто компилятор языка), во-вторых, интерпретаторы языка обеспечивают превосходные интерактивные отладочные возможности. Интерпретатор позволяет, например, динамично устанавливать значения переменных и условий. В-третьих, большинство языков-запросов к базам данных работают в режиме интерпретации.

5.1.1. Общие сведения

Интерпретатор отличается от компилятора тем, что фаза генерации кода обычно заменяется фазой эмуляции элементов промежуточного представления или объектной модели языка [1, 2]. Кроме того, в интерпретаторе, как правило, не проводится оптимизация промежуточного представления, а сразу же осуществляется его эмуляция. Обобщенная структура интерпретатора приведена на рис. 5.1.

При разработке интерпретатора следует учесть тот факт, что каждая лексема имеет два формата: внешний и внутренний. Внешний формат — это строка символов, с помощью которой пишется программа на каком-либо языке программирования (например, «`while`» — это внешняя форма ключевого слова

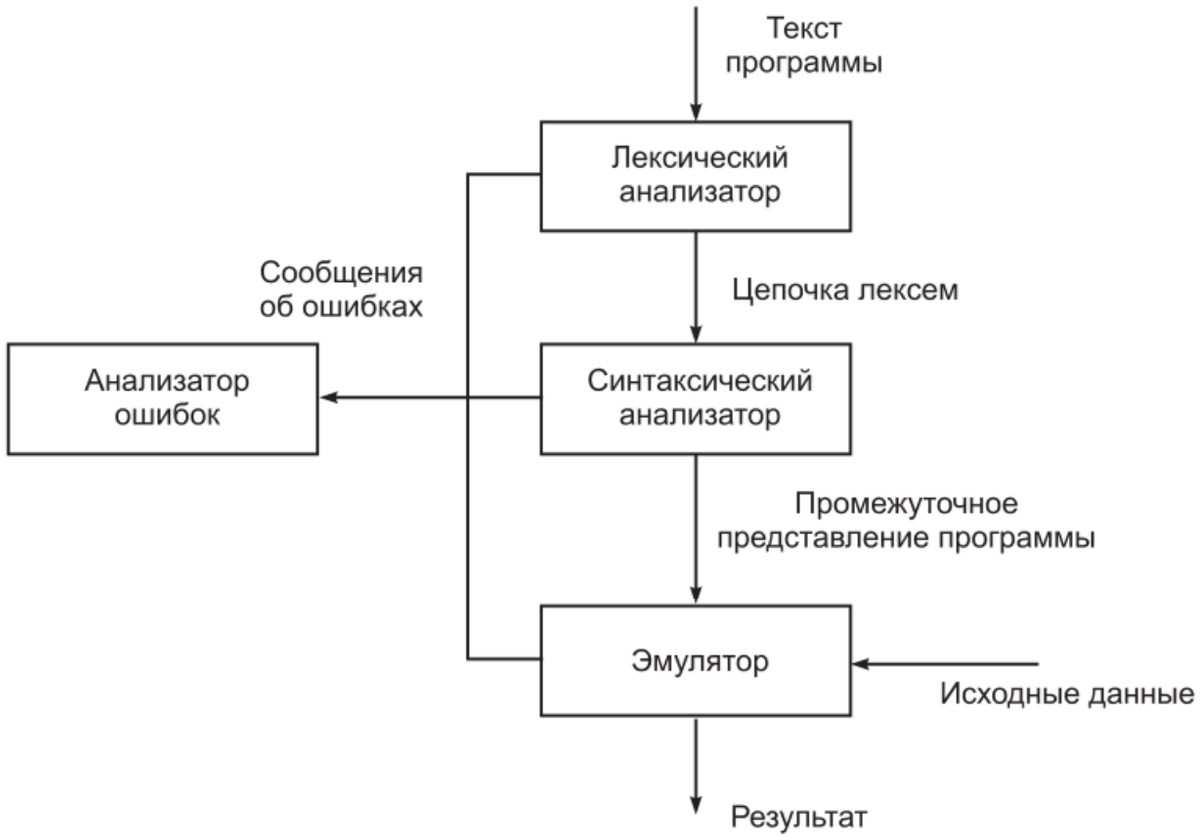


Рис. 5.1. Обобщенная структура интерпретатора

`while` языка C). Создание интерпретатора из расчета, что каждая лексема используется во внешнем формате, достаточно непрофессионально. При серьезном подходе к программированию следует ориентироваться на внутренний формат лексемы, который является просто числом, и разрабатывать интерпретатор исходя из профессиональной точки зрения на проблему. В этом случае каждой лексеме присваивается внутренний номер. Например, ключевое слово `while` может иметь порядковый номер 1, `if` — 2 и т. д. Преимущество внутреннего формата заключается в том, что программы, обрабатывающие числа, быстрее программ, обрабатывающих строки. Для реализации такого подхода необходима функция, которая считывает из входного потока данных очередную лексему и преобразует ее из внешнего формата во внутренний.

Очень важно знать, какой тип лексемы возвращен. Например, для анализатора выражений существенно, является ли следующая лексема числом, оператором или переменной. Значение типа лексемы для анализа в целом станет очевидным непосредственно при разработке интерпретатора.

Некоторые функции интерпретатора нуждаются в повторном просмотре лексемы, т. е. лексема обязательно должна быть возвращена во входной поток, в частности, это необходимо, когда

очередная лексема является началом выражения. Тогда она помещается обратно во входной поток и вызывается функция обработки и вычисления этого выражения [16].

5.1.2. Спецификация языка CBAS

Рассмотрим метод разработки интерпретатора для условного языка CBAS. Данный язык является комбинацией языков программирования C и BASIC и значительно упрощен по сравнению с реальными языками.

Формальная грамматика, порождающая этот язык, приведена ниже.

```
<character>: 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' |
 'h' | 'I' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' |
 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' |
 'z' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' |
 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' |
 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' |
 '_'

<id_end>:
       | <character><id_end>
<identifier>: <character><id_end>
<digit>: '0' | '1' | '2' | '3' | '4' | '5' | '6' |
         '7' | '8' | '9'
<number_end>:
       | <digit><number_end>
<number>: <digit><number_end>
<expression>: <term> '+' <expression>
       | <term> '-' <expression>
       | <term>
<term>: <factor> '*' <term>
       | <factor> '/' <term>
       | <factor>
<factor>: <identifier> | <number> | (<expression>)
<relop>: '<' | '>' | '= =' | '!='
<bool_expression>: <expression> <relop> <expression>
<assign>: <identifier> '=' <expression>
```

```
<string>: '"' <любое количество отличных от '"'
                    символов> '"'
<print>: 'print' <print_end> ';'
<print_end>: <string>
       | <expression>
       | <string> ',' <print_end>
       | <expression> ',' <print_end>
<scan>: 'scan' <identifier> ';'
<for>: 'for' <identifier> '=' <expression> 'to'
          <expression>
<if>: 'if' <bool_expression>
<else>:
       | 'else' '{' <statement> '}'
<statement>:
       | <print> <statement>
       | <scan> <statement>
       | <assign> <statement>
       | <for> '{' <statement> '}'
       | <if> '{ <statement> '}' <else>
<program>: <statement>
```

Все переменные состоят только из букв и символа подчеркивания и являются глобальными. Такая конструкция, как объявление переменных, в языке CBAS не предусмотрена. Все переменные имеют тип данных, аналогичный `int`. Транслятор при транслировании исходной программы строит таблицу переменных. Таблица представляет собой пары «имя—значение». При обнаружении идентификатора транслятор просматривает таблицу в поисках такого же имени. Если его там нет, то создается новое вхождение в таблицу, и переменной присваивается какое-то начальное значение (например, `0`).

Оператор `print` служит для вывода информации на экран. Он может печатать как символьные строки, так и значения выражений.

Оператор `scan` служит для считывания с клавиатуры значений в переменную. Выполнение программы приостанавливается, пока пользователь не введет какое-либо число (так как все переменные имеют тип данных `int`).

Цикл `for` работает следующим образом. Переменной в цикле присваивается начальное значение, которое инкрементируется на единицу при каждой итерации. Выход из цикла происходит, когда переменная цикла БОЛЬШЕ ИЛИ РАВНА значению верхней границы цикла.

5.1.3. Некоторые особенности построения интерпретаторов

Как уже упоминалось, интерпретатор оперирует внутренним представлением для лексем. С целью преобразования во внутренний формат используется таблица соответствия ЛЕКСЕМА ⇔ ⇔ ЧИСЛО, которая описывается следующей структурой:

```
struct commands {/* Вспомогательная структура ключевых
                    слов анализатора */
    std::string command;
    int token;
} table[];
```

Основной цикл работы интерпретатора состоит в пошаговом выполнении всех инструкций исходной программы. Все интерпретаторы выполняют операции путем считывания лексемы программы и выбора необходимой функции для ее выполнения. Например, если очередная лексема — это ключевое слово '`if`', то запускается функция `process_if()`. В качестве параметра функция принимает указатель на обрабатываемую часть программы. Это может быть номер строки и символа в файле, какая-либо другая информация. Так как язык CBAS подразумевает вложенность конструкций, то каждая функция обработки той или иной лексемы вместе с выполнением специфических для данной лексемы функций должна вызывать общую функцию обработки.

Пусть, например, программа состоит исключительно из циклов `for` и условий `if`. Тогда интерпретатор условно можно представить в виде следующих функций:

```
void process_if(char**);
void process_for(char**);

void main_loop(char* programm){
  int token = get_next_lexem(&program);
  switch (token){
    case IF:
      process_if(&program);
      break;
    case FOR:
      process_for(&program);
      break;
```

```
        default:
            //ошибка
    }
    }

    void process_if(char** pointer){
        //специфичные действия для if
        main_loop(*pointer);
    }
    void process_for(char** pointer){
        //специфичные действия для for
        main_loop(*pointer);
    }
```

5.1.4. Циклы

В языке CBAS, так же, как и в языке C, допускается вложенность цикла `for`. Особенностью реализации этого оператора является сохранение информации о каждом вложенном цикле со ссылкой на внешний цикл, для чего используется работа стековой структуры: начало цикла, информация о значении управляющей переменной цикла и ее конечном значении, а также место расположения цикла в теле программы заносятся в стек.

Каждый раз, при достижении знака '`}`', соответствующего концу цикла `for`, из стека извлекается информация о значении управляющей переменной, затем ее значение пересчитывается и сравнивается с конечным значением цикла. Если значение управляющей переменной цикла достигло своего конечного значения, выполнение цикла прекращается и выполняется оператор программы, следующий за циклом. В противном случае, в стек заносится новая информация, и выполнение цикла начинается с его начала. Таким же образом обеспечивается интерпретация и выполнение вложенных циклов. В стекоподобной структуре вложенных циклов каждый `for` должен быть закрыт соответствующей '`}`'.

Для реализации оператора цикла `for` стек должен иметь следующую структуру:

```
struct for_stack {
   int var; /* счетчик цикла */
   int target; /* конечное значение */
```

```
    char* location;
} fstack[FOR_COUNT]; /* стек для цикла for */
int ftos; /* индекс начала стека FOR */
```

Константа `FOR_COUNT` ограничивает уровень вложенности цикла. Переменная `ftos` всегда имеет значение индекса начала стека.

Вместо явного использования стековой структуры можно воспользоваться механизмом локальных переменных языка, на котором реализуется интерпретатор. В этом случае вложенные операторы обрабатываются с помощью рекурсивного вызова соответствующих процедур, и необходимая информация сохраняется в локальных переменных вызывающей процедуры.

5.1.5. Порядок построения выражений

Известно множество вариантов анализа и вычисления выражений. Для использования полного синтаксического анализатора рекурсивного спуска следует представить выражение в виде рекурсивной структуры данных. Это означает, что выражение определяется в термах самого себя. Если выражение определяют с использованием только символов «+», «–», «*», «/» и скобок, то все выражения могут быть определены с использованием следующих правил:

Выражение = > Терм [+Терм][-Терм];
Терм = > Фактор [*Фактор][/Фактор];
Фактор = > Переменная, Число или (Выражение).

Очевидно, некоторые части в выражении могут отсутствовать вообще. Квадратные скобки означают именно такие необязательные элементы выражения. Символ '=>' имеет смысл «продуцирует».

Фактически выше перечислены правила, которые обычно называют правилами вывода выражения. В соответствии с ними дадим определение терма: «Терм является произведением или отношением факторов».

Приоритет операторов безусловен в описанных выражениях, т. е. вложенные элементы включают операторы с более высоким приоритетом.

Рассмотрим теперь ряд примеров.

Пример 5.1. Выражение

```
10+5*B
```

содержит два терма: «10» и «5*B». Они, в свою очередь, состоят из трех факторов: «10», «5» и «B», содержащих два числа и одну переменную.

Пример 5.2. Выражение

```
14*(7-C)
```

содержит два фактора «14» и «(7-C)», которые, в свою очередь, состоят из числа и выражения в скобках. Выражение в скобках вычисляется как разность числа и переменной.

Можно преобразовать правила вывода выражений в множество общих рекурсивных функций, что и является зачастую основной формой синтаксического анализатора рекурсивного спуска. На каждом шаге анализатор такого типа выполняет специфические операции в соответствии с установленными алгебраическими правилами. Работу этого процесса рассмотрим на примере анализа выражения и выполнения арифметических операций.

Пример 5.3. Пусть на вход анализатора поступает следующее выражение:

```
9/3-(100+56)
```

Тогда анализатор работает по нижеследующей схеме.

1. Берем первый терм: «9/3».
2. Берем каждый фактор и выполняем деление чисел, получаем результат «3».
3. Берем второй терм: «(100+56)». В этой точке стартует рекурсивный анализ второго выражения.
4. Берем каждый фактор и суммируем их между собой, получаем результат 156.
5. Берем число, вернувшееся из рекурсии, и вычитаем его из первого: 3-156. Получаем итоговый результат «-153».

Следует учитывать особенности рекурсивного разбора выражений.

1. Приоритет операторов является безусловным в продукционных правилах и определен в них.

2. Этот метод синтаксического анализа и вычисления выражений подобен тому, который используется для выполнения таких же операций.

Результатом разработки является программа-интерпретатор, которая считывает каждую строчку входного текста (программы на языке CBAS) до конца строки, распознает ошибочные конструкции, выдает сообщения об ошибках и исполняет содержащийся в строке код, если ошибок не обнаружено.

5.2. Разработка процессора языка разметки документа

Язык разметки в компьютерной терминологии — набор символов или последовательностей, вставляемых в текст для передачи информации о его выводе или строении. Языки разметки используются везде, где требуется вывод форматированного текста: в типографии (SGML, TeX, PS, PDF), пользовательских интерфейсах компьютеров (troff, Microsoft Word, OpenOffice), всемирной сети Интернет (HTML, XML) [17, 18].

Например, TeX — система компьютерной верстки, разработанная американским почетным профессором Дональдом Кнутом в целях создания компьютерной типографии. В нее входят средства для секционирования документов, для работы с перекрестными ссылками и для набора сложных математических формул. Документы набираются на собственном языке разметки в виде обычных ASCII-файлов. Эти файлы транслируются специальной программой в файлы, «независимые от устройства», которые могут быть отображены на экране или напечатаны. Ядро TeX представляет собой язык низкоуровневой разметки, содержащий команды отступа и смены шрифта.

Каждый документ имеет три составляющие — содержание (смысловое наполнение), структуру и внешнее представление. Структура документа позволяет правильно определить составляющие его части и взаимоотношения между ними. Внешнее представление направлено на повышение эффективности восприятия информации читателем, что достигается за счет выделения смысловых частей документа теми или иными средствами, доступными для данной формы представления.

5.2.1. Некоторые способы разметки

В документе, помимо смыслового наполнения, должна содержаться некоторая метаинформация, позволяющая определить его структуру и внешнее представление. Такая метаинформация называется *разметкой документа*.

Разметка документа преследует следующие две основные цели:

- выделение смысловых частей (логических элементов) документа и связей между ними;
- указание действий, которые необходимо осуществлять с этими элементами.

Для достижения первой цели предназначена *структурная разметка*. Действия, направленные на получение внешнего представления, задаются *разметкой представления*.

В качестве примеров приведены два возможных способа разметки.

Пример 5.4. Структурная разметка:

```
<div1 type="Section">
   <head>Введение</head>
   <p>Так уж сложилось, что большую часть информации
   человек предпочитает хранить в виде документов.</p>
</div1>
```

Пример 5.5. Разметка представления:

```
<font face="Arial Bold" size=16>1. Введение<hspace
size=20>
<tab size=5><font face="Times New Roman" size=12>
Так уж сложилось, что большую часть информации человек
предпочитает хранить в виде документов
```

В первом случае описан раздел, который имеет заголовок и текст в виде абзаца, т. е. определяет структуру документа. Структурная разметка говорит о том, как текст устроен, т. е. из каких частей состоит, и как эти части друг с другом соотносятся.

Во втором случае показано, каким образом данный текст должен быть отображен на бумаге или на мониторе — выделить шрифтом Arial Bold размера 16, отступить по вертикали 20, сделать табуляцию 5, выделить шрифтом Times New Roman размера 12. Здесь приведена разметка представления документа, которая говорит о том, что делать с текстом, как его отображать.

Исторически разметка представления появилась раньше, и в течение длительного времени была ориентирована исключительно на внешний (бумажный) вид документа. Но в последнее время быстрый рост числа документов, их создание, хранение и использование в электронном виде, автоматизированная обработка и обмен документами предъявили новые требования к разметке, в числе которых — независимость от среды представления, возможность осуществления эффективного поиска, возможность повторного использования как документа целиком, так и отдельных его элементов.

Быстрый рост количества документов привел к тому, что поиск нужной информации занимает теперь все больше и больше времени. Так, простой контекстный поиск в Интернете информации об авторе статей даст огромное количество ссылок на те места, где встречается фамилия автора. После этого придется либо просмотреть все полученные ссылки, либо задавать дополнительную информацию для сужения области поиска. Если бы мы могли сразу указать, что фамилию следует искать только среди авторов журнальных статей технического плана, это во много раз упростило бы поиск, но документы, среди которых ведется поиск, необходимо было бы предварительно разметить с явным выделением элементов «автор», «тематика» и т. п.

Возможность повторного использования документа или отдельных его частей приводит к тому, что в настоящее время при работе используются шаблоны с изменением лишь некоторой существенной для данного случая информации. Но делается это преимущественно вручную. Если говорить об автоматизированном формировании, связывании, повторном использовании документов, то перечисленные операции становятся возможными только тогда, когда документы как информационные объекты являются структурированными, а используемая метаинформация полно и ясно описывает характеристики каждого элемента документа.

Все вышеназванные задачи можно решить, используя исключительно структурный подход при разметке документов. Именно структурная разметка позволяет выделять смысловые элементы, определять их связи с другими элементами как в рамках одного документа, так и вне этих рамок.

Характерной особенностью многих языков разметки является блочная структура всего документа, т. е. документ представляет собой блок, который состоит из блоков, которые в свою очередь состоят из блоков и т. д. (рис. 5.2).

Рис. 5.2. Пример блочной структуры документа

Разметка текста осуществляется с помощью специальных управляющих кодов, присутствующих в документе. Одним из вариантов таких кодов являются ***теги*** (***tag***) — ключевые слова, описывающие структуру разметки. Теги могут иметь *атрибуты*, которые задают разметку представления. Например:

```
<center color=red background=black> Текст красного цвета
на черном фоне в центре страницы </center>
```

Теги `<center>...</center>` образуют логические скобки, которые задают вывод заключенного между ними текста в центре. Тег имеет атрибуты `color` и `background`, которые задают цвет текста и фона соответственно.

Вложенность блоков. Как уже упоминалось, языки разметки имеют блочную структуру. Предположим, что есть пара тегов `<block>...</block`, задающих блок, и теги `<row>...</row>` и `<column>...</column>`, задающие строку и столбец в этих блоках. Тогда структура имеет вид, представленный на рис. 5.3.

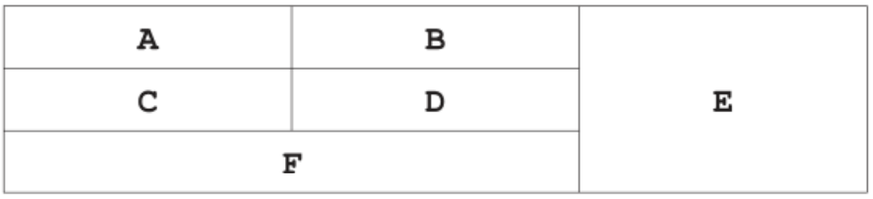


Рис. 5.3. Пример вложенных блоков

Вышеприведенная блочная структура документа описывается следующим кодом:

```
<block>
  <column>
    <row>
      <block>
        <row>
<column> A </column>
```

```
<column> B </column>
        </row>
<column> C </column>
<column> D </column>
        <row>
        </row>
      </block>
    </row>

    <row> F </row>
  </column>

  <column> E </column>
</block>
```

5.2.2. Спецификация языка разметки eMark

Данный язык разметки предназначен для оформления вывода текста на стандартную консоль, которая имеет 24 строки по 80 символов каждая. Соответственно страница выводимого текста может содержать максимум 24 строки и 80 столбцов. Перечислим поддерживаемые теги и их атрибуты:

1) `<block rows =` *число* `columns =` *число*`> </block>` — атрибуты `rows` и `columns` задают число строк и столбцов в текущем блоке;

2) `<column valign=`*вертикальное_выравнивание*
`halign =` *горизонтальное_выравнивание* `textcolor =` *цвет*
`bgcolor =` *цвет* `width =` *число*`> </column>`;

вертикальное_выравнивание — одно из значений `top`, `center`, `bottom` (выравнивание по верхнему краю, по центру и по нижнему краю соответственно);

горизонтальное_выравнивание — одно из значений `left`, `center`, `right` (выравнивание по левому краю, по центру и по правому краю соответственно);

цвет — цифра от `0` до `15` включительно, соответствующая одному из цветов, в которые может быть раскрашена консоль;

число — ширина столбца в символах.

3. `<row valign =` *вертикальное_выравнивание* `halign =` *горизонтальное_выравнивание* `textcolor =` *цвет* `bgcolor =` *цвет* `height =` *число*`>` `</row>` — смысл атрибутов аналогичен одноименным атрибутам тега `<column>`. Атрибут `height` задает высоту строки как количество строчек консоли.

Дополнительные условия и ограничения:

1. Документ должен состоять как минимум из одного тега `<block>`.

2. Атрибуты `rows` и `columns` тега `<block>` обязательны и не могут быть больше высоты (ширины) элемента, в который вложен данный тег.

3. Количество вложенных тегов `<column>` и `<row>` для каждого тега `<block>` должно соответствовать значению атрибутов `columns` и `rows`.

4. Значения по умолчанию для тегов:

- `valign = top`;
- `haligh = left`;
- `textcolor = 15` (белый);
- `bgcolor = 0` (черный);
- `width` (только для последнего тега `<column>` в теге `<block>`, для остальных должно быть указано явно) равно значению ширины блока за вычетом суммы ширин всех предыдущих столбцов в блоке;
- `height` (только для последнего тега `<row>` в теге `<block>`, для остальных должно быть указано явно) равно значению высоты блока за вычетом суммы высот всех предыдущих строк в блоке;
- `columns = 1` и `rows = 0` только для внешнего тега `<block>`, для остальных должно быть указано явно.

5. Каждому открывающему тегу должен соответствовать закрывающий тег.

Пример 5.6. Описание простейшего документа на языке разметки eMark:

```
<block>
   <column>»Hellow world» example using eMark</column>
</block>
```

5.3. Разработка ядра скриптового языка

Скриптовый язык (от *англ.* scripting language), или язык сценариев, — язык программирования, разработанный для записи «сценариев», последовательностей операций, которые пользователь может выполнять на компьютере. Простые скриптовые язы-

ки раньше часто называли *языками пакетной обработки* (batch languages или job control languages). Сценарии всегда интерпретируются, а не компилируются [6, 19].

5.3.1. Понятие сценария

В прикладной программе сценарий (*скрипт*) — это программа, автоматизирующая некоторую задачу, которую без сценария пользователь делал бы вручную, используя интерфейс программы.

Поскольку сценарии интерпретируются из исходного кода динамически при каждом исполнении, они выполняются обычно значительно медленнее готовых программ, оттранслированных в машинный код на этапе компиляции, поэтому сценарные языки не применяются для написания программ, требующих оптимальности и быстроты исполнения. Тем не менее из-за простоты они часто используются для написания небольших, одноразовых («проблемных») программ. В плане быстродействия скриптовые языки подразделяют на языки *динамического разбора* (sh, command.com) и *предварительно компилируемые* (Perl). Языки динамического разбора считывают инструкции из файла программы минимально требующимися блоками и исполняют эти блоки, не читая дальнейший код. Предкомпилируемые языки вначале считывают всю программу, компилируют ее либо в машинный код, либо в какой-то внутренний формат, и лишь затем исполняют получившийся код.

Сценарии представляют собой смесь достаточно низкоуровневых конструкций языков программирования (например, циклы, условия, переменные) и функций («указаний», ключевых слов скриптового языка), которые в большой степени приближены к естественным языкам. Для примера рассмотрим программную реализацию на некотором абстрактном скриптовом языке алгоритма угощения бутербродами группы людей.

Пример 5.7. Сценарий кормления группы людей:

```
foreach персона from группа {
 if ЧеловекГолоден(персона){
    бутерброд = ПриготовитьБутерброд(new КусокХлеба(),
     new КусокКолбасы());
    ОтдатьБутерброд(персона, бутерброд);
  }
}
```

Скрипты могут вызывать из своего кода другие скрипты. Например, если создать скрипт «НакормитьЧеловека» с единственным параметром *персона*, то вышеприведенный пример сводится к двум скриптам.

Пример 5.8. Вложенные скрипты.

```
//скрипт «НакормитьЧеловека»
НакормитьЧеловека(персона){
  бутерброд = ПриготовитьБутерброд(new КусокХлеба(),
   new КусокКолбасы());
  ОтдатьБутерброд(персона, бутерброд);
}
//скрипт «НакормитьГруппу»
НакормитьГруппу(){
  foreach персона from группа {
    if ЧеловекГолоден(персона){
     НакормитьЧеловека(персона);
    }
  }
}
```

Кроме того, существует разновидность скриптовых языков (сценариев), которые используются в компьютерных играх. Их особенность в отличие от скриптов, автоматизирующих некоторую последовательность действий, состоит в том, чтобы предусмотреть при программной реализации (но необязательно) секции инициализации (например, инициализация игровой сцены, связь с другими сценами).

Игровой скрипт — своего рода управляющая модель, виртуальный режиссер событий, происходящих в игре. Благодаря гибкости большинства скриптовых языков сценарии получаются сложными и интересными. При разработке можно предусмотреть действия игрока и прописать соответствующую реакцию системы. Скрипт — это как бы продолжение основного программного кода игры, т. е. структура скрипта соответствует структуре самой игры. Обычно стремятся создать удобную и понятную систему классов и объектов. Манипуляции ими в скрипте создают манипуляции, отождествленные с ними явлениями в самой игре. В этом и состоит концепция скриптового управления. Вообще метод скриптового программирования очень подходит под определение «виртуальная машина» (термин появился еще до возникновения современных компьютеров). Виртуальная

машина — некая математическая модель, среда, которая исполняет написанную для нее программу, хотя сама, в сущности, является программой.

5.3.2. Спецификация скриптового языка rManager

Рассмотрим простой язык сценариев, автоматизирующий работу робота-сапера. Файл скрипта состоит из двух частей — инициализации сцены (размеров, количества мин, начального положения робота) и основного цикла, который задает последовательность действий робота.

Инициализация сцены:

```
init {
    переменная = значение
    переменная = значение
    ...
}
```

Инициализирующая часть скрипта начинается с ключевого слова `init` и инициализирует заключенные в фигурные скобки переменные начальными значениями. Эта часть скрипта обязательна, и всегда расположена перед исполняющей частью. В качестве переменных выступают следующие глобальные переменные сцены:

- `scene_width` — ширина минного поля;
- `scene_height` — высота минного поля;
- `mine_quantity` — число мин;
- `robot_x` — координата *X* робота на минном поле в начальный момент;
- `robot_y` — координата *Y* робота на минном поле в начальный момент.

Все параметры, за исключением количества мин, задаются в «клеточках», т. е. мина и робот занимают на поле одну условную «клеточку». При перемещении робота на одну позицию в любую сторону он перемещается именно на одну клеточку минного поля. Значение переменной `mine_quantity` не может быть больше произведения `scene_width * scene_height – 1` (т. е. на минном поле все клетки заминированы за исключением одной, в которой и должен находиться робот в начальный момент).

В качестве значений любой из переменных может использоваться встроенная функция `rand()`, которая генерирует случайное число в диапазоне от `0` до предопределенного числа `RAND_MAX_VALUE` (на усмотрение разработчика).

Основной цикл:

```
run {
    блок операторов
}
```

Исполняющая часть скрипта расположена за инициализирующей частью и начинается с ключевого слова `run`. В фигурных скобках описаны действия робота, которые выполняет транслятор.

Для того чтобы выполнить задачу (т. е. разминировать все поле), скриптовый язык должен поддерживать следующие ключевые слова и функции (как минимум):

- `foreach_mine` — ключевое слово, организующее цикл по всем неразминированным минам;
- `goto_next_mine()` — функция передвижения робота к следующей неразминированной мине;
- `demine()` — функция разминирования мины, у которой в данный момент находится робот;
- `report("сообщение робота")` — функция вывода на экран сообщения от робота (например, «Работа окончена»).

Функции `goto_next_mine()` и `demine()` описаны без параметров. Правильнее было бы описать их следующим образом:

- `goto_next_mine(x, y)` — записывает в переменные `x` и `y` координату мины. Робот при этом перемещается не на ту же самую клетку, где находится мина (иначе он взорвется), а на соседнюю;
- `demine(x, y)` — координаты мины, которые вернула функция `goto_next_mine()`.

При таком подходе необходимо ввести ключевое слово для объявления переменных, например `var x, y`.

Пример 5.9. Скрипт для решения игровой задачи — разминирования поля:

```
init { /* Инициализирующая часть */
   scene_width = 50
   scene_height = 50
   robot_x = 0
   robot_y = 0
```

```
mine_quantity = 1000
}

run { /* Основная часть */
   report("Starting demine process")
   foreach_mine {
      var mine_x, mine_y
      goto_next_mine(mine_x, mine_y)
      report("Mine found. Demining...")
      demine(mine_x, mine_y)
      report("done")
   }
   report("Task complete")
}
```

Реализация функции `goto_next_mine()` предусматривается при разработке интерпретатора, к примеру, с помощью волнового алгоритма. В крайнем случае есть примитивный способ — хранить массив всех неразминированных мин, отсортированный по возрастанию координат, и передвигать робота к той мине, которая ближе всего к его текущему местоположению.

Контрольные вопросы

1. Чем интерпретатор отличается от компилятора?
2. Что такое внутреннее представление лексем?
3. Охарактеризуйте особенности построения интерпретаторов.
4. Каковы правила внутреннего представления выражений?
5. Как интерпретатор рассматривает операторы языка программирования?
6. Что такое разметка документа?
7. Приведите примеры языков разметки.
8. Какова структура документа?
9. Что такое блочная структура документа.
10. Чем различаются структурная разметка и разметка представления документа?
11. Что такое теги?
12. Охарактеризуйте понятие сценария (скрипта).
13. Какие виды скриптовых языков вы знаете?
14. Приведите примеры скриптов.

Рекомендуемая литература

1. *Романов Е. Л.* Основы построения трансляторов: Конспект лекций. — Новосибирск: НГТУ, 2001. — http://ermak.cs.nstu.ru

2. *Кадан А. М.* Системное программное обеспечение: Конспект лекций. — Гродно: Гродненский государственный университет им. Я. Купалы, 2004. — http://mf.grsu.by/Kafedry/kaf001/ academic_process

3. *Ахо А., Ульман Дж.* Теория синтаксического анализа, перевода и компиляции. — В 2 т. — М.: МИР, 1978.

4. *Мартыненко Б. К.* Языки и трансляции: учеб. пособие. — СПб.: Изд-во Санкт-Петербургского университета, 2004.

5. *Волкова И. А., Руденко Т. В.* Формальные грамматики и языки. Элементы теории трансляции: учеб. пособие. — М.: Изд-во МГУ, 1996.

6. Википедия. — http://www.ru.wikipedia.org

7. *Пентус А. Е., Пентус М. Р.* Теория формальных языков: учеб. пособие. — М.: Изд-во МГУ, 2004.

8. *Фомичев В. С.* Формальные языки, грамматики и автоматы: Конспект лекций. — СПб.: Санкт-Петербургский государственный электротехнический университет, 2001.

9. *Серебряков В. А., Галочкин М. П.* Основы конструирования компиляторов. — М.: МГУ, 1999.

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

11. *Хантер Р.* Проектирование и конструирование компиляторов. — М.: Финансы и статистика, 1984.

12. *Ахо А., Сети Р., Ульман Дж. Д.* Компиляторы: принципы, технологии и инструменты. — М.: Вильямс, 2001.

13. *Малявко А. А.* Теория формальных языков: учеб. пособие: В 3 ч. — Новосибирск: Изд-во НГТУ, 2002.

14. *Касьянов В. Н., Поттосин И. В.* Методы построения трансляторов. — Новосибирск: Наука, 1986.

15. Обратная польская нотация. — http://algolist.manual.ru/maths/misc/revpn.php

16. Интерпретатор алгоритмического языка. — http://kit.kulichki.net/ compilers/algol/desc.htm

17. Языки программирования, разметки, стандарты, спецификации во всемирной сети. — http://selikoff.ru/webmaster/standa rty_i_programm irovanie_html/

18. Язык гипертекстовой разметки HTML. — http://www.citfo rum.ru/programming/khramtsov/html.shtml#1

19. Современные языки сценариев. — http://docs.kde.org/kde3/ru/kdevelop/kdevelop/unixdev-scripting-languages.html

Практические задания

К ГЛАВЕ 1

1. Дана грамматика G. Определить тип и язык, порождаемые этой грамматикой.

а) P: $S \to 0A1$ $0A \to 00A1$ $A \to \lambda$	б) P: $S \to B \mid AS \mid BS$ $A \to a$ $B \to b$
в) P: $S \to A \mid SA \mid SB$ $A \to a$ $B \to b$	г) P: $S \to A \mid AS \mid BS$ $A \to a$ $B \to b$
д) P: $S \to A \mid SA \mid SB \mid B$ $A \to a$ $B \to b$	е) P: $S \to 0A1$ $0A \to 00A11$ $A \to 01$
ж) P: $S \to A \mid AS \mid BS \mid B$ $A \to a$ $B \to b$	з) P: $S \to 0A1$ $0A \to 00A11$ $A \to \lambda$

2. Построить грамматику, порождающую заданный язык.

а) Множество слов словаря T, начинающихся с буквы a. $T = \{a, b\}$	б) Множество чисел с порядком. Примеры: 3.2E-2, .5E+4, 162E+3, 34E+20
в) $L_1 = \{1^{3n+2}0^n \mid n \geq 0\}$	г) $L_2 = \{1^{3n+2}0^n \mid n \geq 1\}$
д) $L_3 = \{a^n b \mid n \geq 0\}$	е) $L_4 = \{ab^n c \mid n \geq 0\}$
ж) $L_5 = \{ab^n c \mid n \geq 1\}$	з) $L_6 = \{ab^n \mid n >= 0\}$

Окончание задания 2

и) Множество цепочек, начинающихся символом $ и оканчивающихся символом ?, между которыми расположена непустая последовательность из знаков + и –, не содержащая двух одинаковых символов, стоящих рядом ($ + – ?, $ + ?, $ + – + – + ?, $ – + – + – ?)	к) Множество правильных выражений, состоящих из знаков &, ∨ (конъюнкция, дизъюнкция), которые могут соединяться отношениями. Отношение строится из двух идентификаторов, соединенных знаками >, <, =, ≠. Например, $x > y \vee x > 2$ или $x = a \;\&\; x > b \vee x < c$
л) $L_7 = \{a_1a_2 \ldots a_na_n \ldots a_2a_1 \mid a_i = 0 \mid 1,\ n \geq 1\}$	м) $L_8 = \{\alpha c \beta c \gamma c \mid \alpha, \beta, \gamma$ — любые цепочки из a и $b\}$
н) $L_9 = \{b^n a \mid n >= 1\}$	о) $L_{10} = \{(a^{2m}b^m)^n \mid m \geq 1,\ n \geq 0\}$
п) Множество составных идентификаторов. Составной идентификатор — это несколько обычных идентификаторов, разделенных точкой (PQ.F11, SICN.X1.R, BL31.IN3.A6)	р) Множество правильных скобочных выражений. Например, (( )( )); ( )( )( )

3. К какому типу по Хомскому относится грамматика?

а) P: $S \to a \mid Ba$ $B \to Bb \mid b$	б) P: $S \to Ab$ $A \to Aa \mid ba$
в) P: $S \to 0A1 \mid 01$ $0A \to 00A1$ $A \to 01$	г) P: $S \to AB$ $AB \to BA$ $A \to a$ $B \to b$

4. Дан язык $L_{11} = \{1^{3n+2}0^n \mid n \geq 0\}$. Построить для цепочки 1111111100:

а) левосторонний вывод;

б) правосторонний вывод;

в) дерево разбора.

5. Дана грамматика. Построить вывод заданной цепочки.

а) P: $S \to aSBC \mid abC$ $bB \to bb$ $bC \to bc$ $cC \to cc$ $CB \to BC$ Цепочка *aaabbbccc*	б) P: $S \to T \mid T+S \mid T-S$ $T \to F \mid F*T$ $F \to a \mid b$ Цепочка *a-b*a+b*

6. Принадлежит ли цепочка 0101 языку L_{12}^* для $L_{12} = \{\lambda, 0, 10\}$.

7. Можно ли в грамматике с правилами $P_{13} = \{S \to 0U \mid 1S \mid 0, U \to 0U \mid 1\}$ вывести цепочку 1100?

8. В грамматике с правилами $P_{14} = \{S \to 0SU \mid 1,\ U \to U1 \mid 0S \mid \lambda\}$ вывести все цепочки длиной 3.

9. Перечислить все цепочки длиной 3 в языке L_{14}^* для $L_{14} = \{\lambda, 0, 01\}$.

10. Построить все сентенциальные формы для грамматики с правилами:

P: $S \to A+B \mid B+A$;
$A \to a$;
$B \to b$.

11. Дана грамматика. Построить дерево вывода для заданной цепочки восходящим и нисходящим методами.

а) P: $S \to S0 \mid S1 \mid D0 \mid D1$ $H \to 0 \mid 1 \mid H0 \mid H1$ $D \to H$ Цепочка 10.1001	б) P: $S \to$ if B then $S \mid B = E$ $B \to a \mid b$ $E \to B \mid B+E$ Цепочка if a then $b = a+b+$

12. Эквивалентны ли грамматики с правилами:

а) $S \to AB$ $A \to a \mid Aa$ $B \to b \mid Bb$	б) $B \to b$ $A \to a$ $S \to AS \mid SB \mid AB$

13. Показать, что грамматика G неоднозначна.

P: $S \to abC \mid aB$;

$B \to bc$;

$bC \to bc$.

14. Показать, что грамматика G с правилами P: $E \to E+E \mid E*E \mid (E) \mid i$ неоднозначна. Как описать этот же язык с помощью однозначной грамматики?

К ГЛАВЕ 2

1. Построить регулярную грамматику, эквивалентную грамматике с правилами.

а) P: $S \to A \mid AS$ $A \to a \mid bb$	б) P: $S \to A.A$ $A \to B \mid BA$ $B \to 0 \mid 1$
в) P: $A \to B \mid C$ $B \to 0B \mid 1B \mid 01$ $C \to 0D \mid 1C \mid \lambda$ $D \to 0C \mid 1D$	в) P: $S \to C*) \mid K\}$ $C \to (* \mid Ca \mid C\} \mid C(\mid C* \mid C)$ $K \to \{ \mid K(\mid K* \mid K) \mid K\{$

2. Построить регулярную грамматику и диаграмму состояний для заданного языка.

а) Множество цепочек из алфавита $\{a, /, *\}$, содержащих правильные комментарии языка С. Например, $/*aaa*/$	б) $L = \{(abb)^k\perp \mid k \geq 1\}$
в) Множество целых восьмеричных чисел, которые не должны начинаться с 0	г) Множество цепочек в алфавите $\{a, b\}$, в которых символ a не встречается 2 раза подряд

3. Построить леволинейную регулярную грамматику, эквивалентную данной праволинейной, допускающую детерминированный разбор и ее диаграмму состояний.

а) P: $S \to 0S \mid 0B$ $B \to 1B \mid 1C$ $C \to 1C \mid \perp$	б) P: $S \to aA \mid aB \mid bA$ $A \to bS$ $B \to aS \mid bB \mid \perp$

4. Определить язык, порождаемый грамматикой. Построить конечный автомат по заданной грамматике (если нужно, преобразовать грамматику в регулярную леволинейную).

а) P: $S \to 0S \mid S0 \mid D$ $D \to DD \mid 1A \mid \lambda$ $A \to 0B \mid \lambda$ $B \to 0A \mid 0$	б) P: $S \to C\perp$ $B \to B1 \mid 0 \mid \lambda$ $C \to B1 \mid C1$ $D \to D1 \mid 1$
в) P: $S \to C\perp$ $B \to B1 \mid 0 \mid D0$ $C \to B1 \mid C1$ $D \to D0 \mid 0$	г) P: $S \to C\perp$ $C \to B1$ $B \to 0 \mid D0$ $D \to B1$
д) P: $S \to A\perp$ $A \to Ab \mid Bb \mid b$ $B \to Aa$	е) P: $S \to 1U \mid \lambda$ $U \to 12V \mid 0$ $V \to 1 \mid 0S$
ж) P: $S \to S0 \mid A1 \mid 0$ $A \to S1 \mid 0$	з) P: $S \to A0$ $A \to A0 \mid S1 \mid 0$

5. Построить детерминированный конечный автомат для заданного языка.

а) Множество всех цепочек из алфавита {0, 1}, в которых каждые пять последовательных символов содержат хотя бы два нуля	б) Множество всех цепочек из алфавита {0, 1}:, которые начинаются или оканчиваются (или и то и другое) последовательностью 01
в) Множество всех цепочек из алфавита {0, 1}, у которых на третьей позиции справа стоит 1	г) Множество цепочек в алфавите {0, 1}, в которых не встречается подцепочка 010
д) Множество цепочек в алфавите {0, 1}, оканчивающихся на 01	е) Множество всех цепочек в алфавите {0, 1}, содержащих три нуля подряд

6. Построить регулярную грамматику конечного автомата.

а) $K_1 = (\{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_1\})$ с функцией перехода δ: $\delta(q_0, 0) = q_0$; $\delta(q_0, 1) = q_1$	б) $K_2 = (\{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_1\})$ с функцией перехода δ: $\delta(q_0, 0) = q_1$; $\delta(q_1, 0) = q_0$.

7. Является ли язык, порождаемый грамматикой P: $S \to S0S \mid 1$ регулярным?

8. Дан автомат $K = (\{A, B, C\}, \{0, 1\}, \delta, A, \{B, C\})$ с функцией перехода: $\delta(A, 0) = B$, $\delta(A, 1) = C$, $\delta(B, 0) = C$, $\delta(C, 1) = B$. Построить эквивалентный автомат с одним конечным состоянием.

9. Детерминировать конечный автомат.

а) $K_3 = (\{q_0, q_1\}, \{a, b, c\}, \delta, q_0, \{q_1\})$ с функцией перехода δ: $\delta(q_0, b) = \{q_0, q_1\}$; $\delta(q_0, c) = q_0$; $\delta(q_1, a) = q_1$; $\delta(q_1, c) = \{q_0, q_1\}$	б) $K_4 = (\{s_1, s_2, s_3\}, \{0, 1\}, \delta, s_1, \{s_2\})$ с функцией перехода δ: $\delta(s_1, 0) = s_2$; $\delta(s_2, 1) = \{s_2, s_3\}$; $\delta(s_3, 0) = s_1$.

10. Построить регулярную грамматику по конечному автомату $M_1 = (\{p, q, r\}, \{0, 1\}, \delta, p, \{r\})$ со следующей таблицей перехода:

Состояния	Символы	
	0	**1**
p	q	p
q	r	p
r	r	r

11. Детерминировать конечный автомат, приведенный на рис. П2.1.

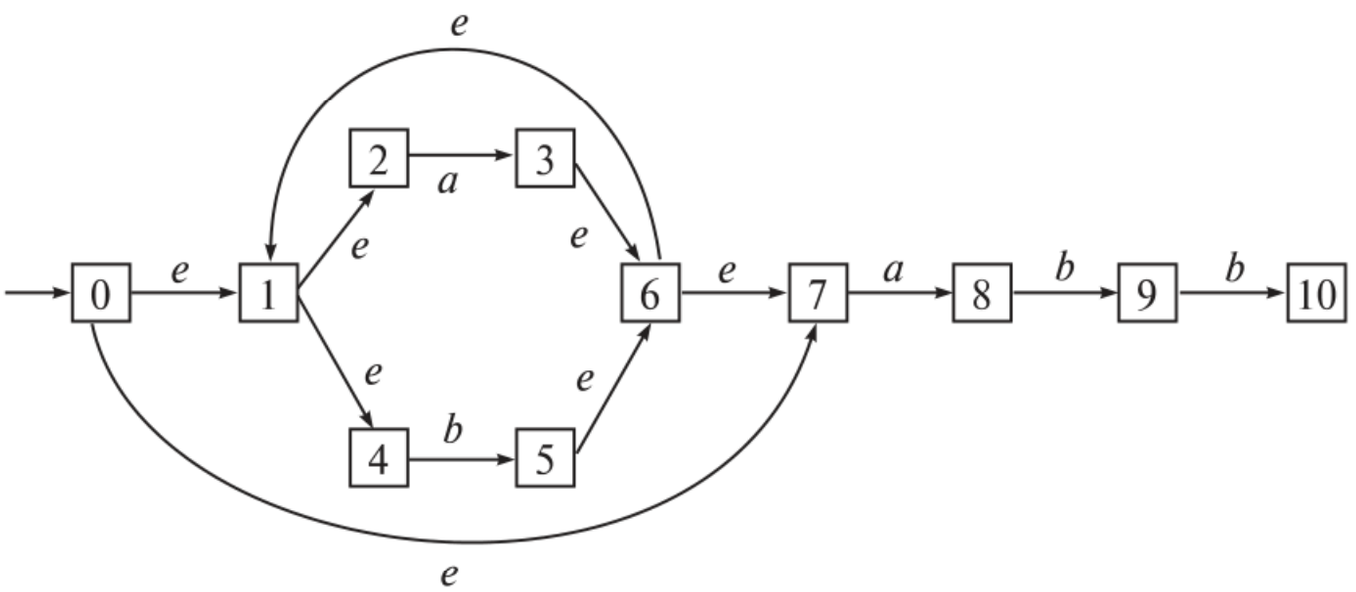


Рис. П2.1. Конечный автомат

12. Минимизировать конечный автомат, приведенный на рис. П2.2.

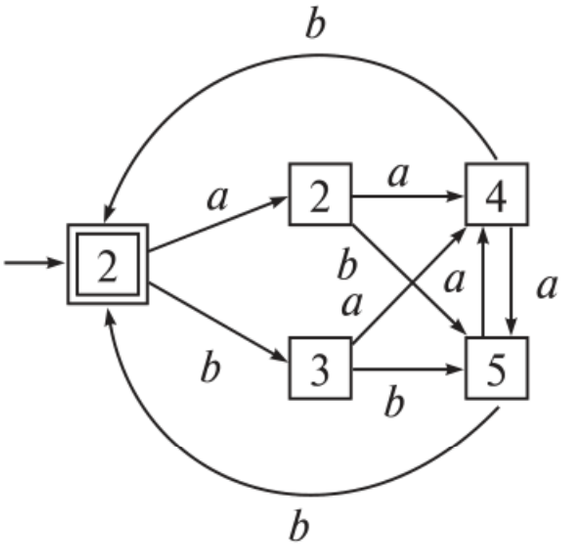


Рис. П2.2. Конечный автомат

13. Построить детерминированный конечный автомат по $M_2 = (\{p, q, r, s, t\}, \{1, 2, 3\}, \delta, p, \{r\})$ с функцией перехода, заданной таблицей:

Состояния	Символы		
	1	**2**	**3**
p	p, q	p, r	p, s
q	r	q	q
r	r	r, t	r
s	s	s	s, t
t	—	—	—

14. Даны грамматики G_1 и G_2 и языки $L(G_1)$и $L(G_2)$ с правилами:

а) $P: S \to 0C \mid 1B \mid \lambda$ $B \to 0B \mid 1C \mid \lambda$ $C \to 0C \mid 1C$	б) $P: S \to 0D \mid 1B$ $B \to 0C \mid 1C$ $C \to 0D \mid 1D \mid \lambda$ $D \to 0D \mid 1D$

Построить регулярную грамматику для $L(G_1) \cap L(G_2)$.

1. Тестовые задания для проверки программы (к практическим заданиям гл. 2)

Автомат 1	Автомат 2	Автомат 3
1. *q*0,*a*=*q*2	*q*0, \=*f*0	*q*0, *a*=*q*1
*q*0, *b*=*q*1	*q*0, /=*q*1	*q*0, *a*=*q*4
*q*1, *a*=*q*3	*q*1, *a*=*q*2	*q*0, *a*=*q*3
*q*2, *a*=*q*3	*q*1, +=*q*3	*q*1, *c*=*q*1
*q*2, *b*=*q*3	*q*2, "=*q*3	*q*1, *c*=*q*5
*q*3, =*q*3	*q*2,,=*q*4	*q*1, *d*=*q*2
*q*3, +=*q*4	*q*3, *e*=*f*0	*q*2, *a*=*q*5
*q*4, =*q*4	*q*3, *f*=*q*0	*q*2, *d*=*f*0
*q*4, *c*=*q*5	*q*3, *g*=*q*5	*q*3, *e*=*q*4
*q*5, *d*=*q*6	*q*4, ;=*q*2	*q*3, *e*=*f*1
*q*6, =*q*6	*q*4, 8=*f*2	*q*4, *b*=*f*1
*q*6, *=*q*7	*q*5, +=*q*5	*q*4, *b*=*q*5
*q*7, =*q*7	*q*5, *=*f*2	*q*5, *f*=*f*0
*q*7, ==*q*10		
*q*7, *e*=*q*8		
*q*8, *e*=*q*8		
*q*8, ==*q*10		
*q*8, =*q*9		
*q*9, =*q*10		
*q*9, ==*q*10		
*q*10, 3=*q*11		
*q*11, 5=*q*12		
*q*12, 7=*f*0		

2. Пример входного файла автомата

*q*0, =*q*0
*q*0, *f*=*q*1
*q*1, *o*=*q*2
*q*2, *r*=*q*3
*q*3, =*q*3
*q*3, (=*q*4
*q*4, =*q*4
*q*4, *i*=*q*5
*q*5, *n*=*q*6
*q*6, *t*=*q*7
*q*7, =*q*7
*q*7, *a*=*q*8
*q*7, *b*=*q*8
*q*7, *c*=*q*8
*q*7, *A*=*q*8
*q*7, *B*=*q*8
*q*7, *C*=*q*8
*q*8, *a*=*q*8
*q*8, *b*=*q*8
*q*8, *c*=*q*8
*q*8, *A*=*q*8
*q*8, *B*=*q*8
*q*8, *C*=*q*8
*q*8, ==*q*10
*q*8, =*q*9
*q*9, =*q*9
*q*9, ==*q*10
*q*10, =*q*10
*q*10, 0=*q*11
*q*10, 1=*q*11
*q*10, 2=*q*11
*q*10, 3=*q*11
*q*10, 4=*q*11
*q*10, 5=*q*11
*q*10, 6=*q*11
*q*10, 7=*q*11
*q*10, 8=*q*11
*q*10, 9=*q*11
*q*11, 0=*q*11
*q*11, 1=*q*11
*q*11, 2=*q*11
*q*11, 3=*q*11
*q*11, 4=*q*11
*q*11, 5=*q*11
*q*11, 6=*q*11
*q*11, 7=*q*11
*q*11, 8=*q*11
*q*11, 9=*q*11
*q*11, ;=*q*13
*q*11, =*q*12
*q*12, =*q*12
*q*12, ;=*q*13
*q*13, =*q*13
*q*13, *a*=*q*14
*q*13, *b*=*q*14
*q*13, *c*=*q*14
*q*13, *A*=*q*14
*q*13, *B*=*q*14
*q*13, *C*=*q*14
*q*14, *a*=*q*14
*q*14, *b*=*q*14
*q*14, *c*=*q*14
*q*14, *A*=*q*14
*q*14, *B*=*q*14
*q*14, *C*=*q*14
*q*14, =*q*15
*q*14, <=*q*16
*q*14, >=*q*16
*q*15, =*q*15
*q*15, =<q16
*q*15, >=*q*16
*q*16, =*q*16
*q*16, 0=*q*17
*q*16, 1=*q*17
*q*16, 2=*q*17
*q*16, 3=*q*17
*q*16, 4=*q*17
*q*16, 5=*q*17
*q*16, 6=*q*17
*q*16, 7=*q*17
*q*16, 8=*q*17
*q*16, 9=*q*17
*q*17, 0=*q*17
*q*17, 1=*q*17
*q*17, 2=*q*17
*q*17, 3=*q*17
*q*17, 4=*q*17
*q*17, 5=*q*17
*q*17, 6=*q*17
*q*17, 7=*q*17
*q*17, 8=*q*17
*q*17, 9=*q*17
*q*17, =*q*18
*q*17, ;=*q*19
*q*18, =*q*18
*q*18, ;=*q*19
*q*19, =*q*19
*q*19, *a*=*q*20
*q*19, *b*=*q*20
*q*19, *c*=*q*20
*q*19, *A*=*q*20
*q*19, *B*=*q*20
*q*19, *C*=*q*20
*q*20, *a*=*q*20
*q*20, *b*=*q*20
*q*20, *c*=*q*20
*q*20, *A*=*q*20
*q*20, *B*=*q*20
*q*20, *C*=*q*20
*q*20, =*q*21
*q*20, +=*q*22
*q*21, =*q*21
*q*21, +=*q*22
*q*22, +=*q*23
*q*23, =*q*23
*q*23,)=f

Приведенный автомат разбирает строку:

for ([*abc*] = [0–9]; [*abc*] [<|>] [0–9]; [*abc*]++)

Здесь [*abc*] — строка произвольной длины, состоящая из символов *a*, *b*, *c*, *A*, *B*, *C*;

[0–9] — строка произвольной длины, состоящая из символов цифр;

[< | >] — или символ «<», или символ «>».

3. Пример работающей программы (операция детерминирования не производится)

```
#include <string>
#include <vector>
#include <algorithm>
#include <iostream>
#include <fstream>

typedef unsigned char uchar;

using namespace std;

class format_error: public runtime_error {
public:
 format_error(const char* msg): runtime_error(msg){}
};
class StateReader{ //class for reading states from file
public:
 struct ElementarySwitch{ // one switch for automat
  int initialState; // number of current state
  uchar letter; // reading symbol
  bool isTerminalState; // is next state terminal?
  int nextState; // number of next state
  ElementarySwitch():
   initialState(-1),
   letter(0),
   isTerminalState(false),
   nextState(-1)
   {}

 // next 2 operators — for sorting states array
    friend bool operator > (const ElementarySwitch& el,
    const ElementarySwitch& er){
  if (el.initialState > er.initialState) return true;
  if (el.initialState == er.initialState){
  if (el.letter > er.letter) return true;
  if (el.letter == er.letter){
  if (el.isTerminalState != er.isTerminalState) return
          er.isTerminalState;
  if (el.nextState > er.nextState) return true;
  }
 }
return false;
}
 friend bool operator < (const ElementarySwitch& el,
   const ElementarySwitch& er){
```

```
return !operator>(el, er);
}
};

typedef vector<ElementarySwitch> StatesSwitchArray;

StateReader(const char* filename);
~StateReader() {stateFile.close();}

protected:
StatesSwitchArray statemachineStates; // array of switches
for automat

private:
 ifstream stateFile; // stream for reading file
};

StateReader::StateReader(const char*
      filename):stateFile(filename){
 if(!stateFile.is_open()) throw runtime_error("Invalid
      states file"); // can't open file

string tmpStr;
while(getline(stateFile, tmpStr)){
if (tmpStr.size() == 0) continue; // skip empty string
// several check for input file format
if (tmpStr[0] != 'q' && tmpStr[0] != 'Q')
 throw format_error("Line must begin with 'q' letter");
 string::size_type commaPos = tmpStr.find(',');
if (commaPos == string::npos)
 throw format_error("There is no comma");
string stateNumber = tmpStr.substr(1, commaPos - 1);
ElementarySwitch tmpSw; // prepare next elementh in array
tmpSw.initialState = atoi(stateNumber.c_str());
if (tmpSw.initialState == 0 && stateNumber[0] != '0')
 throw format_error("State number must contains digits
                        only");
tmpSw.letter = tmpStr[commaPos + 1];
if (tmpStr[commaPos + 2] != '=')
 throw format_error("Expected '=' sign");
switch (tmpStr[commaPos + 3]){
 case 'f':
 case 'F':
  tmpSw.isTerminalState = true;
  break;
 case 'q':
 case 'Q':
```

```
  tmpSw.isTerminalState = false;
  break;
default:
 throw format_error("Next state must begin with 'q' or 'f'
                     letter");
}
stateNumber = tmpStr.substr(commaPos + 4);
tmpSw.nextState = atoi(stateNumber.c_str());
if (tmpSw.nextState == 0 && stateNumber[0] != '0')
 throw format_error("State number must contains digits
                            only");

statemachineStates.push_back(tmpSw); // add one switch
      to array of switches
}
}

class StateMachine: public StateReader{
public:
 StateMachine(const char* filename);
 bool isDeterministic() const {return deterministic;}
 bool hasHangs() const {return hangs;} // means that graph
                   contains isolated node(s)
 const StateReader::StatesSwitchArray& GetSwitches() const
 {return statemachineStates;} // just for printing
bool isExpressionCorrect(const string& expression, int&
errorPos);

protected:
 void SortStates();
 bool _isDeterministic();
 bool _hasHangs();
 int _findNextIndex(int curState, uchar sym);

private:
 bool deterministic;
 bool hangs;
};

StateMachine::StateMachine(const char* filename):
 StateReader(filename),
 deterministic(true),
 hangs(false)
{
 SortStates();
 //some little check of machine
 if (statemachineStates.size() == 0)
  throw runtime_error("Automat is empty");
```

```
 if (statemachineStates[0].initialState != 0)
  throw runtime_error("There is no initial state");
 size_t ln = statemachineStates.size();
 bool hasFinalState = false;
 for (size_t i = 0; i < ln; i++)
if (hasFinalState = statemachineStates[i].isTerminalState)
break;
if (!hasFinalState)
throw runtime_error("There is no final state");

deterministic = _isDeterministic(); // check if automat
            is deterministic
 hangs = _hasHangs(); // check if may be hangs
}

bool StateMachine::_isDeterministic(){
 size_t ln = statemachineStates.size(); // count of
            elements in array
 bool isDet = true;
 for (size_t i = 1; i < ln; i++)
  if (statemachineStates[i-1].initialState ==
   statemachineStates[i].initialState &&
   statemachineStates[i-1].letter ==
   statemachineStates[i].letter &&
      (statemachineStates[i-1].isTerminalState !=
  statemachineStates[i].isTerminalState ||
   statemachineStates[i-1].nextState !=
  statemachineStates[i].nextState))
 {
 isDet = false;
 break;
 };

 return isDet;
}

bool StateMachine::_hasHangs(){
 size_t ln = statemachineStates.size();
 bool isHangs = false;
 for (size_t i = 0; i < ln; i++){
 if (!statemachineStates[i].isTerminalState){
  bool found = false;
  // very bad algorithm to search in _SORTED_ array.
       I was laziness to do better :->
  for (size_t j = 0; j < ln; j++){
```

```
    if (statemachineStates[i].nextState ==
    statemachineStates[j].initialState){
     found = true;
     break;
   }
}
 if (!found){
  isHangs = true;
  break;
 }
}
}
return isHangs;
}

void StateMachine::SortStates(){
 sort(statemachineStates.begin(),
statemachineStates.end()); // common sorting algorithm
         from <algorithm>
 }

int StateMachine::_findNextIndex(int curState, uchar sym){
 int found = -1;
 size_t ln = statemachineStates.size();

 // very bad algorithm to search in _SORTED_ array
for (size_t j = 0; j < ln; j++){
 if (statemachineStates[j].initialState == curState &&
    statemachineStates[j].letter == sym){
    found = j;
    break;
 }
}
return found;
}

bool StateMachine::isExpressionCorrect(const string&
     expression, int& errorPos){
if ( !deterministic || hangs)
  throw runtime_error("This automat cannot check
      expression");
// emulate automat's task
int currentState = 0;
size_t strLen = expression.size();
for (int i = 0; i < strLen; i++){
 int idx = _findNextIndex(currentState, expression[i]);
 if (idx < 0){
```

```
  errorPos = i;
 return false;
}
if (statemachineStates[idx].isTerminalState){
if (i == strLen - 1) return true;
  errorPos = i + 1;
  return false;
}
 currentState = statemachineStates[idx].nextState;
}
errorPos = strLen;
return false;
}

int _tmain(int argc, _TCHAR* argv[]) {
 try{ // try to create object of StateMachine
  StateMachine sr("states.txt");
  StateReader::StatesSwitchArray::const_iterator it;
     // another way to access to array's elements
  for (it = sr.GetSwitches().begin(); it !=
  sr.GetSwitches().end(); it++)
  cout << "q" << it->initialState << "," << it->letter <<
"=" << (it->isTerminalState ? "f" : "q") <<
  it->nextState << endl;
  cout << "There are" << (sr.hasHangs() ? "" : "n't") <<
  " hangs" << endl;
  cout << "Automat is" << (sr.isDeterministic() ? "" :
 "n't") << " deterministic" << endl;

string testExpr;
 cout << "Please, enter expression to check ";
 cin >> testExpr;
 // for test with related "states.txt" file
 // testExpr = " for( int abAccc= 943 ; a<478; bbc++ )";
 // cout << testExpr << endl;
 int err;
 bool res = sr.isExpressionCorrect(testExpr, err);
 if (res) cout << "Expression is correct!" << endl;
 else cout << "Incorrect expression. Error position: " <<
 err << endl;
 }
catch(const exception& err){
 cerr << err.what() << endl;
 }
 return 0;
}
```

Лабораторная работа № 1

КОНЕЧНЫЕ ДЕТЕРМИНИРОВАННЫЕ АВТОМАТЫ. ПРЕОБРАЗОВАНИЕ НЕДЕТЕРМИНИРОВАННОГО КОНЕЧНОГО АВТОМАТА К ДЕТЕРМИНИРОВАННОМУ

Цель работы:

- изучить свойства лексических анализаторов на основе конечных автоматов;
- освоить разбор лексических цепочек символов;
- научиться преобразовывать недетерминированный конечный автомат к детерминированному.

Подготовка к работе

1. Прочитать соответствующие разделы в изданиях [1—5, 8—14].

2. Ознакомиться с теоретическим материалом, изложенным в подразд. 2.1—2.3 данного учебного пособия.

Задание. Написать программу, реализующую работу конечного автомата.

Входные данные

1. Текстовый файл, описывающий граф переходов конечного автомата (приложение 1). Файл представляет собой набор строк. В каждой строке задается *одно* правило перехода в следующем виде:

$q<N>,<C>=<q\,|\,f><N>$

Здесь символ q обозначает состояние автомата; f — конечное состояние автомата; $<N>$ — произвольное число, обозначающее номер состояния; $<C>$ — **один** символ входной цепочки.

Пример. $q12,\ g=f0$ — запись означает, что если автомат находится в состоянии № 12 и читает с ленты символ 'g', то он переходит в конечное состояние с номером 0.

Дополнительные условия:

- количество строк в файле (возможных переходов) — не ограниченно;
- начальное состояние автомата (с которого начинается его работа) — $q0$;
- строки в файле не обязаны быть отсортированы по какому-либо критерию;
- состояния автомата необязательно нумеруются последовательно.

2. Строка символов, которую нужно проанализировать с помощью построенного автомата и дать заключение о возможности (или невозможности) разбора этой строки с помощью данного автомата (приложение 2).

Выходные данные

1. Заключение о детерминированности или недетерминированности заданного автомата.

2. В случае недетерминированного автомата вывести переходы для соответствующего ему детерминированного автомата (в виде, соответствующем входному файлу).

3. Заключение о возможности (невозможности) разбора автоматом введенной строки символов.

Выполняемые функции:

- считать из файла автомат (файл может содержать синтаксические ошибки; заданный с его помощью автомат, возможно, недетерминирован; возможно, граф переходов содержит висячие вершины);
- вывести информацию об автомате (детерминирован/нет, количество конечных состояний и другие свойства);
- детерминировать автомат, вывести таблицу переходов для нового автомата;
- разобрать входную строку;
- дать заключение о возможности/невозможности ее разбора.

В приложении 3 приводится пример программы разбора входной цепочки с помощью недетерминированного автомата.

Лабораторная работа № 2
РАЗРАБОТКА ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА

Цель работы: разработать лексический анализатор цепочек условного языка программирования с обработкой лексических ошибок.

Подготовка к работе

1. Изучить разделы по тематике в изданиях [1—6, 8—14].

2. Ознакомиться с теоретическим материалом, изложенным в подразд. 2.4 данного учебного пособия.

Задание. Разработать класс-лексический анализатор.

Входные данные

1. Файл с описанием регулярных определений для распознаваемых токенов. Каждая строка в файле описывает один токен. Строка имеет следующий формат:

<имя токена> <один или более пробелов> <регулярное выражение для токена>

Если в <регулярное выражение для токена> входит <имя токена>, то оно заключается в фигурные скобки (т. е. {<имя токена>}).

Символы '-' (диапазон), '|' (логическое «или»), '+' (одно или более повторений), '*' (нуль или более повторений), '?' (нуль или одно вхождение), '.' (любое количество любых символов), '(', ')', '[' и ']' (группировка), '{' и '}' (ограничивают имя токена) имеют специальные значения, поэтому для того, чтобы использовать их в качестве обычных символов, перед ними ставится символ '\'. Для задания символа '\' используется запись '\\'. Последовательность \t соответствует символу табуляции, \n — символу новой строки.

Пример. Входной файл для лексического анализатора:

```
if          if
then        then
else        else
relop       < | <= | = | <> | > | >=
letter      [A-Za-z]
digit       [0-9]
id          (_|{letter})(_|{letter}|{digit})*
num         {digit}+(\.{digit}+)?((E|e)(+|\-)?{digit}+)?
ws          [ \t\n]+
string       \».\»
```

При выполнении лабораторной работы допускается расширить синтаксис входного файла.

2. Файл с «программой», которую должен обработать лексический анализатор.

Выходные данные

1. Пары «<имя токена>-<атрибут-значение>» для каждой распознанной лексемы (для простоты в виде двух строк <имя токена>-<считанная лексема>).

2. Если лексема не распознается, то вернуть ошибку и указать номер строки и символа, где произошла ошибка разбора.

Выполняемые функции:

- чтение файла с описанием токенов. При чтении (и дальнейшем разборе) необходимо обеспечить некоторую минимальную проверку корректности синтаксиса данного файла (программа не должна зависать при анализе);
- чтение файла с «программой» (по большому счету — последовательность лексем) и вывод на экран (или в другой файл) результатов разбора.

К ГЛАВЕ 3

1. Написать КС-грамматику для языка L_i, построить левосторонний вывод для цепочки.

а) P: $L_1 = \{a^{2n}b^m c^{2k} \mid m = n + k,$ $m > 1\}$ Цепочка *aabbbcccc*	б) P: $L_2 = \{1^n 0^m 1^p \mid n + p > m;$ $n, p, m > 0$ Цепочка 110000111

2. Построить МП-автомат по КС-грамматике с правилами:

а) P: $S \to U1V \mid 0$ $V \to 1U \mid 0$ $U \to 0V \mid 1$	б) P: $S \to a \mid 1UV$ $U \to bU \mid b$ $V \to aU \mid a$
в) P: $S \to 0SS1 \mid 1$	г) P: $S \to +SS \mid *SS \mid c$
д) P: $S \to aB$ $B \to baA$ $A \to c \mid cB$ $B \to b$	е) P: $S \to AS \mid a$ $A \to BA \mid b$ $B \to Aa$
ж) P: $S \to SS \mid 1S0 \mid 0S1$	з) P: $S \to 1S0 \mid 0S1 \mid \lambda$
и) P: $S \to 1S \mid 0U$ $U \to 1U \mid 01$	к) P: $S \to AS \mid b$ $A \to SA \mid a$
л) P: $S \to 0S \mid 1S \mid \lambda$	м) P: $S \to SS0 \mid 1$

3. Построить МП-автомат для языка.

а) Множество цепочек в алфавите {0, 1, (,)}, в которых правильная структура открывающих и закрывающих скобок. Например, (0100(11)0(10))()	б) Множество цепочек в алфавите {0, 1} с одинаковым количеством нулей и единиц. Например, 0011000111
в) $L_3 = \{0^n 1^{(3*m)} \mid m = n\}$	г) $L_4 = \{0^n 1^{(m+2)} \mid m = n\}$
д) $L_5 = \{0^n 1^{(2*m+1)} \mid m = n\}$	е) $L_6 = \{a^n b a^{(n+1)} \mid n >= 1\}$

4. Построить приведенную грамматику, эквивалентную заданной (удалить бесполезные символы, цепные и аннулирующие правила).

а) P: $S \to 0a0 \mid 1b1 \mid bb$ $a \to c$ $b \to s \mid a$ $c \to s \mid \lambda$	б) P: $I \to ABC$ $A \to BB \mid e$ $B \to CC \mid a$ $C \to AA \mid b$
в) P: $I \to aM$ $M \to A$ $A \to aA \mid B$ $B \to bB \mid b$	г) P: $S \to AB \mid CA$ $A \to a$ $B \to BC \mid AB$ $C \to aB \mid b$
д) P: $S \to 01V \mid U1 \mid 1$ $Z \to S1 \mid 0$ $U \to 1S$ $V \to Z \mid 0$ $Y \to 0$	е) P: $S \to aBCD \mid aG \mid bCCGS$ $A \to SCD \mid bG \mid d$ $B \to dFS \mid aBC$ $C \to aGS \mid aGS$ $D \to aAC \mid d$ $G \to aCG \mid a$ $F \to AB \mid aF$
ж) P: $I \to ac \mid bA$ $A \to cBC$ $B \to aIA$ $C \to bc \mid d$	з) P: $I \to ABC$ $A \to BB \mid \lambda$ $B \to CC \mid a$ $C \to AA \mid b$
и) P: $S \to SRT \mid c$ $R \to aRa \mid b$ $T \to aT$	к) P: $I \to aIa \mid bAd \mid c$ $A \to cBd \mid aAd$ $B \to dAf$
л) P: $D \to Sc$ $S \to SS \mid aSb \mid \lambda$	м) P: $I \to aIb \mid c$ $A \to bI \mid a$

5. Построить множества FIRST и FOLLOW для нетерминалов грамматики.

а) P: $S \to T0$ $T \to AB \mid 1B$ $A \to 2A \mid \lambda$ $B \to 3B \mid \lambda$	б) P: $S \to S+T \mid T$ $T \to T*E \mid T/E \mid E$ $E \to -F \mid F$ $F \to (S) \mid a$
в) P: $S \to S$ or $T \mid T$ $T \to T$ xor $E \mid E$ $E \to E$ and $F \mid F$ $F \to a < a \mid a > a \mid a = a$	г) P: $S \to -B$ $B \to T \mid B\&T$ $T \to J \mid T\hat{}J$ $J \to (B) \mid p$
д) P: $\Pi \to \perp B \perp$ $B \to T \mid B+T$ $T \to M \mid T*M$ $M \to u \mid (B)$	е) P: $S \to bMb$ $M \to (L \mid a$ $L \to Ma)$

6. Привести грамматику к нормальной форме Хомского.

а) P: $S \to aUbU \mid \lambda$ $U \to S \mid ba$	б) P: $S \to ASA \mid \lambda$ $A \to aAS \mid a$ $B \to SbS \mid A \mid bb$

7. Привести грамматику к нормальной форме Грейбах.

а) P: $S \to 01V \mid U1 \mid 1$ $Z \to S1 \mid 0$ $U \to 1S$ $V \to Z \mid 0$ $Y \to 0$	б) P: $S \to XY \mid 1$ $X \to ZT$ $Z \to S0$ $Y \to 0$ $T \to 1$
в) P: $I \to aA \mid Ic \mid Ab$ $A \to d$	г) P: $S \to S0 \mid U \mid 0$ $U \to U1 \mid S \mid 1$
д) P: $S \to RT$ $T \to b \mid UR$ $U \to VT$ $V \to RT$ $R \to a$	е) P: $S \to AB$ $B \to AB \mid b$ $A \to BB \mid a$

8. Построить матрицу предшествования для заданной грамматики.

а) $P: S \to 0S1 \mid U$ $U \to 0S1 \mid 1$	б) $P: S \to S1 \mid U1$ $U \to S1 \mid 11$
в) $P: S \to bMb$ $M \to (L \mid a$ $L \to Ma)$	г) $P: E \to T+E \mid T$ $T \to F*T \mid F$ $F \to (E) \mid a$

9. Обладает ли заданная грамматика свойством $LL(1)$?

а) $P: S \to AaAb \mid BaBb$ $A \to 0 \mid AB$ $B \to Ab \mid A$	б) $P: S \to AB$ $A \to 0A1 \mid \lambda$ $B \to 1B \mid 1$
в) $P: S \to T0$ $T \to AB \mid 1B$ $A \to 2A \mid \lambda$ $B \to 3B \mid \lambda$	г) $P: S \to aB$ $B \to baA$ $A \to c \mid cB$ $B \to b$

10. Построить МП-автомат, допускающий следующие такты:

$...(a,0110,\#) \vdash (b,0110,1\#) \vdash (b,110,1\#) \vdash (b,10,11\#) \vdash (c,0,\#)...$

Лабораторная работа № 3
НЕДЕТЕРМИНИРОВАННЫЕ МАГАЗИННЫЕ АВТОМАТЫ

Цель работы:

- изучить свойства синтаксических анализаторов на основе магазинных автоматов;
- научиться создавать автоматы с магазинной памятью на основе КС-грамматик;
- освоить синтаксический разбор цепочек символов.

Подготовка к работе

1. Изучить соответствующие разделы в изданиях [1—4, 8—14].

2. Ознакомиться с теоретическим материалом, изложенным в подразд. 3.4 данного учебного пособия.

Задание. Написать программу, реализующую работу недетерминированного магазинного автомата.

Входные данные

1. Текстовый файл с описанием грамматики, для которой строится магазинный автомат.

Каждая строка в файле может задавать несколько правил грамматики с одинаковой левой частью. В этом случае правила отделяются друг от друга символом '|'. Для отделения левой части правила от правой используется символ '>'. В одной строке может быть несколько символов '>'. В этом случае первый из них трактуется как символ, отделяющий правую и левую части продукции, все последующие — как терминальный символ. Все нетерминалы задаются с помощью прописных букв латинского алфавита. Все символы, не описанные выше, являются терминальными символами. Правил в грамматике (а значит и во входном файле) — неограниченное количество.

Рекомендуется считать пробельные символы в файле незначащими, а для терминала «пробел» использовать какой-нибудь другой символ (например, ~, либо заключать пробел в апострофы).

Пример. Входной файл с грамматикой

$E > E+T \mid T$

$T > T*F \mid F$

$F > (E) \mid a \mid \sim$

2. Строка символов, которую нужно проанализировать с помощью построенного автомата и дать заключение о допустимости (или недопустимости) автоматом данной цепочки символов.

Выходные данные

1. Значение множеств P, H и т. д.

2. Список команд магазинного автомата.

3. Цепочка конфигураций магазинного автомата, полученная в процессе его работы.

4. Заключение о допустимости (или недопустимости) автоматом цепочки символов.

Тестовые задания для проверки программы

Грамматика 1	Грамматика 2	Грамматика 3
$E > mT \mid !T \mid T$ $T > /P/$ $P > R \mid S$ $R > C–C$ $C > a \mid b \mid c \mid 0 \mid >$ $S > C \mid CS$	$E > C \mid CS$ $C > a \mid b \mid x \mid y$ $S > C \mid D \mid CS \mid DS$ $D > 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$	$E > a \mid Sa \mid bS$ $S > a \mid b \mid Sa \mid Sb$

Лабораторная работа 4
РАЗРАБОТКА ПРОСТЕЙШЕГО СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА

Цель работы: научиться создавать простые синтаксические анализаторы, реализующие внутреннее представление выражения в компьютере.

Подготовка к работе

1. Изучить соответствующие разделы в изданиях [3, 15].
2. Ознакомиться с теоретическим материалом, изложенным в подразд. 3.6 данного учебного пособия.

Задание. Написать программу-аналог калькулятора.

Входные данные

Произвольная строка символов (арифметическое выражение).

Выходные данные

1. Заключение о корректности входной строки как арифметического выражения.
2. Вычисленное значение этого выражения.
3. Информация о типе ошибки в случае некорректной входной строки (например, «несоответствие количества (и)»).

Синтаксический анализатор должен распознавать следующие лексемы:

1. Знаки арифметических операций: '+', '-', '*', '/' .
2. Скобки '(' и ')' .

3. Действительные числа (т. е. дробные). Например, 12, 66.6, 0.54, 221.

4. Имя встроенной функции (например, sin(x) или max(a, b)).

Выполняемые функции:

- вычислить значение введенного выражения с учетом приоритета операций. При этом необходимо обрабатывать исключительные ситуации (например, деление на 0);
- реализовать поддержку встроенной функции. При этом аргументы функции являются выражениями (в том числе содержат эту функцию);
- реализовать возможность сравнительно простого изменения встроенной функции по требованию преподавателя (например, заменить log(a, b) на pow(a, b)). Встроенная функция имеет наивысший приоритет. Скобки служат для изменения порядка действий.

Лабораторная работа № 5
РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА ЗАДАННЫХ КОНСТРУКЦИЙ ЯЗЫКА С

Цель работы: научиться создавать синтаксический анализатор, обнаруживающий наибольшее число ошибок.

Подготовка к работе

1. Изучить соответствующие разделы в изданиях [3, 9, 12].

2. Ознакомиться с теоретическим материалом, изложенным в подразд. 3.7 данного учебного пособия.

Задание. Написать синтаксический анализатор, обнаруживающий наибольшее число ошибок, для приведенной ниже грамматики (данная грамматика является упрощенным вариантом грамматики языка С):

```
<program>: <type> 'main' '(' ')' '{' <statement> '}'
<type>: 'int'
  | 'bool'
  | 'void'
<statement>:
    | <declaration> ';'
    | '{' <statement> '}'
    | <for> <statement>
```

```
    | <if> <statement>
    | <return>
<declaration>: <type> <identifier> <assign>
<identifier>: <character><id_end>
<character>: 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' |
 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' |
 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' |
 'z' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' |
 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' |
 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' |
 '_'
<id_end>:
    | <character><id_end>
<assign>:
    | '=' <assign_end>
<assign_end>: <identifier>
    | <number>
<number>: <digit><number_end>
<digit>: '0' | '1' | '2' | '3' | '4' | '5' | '6' |
         '7' | '8' | '9'
<number_end>:
    | <digit><number_end>
<for>: 'for' '(' <declaration> ';' <bool_expression>
';' ')'
<bool_expression>: <identifier> <relop> <identifier>
    | <number> <relop> <identifier>
<relop>: '<' | '>' | '==' | '!='
<if>: 'if' '(' <bool_expression> ')'
<return>: 'return' <number> ';'
```

Нетерминалы заключены в < >, терминал (или последовательность терминалов) — в ". Символ | обозначает альтернативу (т. е. логическое «или»). Запись <N1><N2> означает конкатенацию нетерминалов, а запись <N1> <N2> — последовательную запись нетерминалов, разделенных пробельными символами (пробельные символы аналогичны таковым в грамматике языка C). Запись вида <N1>: | <N2> означает, что нетерминал N1 можно заменить либо нетерминалом N2, либо пустой строкой. Стартовым нетерминалом грамматики является <program>.

Назовем язык, описываемый данной грамматикой, C-light.

Входные данные

Файл с программой на языке C-light.

Выходные данные

1. Количество обнаруженных ошибок.
2. Сообщения о месте их возникновения и о их типе.
3. Файл с таблицами, аналогичными табл. 3.6—3.7, приведены в подразд. 3.7 данного учебного пособия.

Восстановление после ошибок реализовать в «режиме паники».

В качестве дополнения может быть реализован механизм восстановления после ошибок в «режиме фразы».

Лабораторная работа № 6
РАЗРАБОТКА ИНТЕРПРЕТАТОРА

Цель работы: изучить свойства интерпретаторов. Научиться создавать простой интерпретатор для условного языка.

Подготовка к работе

1. Изучить соответствующие разделы в изданиях [16].
2. Ознакомиться с теоретическим материалом, изложенным в подразд. 5.1 данного учебного пособия.

Задание. Создать интерпретатор языка CBAS.

Входные данные

Исходный текст программы на языке CBAS.

Выходные данные

Консоль с результатами ввода-вывода исходной программы либо сообщения об ошибках, присутствующих в программе.

Лабораторная работа № 7
РАЗРАБОТКА ПРОЦЕССОРА ЯЗЫКА РАЗМЕТКИ ДОКУМЕНТА

Цель работы:

- изучить свойства языков разметки документов;
- освоить разбор управляющих конструкций данных языков;
- научиться создавать внутреннее представление текста, написанного на языке eMark.

Подготовка к работе

1. Изучить соответствующие разделы в изданиях [17, 18].
2. Ознакомиться с теоретическим материалом, изложенным в подразд. 5.2 данного учебного пособия.

Задание. Написать процессор, обрабатывающий документы на языке eMark.

Входные данные

Текстовый файл, содержащий документ на языке eMark.

Выходные данные

Консоль с отформатированным текстом в соответствии с управляющими инструкциями либо с сообщениями об ошибках в документе.

Лабораторная работа № 8
РАЗРАБОТКА ЯДРА СКРИПТОВОГО ЯЗЫКА rMANAGER

Цель работы:

- изучить свойства скриптовых языков;
- научиться создавать внутреннее представление программы, написанной на языке сценариев rManager.

Подготовка к работе

1. Изучить соответствующие разделы в изданиях [6, 19].
2. Ознакомиться с теоретическим материалом, изложенным в подразд. 5.3 данного учебного пособия.

Задание. Реализовать ядро скриптового языка rManager.

Входные данные

Файл с исходным кодом сценария.

Выходные данные

Процесс работы робота (желательно в графическом виде) или сообщения об ошибках во входном файле.

Примечание. При выполнении лабораторной работы можно расширить спецификацию языка rManager (например, задать в инициализирующей части возможность создания на минном поле препятствий, добавить в скрипт раздел событий, т. е. функций, которые автоматически вызываются скриптом при разминировании, движении робота по неразминированной клетке и т. п.).

Оглавление