Э. А. Опалева
В. П. Самойленко

Языки программирования и методы трансляции

- ***Грамматика и распознающий автомат – формальные модели описания синтаксиса***
- ***Синтаксический анализ – ядро транслятора, определяющего его основные свойства***
- ***Атрибутные грамматики – простой и удобный способ описания семантики языка***

$$G = (\Sigma, N, S, P)$$

$$A \to \beta$$

$$(\alpha A \beta, \alpha' A \beta') \Rightarrow_T (\alpha\gamma\beta, \alpha'\gamma'\beta')$$

```
OUT1:
        i1 := c11;
Label11: if i1 > c12 then goto Label12;
        i2 := c21;
Label21: if i1 > c22 then goto Label2
        top;
OUT2:
        i1 := i1 + c13; goto Label11; Label12
        i2 := i2 + c23; goto Label21; Label22
        i2 := c21;
Label21: if i1 > c22 then goto Label22;
        top;
```

УЧЕБНОЕ ПОСОБИЕ

bhv®

Э. А. Опалева, В. П. Самойленко

Языки программирования и методы трансляции

Рекомендовано УМО по университетскому политехническому образованию для студентов высших учебных заведений, обучающихся по специальности 220400 (230105) – Программное обеспечение вычислительной техники и автоматизированных систем

Санкт-Петербург
«БХВ-Петербург»
2005

УДК 681.3.068+800.92
ББК 32.973-018.1я73
О-60

Опалева Э. А., Самойленко В. П.

О-60 Языки программирования и методы трансляции. — СПб.: БХВ-Петербург, 2005. — 480 с.: ил.

ISBN 5-94157-327-8

Учебное пособие содержит систематическое изложение теоретических основ перевода и компиляции. Рассмотрены общие вопросы разработки, описания и реализации языков программирования, формальные методы описания синтаксиса и семантики языков программирования, методы синтаксического анализа современных языков программирования. Приводится методика разработки описания перевода и пример использования этой методики для построения атрибутной транслирующей грамматики.

Для студентов и преподавателей вузов

УДК 681.3.068+800.92
ББК 32.973-018.1я73

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. гл. редактора	*Людмила Еремеевская*
Зав. редакцией	*Григорий Добин*
Редактор	*Нина Седых*
Компьютерная верстка	*Татьяны Олоновой*
Корректор	*Ксения Вальская*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Рецензенты:
Бузюков Л.Б., д.т.н., профессор, декан факультета сетей связи, систем коммутации и ВТ, зав. кафедрой цифровой вычислительной техники и информатики Санкт-Петербургского государственного университета телекоммуникаций.

Парфенов В.Г., д.т.н., профессор, декан факультета информационных технологий и программирования, зав. кафедрой информационных систем Санкт-Петербургского государственного университета информационных технологий, механики и оптики.

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 01.06.05.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 38,7.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5 Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-327-8

Оглавление

Введение

В учебных планах подготовки дипломированных специалистов, включенных в Государственный образовательный стандарт подготовки студентов по направлениям 654600 — "Информатика и вычислительная техника" (специальность 220400 — "Программное обеспечение вычислительной техники и автоматизированных систем") и 657100 — "Прикладная математика" (специальность 073000 — "Прикладная математика"), бакалавров по направлениям 552800 — "Информатика и вычислительная техника" и 510200 — "Прикладная математика и информатика" значительное место занимают дисциплины, связанные с изучением теории формальных грамматик, языков и автоматов, методов построения языковых процессоров (компиляторов и интерпретаторов).

Учебный материал, положенный в основу данного учебного пособия, читается авторами с середины 70-х годов в Санкт-Петербургском государственном электротехническом университете (ЛЭТИ им. В. И. Ульянова (Ленина)) в рамках таких дисциплин, как "Теория языков программирования и методы трансляции", "Теория вычислительных процессов и структур", "Теория формальных грамматик и автоматов", "Системное программное обеспечение". Учебное пособие отличается от аналогичных книг тем, что содержит систематическое изложение теоретических основ перевода и компиляции, ориентированное на новые образовательные стандарты и может служить основой базового курса по перечисленным ранее дисциплинам. Наиболее полно содержание данного учебного пособия соответствует учебной программе специальной дисциплины "Теория языков программирования и методы трансляции" Государственного образовательного стандарта, который изучается студентами в VII семестре. К этому времени студенты уже освоили основы информатики, дискретной математики, организации ЭВМ, алгоритмические языки и программирование, структуры данных, поэтому изучение материала данного учебного пособия не должно вызвать у них особых затруднений. Однако учебное пособие может быть полезно не только студентам, изучающим дисциплину "Теория языков программирования и методы трансляции", но и студентам других направлений и специальностей, а также тем, кто самостоятельно собирается создавать несложные языки программирования, в том числе и такие, которые используются в системах автоматизации различных прикладных областей, и разрабатывать языковые процессоры для языков программирования различных уровней сложности.

Издание подобного учебного пособия представляется полезным и своевременным, поскольку имеющаяся литература по теоретическим основам построения языковых процессоров ориентирована в основном на более подготовленных читателей. При изложении материала в учебном пособии широко используются формальные описания. Такой подход, с одной стороны, позволяет наиболее полно описывать алгоритмы, а с другой стороны, прививает студентам навыки, необходимые для самостоятельной работы с научной литературой. Материал учебного пособия излагается по принципу "от простого к сложному", все изучаемые понятия связаны в единую систему, изложение теоретического материала чередуется с рассмотрением большого количества примеров для небольших грамматик, описывающих синтаксические конструкции реальных языков программирования, и экспериментальных грамматик с небольшим числом терминалов, нетерминалов и правил вывода. Каждая глава заканчивается списком вопросов для самоконтроля и заданий для самостоятельной работы, способствующих активному усвоению материала учебного пособия.

Учебное пособие может быть использовано для односеместрового или двухсеместрового курса по теоретическим вопросам проектирования языковых процессоров. Оно состоит из введения и двенадцати глав, разделенных на четыре части.

В первой части книги рассматриваются общие вопросы разработки, описания и реализации языков программирования. Это те вопросы, которыми должен владеть любой высококвалифицированный программист, т. к. их глубокое понимание позволяет создавать более надежные и эффективные программы. Для будущих специалистов по разработке языковых процессоров вопросы, рассматриваемые в первой части, позволят получить прочные знания как о математических моделях, на которых основываются языки программирования, так и о методах и способах их реализации.

Первая часть учебного пособия состоит из двух глав. В *главе 1* приводятся основные парадигмы языков программирования и критерии, наиболее часто используемые для критической оценки существующих и проектируемых языков программирования. В этой главе рассматриваются основные конструкции языков программирования: объекты данных и механизмы их типизации, типы и структуры данных, выражения и управляющие структуры. При рассмотрении основных конструкций языков программирования основное внимание уделяется не синтаксису этих конструкций, а их семантике. В *главе 2* обсуждаются основные методы формального описания синтаксиса языков программирования (форма Бэкуса-Наура и ее модификации, синтаксические диаграммы Вирта), рассматриваются вопросы, связанные с описанием контекстных условий языков программирования. Также в этой главе дается представление о наиболее известных методах формального определения динамической семантики: W-грамматике, операционной, аксиоматической и денотационной семантике.

Вторая часть учебного пособия посвящена формальным методам описания синтаксиса языка. В *главе 3* определяются формальные грамматики и формальные языки, приводится классификация формальных грамматик по Хомскому, рассматриваются эквивалентные преобразования КС-грамматик (исключение бесполезных символов, ε-правил, цепных правил, левой рекурсии), неоднозначность грамматик, свойства КС-языков и свойства детерминированных КС-языков. В *главе 4* дается определение распознающего автомата, рассматриваются типы распознающих автоматов и языки, допускаемые распознающими автоматами. Подробно рассматриваются конечные автоматы и преобразователи, способы их задания, связь автоматных грамматик и конечных автоматов, решение проблемы принадлежности, пустоты и эквивалентности для конечных автоматов. В *главе 5* определяются автоматы с магазинной памятью и их разновидности (расширенные МП-автоматы, детерминированные МП-автоматы), преобразователи с магазинной памятью, рассматривается связь между МП-автоматами и КС-грамматиками. В учебном пособии рассматриваются только те аспекты теории автоматов, которые имеют отношение к построению языковых процессоров, поэтому ряд ее понятий опущен, и, следовательно, учебное пособие нельзя использовать как исчерпывающий курс по теории автоматов. Однако студенты, изучившие материал данного учебного пособия, впоследствии значительно проще воспримут курс теории автоматов, и наоборот: знание основ теории автоматов поможет студентам быстрее освоить материал, изложенный в данном учебном пособии.

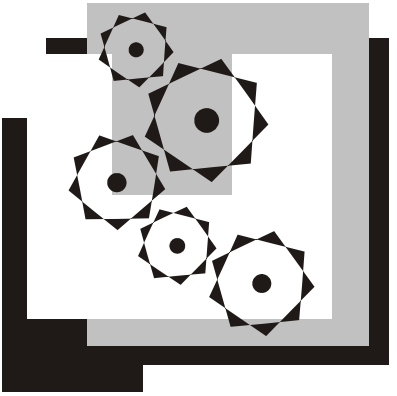
Третья часть учебного пособия посвящена методам синтаксического анализа современных языков программирования. Грамматики, используемые для задания синтаксиса этих языков, позволяют описывать большинство синтаксических конструкций и вместе с тем допускают применение простых детерминированных алгоритмов синтаксического анализа. В *главе 6* даются определение и алгоритмы нисходящего и восходящего разбора, рассматриваются вопросы моделирования работы недетерминированных преобразователей с магазинной памятью, лежащих в основе нисходящих и восходящих синтаксических анализаторов, приводится малоизвестный детерминированный алгоритм Эрли, позволяющий построить правый разбор для входной цепочки, порождаемой произвольной КС-грамматикой. В *главе 7* рассматриваются алгоритмы разбора для $LL(1)$-грамматик ("1-предсказывающий" алгоритм и метод рекурсивного спуска), в *главе 8* — алгоритм разбора "перенос-свертка" и его использование для некоторых подклассов $LR(k)$-грамматик ($LR(0)$- и $SLR(1)$-грамматик). В *главе 9* рассматривается использование алгоритма "перенос-свертка" для грамматик предшествования (простого, слабого, операторного). В конце главы приводится описание языка Флойда-Эванса, ориентированного на разработку синтаксических анализаторов, и рассматривается пример применения этого языка для построения детерминированного восходящего анализатора. При отборе и изложении материала *глав 7—9* большое внимание было уделено не просто методам

синтаксического разбора, а возможности их расширения для описания синтаксически управляемого атрибутного перевода с помощью атрибутных транслирующих грамматик.

В четвертой части учебного пособия излагаются формальные методы описания и реализации синтаксически управляемого перевода. *Глава 10* посвящена промежуточным формам представления программы после этапа синтаксического анализа. В ней рассматриваются как традиционные (польская инверсная запись, тетрады, триады, косвенные триады), так и современные (byte-code JVM) формы представления. В *главе 11* дается формальное определение перевода. В качестве формальных моделей описания перевода рассматриваются схемы синтаксически управляемого перевода, транслирующие грамматики и атрибутные транслирующие грамматики. Приводится методика разработки описания перевода и пример использования этой методики для построения атрибутной транслирующей грамматики. В *главе 12* рассмотрены вопросы разработки и реализации синтаксически управляемого перевода. В этой главе описано, каким образом нужно доработать рассмотренные в третьей части учебного пособия алгоритмы нисходящего и восходящего разбора и каким требованиям при этом должна удовлетворять атрибутная транслирующая грамматика, чтобы можно было выполнять синтаксически управляемый перевод заданным методом.

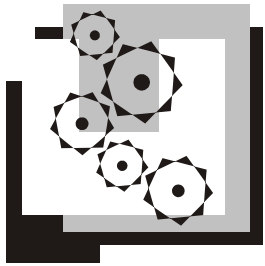
Введение, *главы 1*, *2*, *4*, *5*, *12*, *разд. 7.5*, *10.4*, *11.5—11.6* написаны Эльвирой Александровной Опалевой, а *главы 3*, *6*, *8*, *9*, *разд. 7.1—7.4*, *10.1—10.3*, *11.1—11.4* — Владимиром Петровичем Самойленко. Общее редактирование учебного пособия выполнено Э. А. Опалевой.

В заключение авторы хотят выразить признательность своим родным за их помощь и поддержку в течение всего времени работы над книгой, благодаря которым и смог появиться на свет этот труд. Также хочется выразить благодарность сотрудникам издательства "БХВ-Петербург" Елене Кондаковой, Людмиле Еремеевской и Нине Седых за их терпеливое и доброжелательное отношение в процессе подготовки текста учебного пособия и его корректуры, которые авторам не всегда удавалось делать в срок.



Часть I

Языки программирования



Глава 1

Основные концепции языков программирования

Любую систему обозначений и согласованную с ней систему понятий, которую можно использовать для описания алгоритмов и структур данных, в первом приближении можно считать языком программирования. Однако в настоящем учебнике речь пойдет только об универсальных языках программирования, которые нашли широкое применение при разработке программ в различных областях человеческой деятельности. Каждый из этих языков создавался для определенных целей и имеет свои достоинства и недостатки. Для того чтобы эффективно использовать и реализовывать языки программирования, необходимо хорошо знать фундаментальные понятия, лежащие в основе их построения.

Знание концептуальных основ языков программирования с точки зрения использования и реализации базовых языковых конструкций позволит:

- более обоснованно выбрать язык программирования для реализации конкретного проекта;
- разрабатывать более эффективные алгоритмы;
- систематически пополнять набор полезных языковых конструкций;
- ускорить изучение новых языков программирования;
- использовать полученные знания как методологическую основу для разработки новых языков программирования;
- получить базовые знания, необходимые для разработки трансляторов для языков программирования, поддерживающих разные вычислительные модели.

Используемые программистами языки программирования отличаются как своим синтаксисом, так и функциональными возможностями. Различия в синтаксисе играют незначительную роль при изучении концептуальных основ языка программирования, в то время как наличие или отсутствие тех или иных функциональных возможностей существенно влияет на реализацию и область применения языка.

1.1. Парадигмы языков программирования

На сегодняшний день имеются четыре основные парадигмы языков программирования [36, 38], отражающие вычислительные модели, с помощью которых описывается большинство существующих методов программирования:

- императивная;
- функциональная;
- декларативная;
- объектно-ориентированная.

1.1.1. Императивные языки

Императивные (*процедурные*) *языки* — это языки программирования, управляемые командами, или операторами языка. Основной концепцией императивного языка является состояние компьютера — множество всех значений всех ячеек (слов) памяти компьютера. Данная модель вытекает из особенностей аппаратной части компьютера стандартной архитектуры, названной "фон-неймановской" в честь одного из ее авторов — американского математика Джона фон Неймана. В таком компьютере данные, подлежащие обработке, и программы хранятся в одной памяти, называемой *оперативной*. Центральный процессор получает из оперативной памяти очередную команду на входном языке, декодирует ее, выбирает из памяти указанные в качестве операндов входные данные, выполняет команду и возвращает в память результат.

Программа на императивном языке представляет собой последовательность команд (операторов), которые выполняются в порядке их написания. Выполнение каждой команды приводит к изменению состояния компьютера. Основными элементами императивных языков программирования, ориентированных на фон-неймановскую архитектуру, являются переменные, моделирующие ячейки памяти компьютера, и операторы присваивания, осуществляющие пересылку данных. Выполнение оператора присваивания может быть представлено как последовательность обращений к ячейкам памяти за операндами выражения из правой части оператора присваивания, передача их процессору, вычисление выражения и возвращение результата вычисления в ячейку памяти, представляющую собой переменную из левой части оператора присваивания.

Императивные языки поддерживают, как правило, один или несколько итеративных циклов, различающихся синтаксисом. Итеративные циклы в фон-неймановской архитектуре выполняются быстро за счет того, что команды программы хранятся в соседних ячейках памяти компьютера.

Большинство императивных языков включает в себя конструкции, позволяющие программировать рекурсивные алгоритмы, но их реализация на

компьютерах с фон-неймановской архитектурой не эффективна, что связано с необходимостью программного моделирования стековой памяти.

К императивным языкам относятся такие распространенные языки программирования, как ALGOL-60 [1], BASIC [15, 51], FORTRAN [13, 14], PL/1 [56], Ada [16], Pascal [19, 21], C [57], C++ [53], Java [33, 60].

1.1.2. Языки функционального программирования

В *языках функционального программирования* (*аппликативных* языках) вычисления в основном производятся путем применения функций к заданному набору данных. Разработка программ заключается в создании из простых функций более сложных, которые последовательно применяются к начальным данным до тех пор, пока не получится конечный результат. Типичная программа, написанная на функциональном языке, имеет следующий вид:

$$функция_n\ (\ ...\ функция_2\ (функция_1\ (данные))\ ...\ ).$$

На практике наибольшее распространение получили язык функционального программирования LISP [50, 62] и два его диалекта: язык Common LISP [64] и язык Scheme. Основной структурой данных языка LISP являются связные списки, элементами которых могут быть либо атомы (идентификаторы или числовые константы), либо другие связные списки. При этом список

$$(K\ L\ M\ N)$$

в терминах данных интерпретируется как список из четырех элементов *K*, *L*, *M*, *N*, а в терминах программ — как функция *K* с аргументами *L*, *M* и *N*.

Несмотря на то, что многие ученые в области компьютерных наук указывают на преимущества языков функционального программирования по сравнению с императивными, языки функционального программирования не получили широкого распространения из-за невысокой эффективности реализации их на компьютерах с фон-неймановской архитектурой. На компьютерах с параллельной архитектурой трансляторы для функциональных языков реализуются более эффективно, однако они еще не конкурентоспособны по сравнению с реализациями императивных языков.

Кроме языка LISP, основной областью применения которого являются системы искусственного интеллекта, известны и другие языки функционального программирования: ML (MetaLanguage) [58, 63], Miranda и Haskell [37, 39]. Язык ML наряду с функциональным программированием поддерживает императивное программирование, но, в отличие от императивных языков, функции в языке ML могут быть полиморфными и передаваться между подпрограммами как параметры.

Замечание

Программирование как на императивных, так и на функциональных языках является *процедурным*. Это означает, что программы на этих языках содержат указания, *как* нужно выполнять вычисления.

1.1.3. Декларативные языки

Декларативные языки программирования — это языки программирования, в которых операторы представляют собой объявления или высказывания в символьной логике. Типичным примером таких языков являются *языки логического программирования* (языки, основанные на системе правил).

В программах на языках логического программирования соответствующие действия выполняются только при наличии необходимого разрешающего условия. Программа на языке логического программирования схематично выглядит следующим образом:

разрешающее условие 1 → *последовательность операторов 1*

разрешающее условие 2 → *последовательность операторов 2*

...

разрешающее условие n → *последовательность операторов n*

В отличие от императивных языков, операторы программы на языке логического программирования выполняются не в том порядке, как они записаны в программе. Порядок выполнения операторов определяется системой реализации правил.

Характерной особенностью декларативных языков является их *декларативная семантика*. Основная концепция декларативной семантики заключается в том, что смысл каждого оператора не зависит от того, как этот оператор используется в программе. Так, смысл заданного высказывания в языке логического программирования можно точно определить по самому оператору. Декларативная семантика намного проще семантики императивных языков, что может рассматриваться как преимущество декларативных языков над императивными.

Наиболее распространенным языком логического программирования является язык Prolog [40, 59]. Основными областями применения языка Prolog являются экспертные системы, системы обработки текстов на естественных языках и системы управления реляционными базами данных. К языкам программирования, основанным на системе правил, можно отнести языки синтаксического разбора (например, YACC — Yet Another Compiler Compiler), в которых синтаксис анализируемой программы рассматривается в качестве разрешающего условия.

1.1.4. Объектно-ориентированные языки

Концепция объектно-ориентированного программирования складывается из трех ключевых понятий: *абстракция данных*, *наследование* и *полиморфизм*.

Абстракция данных позволяет *инкапсулировать* множество объектов данных (члены класса) и набор абстрактных операций над этими объектами данных

(методы класса), ограничивая доступ к данным только через определенные абстрактные операции. Инкапсуляция позволяет изменять реализацию класса без плохо контролируемых последствий для программы в целом.

Наследование — это свойство классов создавать из базовых классов производные, которые наследуют свойства базовых классов и могут содержать новые элементы данных и методы. Наследование позволяет создавать иерархии классов и является эффективным средством внесения изменений и дополнений в программы.

Полиморфизм означает возможность одной операции или имени функции ссылаться на любое количество определений функций, зависящих от типа данных параметров и результатов. Это свойство объектно-ориентированных языков программирования обеспечивается динамическим связыванием сообщений (вызовов методов) с определениями методов.

В основе объектно-ориентированного программирования лежит объектно-ориентированная декомпозиция. Разработка объектно-ориентированных программ заключается в построении иерархии классов, описывающих отношения между объектами, и в определении классов. Вычисления в объектно-ориентированной программе задаются сообщениями, передаваемыми от одного объекта к другому.

Объектно-ориентированная парадигма программирования является попыткой объединить лучшие свойства других вычислительных моделей. Наиболее полно объектно-ориентированная концепция реализована в языке Smalltalk [61]. Поддержка объектно-ориентированной парадигмы в настоящее время включена в такие популярные императивные языки программирования, как Ada 95, Java и C++.

1.2. Критерии оценки языков программирования

Каждый из языков программирования, используемых в настоящее время (FORTRAN, Ada, C, Pascal, Java, ML, LISP, Perl, Postscript, Prolog, C++, Smalltalk, Forth, APL, BASIC, HTML, XML), имеет свои преимущества и недостатки, но, тем не менее, все они относительно удачны по сравнению с сотнями других языков, которые были разработаны и реализованы, использовались какое-то время, но так и не нашли широкого применения.

Некоторые успехи или неудачи языка могут быть внешними по отношению к нему. Так, например, использование языка Ada в США для разработки приложений в проектах Министерства обороны было регламентировано Правительством. Аналогично часть успеха языка FORTRAN можно отнести к большой поддержке его различными производителями вычислительной техники, которые потратили много усилий на его реализацию и подробное

описание. Широкое распространение таких языков программирования, как LISP и Pascal, объясняется их использованием в качестве объектов теоретического изучения студентами, специализирующимися в области языков программирования и методов их реализации.

Рассмотрим свойства, которыми должны в той или иной мере обладать языки программирования.

1.2.1. Понятность

Понятность (удобочитаемость) конструкций языка — это свойство, обеспечивающее легкость восприятия программ человеком. Это свойство языка программирования зависит от целого ряда факторов, начиная с выбора ключевых слов и заканчивая возможностью построения модульных программ.

Понятность конструкций языка зависит от выбора такой нотации языка, которая позволяла бы при чтении текста программы легко выделять основные понятия каждой конкретной части программы, не обращаясь к другой документации на программу.

Высокая степень понятности конструкций языка полезна с различных точек зрения:

- уменьшаются требования к документированию проекта, если текст программы является центральным элементом документации;
- понятность конструкций языка позволяет легче понимать программу и, следовательно, быстрее находить ошибки;
- высокая степень понятности конструкций языка позволяет легче сопровождать программу.

Это особенно справедливо для программ с большим жизненным циклом, когда поддержание обновляемой сопроводительной документации в условиях неизбежного множества последовательных модификаций может оказаться весьма трудоемким делом.

Очевидно, что реализация требований понятности конструкций языка во многом зависит от программиста, который должен по возможности лучше структурировать свою программу и так располагать ее текст, чтобы подчеркнуть структуру программы. Вместе с тем важную роль играет синтаксис и структура языка, используемого программистом. Слишком лаконичный синтаксис может оказаться удобным при написании программ, но вместе с тем усложнить их модификацию. Так программы на APL [39] настолько непонятны, что даже авторы спустя несколько месяцев после завершения работы над программой затрудняются в их интерпретации. Понятный язык программирования характеризуется тем, что конструкции, обозначающие разные понятия, выглядят по-разному, т. е. семантические различия языка отражаются в его синтаксисе.

На нижних уровнях программных конструкций язык должен обеспечивать возможность четкой спецификации того, какие объекты данных подвергаются обработке и как они используются. Эта цель достигается выбором идентификаторов и спецификацией типов данных. В язык нельзя вводить такие ограничения, как максимальная длина идентификаторов или только определенные фиксированные типы данных.

Алгоритмические структуры должны выражаться в терминах легко понимаемых структур управления, таких как `if ... then ... else` и т. п. Ключевые слова не следует сводить к аббревиатурам, символы операций должны отображать их смысл (следствием этого является перегрузка операций в языках программирования).

На более высоких уровнях программных конструкций язык должен обеспечивать возможность реализации различных абстракций (определять типы данных, структуры данных и операции над ними), а также разбиения программы на модули и управления областями действия имен.

Как правило, неизбежной платой за выполнение требования высокой степени понятности конструкций языка программирования является увеличение длины программы.

1.2.2. Надежность

Под надежностью понимается степень автоматического обнаружения ошибок, которое может быть выполнено транслятором или операционной средой, в которой выполняется программа. Надежный язык позволяет выявлять большинство ошибок во время трансляции программы, а не во время ее выполнения. Это желательно по двум причинам:

- чем раньше при разработке программы обнаружена ошибка, тем меньше стоимость самого проекта;
- трансляция может быть выполнена на любой машине, воспринимающей входной язык, в то время как тестирование оттранслированной программы должно выполняться на целевой машине либо с использованием программ интерпретации, специально разработанных для тестирования.

Существует несколько способов проверки правильности выполнения программой своих функций: использование формальных методов верификации программ, проверка путем чтения текста программы, прогон программы с тестовыми наборами данных. На практике для проверки правильности программ, как правило, используют некоторую комбинацию этих способов. При этом для обнаружения ошибок во время прогона программы необходимо включать в нее дополнительные операторы вывода промежуточных результатов, которые после отладки программы должны быть удалены.

Принципиальным средством достижения высокой надежности языка, поддерживаемым на этапе трансляции, является *система типизации данных*.

Предположим, что массив целых чисел A, содержащий 100 элементов, индексируется целой переменной i. Надежный язык должен обеспечить выполнение условия 1 ≤ i ≤ 100 в любом месте программы, где встречается A[i]. Это можно сделать двумя способами:

- включить в программу явную проверку значений индекса перед каждым обращением к элементу массива;
- специфицировать область значений при описании переменной i. В этом случае проверка значения переменной i будет выполняться во время присваивания ей значения.

Второй способ является более предпочтительным, т. к. во многих случаях законность изменений значения переменной i может быть проверена во время компиляции программы. Одновременно увеличивается удобочитаемость программы, т. к. область значений, принимаемых каждой переменной, устанавливается явно.

Безусловно, имеется предел числа ошибок, которые могут быть обнаружены любым транслятором. Например, логические ошибки в программе не могут быть обнаружены автоматически. Однако ошибок такого рода будет возникать меньше, если сам язык программирования поощряет программиста писать ясные, хорошо структурированные программы, предоставляя ему возможность выбора подходящих языковых конструкций. Отсюда следует, что надежность языка программирования связана с его удобочитаемостью.

1.2.3. Гибкость

Гибкость языка программирования проявляется в том, сколько возможностей он предоставляет программисту для выражения всех операций, которые требуются в программе, не заставляя его прибегать к вставкам ассемблерного кода или различным ухищрениям.

Как правило, при разработке языка обеспечивается достаточная гибкость для соответствующей выбранной области применения, но не больше.

Замечание

Требование гибкости языка конфликтует с требованием надежности, поэтому при выборе языка программирования для решения конкретной задачи необходимо понимать, на основании каких требований сделан выбор и на какие компромиссы при этом придется идти. Например, критерий гибкости особенно существенен при решении задач в режиме реального времени, где может потребоваться работа с широким спектром нестандартного периферийного оборудования. В тех же случаях, когда сбой в программе может привести к тяжелым последствиям, например при управлении работой атомной электростанции, на первый план выступает такая характеристика языка, как надежность.

1.2.4. Простота

Простота языка обеспечивает легкость понимания семантики языковых конструкций и запоминания их синтаксиса. Простой язык предоставляет ясный, простой и единообразный набор понятий, которые могут быть использованы в качестве базовых элементов при разработке алгоритма. При этом желательно иметь минимальное количество различных понятий с как можно более простыми и систематизированными правилами их комбинирования — язык должен обладать свойством *концептуальной целостности*. Концептуальная целостность языка включает в себя три взаимосвязанных аспекта: экономию, ортогональность и единообразие понятий.

Экономия понятий языка предполагает использование минимального числа понятий.

Ортогональность понятий означает, что между ними не должно быть взаимного влияния. В ортогональном языке любые языковые конструкции можно комбинировать по определенным правилам. Например, выражение и условный оператор некоторого языка программирования ортогональны, если любое выражение можно использовать внутри условного оператора.

Единообразие понятий требует согласованного, единого подхода к описанию и использованию всех понятий.

Для достижения простоты не обязательно избегать сложных языковых конструкций, но следует стараться не накладывать случайных ограничений на их использование. Так, при реализации массивов следует дать возможность программисту объявлять массивы любого типа данных, допускаемого языком. Если же, наоборот, в языке запретить использование массивов некоторого типа, то в результате язык окажется скорее сложным, чем простым, т. к. основные правила языка изучить и запомнить проще, чем связанные с ними ограничения.

Простота уменьшает затраты на обучение программистов и вероятность совершения ошибок, возникающих в результате неправильной интерпретации программистом языковых конструкций. Естественно, упрощать язык можно до определенного предела. Язык высокого уровня с неадекватными управляющими операторами и структурами данных вряд ли можно назвать хорошим.

Наиболее простыми являются языки функционального программирования, т. к. они основаны на использовании одной конструкции — вызова функции, который может легко комбинироваться с другими вызовами функций.

1.2.5. Естественность

Язык должен содержать такие структуры данных, управляющие структуры и операции, а также иметь такой синтаксис, которые позволяли бы отражать в программе логические структуры, лежащие в основе реализуемого алгоритма.

Наличие различных парадигм программирования напрямую связано с необходимостью реализации последовательных, параллельных и логических алгоритмов. Язык, соответствующий определенному классу алгоритмов, может существенно упростить создание программ для конкретной предметной области.

1.2.6. Мобильность

Язык, независимый от аппаратуры, предоставляет возможность переносить программы с одной платформы на другую с относительной легкостью. Это позволяет распределить высокую стоимость программного обеспечения на ряд платформ.

На практике добиться мобильности довольно трудно, особенно в системах реального времени, в которых одна из задач проектирования языка заключается в максимальном использовании преимуществ базового машинного оборудования. Особенно трудно поддаются решению проблемы, связанные с различающимися длинами слов памяти.

На мобильность значительно влияет уровень стандартизации языка. Для языков, имеющих стандартное определение, таких как Ada, FORTRAN, C и Pascal, все реализации языка должны основываться на этом стандарте. Стандарт является единственным способом обеспечения единообразия различных реализаций языка. Международные стандарты разрабатываются Организацией Международных Стандартов (ISO — International Standards Organization). Стандарты обычно создаются для популярных, широко используемых языков программирования.

1.2.7. Стоимость

Суммарная стоимость использования языка программирования складывается из нескольких составляющих. В нее входят:

- стоимость обучения языку;
- стоимость создания программы;
- стоимость трансляции программы;
- стоимость выполнения программы;
- стоимость сопровождения программы.

Стоимость обучения языку определяется степенью сложности языка.

Стоимость создания программы зависит от языка и системы программирования, выбранных для реализации конкретного приложения. Наличие в языке программирования развитых структур данных и конструкций позволяет эффективно использовать язык только при условии его надежной, эффективной и хорошо документированной реализации. Для сокращения времени создания программы необходимо иметь систему программирования, которая включает в себя специализированные текстовые редакторы, тестирующие

пакеты, средства для поддержки и модификации нескольких версий программы, развитый графический интерфейс и др.

Стоимость трансляции программы тесно связана со стоимостью ее выполнения. Совокупность методов, используемых транслятором для уменьшения объема и/или сокращения времени выполнения оттранслированной программы, называется *оптимизацией*. Чем выше степень оптимизации, тем качественнее получается результирующая программа и сильнее уменьшается время ее выполнения, но при этом возрастает стоимость трансляции. Разрешение конфликта между стоимостью трансляции и стоимостью выполнения программы осуществляется на этапе создания транслятора в результате анализа области его применения. Так для трансляторов, разрабатываемых для учебных целей, оптимизация не требуется, в то время как для промышленных трансляторов, с помощью которых создаются многократно используемые программы, требуется высокая степень оптимизации.

Язык программирования, стоимость реализации которого велика, потенциально имеет меньше шансов на широкое распространение. Одной из причин повсеместного распространения языка Java является бесплатное распространение его реализаций сразу после разработки первой версии языка.

Стоимость выполнения программы существенна для программного обеспечения систем реального времени. Системы реального времени должны обеспечивать высокую пропускную способность, чтобы не нарушать ограничений, накладываемых управляемым производственным или технологическим процессом или внешним оборудованием. Поскольку необходимо гарантировать определенное время реакции системы, следует избегать языковых конструкций, ведущих к непредсказуемым издержкам времени выполнения программы (например, при сборке мусора в схеме динамического распределения памяти). Для обычных приложений все снижающаяся стоимость машинного оборудования и все возрастающая стоимость разработки программ позволяют считать, что скорость выполнения программ на сегодняшний день не столь критична.

В *стоимость сопровождения программы* входят затраты на исправление дефектов и модификацию программы в связи с обновлением аппаратуры или расширением функциональных возможностей. Стоимость сопровождения программы в первую очередь зависит от удобочитаемости языка, поскольку сопровождение обычно выполняется лицами, не являющимися разработчиками программного обеспечения.

1.3. Объекты данных в языках программирования

Любая программа для ЭВМ, базирующейся на архитектуре фон Неймана, представляет собой набор операций, которые применяются к определенным данным в заданной последовательности. При программировании на машин-

ном языке необходимо точно знать, как данные представляются в виде последовательности битов в памяти машины и какие машинные команды необходимо использовать для реализации требуемых операций. Язык программирования предоставляет программисту абстрактную модель, в которой объекты данных и операции специфицированы в проблемно-ориентированных терминах. Под объектом данных будем понимать один или несколько однотипных элементов данных, объединенных в одно целое. Объект данных называется *элементарным*, если представляющее его значение является единым целым. В противном случае, если объект данных представляет собой совокупность некоторых других объектов, будем называть его *структурным*. Рассмотрим более подробно вопросы, связанные с определением и использованием объектов данных в языках программирования.

1.3.1. Имена

Имя (*идентификатор*) — это строка символов, используемая для обозначения некоторой сущности в программе. Такими сущностями могут быть переменные, типы, метки, подпрограммы, формальные параметры и другие конструкции языков программирования. В общем случае идентификаторы не имеют какого-либо смысла, а используются только в качестве имен программных объектов или их атрибутов.

В большинстве языков программирования идентификатор представляет собой конечную последовательность букв и цифр, которая начинается с буквы и может содержать соединительный символ подчеркивания '_'. В некоторых языках (C, C++, Java) различаются строчные и прописные буквы. При таком подходе увеличивается пространство возможных имен, но ухудшается надежность языка.

Ключевые слова — это имена, имеющие особое значение только в определенном контексте, например: **`begin`**, **`end`**, **`if`**. В некоторых языках программирования (Pascal, Ada, C, C++, Java) ключевые слова являются зарезервированными, т. е. не могут использоваться в качестве имен. В других языках (FORTRAN, PL/1, APL, BASIC) разрешается переопределять ключевые слова. Возможность переопределения ключевых слов ухудшает надежность и удобочитаемость языка, усложняет процесс компиляции, поэтому при разработке нового языка лучше объявлять ключевые слова зарезервированными.

Многие языки программирования содержат *предопределенные* имена, имеющие конкретный смысл, но не являющиеся ключевыми словами. Это, как правило, имена встроенных типов данных или функций. Для того чтобы компилятор мог правильно определять атрибуты предопределенных имен, их объявления должны быть видимы компилятору. В некоторых языках программирования, например в C и C++, программисты могут использовать имена, предопределенные в библиотеках. Доступ компилятора к таким именам возможен через соответствующие заголовочные файлы.

Предопределенные имена можно переопределять. Например, в языке Pascal идентификатор `sin` считается именем функции, значение которой равно синусу ее аргумента. Программист может определить в своей программе другой идентификатор `sin`, но в этом случае предопределенная в языке функция `sin` будет не доступна. Так же, как и в случае переопределения ключевых слов, переопределение предопределенных имен в большинстве случаев нежелательно.

1.3.2. Константы

Константа — это объект данных, имя которого связано со значением (значениями) в течение всего времени жизни. В языках программирования используются константы двух видов:

- литералы;
- именованные константы.

Литерал представляет собой буквальную запись значения константы. Например, 25 — это десятичная форма записи целочисленной константы, представляющей собой объект данных со значением 25. Форма записи значений литералов предопределенного типа задается в языке.

Именованная константа (константа, определяемая программистом) — это объект данных, который связывает имя с буквальным значением константы. Значения именованных констант известны во время компиляции, поэтому компилятор будет обнаруживать все ошибки, связанные с попыткой присвоения именованной константе нового значения.

1.3.3. Переменные

Переменная — это объект данных, который явным образом определен и именован в программе. *Простая переменная* — это именованный элементарный объект данных. Переменные можно характеризовать с помощью следующих атрибутов:

- имя;
- адрес;
- значение;
- тип;
- время жизни;
- область видимости.

Имя переменной — это идентификатор, используемый в программах для ссылки на значение переменной. Связывание объекта данных с одним или несколькими именами, с помощью которых можно ссылаться на объект

данных, осуществляется при помощи объявлений и может изменяться при входе и выходе из подпрограмм (блоков).

Переменная представляет собой абстракцию области памяти — ячейки или совокупности ячеек памяти компьютера. *Адрес* переменной — это адрес области памяти, с которой связана данная переменная. Для связывания с переменой свободная область памяти соответствующего размера извлекается из пула доступной памяти. Этот процесс называется *выделением* памяти. Разрыв связи между некоторой областью памяти и переменной называется *освобождением* памяти. При освобождении памяти ее область, с которой разрывается связь, возвращается обратно в пул доступной памяти. Выделение и освобождение памяти выполняется специальными *программами управления памятью*, которые недоступны программисту.

Во многих языках программирования одно и то же имя можно связать с разными адресами памяти. Например, в программе могут быть определены две подпрограммы `sub1` и `sub2`, в каждой из которых определяется переменная с одним и тем же именем, например `temp`. В этом случае имя `temp` определяет две *разные* переменные, которые будут связаны с разными областями памяти. Аналогичным образом решается проблема определения переменных в блочных структурах, когда переменная с одним и тем же именем описана во внешнем и внутреннем блоках.

Возможен случай, когда несколько имен переменных связаны с одной и той же областью памяти. Такие переменные называются *альтернативными* (alias-именами).

Например, в языке FORTRAN альтернативные имена могут создаваться с помощью оператора `EQUIVALENCE`, а в языках Ada и Pascal — с помощью вариантных записей. Альтернативные имена широко используются в языках манипулирования данными в системах управления базами данных для получений виртуальных копий одной и той же таблицы. Альтернативные имена экономят память, позволяя связывать с одной областью памяти несколько переменных. Однако такой подход ухудшает удобочитаемость и надежность программ, создает серьезные трудности при верификации и модернизации программ, содержащих альтернативные имена.

Значение переменной — это содержимое ячейки или совокупности ячеек памяти (определенная комбинация битов), связанных с данной переменной. Связывание переменной со своим значением осуществляется в процессе выполнения программы, обычно в результате выполнения оператора присваивания. Переменная сохраняет присвоенное ей значение до тех пор, пока этой переменной не будет присвоено новое значение, при этом предыдущее значение переменной теряется безвозвратно. С каждой переменной связывается определенный тип значений, которые она может принимать.

Тип переменной связывает переменную с множеством значений, которые она может принимать. Переменная должна быть связана с определенным типом до того, как к ней можно будет обращаться.

Время жизни переменной — это время, в течение которого переменная связана с определенной областью памяти (*более подробную информацию см. в разд. 1.5*).

Область видимости переменной — это последовательность операторов программы, из которых можно обратиться к этой переменной (*более подробную информацию см. в разд. 1.6*).

1.4. Механизмы типизации

Типы могут определяться статически и динамически. При *статическом* определении типа связывание осуществляется при трансляции программы, а при *динамическом* — во время выполнения программы.

1.4.1. Статические и динамические типы данных

Статическое определение типа может быть выполнено путем явного или неявного объявления переменных. *Явное объявление* имеет место, если в программу включен специальный оператор объявления типа, устанавливающий тип для переменных, перечисленных в операторе. *Неявное объявление* связывает переменные с типами посредством принятых по умолчанию соглашений. При этом первое появление имени в программе расценивается как объявление переменной.

Большинство языков программирования требует явного объявления переменных. В случае неявного объявления в языке должны быть заданы правила, с помощью которых определяется тип переменных. Например, в программах на языке FORTRAN, если идентификатор не был объявлен явно, по умолчанию считается, что он имеет тип `INTEGER`, если он начинается с одной из букв I, J, K, L, M или N; в противном случае идентификатор является именем переменной типа `REAL`. В языке Perl имена, начинающиеся с символа '@', являются именами массивов, а идентификаторы, начинающиеся с символа '$', могут именовать только переменные числового и строкового типов. Неявные объявления ухудшают надежность программ, т. к. необъявленные по ошибке переменные будут неправильно связаны с типами, устанавливаемыми по умолчанию, что может привести к непредсказуемым ошибкам, которые сложно обнаружить.

При динамическом связывании переменная связывается с типом в момент присваивания переменной значения. При этом тип переменной будет идентичен типу присваиваемого значения. Основным преимуществом динамического связывания переменных с типом является то, что в этом случае обеспечивается высокая гибкость языка. Например, программист может написать программу сортировки элементов массива, который может содержать данные любого типа. Переменные, предназначенные для хранения элементов массива, будут связываться с соответствующим типом во время

ввода данных. Классическим примером динамического связывания переменных с типом является язык APL, в котором корректной является следующая последовательность операторов:

```
A ← 1, 100, 1000
A ← 2.5
```

Независимо от предыдущего значения типа переменная `A` после выполнения первого оператора присваивания будет представлять собой целочисленный массив, содержащий 3 элемента: 1, 100 и 1000. В результате выполнения второго оператора присваивания переменная `A` станет простой переменной, содержащей вещественное число 2,5.

В языках функционального программирования (ML, Miranda, Haskell) распространен механизм логического вывода типа. Например, в языке ML объявление функции:

```
Fun circ_length (r) = 6.28318 * r;
```

определяет функцию, аргумент и результат которой имеют вещественный тип. Вещественный тип логически выводится из типа константы, входящей в выражение.

Динамическое связывание типов имеет ряд недостатков:

- снижается возможность обнаружения транслятором ошибок по сравнению со статическим связыванием типов, т. к. при динамическом связывании во время трансляции отсутствует информация о типе переменных;
- при реализации динамического связывания вся информация о типах переменных должна сохраняться в течение всего времени работы программы, что требует значительных дополнительных ресурсов памяти, связанных с необходимостью хранить данные различных типов;
- динамическое связывание типов приводит к увеличению времени работы программы за счет программной реализации механизмов связывания.

Языки программирования с динамическим механизмом связывания типов часто реализуются в виде интерпретаторов в связи со сложностью реализации динамического изменения типов на машинном языке.

Ключевым фактором, обеспечивающим необходимый уровень надежности языка программирования, является механизм типизации, реализованный в языке.

Различают следующие механизмы типизации:

- слабая типизация;
- строгая типизация.

1.4.2. Слабая типизация

В языках программирования со слабой типизацией информация о типе данных используется только для обеспечения корректности программ на машинном уровне.

Выделяют следующие недостатки слабой типизации:

1. Операция, которая может восприниматься компьютером как корректная, может быть некорректной на абстрактном уровне программы.

   Пусть имеется фрагмент программы на языке C:

   ```
   char c;
   c = 7;
   ```

   На машинном уровне символы представляются целыми числами от 0 до 255, поэтому приведенный оператор присваивания корректен. Если программист на абстрактном уровне программы допустил ошибку, собираясь написать `c = '7'`, то при слабой типизации такого рода ошибки не отслеживаются.

2. При работе с разными типами слабо типизированный язык предусматривает автоматическое выполнение операций преобразования типа.

   Рассмотрим фрагмент программы на языке C:

   ```
   int i;
   float x;
    ...
   i = x;
   ```

   Приведенный фрагмент программы синтаксически корректен, но требует преобразования вещественной переменной `x` из ее внутреннего представления с плавающей точкой в целое представление. Если при этом значение `x` выходит за пределы области допустимых целых чисел, реализуемых компилятором, то на этапе выполнения программы будет обнаружена ошибка.

   Если переменные `i` и `k` — целого типа, а `x` — вещественного, то для определения последовательности, в которой будут выполняться операции преобразования типа при выполнении в языке C оператора присваивания

   ```
   k = x - i;
   ```

   необходимо знать, какой порядок преобразования типов определен в компиляторе. Возможны следующие варианты:

   - `x` преобразуется к целому типу;
   - `i` преобразуется к вещественному типу, а затем `x - i` — к целому.

3. Слабая типизация позволяет определять для некоторых типов некорректные операции.

   Например, в языке C можно использовать для целых операндов логические операции. Если длина слова равна 16 битам, оператор `i = (k << 12) || 1` упаковывает значение `k` в четырех левых разрядах `i` и засылает `1` в крайний правый разряд слова.

Замечание

Увеличение гибкости, обеспечиваемое слабой типизацией, приводит к уменьшению удобочитаемости программы и к необходимости дополнительного контроля во время выполнения программы.

1.4.3. Строгая типизация

Строгая типизация является крайне полезным механизмом типизации, который обеспечивает полный контроль типов (статический и динамический). Строгая типизация существенно повышает надежность и ясность программы, т. к. всем операциям обеспечивается корректность на абстрактном уровне языка.

В строго типизированном языке:

- каждый объект данных обладает уникальным типом;
- каждый тип определяет множество значений и множество операций;
- в каждой операции присваивания тип присваиваемого значения и тип переменной, которой присваивается значение, должны быть эквивалентны;
- каждая примененная к объекту данных операция должна принадлежать множеству операций, допустимых для объектов данного типа;
- преобразование типа должно задаваться явно, например `i = (int) x`.

1.4.4. Производные типы

Если в языке имеются только предопределенные типы, то надежность такого языка невелика вследствие малого числа и широкого диапазона значений предопределенных типов. Существенный результат при разработке механизма строгой типизации можно получить путем введения в язык типов данных, определенных пользователем.

Тип данных можно рассматривать как *факторизацию* определенных свойств, являющихся общими для некоторого класса объектов. Если множество типов данных ограничено предопределенными типами, то все объекты, которые представлены, например, вещественным типом, должны принадлежать

одному и тому же классу, что может не соответствовать смыслу и использованию таких объектов в программе.

Программа на языке Pascal, читающая текущее значение веса товара и вычисляющая суммарный вес десяти товаров, представлена в листинге 1.1.

Листинг 1.1

```
program sum (input, output);
  var
    temp_weight, sum_weight: integer;
    i: integer;
begin
  sum_weight := 0;
  for i := 1 to 10 do
  begin
    read (temp_weight);
    sum_weight := sum_weight + temp_weight
  end;
  writeln (sum_weight)
end.
```

В этой программе переменные `temp_weight`, `sum_weight` и `i` являются *целыми*, хотя и принадлежат к двум логически разным классам объектов: вес и индекс. Если по ошибке внутри цикла написать оператор:

```
sum_weight := sum_weight + i;
```

то компилятор не сможет ее обнаружить, т. к. переменные `sum_weight` и `i` одного типа. Для того чтобы компилятор выявлял семантические ошибки такого типа, многие языки программирования разрешают определять в программе собственные (*производные*) типы на основе базовых (*порождающих*) типов. Определим новые типы `weight` (вес) и `index` (индекс), используя базовый тип `integer`. Теперь описанные в программе переменные типа `weight` будут логически отличаться от переменных типа `index`.

С использованием производных типов `weight` и `index` приведенная ранее программа может быть переписана так, как приведено в листинге 1.2.

Введение в язык производных типов дает возможность компилятору обнаруживать ошибки вида `sum_weight := sum_weight + i`. Обнаруживается ли такая ошибка на самом деле, зависит от реализованного в компиляторе метода определения эквивалентности типов. При реализации производных типов необходимо также решить вопрос: какие атрибуты производный тип должен наследовать от порождающего типа.

Листинг 1.2

```
program sum1 (input, output);
  type
    weight = integer;  {вес}
    index = integer;   {индекс}
  var
    temp_weight, sum_weight: weight;
    i: index;
begin
  sum_weight := 0;
  for i := 1 to 10 do
  begin
    read (temp_weight);
    sum_weight := sum_weight + temp_weight
  end;
  writeln (sum_weight)
end.
```

1.4.5. Эквивалентность типов

Возможны два метода определения эквивалентности типов:

- с использованием структурной эквивалентности;
- с использованием именной эквивалентности.

При использовании *структурной эквивалентности* два объекта принадлежат эквивалентным типам, если у них одинаковая структура. Фактически при структурной эквивалентности производные типы являются синонимами для имени порождающего типа, и компилятор должен разрешать присваивания типа `sum_weight := i`, т. к. структурно и `sum_weight`, и `i` представляются как целые. Для производных типов структурная эквивалентность обеспечивает бо?льшую наглядность и ясность программы, не увеличивая ее надежности.

При использовании *именной эквивалентности* два объекта принадлежат эквивалентным типам только в том случае, если они описаны с помощью одного и того же имени типа. При использовании именной эквивалентности компилятор не будет допускать операторы вида `sum_weight := i`. Именная эквивалентность является гораздо более ограничивающей формой строгой типизации, чем структурная.

Преимущества именной эквивалентности заключаются в следующем:

- именная эквивалентность обеспечивает бо?льшую надежность компилятора;

- компилятор упрощается за счет того, что для именной эквивалентности не нужно разрабатывать алгоритмы сравнения шаблонов, необходимые при реализации структурной эквивалентности для сложных структурных типов данных.

Можно указать на следующие недостатки именной эквивалентности:

- необходимость явного определения функций преобразования типа;
- ограничение возможности использования описаний *анонимного* типа (подробно данный вопрос будет рассмотрен в *разд. 1.4.8*).

1.4.6. Наследование атрибутов

Множество атрибутов типа включает в себя:

- множество значений;
- литеральные обозначения констант;
- множество определенных для типа операций.

При использовании структурной эквивалентности производный тип может унаследовать все атрибуты порождающего типа. В случае именной эквивалентности производный тип наследует от порождающего типа множество значений и литеральные обозначения констант, но, как правило, может наследовать не все его операции.

Рассмотрим программу вычисления площади прямоугольника (листинг 1.3).

Листинг 1.3

```
program square_figure (input, output);
  type
    length = real;    {длина}
    square = real;    {площадь}
  var
    a, b: length;
    s: square;
begin
  read (a, b);
  s := a * b;
  writeln (s)
end.
```

Если операция умножения наследуется от вещественного типа, то разумно предположить, что и тип результата операции `a * b` совпадает с типом

операндов, т. е. length. Следовательно, оператор присваивания s := a * b недопустим!

Можно использовать функцию преобразования типа s := square(a * b), но это не логично, т. к. назначение производных типов состоит в запрещении некорректных операций, а не в ограничении корректных. С другой стороны, оператор a := a * b синтаксически корректен, но является бессмысленным с точки зрения размерности.

Замечание

Для обеспечения высокой надежности за счет использования производных типов язык должен обладать *механизмом переопределения существующих операций* и *механизмом определения новых операций*.

В современных языках программирования операции, как правило, *перегружены*, т. е. операции для различных типов операндов имеют одинаковое обозначение. Например, выполнение операции сложения целых чисел отличается от выполнения операции сложения вещественных чисел, хотя имеют одно и то же обозначение — знак '+'. Тип операции в каждом конкретном случае определяется компилятором путем исследования типов операндов и предполагаемого результата.

Обычно операция присваивания и операции отношения "равно" и "не равно" *корректны для любого типа*. Для других же операций необходимо предусмотреть механизм, с помощью которого множество операций, наследуемое от порождающего типа, должно ограничиваться, а затем расширяться за счет определенных пользователем операций.

1.4.7. Ограничения

Для повышения надежности языка обычно ограничивают не только множество операций, наследуемое от базового типа, но и наследуемое множество значений.

Ограничение диапазона можно включить непосредственно в определение производного типа, например:

```
type index = 1..100;
```

В этом случае тип index определяет множество целых значений в диапазоне от 1 до 100.

Ограничение множества значений, наследуемого производным типом, до множества необходимых значений повышает ясность и надежность программы и оптимизирует результирующую программу за счет исключения проверок принадлежности значений переменных заданному диапазону.

Если переменная i описана с использованием типа index, то при компиляции следующего фрагмента программы:

```
sum := 0;
for i := 1 to 100 do sum := sum + a[i];
```

компилятор может не вставлять код для проверки истинности условий $i \geq 1$ и $i \leq 100$ при вычислении элемента массива a[i].

1.4.8. Подтипы

Иногда оказывается логически нецелесообразным вводить новый тип только для введения ограничений. В некоторых языках имеется возможность ограничить диапазон значений объектов из заданного класса без изменения их статуса как членов этого класса, используя понятие *подтипа*. Например, описание в языке Ada

```
sybtype shortint = integer range –128 .. 127
```

определяет подтип shortint. Объекты подтипа shortint могут применяться совместно с объектами типа integer без использования явного преобразования типа.

С точки зрения программиста, единственное различие между производным типом и подтипом заключается в том, что производный тип вводит совершенно новый класс объектов, отличный от всех остальных, в то время как подтип просто накладывает ограничения на некоторый тип.

Реализация же производных типов и подтипов может быть выполнена совершенно по-разному, поскольку производный тип можно представлять на базовом оборудовании наиболее эффективным способом, а подтип должен быть реализован так же, как и порождаемый тип, но с учетом ограничений.

1.4.9. Анонимные типы и подтипы

На практике довольно часто встречаются случаи, когда введение нового типа или подтипа нецелесообразно, поскольку требуется объявить один или два объекта этого типа. Существуют языки программирования, в которых имеется возможность не использовать имена типов (подтипов) при объявлении переменных — можно объявлять переменные *анонимного* типа. Например, в языке Ada переменные перечислимого типа stud_answer (ответ студента) и etalon (эталон) могут быть объявлены одним из следующих способов:

```
var stud_answer, etalon: (yes, no);
```

или

```
var stud_answer: (yes, no);
    etalon: (yes, no);
```

Если компилятор поддерживает структурную эквивалентность типов, то объявления анонимных типов могут обрабатываться точно так же, как и объявления обычных типов, т. к. при проверке эквивалентности типов сравниваются только структуры объектов. Поэтому в случае использования структурной эквивалентности типов оба объявления переменных `stud_answer` и `etalon` являются синтаксически корректными.

Рассмотрим использование анонимных типов в случае именной эквивалентности типов. Первое объявление переменных `stud_answer` и `etalon` связывает их с одним и тем же типом. Но если переменные `stud_answer` и `etalon` описаны раздельно, нет никаких оснований считать, что они имеют один и тот же тип. В этом случае `stud_answer` и `etalon` рассматриваются как переменные уникального анонимного типа, несовместимые ни с какими другими объектами, в результате чего оператор присваивания:

```
stud_answer := etalon;
```

является синтаксически некорректным. Более того, невозможно предложить какой-нибудь способ приведения переменных `stud_answer` и `etalon` к одному типу.

Рассмотрим еще один пример объявлений переменных:

```
var  j: integer 1..10;
     k: integer 1..10;
```

В случае именной эквивалентности типов имеет место неоднозначность объявлений переменных `j` и `k`, поскольку специфицируемые для `j` и `k` типы могут интерпретироваться либо как производные типы с ограничениями, либо как подтипы с ограничениями.

Замечание

Именная эквивалентность существенно ограничивает использование на практике анонимных типов и заставляет программистов вводить новые типы в тех случаях, когда можно было бы обойтись и без них.

1.5. Время жизни переменных

Время жизни переменной начинается в момент связывания ее с определенной областью памяти и заканчивается при разрыве этой связи.

В зависимости от времени жизни все переменные можно разделить на четыре категории:

- статические переменные;
- автоматические переменные;
- явные динамические переменные;
- неявные динамические переменные.

Статические переменные связываются с областью памяти во время трансляции программы и остаются связанными с этой областью до конца выполнения программы. Глобальные переменные и переменные, объявляемые в подпрограммах, которые должны сохранять свое значение между отдельными выполнениями подпрограмм, реализуются как статические переменные.

Автоматическими называются переменные со статическим связыванием типов, которые связываются с областью памяти при обработке операторов объявления переменных во время выполнения программы. Память автоматическим переменным выделяется из стека. Обычно в языках программирования автоматическими являются локальные переменные, объявленные в процедурах или блоках. Время жизни автоматических переменных начинается в момент обработки операторов объявления переменных во время выполнения программы и заканчивается при выходе из блока или при возврате подпрограммой управления вызывающему оператору.

Явные динамические переменные — это переменные, которые связываются с соответствующей областью памяти во время выполнения программы при обработке оператора создания новых программных объектов, такого, как оператор `new` в языке C++, использующего в качестве операнда тип создаваемой переменной. Так же, как и автоматические переменные, явные динамические переменные связываются с типом статически. При выполнении оператора создания новой переменной создается явная динамическая переменная, имеющая тип операнда, и возвращается *указатель* на нее.

В некоторых языках программирования имеются операторы уничтожения явных динамических переменных, например оператор `delete` в языке C++. В других языках, например в Java, освобождение памяти, занятой явными динамическими переменными, производится во время *сборки мусора* — специальной процедуры освобождения недоступных областей памяти, выполняемой в случае, если весь пул доступной памяти исчерпан.

Таким образом, началом времени жизни явных динамических переменных является время выполнения оператора создания такой переменной, а концом — время разрыва связи между динамической переменной и соответствующей областью памяти.

Использование указателей, содержащих адреса динамических переменных, связано с двумя проблемами:

- наличие висячих указателей;
- потерянные динамические переменные (мусор).

Висячий указатель — это указатель, содержащий адрес динамической переменной, уже удаленной из памяти. Висячие указатели появляются в случае, когда явно удаляются из памяти динамические переменные, на которые ссылалось несколько указателей (при удалении динамической переменной удаляется только один указатель, сформированный при ее создании). Они

могут привести к непредсказуемым ошибкам, включая сбои в работе программы управления памятью. Поскольку висячие указатели не могут быть обнаружены при трансляции, в некоторых языках, например в Java, отсутствуют возможности явного уничтожения указателей.

Потерянная динамическая переменная (*мусор*) — это размещенная в памяти динамическая переменная, недоступная из прикладной программы. Такие переменные возникают, когда динамические переменные находятся в памяти, а пути доступа к ним (указатели) уничтожены.

Неявные динамические переменные — это переменные, которые связываются с соответствующей областью памяти только при присвоении им значений. Время жизни неявной динамической переменной определяется с момента присваивания этой переменной значения до момента присваивания ей нового значения.

1.6. Область видимости переменных

Правила видимости переменных определяют, каким образом ссылки на переменные, объявленные вне выполняющейся в данный момент подпрограммы (блока), связаны с объявлениями этих переменных. Такое связывание называется *ассоциацией* и может быть представлено в виде пары, состоящей из идентификатора и связанного с ним объекта данных. Рассмотрим типовой процесс создания, использования и уничтожения ассоциаций.

1. В начале выполнения главной программы (главной функции) имя каждой объявленной в ней переменной связывается с конкретным объектом данных (областью памяти для хранения значения переменной), а каждое имя подпрограммы, вызываемой в главной программе, — с конкретным определением подпрограммы.
2. При выполнении главной программы с помощью *операций обработки ссылок* для каждого идентификатора определяется ассоциированный с ним объект.
3. При обработке вызова каждой подпрограммы создается новое множество ассоциаций: имена локальных переменных, объявленных в подпрограмме, и имена формальных параметров связываются с конкретными объектами данных.
4. При выполнении подпрограммы операции обработки ссылок определяются конкретные объекты, ассоциированные с каждым идентификатором. При этом ссылки могут быть как ссылками на ассоциации, созданные при входе в подпрограмму, так и на глобальные ассоциации.
5. При выходе из подпрограммы созданные в ней ассоциации уничтожаются.

6. Когда управление возвращается в главную программу, доступными становятся только глобальные ассоциации.

Такая модель позволяет программе (подпрограмме) иметь множество ассоциаций, доступных для разрешения ссылок во время их выполнения, а программисту — управлять памятью.

Среда ссылок (множество ассоциаций подпрограммы) обычно не изменяется при выполнении подпрограммы и включает в себя:

- *среду локальных ссылок* (*локальную среду*), в которую входят все ассоциации, созданные при передаче управления данной подпрограмме. В нее включаются формальные параметры, локальные переменные и подпрограммы, *определенные в данной подпрограмме*;
- *среду нелокальных ссылок*, содержащую ассоциации, которые могут использоваться в подпрограмме, но были созданы *до момента входа в данную подпрограмму*;
- *среду глобальных ссылок*, созданную в начале выполнения главной программы и являющуюся частью среды нелокальных ссылок;
- *среду предопределенных ссылок*, задаваемую непосредственно в определении языка, которую можно использовать в любой программе или подпрограмме.

Листинг 1.4 иллюстрирует понятие среды ссылок на примере схемы программы на языке Pascal.

Листинг 1.4

```
program main;
   var A, B, C: integer;
   procedure Sub1 (A: integer);
      var C, D: integer;
      procedure Sub11 (C: integer);
         var D: integer;
         begin {Sub11}
            <операторы Sub11>
      end; {Sub11}
      begin {Sub1}
         <операторы Sub1>
   end; {Sub1}
   procedure Sub2 (B: integer);
      var A: integer;
      procedure Sub21 (A: integer);
         var D: integer;
```

```
      Среда ссылок для Sub1
Локальные:     A, C, D, Sub11
Нелокальные:   B, Sub1 из main
     Среда ссылок для Sub11
Локальные:     C, D
Нелокальные:   A, Sub1 из Sub1,
               B из main

      Среда ссылок для Sub2
Локальные:     A, B, Sub21, Sub22
Нелокальные:   C, Sub2 из main
     Среда ссылок для Sub21
Локальные:     A, D
```

```
        begin {Sub21}
            <операторы Sub21>
      end; {Sub21}
      procedure Sub22 (D: integer);
         var A, B, C: integer;
         begin {Sub22)}
            <операторы Sub22>
      end; {Sub22}
   end; {Sub2}
   begin
      <операторы main>
end.
```

```
Нелокальные:   B, Sub21 из Sub2,
               C из main

     Среда ссылок для Sub22
Локальные:     A, B, C, D
Нелокальные:   Sub22 из Sub2
```

Каждый идентификатор в этой программе объявлен несколько раз. Например, идентификатор `A` именует:

- переменную в программе `main`;
- формальный параметр процедуры `Sub1`;
- переменную в процедуре `Sub2`;
- формальный параметр процедуры `Sub21`;
- переменную в процедуре `Sub22`.

В листинге 1.4 справа от текста программы приведены среды ссылок для процедурных блоков программы. Например, среда ссылок для процедуры `Sub2` состоит из локальной среды, включающей в себя формальный параметр `B`, локальную переменную `A` и имена процедур `Sub21` и `Sub22`, объявленных в процедуре `Sub2`. Нелокальная (в данном случае глобальная) среда ссылок для процедуры `Sub2` включает в себя глобальную переменную `C` и имя объявленной в программе `main` процедуры `Sub2`.

Среда предопределенных ссылок для данной программы не показана (она рассматривается как блок, внешний по отношению к самому внешнему блоку программы). В языке Pascal она состоит из предопределенных констант (например, `MAXINT`) и имен подпрограмм (например, `sin`, `sqrt`, `read` и `write`). Для каждого из этих предопределенных имен можно создать новую ассоциацию с помощью явного объявления, сделав невидимой для части программы или всей программы предопределенную ассоциацию.

В связи с тем, что множества глобальных и локальных идентификаторов, именующих величины различного вида, могут пересекаться, т. е. входить в различные ассоциации и среды ссылок, возникает *проблема разрешения ссылок* — связывание идентификатора и значения соответствующей величины. Эта проблема решается введением в язык понятия *области видимости* [37, 39].

Замечание

Ассоциация для идентификатора *видима* в подпрограмме, если она является частью ее среды ссылок. Ассоциация для идентификатора *скрыта* от подпрограммы, если она не входит в среду ссылок выполняющейся в данный момент времени подпрограммы.

Если при входе в подпрограмму некоторый идентификатор переопределяется, то ассоциация, в которую он входил до этого момента времени, становится скрытой.

Различают статическую и динамическую области видимости имен.

Статическая область видимости — это область видимости, которая может быть определена во время трансляции программы. Большинство статических областей видимости связано с определением подпрограмм и блоков.

Для определения статических областей видимости имен удобно пользоваться графическим представлением структуры программы. На рис. 1.1 приведен граф программы из листинга 1.4. На этом рисунке узлы графа помечены именами блоков программы, при этом в скобках указаны идентификаторы, объявленные в данном блоке.

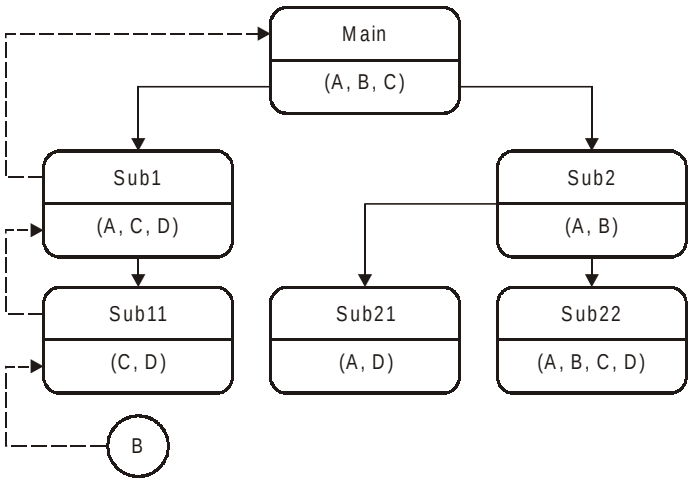


Рис. 1.1. Поиск ассоциации для идентификатора B

Если нам необходимо определить, например, ассоциацию для идентификатора `B`, встречающегося в теле процедуры `Sub11`, то для этого нужно выполнить следующие действия:

1. Определить, является ли идентификатор `B` локальной переменной или формальным параметром процедуры `Sub11`. Так как `B` не удовлетворяет

этому условию, то необходимо перейти в охватывающий блок — к процедуре Sub1.

2. Определить, является ли идентификатор B локальной переменной или формальным параметром процедуры Sub1. Имя B не является именем локальной переменной или формального параметра Sub1, следовательно, переходим к блоку, в котором описана процедура Sub1.
3. Имя B — это имя глобальной переменной программы, и в процедуре Sub11 используется именно эта ассоциация.

Правила статической области видимости позволяют определить множество различных типов связей между ссылками на имена и их объявлениями только один раз во время трансляции программы. К таким типам относятся:

- связывание имени переменной с объявлением переменной;
- связывание имени константы с объявлением константы;
- связывание имени типа с объявлением типа;
- связывание формального параметра со спецификацией формального параметра;
- связывание вызова подпрограммы с объявлением подпрограммы;
- связывание метки оператора, используемой в операторе перехода, с меткой конкретного оператора.

В каждом из перечисленных случаев во время трансляции могут быть сделаны упрощения, повышающие эффективность выполнения программы.

Правила статической видимости имен упрощают также понимание программы, т. к. позволяют связать каждое имя, используемое в программе, с объявлением этого имени без необходимости отслеживания процесса выполнения программы.

Несмотря на очевидные преимущества статических областей видимости имен, этот метод нелокального доступа к объектам программы имеет ряд недостатков:

1. Можно по ошибке вызвать подпрограмму, вызов которой не должен допускаться. Это будет обнаружено только при выполнении программы, что повышает стоимость исправления ошибки.
2. При использовании статической области видимости имеет место слишком интенсивное обращение к данным. Например, все переменные, объявленные в главной программе, видимы для всех подпрограмм.
3. Предположим, что после разработки и тестирования программы потребовалось внести в нее изменения, например, в программе из листинга 1.4 (см. рис. 1.1) необходимо открыть доступ из подпрограммы Sub11 к некоторым переменным из области видимости подпрограммы Sub22. Для

решения данной проблемы можно перенести объявление подпрограммы Sub11 внутрь области видимости подпрограммы Sub22. Однако в этом случае из подпрограммы Sub11 уже нельзя будет обращаться к области видимости подпрограммы Sub1, что, по-видимому, неправильно. Другим решением является объявление переменных, объявленных в подпрограмме Sub22 и необходимых в подпрограмме Sub11, в главной программе main. Это позволит обращаться к этим переменным из любой подпрограммы, что может привести к некорректному доступу. Например, неверно написанный идентификатор подпрограммы может быть воспринят транслятором не как ошибка, а как ссылка на идентификатор из области видимости главной программы.

4. При объявлении всех переменных в главной программе значительно ухудшается такая ее характеристика, как удобочитаемость. Это объясняется тем, что объявления переменных находятся на значительном расстоянии от места их использования.

Динамическая область видимости имен, реализованная в таких языках программирования, как APL и Snobol, представляет собой множество активаций подпрограмм, в которых ассоциации идентификаторов видимы *во время выполнения программы*. Динамическая область видимости имен опирается не на пространственную взаимосвязь подпрограмм, а на последовательность их вызовов.

Предположим, что в подпрограмме example (листинг 1.5) к нелокальным ссылкам применяются правила динамической видимости имен.

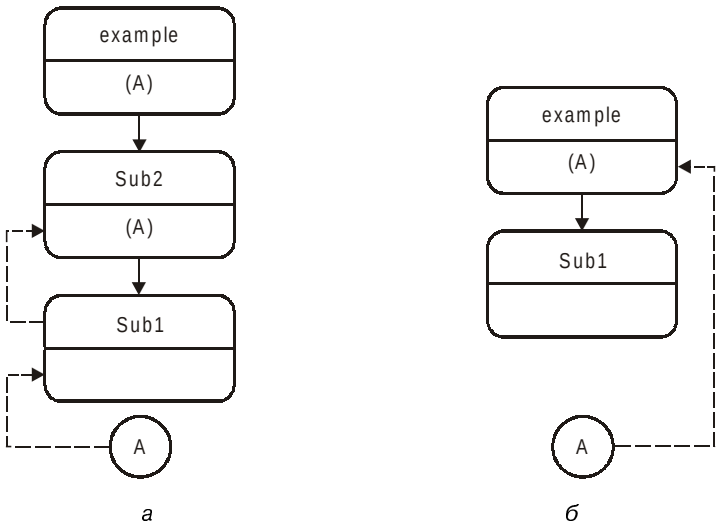
Для того чтобы определить корректное значение переменной A во время выполнения программы, необходимо установить, нет ли идентификатора A среди локальных объявлений. Если поиск среди локальных объявлений окажется неуспешным, следует перейти к рассмотрению объявлений вызывающей процедуры. Если объявление идентификатора A отсутствует в области локальных объявлений вызывающей процедуры, поиск для этой процедуры продолжается в вызывающей процедуре и так далее, пока не будет найдено объявление переменной A. Если в результате выполнения описанной процедуры объявление переменной A не будет обнаружено, то возникнет *ошибка времени выполнения* программы.

Листинг 1.5

```
procedure example;
   var A: integer;
   procedure Sub1;
      begin  {Sub1}
         <операторы Sub1>
   end;   {Sub1}
```

```
    procedure Sub2;
       var A: integer;
       begin  {Sub2}
          <операторы Sub2>
    end;   {Sub2}
    begin  {example}
          <операторы example>
end;
```

Возможные вызовы процедуры Sub1 приведены на рис. 1.2.



а *б*

Рис. 1.2. Возможные вызовы процедуры Sub1

В первом случае процедура example вызывает процедуру Sub2, которая в свою очередь вызывает процедуру Sub1, поэтому обращение к переменной A в процедуре Sub1 является обращением к переменной A, объявленной в процедуре Sub2. Во втором случае процедура Sub1 вызывается непосредственно из процедуры example и обращение будет осуществляться к переменной A, объявленной в процедуре example.

Использование динамической области видимости имен приводит к тому, что подпрограммы *всегда* выполняются в непосредственной среде вызывающей процедуры, поэтому они менее надежны. Обращение к нелокальным переменным в языках с динамической областью видимости имен занимает значительно больше времени, чем в языках со статической областью видимости.

1.7. Типы данных

Тип данных — это некоторый класс объектов данных вместе с набором операций для создания и работы с ними. В каждом языке программирования имеется некоторый набор встроенных *элементарных* типов данных. Дополнительно язык может предоставлять возможности, позволяющие программисту определять новые типы данных.

Различают следующие основные элементы *спецификации* типа данных:

- *атрибуты*, характеризующие объекты данных заданного типа;
- *значения*, которые могут принимать объекты данных заданного типа;
- *операции*, которые допустимы над объектами данных заданного типа.

Если во время выполнения программы атрибуты объекта данных могут изменяться, их текущие значения должны храниться в *дескрипторе* (описателе) данных. Дескриптор данных может быть частью объекта данных или храниться в отдельной области памяти.

Под *реализацией* типа данных понимают:

- *способ представления объектов данных* этого типа в памяти компьютера во время выполнения программы;
- *способ представления операций*, определенных для этого типа данных (комбинация аппаратных и программных средств, реализующих конкретные алгоритмы и процедуры над представлениями объектов данных заданного типа в памяти).

Синтаксис типа данных — это набор синтаксических конструкций для представления основных элементов спецификации типа данных. Атрибуты объектов данных обычно представлены синтаксически через *объявления* или *определения* типов. Значения могут представляться литералами или именованными константами. Вызов операций может осуществляться с помощью специальных символов, встроенных процедур и функций, таких как `sin`, `sqrt` или `read`, или неявным образом через комбинации других элементов языка. Конкретный вид синтаксического представления типа данных не имеет большого значения при изучении концептуальных основ языка.

1.7.1. Элементарные типы данных

Элементарными типами данных называются типы данных, не определяемые в терминах других типов. Простейшими из них являются *скалярные* типы данных. Они характеризуется тем, что у каждого объекта имеется только один атрибут. К скалярным типам данных относятся числовые, логический, символьный и указательный типы данных.

1.7.1.1. Числовые типы

Числовые типы включают в себя следующие типы данных:

- целый тип;
- вещественный тип с плавающей точкой;
- десятичный тип с фиксированной точкой.

Целый тип

Наиболее распространенным числовым элементарным типом является *целый тип*. В настоящее время многие компьютеры поддерживают несколько размеров целых чисел, и эти возможности должны быть реализованы в языках программирования. Например, в языке Ada можно использовать три типа целочисленных значений: `SHORT INTEGER`, `INTEGER` и `LONG INTEGER`, а в языке C — четыре: `int`, `short`, `long` и `char`.

Спецификация. Множество чисел целого типа образует ограниченное упорядоченное подмножество бесконечного множества целых чисел, определенного в математике.

Операции над целочисленными объектами данных можно сгруппировать следующим образом:

- *арифметические операции*:
  - *бинарные* операции (*БинОп*): *сложение*, *вычитание*, *умножение*, *деление*, *деление по модулю*. Спецификация бинарной операции имеет вид:

    $$БинОп: \quad integer_1 \times integer_2 \rightarrow integer_3,$$

    где символом '$\times$' обозначен знак операции;
  - *унарные* операции (*УнарОп*): *операция отрицания*, *операция определения абсолютного значения*. Спецификация унарной операции выглядит следующим образом:

    $$УнарОп: \quad integer_1 \rightarrow integer_2;$$

- *битовые операции* (*БитОп*): *побитовое логическое И*, *побитовое логическое ИЛИ*, *сдвиг битов влево*, *сдвиг битов вправо*. При выполнении битовых операций целые числа рассматриваются как последовательности битов. Спецификация битовых операций имеет вид:

  $$БитОп: \quad integer_1 \times integer_2 \rightarrow integer_3;$$

- *операции сравнения* (*СравОп*): *равно*, *не равно*, *больше*, *меньше*, *больше или равно*, *меньше или равно*. Спецификация операций сравнения имеет вид:

  $$СравОп: \quad integer_1 \times integer_2 \rightarrow Boolean;$$

- *операция присваивания* (*ПрисвОп*) — это операция, изменяющая связывание объекта данных со значением. Спецификация для операции присваивания может быть определена одним из двух способов:
  - установить значение, содержащееся в объекте данных $integer_1$, равным значению, содержащемуся в объекте данных $integer_2$, но не возвращать явный результат (изменение значения объекта $integer_1$ является *побочным эффектом* данной операции).

    $$ПрисвОп: \quad integer_1 \times integer_2 \rightarrow void;$$

  - установить значение, содержащееся в объекте данных $integer_1$, равным значению, содержащемуся в объекте данных $integer_2$, а также создать и вернуть новый объект данных $integer_3$, содержащий копию значения $integer_2$.

    $$ПрисвОп: \quad integer_1 \times integer_2 \rightarrow integer_3;$$

  Первый способ характерен для языка Pascal, а второй способ — для языка C.

Реализация. В большинстве случаев предопределенный в языке целочисленный тип данных реализуется при помощи аппаратного представления целых чисел и элементарных арифметических операций и операций сравнения, реализованных аппаратно. Различные способы представления целых чисел приведены на рис. 1.3.

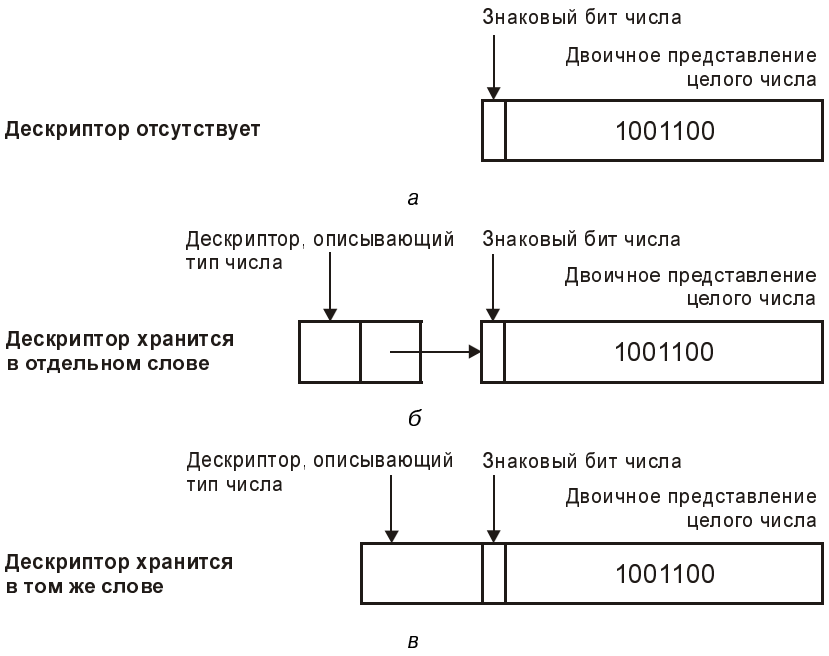


Рис. 1.3. Три способа представления в памяти целых чисел

Вещественный тип с плавающей точкой

Вещественный тип с плавающей точкой включен практически во все языки программирования.

Спецификация. Вещественные числа с плавающей точкой обычно определяются одним атрибутом (`real` или `float`). Множество значений, которое можно представить с помощью вещественных чисел с плавающей точкой, определяется их *точностью* и *диапазоном*. Точность числа определяется числом разрядов мантиссы, а диапазон значений определяется диапазоном изменения экспоненты (от 10^{-38} до 10^{+38} для чисел обычной точности и от 10^{-308} до 10^{+308} для чисел удвоенной точности). В некоторых языках точность, требуемая для представления вещественных чисел, может быть задана программистом количеством значащих цифр в десятичном представлении. Некоторые десятичные числа, например 0.1 или π, не могут быть точно представлены вещественными числами с плавающей точкой, т. к. они не могут быть представлены конечным набором двоичных цифр.

К вещественным числам с плавающей точкой обычно применимы те же операции, что и для целых чисел, за исключением битовых операций. В большинстве языков программирования определены встроенные функции, реализующие широко распространенные операции, такие как `max`, `sqrt`, `sin`. Из-за проблем, связанных с округлением, точного равенства вещественных чисел достичь удается редко, в результате чего разработчик языка может запретить операцию проверки равенства двух вещественных чисел.

Реализация. Представление в памяти компьютера вещественных чисел с плавающей точкой реализуется с помощью аппаратного представления таких чисел. Область памяти, отводимая для хранения вещественных чисел с плавающей точкой, разделяется на две части: *мантиссу* и *экспоненту*. Большинство компьютеров для реализации вещественных чисел с плавающей точкой используют формат, определяемый стандартом IEEE Standard 7754: 32 бита для представления чисел обычной точности и 64 бита для представления чисел удвоенной точности. Формат представления вещественных чисел с плавающей точкой, определяемый стандартом IEEE, приведен на рис. 1.4.

Аппаратное обеспечение большинства компьютеров поддерживает арифметические операции для вещественных чисел с плавающей точкой как обычной, так и удвоенной точности. Операция возведения вещественного числа в целую степень реализуется с помощью операции умножения, а реализация операции возведения вещественного числа в вещественную степень требует программного моделирования.

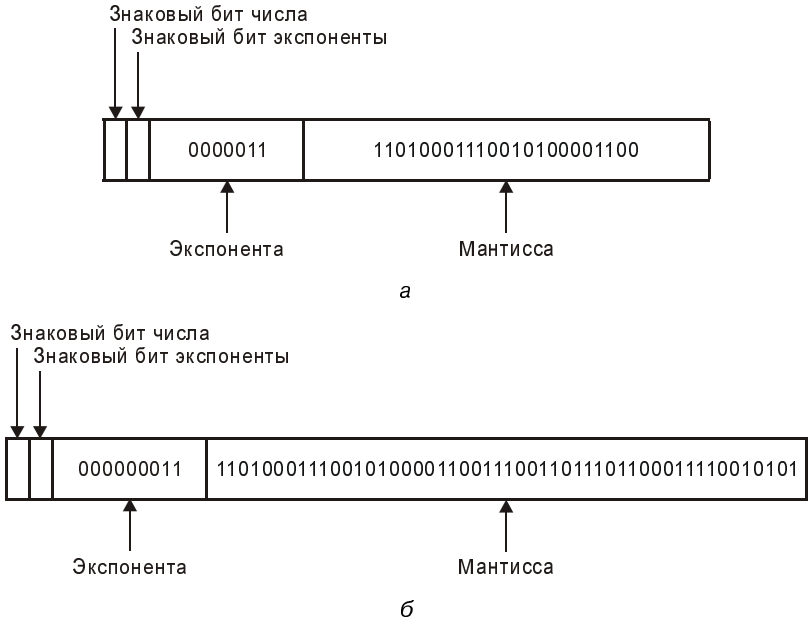


Рис. 1.4. Формат вещественных чисел с обычной (*а*) и двойной (*б*) точностью

Десятичный тип с фиксированной точкой

Выполнение операций над вещественными числами с плавающей точкой связано с ошибками округления, поэтому в коммерческих приложениях широко используется десятичный тип с фиксированной точкой.

Спецификация. Десятичное число с фиксированной точкой представляется как последовательность десятичных цифр фиксированной длины с десятичной точкой, разделяющей целую и дробную части числа. При объявлении десятичного типа с фиксированной точкой местоположение десятичной точки задается явно. Например, в языке PL/1 объявление переменной `X`, содержащей две цифры после десятичной точки, имеет следующий вид:

```
DECLARE  X FIXED DECIMAL (10.2);
```

Реализация. Десятичное число с фиксированной точкой хранится как целое, причем каждая десятичная цифра представляется двоичным четырехразрядным числом. Положение десятичной точки в числе задается атрибутом, называемым *масштабным коэффициентом.*

Арифметические операции и операции сравнения над десятичными числами с фиксированной точкой, как правило, моделируются программно.

Комплексный тип

Комплексный тип обычно включают в языки программирования, ориентированные на решение научно-технических задач, в которых используются комплексные числа. Типичным примером такого языка является FORTAN.

Спецификация. Комплексное число записывается в виде упорядоченной пары вещественных чисел, представляющих его вещественную и мнимую части.

Реализация. Комплексное число обычно хранится как пара вещественных чисел, расположенных в смежных областях памяти.

Операции над комплексными числами моделируются программно.

Рациональный тип

Рациональный тип включается в язык программирования для того, чтобы избежать проблем с округлением и усечением, имеющих место при использовании вещественных типов с плавающей и фиксированной точкой.

Спецификация. Рациональное число обычно имеет вид упорядоченной пары целых чисел неограниченной длины, представляющих его числитель и знаменатель.

Реализация. Рациональное число обычно хранится в виде двух связанных списков, представляющих его числитель и знаменатель. На рис. 1.5 изображено рациональное число 3/2 при условии, что числитель и знаменатель рациональной дроби представлены четырехразрядными двоичными числами.

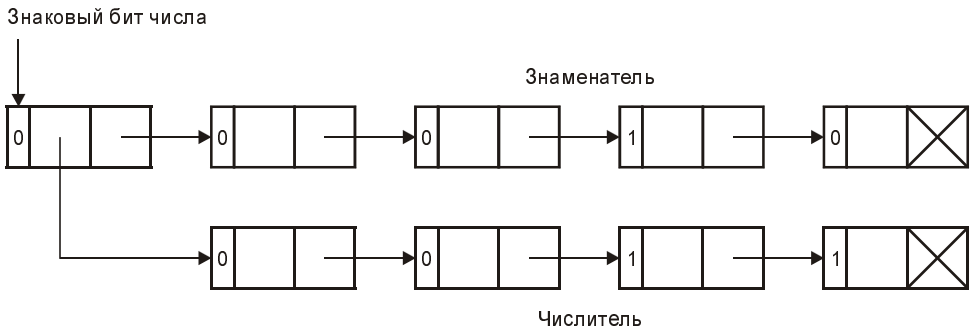


Рис. 1.5. Рациональное число 3/2

Операции над рациональными числами моделируются программно.

1.7.1.2. Логический тип

Логический (булевский) тип данных присутствует в большинстве языков программирования.

Спецификация. Логический тип данных состоит из объектов, которые могут принимать два значения: *истина* (`true`) и *ложь* (`false`), причем `false` < `true`.

Наиболее распространенными операциями для этого типа данных являются операции присваивания:

$$ПрисвОп: \quad Boolean_1 \times Boolean_2 \rightarrow void$$

или

$$ПрисвОп: \quad Boolean_1 \times Boolean_2 \rightarrow Boolean_3$$

и *логические операции* (*умножение*, *сложение* и *отрицание*):

$$ЛогУмн\ (and): \quad Boolean_1 \times Boolean_2 \rightarrow Boolean_3,$$

$$ЛогСлож\ (or): \quad Boolean_1 \times Boolean_2 \rightarrow Boolean_3,$$

$$ЛогОтр\ (not): \quad Boolean_1 \rightarrow Boolean_2.$$

Менее распространены такие логические операции, как *эквивалентность*, *исключающее ИЛИ*, *импликация*, *И-НЕ*, *ИЛИ-НЕ*.

Реализация. Объекты логического типа обычно представляются минимальной адресуемой областью памяти (байтом или словом). Внутри такой области памяти значения *истина* и *ложь* могут быть реализованы одним из двух способов:

- какой-то определенный бит внутри области памяти (например, знаковый бит) используется для представления логического значения (*ложь* = 0, *истина* = 1), а остальные биты игнорируются;
- значению *ложь* соответствуют нули во всех битах отводимой области памяти, а любые другие комбинации единиц и нулей в этой области памяти интерпретируются как значение *истина*.

В некоторых языках программирования, например в языке C, для представления логического типа используются целые числа (*истина* — любое ненулевое значение, а *ложь* — нулевое значение).

Обычно в языках программирования простейшие логические операции (умножение, сложение, отрицание) реализуются аппаратно, а остальные операции моделируются программно.

1.7.1.3. Символьный тип

Символьный тип позволяет создавать объекты данных, значениями которых является одиночный символ.

Спецификация. Множество возможных значений символов обычно задается встроенным в язык программирования перечислением символов, соответствующим стандартному набору (например, ASCII или Unicode). Для символьного типа определены операции присваивания:

$$ПрисвОп: \quad char_1 \times char_2 \rightarrow void$$

или

$$ПрисвОп: \quad char_1 \times char_2 \rightarrow char_3$$

и сравнения:

$$СравОп: \quad char_1 \times char_2 \rightarrow Boolean.$$

Для прямого и обратного отображениий множества символьных значений на подмножество натуральных чисел, являющихся порядковыми номерами этих значений, в некоторые языки программирования (например, в Pascal) встроены функции преобразования, являющиеся обратными по отношению друг к другу:

$$Ord: \quad char \rightarrow integer,$$

$$Chr: \quad integer \rightarrow char,$$

а также функции определения предыдущего (successor) и следующего (predecessor) символов:

$$Pred: \quad char \rightarrow char,$$

$$Succ: \quad char \rightarrow char.$$

Реализация. Символьный тип данных обычно поддерживается аппаратно, т. к. объекты этого типа используются для ввода-вывода.

1.7.1.4. Указатели

Указатели включаются в определение языка с целью обеспечения возможности конструирования произвольных структур данных из объектов разного типа.

Спецификация. Тип данных *указатель* определяет класс объектов данных, значением которых является ссылка на местоположение (адрес) другого объекта данных или пустой указатель (`null`). Имеются два способа определения указателей:

1. Указатели могут ссылаться только на объекты данных одного типа. Такой подход используется в языках программирования, в которых применяется статический контроль и объявления типов (C, Pascal, Ada).
2. Указатели могут ссылаться на объекты данных любого типа в различные моменты выполнения программы. Такой способ используется в языках программирования, в которых реализован динамический контроль типов (например, Smalltalk). Значение типа объекта, на который производится ссылка в текущий момент времени, хранится в дескрипторе.

В некоторых языках (например, в C, C++, Pascal) указатели являются объектами данных, которые можно явно использовать в программе. При этом указатели могут быть как простыми переменными, так и компонентами сложных структур данных. В других языках (например, в Java) указатели являются частью скрытых структур данных, управляемых реализацией языка.

Над указателями определены следующие операции:

- *операция создания* объекта данных фиксированного размера. В результате выполнения этой операции в памяти отводится место для нового объекта,

создается указатель на этот объект данных, которому присваивается значение ссылки (адреса) на этот объект;

- *операция разыменования*. Эта операция использует значение указателя для доступа к объекту данных, на который он ссылается.

Для указателей-переменных, явно употребляемых в программе, обычно можно использовать следующие операции:

- *операция присваивания*. Значение одного указателя присваивается другому указателю. Спецификация операции присваивания имеет вид:

$$ПрисвОп: \quad ptr_1 \times ptr_2 \rightarrow void$$

или

$$ПрисвОп: \quad ptr_1 \times ptr_2 \rightarrow ptr_3;$$

- *операции сравнения равно* и *не равно* для указателей, ссылающихся на объекты одного типа. Спецификация операций сравнения на равенство имеет вид:

$$СравОп: \quad ptr_1 \times ptr_2 \rightarrow Boolean;$$

- *бинарные* арифметические операции (*сложение*, *вычитание*):

$$БинОп: \quad ptr_1 \times ptr_2 \rightarrow ptr_3;$$

- *унарные* арифметические операции (*инкремент* и *декремент*):

$$УнарОп: \quad ptr_1 \rightarrow ptr_2.$$

Реализация. Используются два основных представления значений указателей в памяти компьютера:

- *абсолютный адрес*. Значение указателя представляет собой адрес области памяти, отведенной для объекта данных;
- *относительный адрес*. Значение указателя представляет собой смещение относительно базового адреса области памяти, отводимой для размещения объектов.

Недостатком абсолютной адресации является то, что управление памятью становится более сложным, т. к. ни один объект данных не может быть перемещен в памяти, пока существует указатель, ссылающийся на этот объект.

Арифметические операции над указателями выполняются как над обычными целыми числами, с той лишь разницей, что значения, участвующие в операции, являются адресами оперативной памяти компьютера.

1.7.2. Символьные строки

Символьная строка — это объект данных, образованный последовательностью символов.

Спецификация. Различают символьные строки следующих видов:

- *символьные строки фиксированной длины*. Значениями, присваиваемыми объекту данных, объявленному как символьная строка фиксированной длины, могут быть строки символов только этой длины. Попытка присвоить объекту значение строки символов, длина которой отлична от определенной при его объявлении, приведет к усечению строки до заданной длины или к добавлению пробелов в ее конец;
- *символьные строки переменной длины*, ограниченной максимальным значением. При таком объявлении строки могут содержать меньшее количество символов или быть пустыми;
- *символьные строки неограниченной длины*. Объект данных такого типа может содержать строку любой длины, которая может изменяться в процессе выполнения программы.

Над строками определены следующие операции:

- *операция присваивания*:

$$ПрисвОп: \quad string_1 \times string_2 \rightarrow void$$

или

$$ПрисвОп: \quad string_1 \times string_2 \rightarrow string_3;$$

- *конкатенация* (*объединение*) строк:

$$КонкСтр: \quad string_1 \times string_2 \rightarrow string_3;$$

- *операции сравнения* (*СравОп*): *равно*, *не равно*, *больше*, *меньше*, *больше или равно*, *меньше или равно*. Спецификация операций сравнения имеет вид:

$$СравОп: \quad string_1 \times string_2 \rightarrow Boolean.$$

Строка `A` меньше строки `B`, если первый символ в `A` меньше первого символа строки `B`. Если эти символы совпадают, то сравнение продолжается со второго символа, и так далее до конца строки. При этом если одна из сравниваемых строк длиннее, вторая дополняется в конце пробелами, чтобы их длины были одинаковыми;

- *выбор подстроки* по указанной позиции первого символа и длине или по указанным позициям ее первого и последнего символов:

$$Выбор: \quad string_1 \times integer_1 \times integer_2 \rightarrow string_2;$$

- *выбор подстроки на основе сопоставления с образцом*:

$$ВыборОбр: \quad string_1 \times string_2 \rightarrow integer_1, integer_2.$$

Строка просматривается слева направо и ищется первое вхождение подстроки-образца. Результатом выполнения операции является пара целых чисел, определяющих позиции первого и последнего символов подстроки. Эта операция обычно используется в случаях, когда точная позиция подстроки неизвестна, но определено ее положение по отношению к другим подстрокам. Например, нужно получить первый отличный от пробела символ строки.

Реализация. Каждый из трех способов задания символьных строк по-разному представляется в памяти компьютера (рис. 1.6). Обычно аппаратно поддерживаются только строки фиксированной длины, а для реализации остальных способов требуется программное моделирование. Операции над строками также реализуются путем программного моделирования.

Рис. 1.6. Три представления символьных строк в памяти

1.7.3. Перечислимые типы

Во многих языках программирования имеется тип данных *перечисление*, позволяющий программисту определять переменные, которые могут принимать значения из ограниченного множества допустимых значений.

Спецификация. Перечисление представляет собой упорядоченную последовательность некоторых элементов. Поскольку в программе могут использоваться несколько переменных одного и того же перечислимого типа, обычно он определяется как отдельный тип, которому присваивается имя. Например, в языке Pascal объявления перечислимого типа и переменных этого типа могут выглядеть следующим образом:

```
type colour = (white, red, green, blue, black);
var  circle, square: colour;
```

Для объектов перечислимого типа определены следующие операции:

- *операции сравнения*: *равно*, *не равно*, *больше*, *меньше*, *больше или равно*, *меньше или равно*;
- *операция присваивания*;
- *операции* `succ` и `pred`, которые для данного элемента перечисления определяют соответственно последующий и предыдущий элементы. Операция `succ` не применима к последнему элементу перечисления, а операция `pred` — к первому.

Реализация. Перечислимый тип представляется в памяти компьютера последовательностью неотрицательных целых чисел, каждое из которых является порядковым номером элемента перечислимого типа в последовательности. Поскольку число элементов перечислимого типа, как правило, невелико, используются короткие целые числа. Например, для представления в памяти приведенного ранее перечислимого типа `colour` достаточно трех битов.

Реализация операций над объектами перечислимого типа выполняется с использованием встроенных операций над целыми числами.

1.7.4. Ограниченные типы

Ограниченным типом называется непрерывная последовательность базового (перечислимого, целого, логического) типа. Например, в языке Pascal ограниченный тип и переменную этого типа можно объявить следующим образом:

```
type size = 1 .. 100;
var  index: size;
```

Над ограниченным типом определены все операции, допустимые для базового типа.

Реализация ограниченных типов не имеет особенностей по отношению к реализации соответствующих базовых типов.

1.7.5. Векторы и массивы

1.7.5.1. Векторы

Вектор (*одномерный массив*) — это структура данных, состоящая из фиксированного количества компонентов одного типа. Компонент вектора определяется путем задания *индекса*, который является целочисленным значением или элементом перечислимого типа.

Спецификация. Вектор характеризуется следующими атрибутами:

- *количество компонентов* (*размер*) вектора. Задается явно целочисленным выражением или неявно путем задания диапазона изменения индекса;
- *тип данных компонентов*;
- *список значений индексов*. Обычно задается парой целых чисел, определяющих диапазон значений индекса. Может также задаваться неявно количеством компонентов вектора, при этом значение нижней границы индекса задается по умолчанию (0 или 1). Если диапазон значений индекса задается с помощью перечислимого типа, то значениями индекса могут быть только символьные константы.

Над векторами определены следующие операции:

- *индексация* — операция выбора компонента вектора. Синтаксически операция индексации обычно записывается как имя вектора, за которым в скобках (квадратных или круглых) записывается выражение, значение которого определяет индекс;
- *создание* и *уничтожение* вектора;
- *присвоение значений компонентам* вектора;
- *арифметические операции над векторами* одинаковой размерности (при этом операция выполняется для каждой пары компонентов вектора);
- *специальные операции над векторами* (*скалярное* и *векторное произведения* векторов, *обращение* вектора, *определение длины* вектора).

Реализация. Однородность компонентов вектора позволяет использовать последовательное представление компонентов в памяти, при этом дескриптор вектора, описывающий его атрибуты, может храниться вместе с вектором в смежных областях памяти или раздельно, как показано на рис. 1.7.

Нижнюю *LB* и верхнюю *UB* границы диапазона значений индекса, которые не требуются для доступа к компонентам вектора, обычно сохраняют в дескрипторе во время выполнения программы для того, чтобы можно было проверять, соответствует ли индекс объявленному диапазону. Остальные атрибуты вектора требуются только во время компиляции программы.

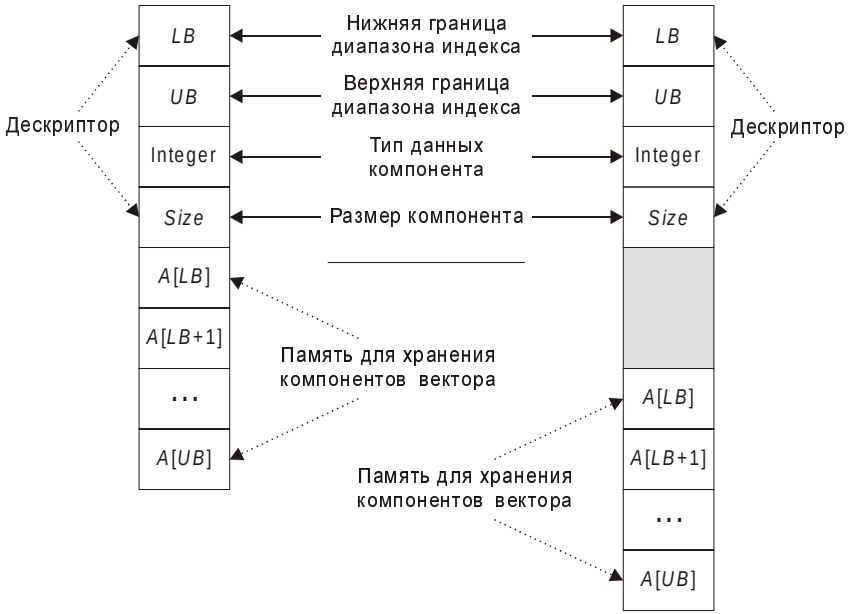


Рис. 1.7. Представление вектора и дескриптора вектора в памяти

При выделении памяти для представления вектора из N компонентов в памяти отводится место размером $N \times Size$ для N компонентов вектора ($Size$ — размер компонента) и D слов для хранения дескриптора.

Если адрес первого компонента вектора $A[LB]$ — B, то адрес компонента вектора $A[i]$ может быть вычислен по формуле:

$$B + (i - LB) \times Size = (B - LB \times Size) + i \times Size,$$

где $(B - LB \times Size)$ — постоянная величина.

Для проверки возможных ошибок, связанных с выходом значения индекса за пределы допустимого диапазона, необходимо до вычисления адреса компонента вектора проверить выполнение двойного неравенства $LB \le i \le UB$.

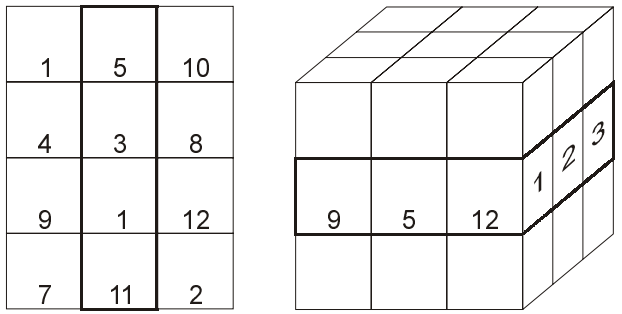
Однородность компонентов и фиксированная размерность векторов упрощают их хранение и доступ к отдельным компонентам. В приведенной ранее формуле доступа предполагается, что размер компонентов вектора $Size$ является целым числом адресуемых единиц памяти (слов или байтов). Если тип компонентов вектора *логический* или *символьный*, то в одном слове можно разместить несколько компонентов. *Упакованное* представление вектора логического или символьного типа в памяти машины позволяет сэкономить значительный объем памяти, но при этом доступ к компонентам упакованного вектора усложняется. По этой причине вектора обычно хранятся в неупакованном виде: каждый компонент располагается в начале отведенного ему слова или байта.

1.7.5.2. Многомерные массивы

В *двухмерном массиве* (*матрице*) компоненты представлены в форме прямоугольной таблицы, состоящей из строк и столбцов. Для выбора компонента матрицы необходимо указать два индекса: *номер строки* и *номер столбца*, на пересечении которых находится компонент. Трехмерный массив представляет собой вектор, компонентами которого являются матрицы. Аналогично можно построить массив любой размерности из массивов меньшей размерности.

Спецификация. Многомерный массив характеризуется тем же набором атрибутов, что и вектор, но для массива требуется указывать диапазон изменения значений индекса (или их количество) для каждого измерения отдельно. Операция индексации для массива синтаксически представляется как имя массива и список индексов — по одному индексу для каждого измерения массива. Например, в языке программирования C++ элемент трехмерного массива A с индексами i, j, k представляется в виде $A[i, j, k]$.

Над массивами определены те же основные операции, что и над векторами, с поправкой на то, что компонент многомерного массива задается несколькими индексами. Специфической операцией для массивов является операция определения *сечения* массива. Определение сечения массива является механизмом обращения к части массива как к единому целому. Например, если A — матрица, то одним из возможных сечений матрицы может быть какой-либо ее столбец (рис. 1.8, *а*). Для многомерных массивов размерности выше второй можно получать многомерные сечения (рис. 1.8, *б*).



а *б*

Рис. 1.8. Сечения массивов: одномерное (столбец) (*а*) и многомерное (плоскость) (*б*)

Реализация. Многомерный массив может рассматриваться как вектор по первому индексу, элементами которого являются массивы, размерность которых на единицу меньше размерности исходного массива. Затем каждый из этих массивов меньшей размерности снова рассматривается как вектор по второму индексу (относительно исходного массива), элементами которого

являются массивы, размерность которых еще раз уменьшается на единицу и т. д. Такое представление массивов в памяти называется *развертыванием по строкам*. На рис. 1.9 приведено представление в памяти трехмерного массива A, описание которого на языке Pascal имеет следующий вид:

```
var A: array [1..3, 1..2, 1..2] of integer;
```

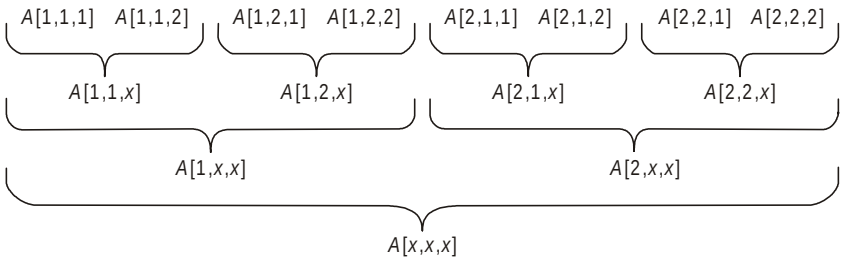


Рис. 1.9. Представление в памяти трехмерного массива

Как видно из рис. 1.9, массив A ($A_{x,\,x,\,x}$) может быть представлен в памяти в виде последовательности трех сечений $A_{i_1,\,x,\,x}$, где i_1 — значение индекса по первому измерению. Каждое из этих сечений можно представить в виде двух сечений $A_{i_1,\,i_2,\,x}$, где i_2 — значение индекса по второму измерению.

Операция индексации для массива реализуется путем вычисления смещения местоположения компонента относительно базового адреса массива B. Например, местоположение элемента матрицы $A[i, j]$ определяется следующим образом:

$$B + (i - LB_1) \times (UB_2 - LB_2 + 1) \times Size + (j - LB_2) \times Size,$$

где LB_1 — нижняя граница диапазона значений первого индекса, а LB_2 и UB_2 — нижняя и верхняя границы диапазона значений второго индекса.

Приведенную формулу можно упростить, выделив постоянные члены:

$D_2 = (UB_2 - LB_2 + 1) \times Size$ — размер строки матрицы,

$V = B - LB_1 \times D_2 - LB_2 \times Size$ — виртуальный начальный адрес.

Тогда:

$$\mathrm{ADDR}(A[i, j]) = V + i \times D_2 + j \times Size.$$

Значения V, D_2 и $Size$ определяются при создании массива и могут быть сохранены в дескрипторе матрицы.

Обобщим приведенную формулу на случай массивов более высоких размерностей. Предположим, что многомерный массив A размерности n имеет следующие диапазоны изменения индексов: $L_1 : U_1$, $L_2 : U_2$, ... , $L_n : U_n$, где L_i — нижняя граница, U_i — верхняя граница i-го измерения массива, а $D_i = U_i - L_i + 1$ — размер массива по i-му измерению.

Пусть размер S_n компонента массива $A[i_1, i_2, \ldots, i_n]$ равен *Size*.

Определим размеры сечений S_i этого массива:

$S_{n-1} = D_n \times S_n$ — размер сечения $A_{i_1, i_2, \ldots, x}$;

$S_{n-2} = D_{n-1} \times S_{n-1}$ — размер сечения $A_{i_1, i_2, \ldots, x, x}$;

...

$S_1 = D_2 \times S_3 \times \ldots \times S_{n-1} \times S_n$ — размер сечения $A_{i_1, x, \ldots, x}$.

Виртуальный начальный адрес массива вычисляется как:

$$V = B - L_1 \times S_1 - L_2 \times S_2 - \ldots - L_n \times S_n.$$

Тогда адрес компонента многомерного массива вычисляется по формуле:

$$\mathrm{ADDR}(A[i_1, i_2, \ldots, i_n]) = V + i_1 \times S_1 + i_2 \times S_2 + \ldots + i_n \times S_n.$$

Размеры сечений S_i и виртуальный начальный адрес массива V вычисляются один раз и хранятся в дескрипторе массива.

1.7.6. Записи

Записью (*структурой*) называют структуру данных, состоящую из фиксированного количества компонентов (*полей*), которые могут быть различных типов. Компоненты записей обозначаются *символическими именами* (*идентификаторами*).

Спецификация. Запись характеризуется следующими атрибутами:

- *количество компонентов записи*;
- *тип данных каждого компонента*;
- *идентификатор для обозначения каждого компонента.*

Типичным примером синтаксиса, используемого для объявления записей, является конструкция `struct`, используемая в языке C:

```
struct book
{ char  name[20];
  char  title[50];
  int   year;
  float price;
} child_book, my_book;
```

Это объявление определяет запись типа `book`, состоящую из четырех компонентов с именами `name`, `title`, `year` и `price`, и переменные `child_book` и `my_book` этого типа.

Одной из операций над записью как неделимой структурой, является операция присваивания одной записи другой того же типа и операции сравнения записей одинаковой структуры на равенство (неравенство).

Для выбора компонента записи обычно используется следующая синтаксическая конструкция: `my_book.title`, `my_book.price` и т. д.

Реализация. Компоненты записи размещаются в памяти в последовательных словах единого блока памяти. Объявление записи позволяет определить размер каждого компонента *Size* и его *смещение Displ* внутри блока памяти во время трансляции программы и поместить его в дескриптор (рис. 1.10).

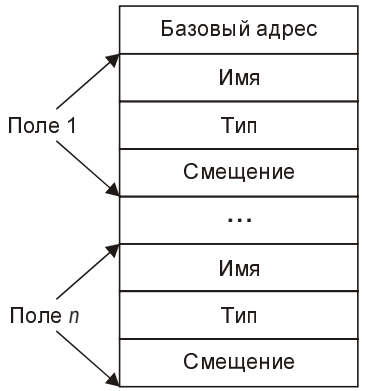


Рис. 1.10. Статический дескриптор записи

Формула доступа, используемая для вычисления местоположения *i*-го компонента записи *Str*, имеет следующий вид:

ADDR(*Str*, *i*) = *B* + $Displ_i$, где *B* — базовый адрес записи.

Области памяти, отводимые для хранения некоторых типов данных, должны начинаться с определенных адресов (например, области памяти для хранения целочисленных данных должны начинаться на границе слова). Так, в рассмотренном ранее примере записи типа `book` область памяти, отводимая под поле `year`, должна начинаться на границе слова (должна быть кратна четырем, если адресуемой единицей памяти является байт), и поэтому 2 байта между полями `title` и `year` не используются.

Операция присваивания одной записи другой может быть реализована как простое копирование содержимого блока памяти, представляющего первую запись, в блок памяти, представляющий вторую запись, а операция сравнения двух записей — как последовательность операций сравнения соответствующих компонентов записей.

1.7.6.1. Записи и массивы со структурированными компонентами

Во многих языках программирования допускается совместное использование компонентов записей и массивов. Например, можно объявить вектор

library, состоящий из тысячи компонентов, каждый из которых является записью типа book:

```
struct book library[1000];
```

Для выбора компонента такой сложной структуры сначала необходимо выбрать компонент вектора, а затем компонент записи, например library[5].price. Запись также может состоять из компонентов, которые в свою очередь являются записями или массивами. Таким образом, можно создавать иерархические структуры, состоящие из нескольких уровней записей и массивов.

1.7.7. Объединения

Объединением называется переменная, которая может содержать в различное время выполнения программы значения разных типов. Примером структуры данных, построенной с использованием объединений, может служить таблица констант, создаваемая транслятором во время трансляции программы. При работе с такой таблицей удобно, чтобы одно ее поле могло содержать значения констант разных типов. Такой тип поля записи является в некотором смысле объединением нескольких типов значений, которые могут содержаться в этом поле.

Спецификация. В языках программирования имеются два типа объединений: *свободные* и *размеченные*. Свободные объединения не требуют проверки значения типа данных при обращении к переменной. При использовании размеченных объединений каждое объединение содержит специальное поле — *тег*, содержащий актуальное значение типа данных переменной. Размеченные объединения обычно интегрируются в одну структуру данных с записями. Такие структуры данных получили название *вариантных записей*. Вариантная запись содержит компоненты, общие для всех вариантов, и один или несколько компонентов, чьи имена и типы данных уникальны для каждого варианта записи. Листинг 1.6 иллюстрирует пример записи языка Pascal, содержащий вариантную часть.

Листинг 1.6

```
type shape = (circle, triangle, rectangle);
     colors = (black, red, white);
     figure =
     record
       case form : shape of
         circle: (diameter : real);
         triangle:  (side1 : real;
```

```
                side2 : real;
                angle : real);
    rectangle: (side1 : real;
                side2 : real)
end;
var my_figure : figure;
```

Реализация. Представление вариантной записи `my_figure` приведено на рис. 1.11. Основное отличие объединения от записи заключается в том, что все поля объединения начинаются с одного адреса области памяти. При этом отводится такой объем памяти, который достаточен для хранения наибольшего варианта (в данном примере наибольшим вариантом является поле `triangle`). Во время выполнения программы для выбора соответствующего варианта не требуется никакого дополнительного дескриптора, т. к. для этих целей используется значение тега, являющегося частью вариантной записи.

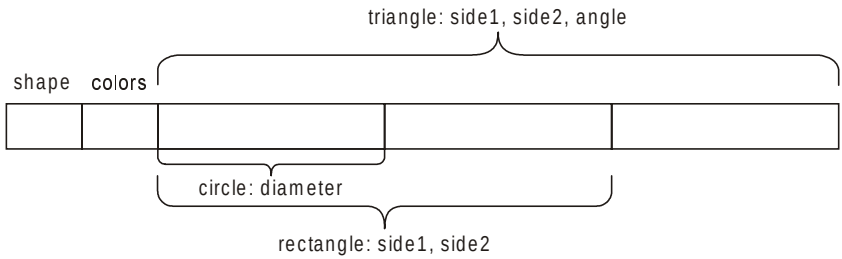


Рис. 1.11. Представление вариантной записи в памяти

Выбор компонента вариантной записи аналогичен выбору компонента обычной записи. Во время трансляции вычисляется смещение требуемого компонента относительно базового адреса блока памяти, выделенного для хранения всей записи. Во время выполнения программы это смещение добавляется к базовому адресу для определения компонента в памяти.

1.7.8. Множества

Множество — это структура данных, представляющая собой *неупорядоченный набор различных значений.*

Спецификация. Значения, образующие множество, являются величинами некоторого порядкового типа, называемого *универсальным.*

Над множествами определены следующие основные операции:

- *определение принадлежности к множеству.* Проверяет, принадлежит ли элемент *A* множеству *M.* Если принадлежит, то возвращает значение `true`, в противном случае — `false`;

- *включение элемента в множество.* Включает элемент A в множество M, если множество не содержит такого элемента;
- *удаление элемента из множества.* Исключает A из множества M, если A является элементом множества;
- *объединение*, *пересечение*, *разность множеств.* Каждая из этих операций создает новое множество M_3, которое либо содержит элементы обоих множеств M_1 и M_2 (случай *объединения*), либо содержит только те элементы, которые принадлежат и множеству M_1, и множеству M_2 (случай *пересечения*), либо содержит только те элементы множества M_1, которых нет в множестве M_2 (случай *разности* множеств M_1 и M_2).

Реализация. Наиболее распространены два способа представления множеств в памяти:

- *представление множеств битовыми строками.* Такое представление целесообразно использовать, если универсальное множество, используемое для представления элементов множества, невелико по размеру (число элементов множества равно длине машинного слова в битах). Предположим, что универсальное множество состоит из N элементов: $\{b_1, b_2, \ldots, b_N\}$. Тогда множество можно представить в виде битовой строки длины N, в которой бит с номером i равен 1, если b_i входит в множество, и 0 в противном случае.

  В случае представления множеств битовыми строками операция включения элемента в множество заключается в замене в битовой строке соответствующего бита на 1, операция удаления — в замене соответствующего бита на 0, а операция определения принадлежности элемента множеству сводится к проверке значения соответствующего бита. Операции объединения, пересечения и вычитания над множествами реализуются при помощи встроенных в аппаратную часть компьютера поразрядных логических операций над битовыми строками: поразрядное логическое ИЛИ, примененное к двум битовым строкам, реализует операцию объединения множеств, поразрядное логическое И — операцию пересечения, а поразрядное логическое И, примененное к первой битовой строке и инверсии второй строки, соответствует операции вычитания;
- *представление множеств с использованием хеширования.* Этот способ представления множеств рекомендуется использовать, если число элементов множества достаточно велико. Для эффективной реализации представления множеств с помощью хеширования необходимо иметь блок памяти, размер которого превышает размер множества в несколько раз. При этом элементы множества будут размещаться в этом блоке памяти не в последовательных словах памяти, а в зависимости от значения хеш-функции.

  Рассмотрим реализацию операции включения нового элемента A, представленного битовой строкой B_A, в множество M, представленное блоком

памяти S. Применим к строке B_A некоторую хеш-функцию, которая разбивает ее на маленькие части, затем их перемешивает и вырезает в полученной строке битов некоторую подстроку, значение которой рассматривается как хеш-адрес I_A. Этот хеш-адрес рассматривается как индекс, указывающий место элемента A в блоке памяти S. Если элемент A принадлежит множеству S, то он должен находиться в том месте блока памяти, на который указывает хеш-адрес I_A. Если элемента A в множестве нет, то строка B_A помещается по адресу I_A. Даже самая хорошая функция хеширования не может гарантировать, что для различных элементов множества будут сгенерированы разные хеш-адреса. Хотя желательно, чтобы хеш-функция распределяла адреса по блоку памяти M равномерно, время от времени возникают так называемые *коллизии*, при которых двум или более элементам множества будет соответствовать один и тот же хеш-адрес. Для разрешения коллизий существует несколько способов:

- *рехеширование*. Можно модифицировать исходную строку битов B_A, например, умножив ее на некоторый постоянный коэффициент k, заново применить к ней функцию хеширования и получить новый хеш-адрес I_A. Если при этом снова будет иметь место коллизия, то повторное хеширование проводится до тех пор, пока не отыщется свободное место в блоке памяти или не обнаружится, что элемент A уже содержится в множестве;
- *последовательный поиск*. Начиная с хеш-адреса, для которого имеет место коллизия, организуется последовательный (циклический) поиск до тех пор, пока не найдется свободное место в блоке памяти или не обнаружится, что элемент A уже содержится в множестве;
- *группировка данных*. Вместо одного сравнительно большого блока памяти S можно использовать вектор указателей, каждый из которых указывает на список элементов с одинаковыми хеш-адресами. В результате хеширования значения B_A определяется указатель на соответствующий список, в котором осуществляется поиск элемента A, и в случае его отсутствия он добавляется в начало списка.

Выбор той или иной функции хеширования зависит от значений элементов множества. Довольно представительный набор функций хеширования и рехеширования приводится в [11], [24]. Использование хеширования для представления множеств позволяет эффективно реализовать операции определения принадлежности элемента к множеству, включения и удаления элементов множества. Такие же операции, как объединение, пересечение и вычитание множеств, реализуются посредством применения последовательности указанных ранее операций, поэтому они малоэффективны.

1.7.9. Списки

Списком называется упорядоченная последовательность произвольных структур данных.

Спецификация. Список характеризуется следующими атрибутами:

- *голова списка* — значение первого элемента списка;
- *хвост списка* — значение последнего элемента списка;
- *тип данных элементов списка.* Этот атрибут не всегда характеризует список, т. к. списки могут быть неоднородными.

Списки редко имеют фиксированную длину. Часто они используются для представления динамических объектов данных, длина которых изменяется в процессе выполнения программы.

Над списками определены следующие операции:

- *создание* и *уничтожение* списка;
- *включение элемента* в список;
- *исключение элемента* из списка;
- *поиск элемента списка с заданным значением*;
- *поиск предыдущего и следующего элементов* списка;
- *упорядочение значений элементов* списка (по возрастанию или убыванию);
- *объединение двух списков в один.*

Реализация. Списки, как правило, представляются в памяти в виде связанных списков. Элемент связанного списка представляет собой запись, одна часть полей которой содержит данные, а другая часть — указатели на другие элементы списка. Различают односвязные, двухсвязные и многосвязные списки. В односвязном списке каждый элемент связан только с одним элементом списка (предыдущим или следующим). В двухсвязном списке каждый элемент списка связан с двумя элементами списка: предыдущим и следующим. В многосвязном списке каждый элемент связан с произвольным числом элементов списка. Это число хранится в виде отдельного поля в элементе.

В некоторых языках программирования определены специальные типы списков. К таким спискам относятся:

- *стеки и очереди. Очередь* — это список, в котором выбор и удаление элементов осуществляются с начала списка, а включение элементов — с конца. *Стек* — это список, в котором операции выбора, включения и удаления элемента выполняются с конца списка. Для очередей и стеков используется как связанное, так и последовательное представление;

- *деревья*. Это список, элементами которого могут быть как списки (поддеревья), так и элементарные объекты данных, если каждый список является компонентом не более чем одного другого списка;
- *ориентированные графы*. Элементы такого списка связаны друг с другом произвольным образом.

1.8. Выражения и операторы присваивания

Операции и данные в языках программирования объединяются в программы и комплексы программ при помощи управляющих структур. Структуры управления последовательностью действий удобно разбить на три группы:

- управление последовательностью действий внутри выражений;
- управление последовательностью действий на уровне операторов;
- управление последовательностью действий при взаимодействии подпрограмм.

В данном разделе мы рассмотрим, каким образом осуществляется управление последовательностью действий внутри выражений.

Выражения являются основными средствами вычислений в языках программирования. Порядок вычисления операторов в выражении определяется правилами ассоциативности и приоритетами, в то время как порядок вычисления операндов в языке часто не определяется, хотя и может влиять на значение выражения.

1.8.1. Арифметические выражения

Арифметические выражения в языках программирования состоят из операндов, операторов, круглых скобок и вызовов функций. Если скобки в выражении отсутствуют, то порядок выполнения операторов при вычислении выражения определяется *приоритетом операторов*. В языках программирования, как правило, поддерживается один и тот же приоритет арифметических операторов, основанный на соответствующих математических правилах *для бинарных операторов*: наивысший приоритет имеет оператор возведения в степень, затем следуют имеющие одинаковый приоритет операторы умножения и деления, за которыми идут операторы сложения и вычитания, имеющие низший приоритет.

Во многих языках программирования, помимо бинарного оператора вычитания, имеется унарный минус, используемый для изменения знака значения операнда. Обычно унарный минус имеет более высокий приоритет, чем бинарный. При использовании унарного минуса внутри выражения желательно применять круглые скобки.

Порядок выполнения операций с одинаковым приоритетом определяется правилами ассоциативности. *Ассоциативность* оператора может быть

правосторонней или левосторонней. Наиболее часто в языках программирования операторы с равным приоритетом выполняются слева направо. Исключение составляет оператор возведения в степень, который является правоассоциативным или вообще не подчиняется правилам ассоциативности (примером такого языка является Ada).

Для изменения правил ассоциативности и приоритета операторов в арифметических выражениях используются круглые скобки. Если в языке все операторы имеют одинаковый приоритет (например, в языке APL), то скобки являются единственным механизмом изменения порядка выполнения операторов.

Порядок вычисления операндов при вычислении выражения безразличен, если ни один из операндов в операторе не имеет побочных эффектов. *Побочный эффект* возникает при изменении функцией одного из своих параметров или глобальной переменной.

Рассмотрим следующее выражение:

```
X + FUNC(X)
```

Если функция `FUNC` не вызывает побочного эффекта, связанного с изменением значения переменной `X`, то значение выражения `X + FUNC(X)` не зависит от порядка вычисления операндов в выражении.

Допустим, что побочный эффект имеет место: функция `FUNC(X)` возвращает удвоенное значение аргумента и присваивает своему параметру значение 1. Если в процессе вычисления выражения `X + FUNC(X)` переменная `X` выбирается из памяти первой и ее значение равно 10, то значение выражения будет равно 30. Однако если первым вычисляется второй операнд выражения, то значение первого операнда после обращения к функции станет равным 1, а значение выражения будет равно 21.

Существует несколько способов решения проблемы, связанной с определением порядка вычисления операндов выражений, содержащих вызовы функций:

- исключить побочный эффект;
- указать в определении языка точный порядок вычисления операндов выражений и потребовать от разработчиков поддерживать этот порядок при реализации языка;
- указать в определении языка, что выражения, содержащие вызовы функций, можно использовать только тогда, когда функции не имеют побочных эффектов.

Каждый из этих способов имеет свои преимущества и недостатки, поэтому оптимального решения этой проблемы нет, что и подтверждается существующими языками программирования.

В большинстве языков программирования имеет место множественное использование операторов (*перегрузка операторов*), когда один и тот же знак операто-

ра может использоваться для выполнения операций над операндами разных типов. Например, знак '+' часто используется для сложения любых операндов числовых типов. При статическом связывании типов для перегруженных операторов языка компилятор выбирает нужный тип операции на основе типов операндов. Поскольку компьютер обычно содержит бинарные операции, выполняемые над операндами одного типа, в языках программирования, допускающих арифметические выражения над операндами разных типов, должны быть определены правила неявного преобразования типов операндов (*правила приведения типа*). Если оператор связывает два операнда, принадлежащих к разным типам, и такой оператор в языке допустим, то компилятор должен применить правила приведения типов к одному из операндов.

Наряду с неявным преобразованием типов, многие языки программирования позволяют использовать явное преобразование типов. Преобразование типа может быть как сужающим, так и расширяющим. *Сужающее преобразование* характеризуется тем, что тип, в который осуществляется перевод, принципиально не может содержать всех значений исходного типа, например преобразование типа `double` в тип `float` языка C. *Расширяющее преобразование*, наоборот, преобразует исходное значение в тип, который может содержать все значения исходного типа или их приближения, например преобразование типа `int` в тип `float` языка C. Хотя расширяющее преобразование, как правило, всегда безопасно, во многих языках программирования при таком преобразовании может теряться точность представления данных. Например, целые числа занимают 32 бит памяти, что соответствует точности представления не менее 9-ти значащих десятичных цифр, в то время как десятичные числа с плавающей точкой размером 32 бит имеют точность 7-ми десятичных цифр.

Во время выполнения программы многие компьютеры способны обнаруживать ошибки, возникающие при вычислении выражений:

- ошибки приведения типов операндов;
- ошибки переполнения памяти, возникающие при попытке занесения слишком большого числа в отведенную для его хранения область памяти;
- ошибки потери значимости, возникающие при попытке занести слишком маленькое число в отведенную для его хранения область памяти;
- ошибки деления на нуль.

Возникновение подобных ошибок, как правило, приводит к выдаче соответствующего диагностического сообщения, прекращению выполнения программы и передаче управления операционной системе.

1.8.2. Логические выражения

Логические выражения состоят из логических операндов (переменных, констант, вызовов функций, возвращающих результат логического типа,

и выражений отношения), круглых скобок и логических операторов. Наиболее часто используются логические операторы И, ИЛИ, НЕ, реже применяются операторы эквивалентности и исключающего ИЛИ. Результатом выполнения логических выражений являются логические значения. Выражение отношения состоит из двух арифметических выражений, разделенных оператором отношения (больше, не меньше, равно, не равно, меньше, не больше). Операторы отношения обычно перегружаются для большого числа типов.

Порядок вычисления логических выражений в большинстве языков программирования определяется *приоритетом логических операторов*: унарный оператор НЕ имеет наивысший приоритет, за ним следует оператор И, а наименьший приоритет имеет оператор ИЛИ. Поскольку арифметические операторы могут использоваться в качестве операндов выражений отношения, которые в свою очередь могут быть операндами логических выражений, то в языке программирования должны быть определены соотношения между приоритетами операторов этих трех категорий. Приоритет операторов отношения всегда ниже приоритета арифметических операций. Логические операторы в некоторых языках имеют более высокий приоритет, чем операторы отношения, а в других языках наоборот: более высокий приоритет имеют операторы отношений. Например, логическое выражение вида:

```
a > 1 or a < 10
```

недопустимо в языке Pascal, а в языке Ada такое выражение синтаксически корректно. Для изменения порядка вычисления логических операторов используются круглые скобки.

При реализации языков программирования с целью уменьшения времени выполнения программы может использоваться *сокращенное вычисление выражений*. Например, если значение переменной `A` меньше 0, то значение логического выражения:

```
(A >= 0) and (B < 0)
```

не зависит от значения второго оператора отношения, поскольку для всех `X` значение выражения `false and X` равно `false`. Такое сокращенное вычисление выражений легко обнаружить, поэтому во время трансляции может быть сгенерирован код, который во время работы программы не будет выполнять лишних действий. В язык программирования Ada включены логические операции `and then` и `or else`, которые вычисляют только свой первый операнд, если его значения достаточно для вычисления всего выражения.

Сокращенное вычисление целесообразно использовать только для логических выражений, поскольку обнаружение в арифметических выражениях подвыражений, равных нулю, является достаточно сложной задачей. Использование сокращенного вычисления выражений может привести к побочным эффектам, вызывающим серьезные ошибки, поэтому при наличии побочных эффектов сокращение вычислений применять не рекомендуется.

1.8.3. Операторы присваивания

Присваивание является одной из основных конструкций в императивных языках программирования, обеспечивающей? механизм, с помощью которого можно динамически изменять связи значений с переменными.

Синтаксис простого присваивания имеет следующий вид:

```
<левая часть оператора присваивания> <оператор присваивания> <выражение>
```

Существует несколько альтернативных решений для реализации присваивания. В некоторых языках программирования (FORTRAN, Pascal, Ada) присваивание существует только в качестве самостоятельного оператора, при этом левой частью оператора присваивания может быть единственная переменная. В языке PL/1 оператор присваивания является также самостоятельным оператором, но в его левой части можно использовать несколько переменных, разделенных запятой, например:

```
COUNT, SUMMA = 0;
```

Оператор присваивания в языках C, C++ и Java может использоваться в качестве операнда в выражении, что позволяет присваивать значения нескольким переменным одновременно, например:

```
COUNT = SUMMA = 0;
```

При этом оператор присваивания рассматривается как обычный бинарный оператор, результат выполнения которого равен значению выражения в правой части, но имеющий побочный эффект, связанный с изменением значения его левого операнда.

Недостатком использования присваивания в качестве операнда выражения является появление еще одного вида побочного эффекта в выражениях.

В языках C++ и Java допускается использование условного оператора присваивания:

```
condition ? var1 : var2 = 0;
```

эквивалентного следующему условному оператору:

```
if (condition) var1 = 0; else var2 = 0;
```

В языках программирования C, C++, Java имеются сокращенные формы для записи операторов присваивания. Так, для многих бинарных операций применяется сокращенная запись оператора присваивания (составной оператор присваивания), в которой переменная из левой части оператора присваивания используется в качестве первого операнда в выражении из правой части. Например, выражение:

```
count += 1;
```

эквивалентно выражению:

```
count = count + 1;
```

В этих языках программирования имеются также два унарных арифметических оператора ++ и – –, сочетающих операции увеличения и уменьшения значения операнда на 1 с присваиванием. Имеются две формы таких операторов: префиксная и постфиксная. Так в операторе присваивания:

```
summa = ++count;
```

значение переменной count сначала увеличивается на 1, а затем присваивается переменной summa.

Постфиксный оператор:

```
summa = count++;
```

сначала присваивает переменной summa значение переменной count, после чего значение переменной count увеличивается на 1.

Тип выражения из правой части оператора присваивания может не соответствовать типу переменной, которой присваивается значение. В этом случае большинство языков программирования используют правила приведения типов, подобные правилам, существующим для выражений, хотя имеются языки, в которых такое приведение запрещено. Во всех языках, в которых допускается приведение типов для операторов присваивания, приведение типов выполняется только после вычисления выражения в правой части.

1.9. Структуры управления на уровне операторов

Структуры управления на уровне операторов могут быть явными или неявными (структурами управления по умолчанию). *Неявные* структуры действуют во всех случаях, если только программист не изменяет их с помощью какой-либо явной структуры. Например, в большинстве языков программирования по умолчанию поддерживается естественная последовательность выполнения операторов в порядке их написания в программе. Для изменения естественного порядка выполнения операторов используются *управляющие структуры* — управляющий оператор и совокупность операторов, выполнение которых он контролирует.

В языках программирования имеются три типа управляющих структур:

- *композиция* — операторы могут быть представлены в виде последовательности, выполняемой как единое целое;
- *ветвление* — две последовательности операторов могут быть альтернативными, при этом в каждом конкретном случае выполняется только одна из них;
- *повторение* — последовательность операторов может выполняться многократно или вообще не выполняться в зависимости от некоторого условия.

Исследования в области языков программирования [20] показали, что наиболее надежной методологией разработки программ является *структурное программирование*. Согласно этой методологии управляющие операторы, включаемые в язык программирования, должны иметь *одну точку входа* и *одну точку выхода*. При чтении структурированной программы порядок выполнения операторов программы соответствует порядку, в котором операторы располагаются в тексте программы. Каждый оператор с одной точкой входа и одной точкой выхода может содержать внутренние ветвления и циклы, но управление между такими операторами должно осуществляться через единственную точку входа-выхода.

1.9.1. Составные операторы

Составной оператор является основной структурой для представления *композиции*. Составной оператор — это последовательность операторов вида:

```
begin
   оператор_1
   ...
   оператор_n
end
```

В языках программирования C, C++, Java, Perl в качестве операторных скобок используются символы '{' и '}'.

Поскольку составной оператор — это тоже оператор, то можно образовывать иерархические конструкции, в которые будут входить составные операторы наряду с другими управляющими операторами.

1.9.2. Условные операторы

Условные операторы предоставляют программисту возможность альтернативного выполнения одного из двух или более операторов (ветвление) или выполнение какого-либо одного оператора при определенных условиях. При этом в качестве выполняемого оператора может выступать как одиночный базовый оператор (оператор присваивания, оператор ввода/вывода), так и любой управляющий оператор. Выбор осуществляется посредством проверки некоторого условия, которое задается логическим выражением.

1.9.2.1. Операторы *if*

Одновариантный условный оператор **if** имеет следующий обобщенный вид:

```
if логическое_выражение then оператор endif
```

Семантика выполнения одновариантного условного оператора заключается в следующем: вычисляется логическое выражение; если значение логического

выражения *истина*, выполняется оператор, следующий за ключевым словом **then**, в противном случае этот оператор не выполняется и управление передается оператору, следующему за условным оператором.

Двухвариантный условный оператор можно представить в следующем обобщенном виде:

```
if логическое_выражение then оператор_1 else оператор_2 endif
```

Семантика выполнения двухвариантного условного оператора выглядит следующим образом: в зависимости от значения логического выражения (*истина* или *ложь*) выполняется соответственно `оператор_1` или `оператор_2`.

Вложенные условные двухвариантные операторы могут использоваться для реализации выбора из нескольких вариантов, например,

```
if flag = 0 then
   if summa = 0 then result := 0 else result := 1;
```

Этот оператор можно интерпретировать двумя различными способами в зависимости от того, к какому из операторов, следующему за ключевым словом **then**, относится оператор, следующий за ключевым словом **else**.

Вариант 1

```
if flag = 0 then
   if summa = 0 then
      result := 0
   else
   result := 1;
```

Вариант 2

```
if flag = 0 then
   if summa = 0 then
      result := 0
   else
      result := 1;
```

Для предотвращения неоднозначности интерпретации порядка выполнения операторов во вложенных условных операторах в приведенном ранее примере необходимо использовать семантические правила. В большинстве языков программирования семантика выполнения условного оператора такова, что оператор, следующий за ключевым словом **else**, всегда относится к ближайшему предшествующему оператору, следующему за ключевым словом **then**, как в варианте 2.

Более надежным решением проблемы неоднозначности вложенных условных операторов является задание порядка выполнения операторов с помощью синтаксиса. Например, разработчики языка ALGOL-60 запретили включать вложенный условный оператор непосредственно после ключевого слова **then**. В случае необходимости использования такой конструкции

вложенный условный оператор должен быть заключен в операторные скобки **begin ... end** одним из следующих способов:

Вариант 1

```
if flag = 0 then
   begin
   if summa = 0 then
      result := 0
   end
else
   result := 1;
```

Вариант 2

```
if flag = 0 then
   begin
   if summa = 0 then
      result := 0
   else
      result := 1
   end;
```

Альтернативой решению, принятому при разработке языка ALGOL-60, является введение специальных ключевых слов, например замыкающих условные операторы. Так, на языке Ada приведенные ранее фрагменты программ можно записать в виде:

Вариант 1

```
if flag = 0 then
   if summa = 0 then
      result := 0
   end if;
else
   result := 1;
end if;
```

Вариант 2

```
if flag = 0 then
   if summa = 0 then
      result := 0;
   else
      result := 1;
   end if;
end if;
```

Реализация. Операторы условного перехода легко реализуются с помощью обычных команд перехода и ветвления, поддерживаемых аппаратной частью компьютера.

1.9.2.2. Операторы многовариантного ветвления

Несмотря на то, что многовариантные операторы ветвления можно построить на основе двухвариантного условного оператора и операторов безусловного перехода, полученные таким образом структуры громоздки, трудны для чтения и потенциально ненадежны. В связи с этим во многие современные языки программирования включен многовариантный оператор ветвления **case**. Этот оператор позволяет выбрать для выполнения одну группу операторов (в простейшем случае один оператор) из произвольного количества групп.

Например, многовариантный оператор ветвления в языке Pascal имеет следующий вид:

```
case выражение of
   список_констант_1: оператор_1;
       ...
   список_констант_n: оператор_n;
else оператор_n+1
end;
```

В этом операторе `выражение` должно быть целого, логического, символьного или перечислимого типа.

Семантика оператора **case** языка Pascal заключается в следующем: вычисляется выражение, значение которого сравнивается с константами из списка констант. Если один из списков констант содержит значение вычисленного выражения, выполняется оператор, связанный с этим списком. Если значение выражения не принадлежит ни одному из списков констант, то управление передается оператору, следующему за ключевым словом **else**.

Списки констант должны быть *взаимоисключающими*, но могут быть не исчерпывающими, т. е. одна и та же константа не может входить в несколько списков, но в списке констант не обязательно должны быть представлены все значения из диапазона, которому может принадлежать значение выражения.

Хотя управляющее выражение в операторе **case** может быть логическим, такая конструкция плохо приемлема для многовариантного выбора. В этом случае для реализации многовариантного выбора более предпочтительным является вложенный оператор ветвления, называемый оператором **elsif**. Далее приводится пример многовариантного выбора на основе оператора **elsif**, включенного в язык Ada:

```
if summa <= 25 then mark := 2;
elsif summa > 25 and summa <= 50 then mark := 3;
elsif summa > 50 and summa <= 75 then mark := 4;
elsif summa > 75 and summa <= 100 then mark := 5;
end if;
```

Обычно операторы **case** и **elsif** инкапсулированы и имеют один вход и один выход.

Реализация. Операторы **case** и **elsif** во избежание повторной проверки значения одной и той же переменной обычно реализуются с помощью *таблицы переходов*. Таблица переходов — это вектор, компоненты которого размещены в памяти последовательно и каждый из них представляет собой безусловную команду перехода на фрагмент кода, реализующий соответствующий вариант.

1.9.3. Операторы цикла

Цикл является основной управляющей структурой для реализации повторяющихся вычислений в императивных языках программирования. Оператор цикла состоит из двух частей — *заголовка* и *тела*. Заголовок управляет количеством повторений тела цикла, в то время как тело обычно является оператором (простым или составным), который определяет действие оператора цикла. Тело цикла может быть произвольным, в то время как для представления заголовка цикла имеется ограниченное число вариантов. Основные варианты заголовков цикла определяются тем, как разработчики языка решили две основные проблемы реализации циклов:

1. Как осуществляется управление числом повторений цикла?
2. В каком месте цикла размещается механизм управления?

1.9.3.1. Простая итерация

Простейший тип заголовка оператора цикла задает, сколько раз должно выполняться тело цикла. Пример такого оператора цикла на языке COBOL выглядит следующим образом:

```
perform body K times
```

(вычислить `K`, а затем выполнить тело цикла `K` раз).

Даже для такого простого случая возникают вопросы:

1. Можно ли значение `K` переопределить внутри тела цикла и, таким образом, изменить количество повторений цикла?
2. Как будет выполняться оператор цикла, если `K` является отрицательным числом или равно нулю?

Ответы на данные вопросы необходимо определить в описании семантики языка.

1.9.3.2. Циклы с параметром

Операторы цикла с параметром имеют переменную, называемую *параметром* (счетчиком) цикла, которая используется в качестве счетчика (или индекса) числа повторений цикла. В таком заголовке задаются *начальное* и *конечное* значения параметра цикла и приращение — разность между двумя последовательными значениями параметра. При определении оператора цикла с параметром необходимо ответить на следующие вопросы:

1. Какой тип и какую область видимости должен иметь параметр цикла?
2. Чему равно значение параметра цикла при выходе из цикла?
3. Следует ли разрешать изменять параметр цикла, его начальное, конечное значения и приращение в теле цикла, и если да, то как влияет такое изменение на управление циклом?

4. Начальное и конечное значения параметра цикла и приращение должны вычисляться только один раз при входе в цикл, или эти вычисления следует делать на каждой итерации?

1.9.3.3. Цикл с предусловием

Типичный цикл с предусловием имеет следующий синтаксис:

```
while логическое_выражение do тело_цикла
```

В цикле с предусловием тело цикла выполняется до тех пор, пока результат логического выражения является истинным.

1.9.3.4. Цикл с постусловием

Цикл с постусловием может быть представлен в виде:

```
do тело_цикла while логическое_выражение
```

Тело цикла с постусловием выполняется до тех пор, пока результат логического выражения не станет ложным. Основное различие между приведенными двумя циклами заключается в том, что в цикле с постусловием тело цикла всегда будет выполнено хотя бы один раз.

В языке Pascal цикл с постусловием **repeat ... until** отличается от операторов **do ... while** тем, что он имеет противоположную логику управляющего логического выражения: тело цикла выполняется до тех пор, пока результат логического выражения имеет значение *ложь*.

1.9.3.5. Циклы, управляемые пользователем

В циклах, управляемых пользователем, программист сам выбирает место расположения механизма управления циклом. Например, выход из бесконечного цикла языка Ada может быть выполнен следующим образом:

```
loop
   ...
   exit when условие;
   ...
end loop;
```

1.10. Подпрограммы

Подпрограммы являются теми базовыми блоками, из которых строится большинство программ, поэтому они включены практически во все языки программирования. Все подпрограммы, за исключением сопрограмм, имеют следующие свойства:

- каждая подпрограмма имеет один вход;
- после выполнения подпрограммы управление всегда возвращается в вызывающую программу (подпрограмму);

- на время выполнения вызываемой подпрограммы выполнение вызывающей программы (подпрограммы) приостанавливается, т. е. в каждый момент времени выполняется только одна подпрограмма.

Подпрограмма — это абстрактная операция, определяемая программистом. Она представляет собой некоторую математическую функцию, которая отображает каждый конкретный набор аргументов в некоторый набор результатов.

1.10.1. Определение подпрограммы

Определение подпрограммы состоит из двух частей: *спецификация* и *реализация*, которые задаются программистом при ее описании.

Спецификация. Спецификация подпрограммы включает:

- *имя* подпрограммы;
- *сигнатуру* подпрограммы, которая задает количество аргументов и результатов (*формальных параметров*), порядок их следования, тип данных для каждого аргумента и результата. Сигнатура не является обязательным элементом и может отсутствовать;
- *действие*, определяемое подпрограммой.

Реализация. Реализация определяется *телом* подпрограммы, которое состоит из *объявлений локальных данных*, определяющих структуры данных, используемых подпрограммой, и *операторов*, задающих действия, которые должна выполнять подпрограмма.

Типичный синтаксис определения подпрограммы приведен в листинге 1.7.

Листинг 1.7

```
procedure SWAP(int X, int Y)    - сигнатура подпрограммы
{ int TEMP;                     - объявление локальной переменной
    TEMP = X;
    X = Y;                      - операторы, задающие действия
    Y = TEMP;
}
```

В некоторых языках программирования (например, в Pascal и Ada) в тело подпрограмм могут входить определения других подпрограмм.

Определение подпрограммы является статическим свойством программы. При вызове подпрограммы создается *активация* подпрограммы, а при завершении выполнения подпрограммы активация разрушается. Во время выполнения программы по единственному определению подпрограммы может быть создано много активаций. Активация имеет определенное время жизни —

промежуток времени между создающим ее вызовом подпрограммы и выходом из подпрограммы, когда активация разрушается.

1.10.2. Формальные и фактические параметры подпрограммы

Объявления и операторы обычно инкапсулированы, поэтому ни локальные данные, ни операторы по отдельности не доступны программисту. Программист может только вызвать подпрограмму с фактическими параметрами и получить результаты. При этом фактические параметры и результат должны иметь тип, определенный спецификацией подпрограммы. Если типы всех формальных параметров и результатов подпрограммы объявлены, то *контроль типов* осуществляется статически во время компиляции. *Приведение типа* фактического параметра к типу формального параметра может выполняться *автоматически*, если это предусмотрено реализацией языка.

Практически во всех языках программирования связывание фактических параметров с формальными осуществляется при вызове подпрограммы на основе их расположения: первый фактический параметр соответствует первому формальному параметру, второй фактический параметр соответствует второму формальному и т. д. Такие формальные параметры называются *позиционными*.

Другой способ установления взаимного соответствия фактических и формальных параметров связан с использованием *ключевых параметров*, в которых фактические параметры указываются вместе с соответствующими формальными. Ключевые параметры могут появляться в списке фактических параметров в произвольном порядке, например:

```
SWAP(Y => SECOND; X => FIRST);
```

В некоторых языках программирования (например, в Ada и FORTRAN 90) разрешается смешивать в одном вызове и ключевые, и позиционные параметры. Единственное ограничение, накладываемое на такой способ указания фактических параметров, заключается в том, что после первого появления ключевого параметра в списке параметров все остальные параметры должны быть ключевыми.

В языках программирования C++, Ada и FORTRAN 90 формальному параметру, объявленному при определении подпрограммы, может не соответствовать никакой фактический параметр. В этом случае в качестве значения фактического параметра принимается значение формального параметра, заданное при его объявлении.

1.10.3. Процедуры и функции

Подпрограммы разделяются на две различные категории: *процедуры* и *функции*.

Процедуры (в некоторых языках они называются подпрограммами) представляют собой последовательности операторов, задающих параметризованные

вычисления. Процедуры определяют новые операторы, которые активизируются операторами вызова.

Процедуры могут вырабатывать результаты двумя способами. Во-первых, процедура может изменять значения переменных, которые не являются параметрами процедуры, но видимы в процедуре и вызывающем модуле. Во-вторых, процедура может изменять значения своих формальных параметров и, тем самым, передавать результаты в вызывающую программу.

Если подпрограмма явно возвращает только один результат, то она называется функцией. В большинстве императивных языков типы переменных, которые могут возвращаться функциями, ограничены неструктурированными типами. Вызов функции с соответствующими фактическими параметрами может участвовать в выражениях везде, где может использоваться переменная. Если при определении некоторой функции используются вызовы этой же функции, то она называется *рекурсивной*. Функция может иметь побочные эффекты, т. е. изменять значения переменных, определенных вне этой функции.

В большинстве языков программирования можно определять как функции, так и процедуры. Если в языке имеются только функции, то процедуру можно определить как функцию, не возвращающую явно никакого значения, указав тип возвращаемого функцией значения как `void`.

1.10.4. Методы передачи параметров

Методы передачи параметров — это способы, которыми параметры передаются в подпрограмму и/или возвращаются из нее. Формальные параметры характеризуются одним из трех режимов передачи параметров:

- формальные параметры могут получать значения от соответствующих фактических параметров (режим ввода);
- формальные параметры могут передавать данные фактическим параметрам (режим вывода);
- формальные параметры могут получать значения от соответствующих фактических параметров и передавать данные фактическим параметрам (режим ввода-вывода).

Существуют два различных механизма передачи данных при передаче параметров:

- фактическое значение параметра физически перемещается в вызывающую или вызываемую подпрограмму или двигается в обоих направлениях;
- передается путь доступа (указатель) к параметру.

Рассмотрим реализацию механизмов передачи данных для каждого режима передачи параметров.

1.10.4.1. Передача по значению

При передаче параметра по значению значение фактического параметра используется для инициализации соответствующего формального параметра, который в дальнейшем рассматривается как локальная переменная (режим ввода).

Передача по значению обычно реализуется путем физического перемещения данных. Режим ввода может быть реализован также с помощью передачи пути доступа к значению фактического параметра в вызывающей подпрограмме, которое должно быть *доступно только для чтения*.

Основным недостатком передачи параметра по значению путем физического перемещения данных является необходимость в дополнительной памяти для хранения формального параметра. Дополнительная память должна выделяться в вызываемой подпрограмме или в некоторой области памяти вне вызывающей и вызываемой подпрограмм.

1.10.4.2. Передача по результату

Передача параметра по результату представляет собой реализацию режима вывода. Соответствующий формальный параметр рассматривается как локальная переменная. Непосредственно перед возвращением управления в вызывающую подпрограмму значение формального параметра передается фактическому параметру, который должен быть переменной.

Реализация передачи по результату также требует дополнительной памяти и операций копирования, как и в случае передачи по значению. Обычно этот способ передачи параметра реализуется на основе физического перемещения данных. При этом разработчик должен решить вопрос, когда нужно вычислять адрес фактического параметра: во время вызова подпрограммы или при возвращении из нее. Например, формальный параметр `i` (индекс элемента массива) изменяется в подпрограмме. Это приведет к тому, что адрес элемента массива с индексом `i` изменится в промежутке между вызовом подпрограммы и возвращением из нее.

1.10.4.3. Передача по значению и результату

Передача параметра по значению и результату представляет собой комбинацию рассмотренных ранее методов передачи параметров. Значение фактического параметра сначала используется для инициализации соответствующего формального параметра, который в дальнейшем рассматривается как локальная переменная. При завершении выполнения подпрограммы значение формального параметра передается обратно фактическому параметру.

Недостаток передачи по значению и результату, как и передачи по значению и передачи по результату, состоит в необходимости хранить параметры в нескольких местах и тратить время на копирование их значений.

1.10.4.4. Передача по ссылке

Метод передачи параметров по ссылке передает путь доступа к данным (адрес) в вызываемую подпрограмму, в результате чего открывается доступ к фактическому параметру в вызывающей подпрограмме.

Преимущество передачи параметра по ссылке заключается в том, что не требуется использовать дополнительную память и копировать значения параметров.

К недостаткам передачи по ссылке относятся:

- увеличение времени доступа к формальному параметру, поскольку требуется еще один уровень косвенной адресации при передаче данных;
- если требуется односторонняя связь с вызываемой подпрограммой (только запись или только чтение), то могут возникнуть ошибочные изменения фактического параметра;
- совмещение имен при передаче параметров по ссылке может привести к противоречиям между формальными параметрами и глобальными именами, видимыми в вызываемой подпрограмме.

1.10.4.5. Передача по имени

Метод передачи параметров по имени — это метод передачи параметров в режиме ввода-вывода, который реализуется таким образом, что фактический параметр заменяется соответствующим формальным параметром всюду в подпрограмме, где он встречается.

При передаче по имени формальный параметр связывается с методом доступа во время вызова подпрограммы, но фактическое связывание параметра с некоторым значением или адресом откладывается до тех пор, пока формальному параметру не будет присвоено какое-нибудь значение либо на него не будет сделана ссылка. Выбор механизма передачи параметра зависит от вида фактического параметра. Если фактический параметр является скалярной величиной, то передача по имени эквивалентна передаче по ссылке. Если фактический параметр представляет собой константу, то передача по имени равносильна передаче по значению. Если фактический параметр представляет собой выражение, содержащее переменную, то выражение вычисляется каждый раз, когда необходимо получить доступ к формальному параметру в то время, когда управление выполнением подпрограммы достигает переменной.

Основное достоинство передачи параметра по имени заключается в гибкости, которая предоставляется программисту. Основным недостатком этого метода является низкая эффективность выполнения связывания параметров по сравнению с другими методами. Кроме этого, метод передачи параметров по имени трудно реализуем. Например, простейшая процедура `SWAP`, приведенная в листинге 1.7, не может быть реализована при передаче параметров по имени.

1.10.5. Сопрограммы

Сопрограмма — это специальная разновидность подпрограммы, в которой вызывающая и вызываемая подпрограммы равноправны, т. е. получая управление от другой подпрограммы, сопрограмма выполняется лишь частично, а затем приостанавливается и возвращает управление вызвавшей ее подпрограмме. При повторной активации выполнение сопрограммы возобновляется с той точки, в которой было приостановлено выполнение. Общим для подпрограмм и сопрограмм является то, что одновременно может выполняться только одна подпрограмма или сопрограмма.

Допустим, что A и B являются сопрограммами. На рис. 1.12 проиллюстрирована передача управления между этими сопрограммами.

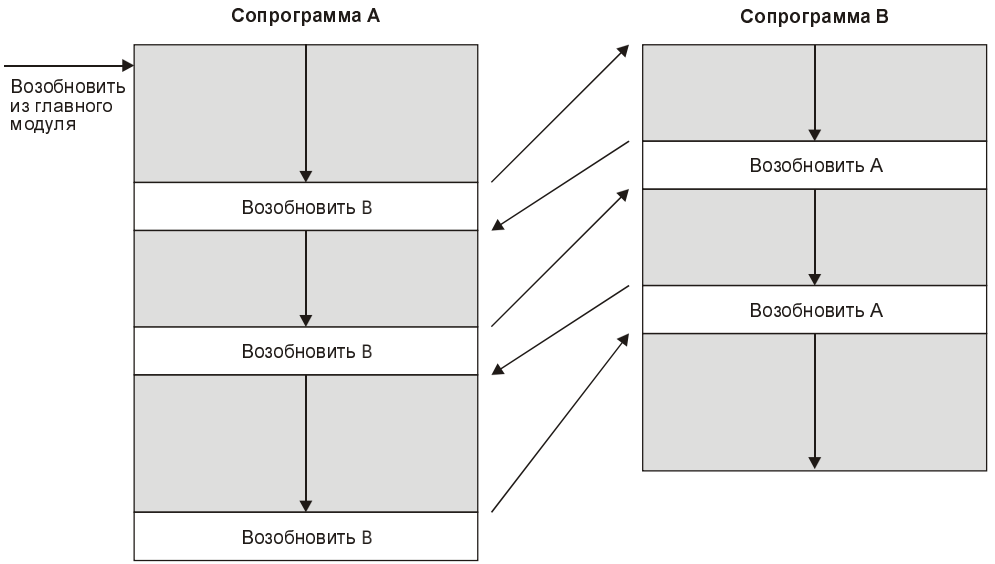


Рис. 1.12. Передача управления между сопрограммами A и B

При возобновлении выполнения сопрограммы ее локальные переменные сохраняют свои значения, полученные во время предыдущего выполнения.

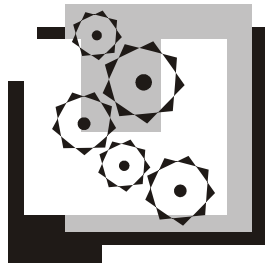
Сопрограммы как управляющие структуры в настоящее время используются только в языках дискретного моделирования. Вместе с тем для многих алгоритмов сопрограммы являются более естественной структурой управления, чем обычная иерархия вызовов подпрограмм. Простая структура сопрограмм может быть смоделирована с помощью оператора **goto** и переменной, определяющей метку оператора, с которого должно возобновляться выполнение сопрограммы.

Контрольные вопросы

1. Какие языки называются императивными?
2. Какие языки относят к языкам функционального программирования?
3. Какие языки являются декларативными?
4. Назовите три основных свойства объектно-ориентированных языков программирования.
5. Какую пользу можно извлечь из знания свойств языка программирования?
6. Как можно увеличить надежность языка программирования?
7. Как влияет удобочитаемость языка программирования на легкость создания программ на этом языке?
8. Что понимается под естественностью языка программирования?
9. Какие два критерия разработки языка программирования конфликтуют между собой?
10. Из каких составляющих складывается суммарная стоимость языка программирования?
11. Какое свойство языка программирования дает возможность более просто переносить программы с одной платформы на другие?
12. Что означает концептуальная целостность языка программирования?
13. Что понимается под объектом данных в языках программирования?
14. Чем отличается литерал от именованной константы?
15. С помощью каких атрибутов можно охарактеризовать переменную?
16. Что означает предопределенное имя?
17. Что такое неявное определение типа?
18. Когда осуществляется статическое связывание типа?
19. Назовите недостатки слабой типизации языка программирования.
20. Назовите основные признаки строгой типизации.
21. Каким образом производный тип наследует атрибуты?
22. Как определяется эквивалентность типов?
23. Чем подтип отличается от производного типа?
24. Что понимается под анонимным типом?
25. С какими проблемами связано использование указателей?
26. Что определяет область видимости переменных?

27. Из каких частей состоит среда ссылок?
28. Каким образом реализуется статическая область видимости имен?
29. Что понимается под динамической областью видимости имен?
30. Какие существуют разновидности числовых типов, и чем определяется их многообразие?
31. Какие операции определены для указателей?
32. Что содержит дескриптор вектора?
33. Как представляются в памяти многомерные массивы?
34. Какими атрибутами характеризуются записи?
35. Чем различаются свободные и размеченные объединения?
36. Какие существуют способы представления множеств в памяти?
37. Каким образом разрешаются коллизии при использовании хеширования?
38. Какие операции над списками наиболее часто используются?
39. Чем определяется порядок вычисления операций в арифметических выражениях?
40. Какие типы побочных эффектов встречаются в выражениях?
41. Что означает приведение типа?
42. Что такое перегруженный оператор?
43. Что означают сужающее и расширяющее преобразования типа?
44. Как выполняется сокращенное вычисление?
45. Что представляет собой составной оператор присваивания?
46. Какие типы управляющих структур имеются в языках программирования?
47. Как реализуются правила интерпретации вложенных условных операторов?
48. В каких случаях и почему более предпочтительным, чем оператор **case**, является многовариантный оператор ветвления **elsif**?
49. Для каких целей используется таблица переходов при реализации многовариантных операторов ветвления?
50. Какие проблемы возникают при разработке операторов цикла с параметром?
51. Какие проблемы возникают при разработке операторов цикла с логическим управлением?
52. Как выполняется оператор цикла, управляемый пользователем?

53. Что представляет собой спецификация подпрограммы?
54. В каком отношении между собой находятся формальные и фактические параметры?
55. Какие существуют режимы и механизмы передачи параметров?
56. Перечислите основные достоинства и недостатки передачи параметров по значению, по результату, по значению и результату.
57. Перечислите основные достоинства и недостатки передачи параметров по ссылке.
58. Перечислите основные достоинства и недостатки передачи параметров по имени.
59. Чем сопрограммы отличаются от обычных подпрограмм?



Глава 2

Описание языка программирования

Вопросы разработки, использования и реализации языков программирования тесно связаны с проблемой *точного описания* (определения) языка. Для определения языка требуется задать множество основных символов языка и описать его синтаксис и семантику. *Синтаксис* языка определяет правила составления корректных программ как цепочек, состоящих из основных символов языка. *Семантика* задает смысловые значения конструкций языка, а также интерпретацию различных синтаксических конструкций языковым процессором. Руководство по языку программирования должно включать в себя описание синтаксиса и семантики каждой конструкции языка, как по отдельности, так и в совокупности с другими конструкциями.

С точки зрения пользователя, описывающего на языке программирования алгоритмы решения своих задач, точное определение языка необходимо для того, чтобы он мог понимать, правильно применять и записывать конструкции этого языка. Разработчикам языковых процессоров (компиляторов, интерпретаторов) точное описание языка требуется для правильной реализации языкового процессора, поскольку вид и смысл языковых конструкций должны быть в точности такими, как их задумал автор языка. Все реализации языкового процессора должны выдавать идентичные результаты для одной и той же программы.

Основными функциями языкового процессора являются:

- контроль соответствия исходной программы описанию языка программирования;
- перевод правильной программы с языка программирования в машинный код, который может быть выполнен на процессоре (виртуальном или физическом).

Для реализации перечисленных функций средства описания синтаксиса и семантики языка должны иметь возможность определять синтаксис,

контекстные условия (статическую семантику языка) и отображение операций, предоставляемых языком программирования, в операции процессора, на котором будет выполняться программа (динамическую семантику языка).

2.1. Определение синтаксиса языка

Разработка языка программирования начинается с определения его синтаксиса. Естественный язык мало пригоден для этой цели, поэтому для точного описания синтаксиса языка программирования нужен некоторый вспомогательный язык. Язык, предназначенный для описания другого языка, называется *метаязыком*.

Метаязык задает систему обозначений, *понятий языка* и образованных из них конструкций, позволяющих представить описываемый язык с помощью определенных ранее понятий и отношений между ними. При этом каждое понятие языка подразумевает некоторую синтаксическую единицу (конструкцию) и определяемые ею свойства программных объектов или процесса обработки данных.

Для описания синтаксиса языков программирования наибольшее распространение получили:

- форма Бэкуса-Наура [1] и ее различные модификации;
- синтаксические диаграммы Вирта [21];
- формальные грамматики.

2.1.1. Форма Бэкуса-Наура

Форма Бэкуса-Наура (БНФ) представляет собой очень естественный способ описания синтаксиса. В БНФ каждое определяемое понятие — это *металингвистическая переменная*. Значением металингвистической переменной может быть любая конструкция из некоторого фиксированного для этого понятия набора конструкций. Каждая металингвистическая форма определяет одну металингвистическую переменную и состоит из двух частей: левой и правой. В левой части записывается определяемая металингвистическая переменная, которая заключается в угловые скобки '<' и '>' (предполагается, что эти скобки являются метасимволами и не принадлежат алфавиту определяемого языка), например: <двоичное число>, <метка>, <арифметическое выражение>. В правой части формы записываются все варианты определения конструкции, задаваемой этой формой. Каждый вариант представляет собой цепочку основных символов определяемого языка и металингвистических переменных. Варианты разделяются металингвистической связкой '|', имеющей смысл "или". Левая и правая части формы разделяются

метасимволом '::=', означающим "по определению есть". Например, следующие металингвистические формы определяют множество целых чисел:

<целое число> ::= <целое без знака> | +<целое без знака> | −<целое без знака>

<целое без знака> ::= <цифра> | <целое без знака ><цифра>

<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Характерной особенностью многих металингвистических форм является наличие в них *рекурсии*. Рекурсия имеет место в том случае, когда для определения конструкций языков программирования используются металингвистические переменные, обозначающие саму определяемую конструкцию. Рекурсия может быть явной или неявной. *Явная рекурсия* используется, например, в определении конструкции "двоичное целое". *Неявная рекурсия* имеет место в случае, когда металингвистическая переменная, обозначающая какую-либо конструкцию, используется на некотором промежуточном шаге определения этой конструкции.

Наличие рекурсивных определений затрудняет чтение и понимание БНФ, хотя и является наиболее удобным способом описания бесконечных языков с помощью конечного числа правил. На практике для описания синтаксиса языков программирования часто используют расширения БНФ, позволяющие более естественно представлять альтернативные, необязательные и повторяющиеся части металингвистических формул. Так, одно из расширений БНФ (РБНФ) разрешает использовать следующие упрощения:

- необязательные элементы синтаксической конструкции заключаются в квадратные скобки '**[**' и '**]**';
- альтернативные варианты могут в случае необходимости заключаться в квадратные скобки для образования многовариантного выбора;
- элементы синтаксической конструкции, повторяющиеся нуль и более раз, заключаются в фигурные скобки '{' и '}'.

С учетом указанных расширений приведенное ранее определение множества целых чисел можно записать в таком виде:

<целое число> ::= **[** + | − **]** <целое без знака>

<целое без знака> ::= <цифра> **{** <цифра> **}**

<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Оба описания эквивалентны, но второе описание более компактно и естественно. Любая синтаксическая конструкция, полученная с помощью РБНФ, может быть получена с помощью БНФ и наоборот.

2.1.2. Синтаксические диаграммы Вирта

Синтаксические диаграммы [21] были предложены Никлаусом Виртом для описания синтаксиса языка Pascal и являются удобной графической формой представления РБНФ.

Элементами синтаксических диаграмм Вирта являются: прямоугольники, кружки или овалы, стрелки. В прямоугольниках записываются имена металингвистических переменных, в кружках или овалах — основные символы языка, а стрелки определяют порядок сочетания металингвистических переменных и основных символов языка для образования определяемой синтаксической конструкции. Каждой синтаксической конструкции соответствует одна диаграмма Вирта. Имя определяемой синтаксической конструкции записывается над стрелкой, входящей в диаграмму (точка входа в диаграмму), которая, как правило, располагается в левом верхнем углу. Любой путь от точки входа в синтаксическую диаграмму к выходу (исходящая из диаграммы стрелка) представляет собой цепочку металингвистических переменных и основных символов языка, соответствующую одному из вариантов правой части РБНФ. На рис. 2.1 приведены синтаксические диаграммы, определяющие множество целых чисел.

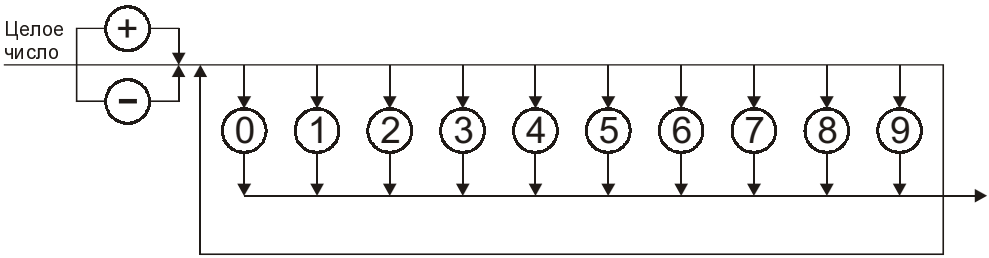


Рис. 2.1. Синтаксические диаграммы для определения множества целых чисел

БНФ, РБНФ и синтаксические диаграммы Вирта дают возможность косвенно включать в формальное описание синтаксиса языков программирования элементы семантики, т. к. в них входят металингвистические переменные, являющиеся осмысленными названиями описываемых конструкций. При использовании автоматических методов анализа языков элементы семантики, заложенные в эти формальные модели, теряют смысл, поэтому в теории и практике проектирования языковых процессоров используются формальные грамматики, которые будут рассмотрены в *гл. 3*.

Формальные грамматики и рассмотренные в *разд. 2.1* формальные модели не позволяют описать синтаксис языка программирования полностью. Некоторые его аспекты, например контекстные условия, могут быть описаны только семантическими правилами.

2.2. Описание контекстных условий

Контекстные условия характеризуют те синтаксические правила языков программирования, которые сложно, а иногда и невозможно описать при

помощи БНФ. В связи с тем, что контекстные условия связаны с описанием синтаксиса языка программирования и анализируются на этапе синтаксического анализа программы, их называют *правилами статической семантики* [39].

Рассмотрим наиболее характерные типы контекстных условий [6]:

1. Правила единственности именования различных объектов программы, таких, как переменные, подпрограммы, типы данных, константы, метки и т. п., устанавливаемые каждым языком программирования. Например:
   - пространства имен различных объектов программы не должны попарно пересекаться;
   - пространства имен различных объектов программы могут пересекаться.

   На самом деле в реальных языках программирования эти правила значительно сложнее. Даже для такого несложного языка, как Pascal, при задании правил именования объектов необходимо учитывать блочную структуру программы и особенности именования полей записей.
2. Контекстные условия, связанные с необходимостью описания идентификатора перед его использованием.
3. Контекстные условия, определяющие соответствие типов величин, входящих в конструкции языка программирования. Например, во многих языках программирования типы операндов операции должны быть совместимы.
4. Контекстные условия, связанные с *количественными ограничениями* и определяемые конкретной реализацией языка. Например, в некоторых реализациях может быть ограничена размерность массивов или глубина вложенности блоков.

Контекстные условия языков программирования могут быть описаны с помощью программных грамматик [6] и атрибутных транслирующих грамматик [29, 30, 34], которые являются расширениями контекстно-свободных грамматик.

2.3. Описание динамической семантики

В большинстве руководств по программированию определение динамической семантики языка дается в виде небольшого пояснительного текста, дополненного несколькими примерами использования конструкций языка или их совокупности. Такое описание семантики может быть интерпретировано пользователями языка неоднозначно, поэтому так же, как и для описания синтаксиса, необходимо иметь методы, позволяющие точно и лаконично определять динамическую семантику языка программирования. Хотя задача формального определения семантики языка решается по времени

столько же, сколько и задача определения синтаксиса, решение, удовлетворяющее большинство пользователей языка (программистов, разработчиков языка, разработчиков трансляторов с языка программирования), до сих пор не получено. Каждый из предложенных методов выглядит очень элегантно, когда применяется к языку, созданному специально для иллюстрации соответствующего метода, но становится громоздким и сложным при попытке применить его для описания семантики реального языка программирования.

Рассмотрим наиболее известные методы формального определения семантики.

2.3.1. Грамматические модели

В основе грамматических моделей описания семантики лежат различные способы расширения грамматик, описывающих синтаксис языков программирования. Наиболее распространенными описаниями такого типа являются транслирующие и атрибутные транслирующие грамматики. Их подробное описание и использование для описания контекстных условий и динамической семантики рассматриваются в *гл. 11.*

Другой способ описания семантики языков программирования основан на использовании двухуровневых грамматик (W-грамматик), которые предложил Аад ван Вейнгаарден для описания языка ALGOL 68 [31].

W-грамматика состоит из двух конечных множеств правил, называемых множествами *метаправил* и *гиперправил*. Метаправила — это БНФ, записываемые в несколько другой форме. Нетерминалы метаправил, называемые *метапонятиями*, записываются прописными буквами, например ЛИТЕРА. В каждом метаправиле определяются все альтернативы данного метапонятия. Символ '::' используется для разделения левой и правой частей метаправила, символ ';' разделяет альтернативы в правой части метаправила, а символ '.' используется для определения конца метаправила. Например, следующие метаправила:

(1) ЛИТЕРА :: **a**; **b**; ... ; **z**; **а**; **б**; ... ; **я**.

(2) ПОНЯТИЕ :: ЛИТЕРА;

ПОНЯТИЕ ЛИТЕРА.

порождают либо метапонятие ЛИТЕРА, которое в свою очередь порождает *протопонятие*, представленное произвольной строчной буквой латинского или русского алфавита, либо метапонятие ПОНЯТИЕ, за которым следует ЛИТЕРА.

Гиперправило — это заготовка, из которой получаются синтаксические правила. Например, гиперправило:

ПОНЯТИЕ **строка**: ПОНЯТИЕ;

ПОНЯТИЕ **строка**, ПОНЯТИЕ.

является прототипом правил, описывающих строки. Оно содержит метапонятия (ПОНЯТИЕ) и протопонятия (**строка**). В гиперправилах для разделения левой и правой частей используется символ ':', а разные протопонятия внутри одной альтернативы разделяются символом ','. Металингвистические формы получаются после замены в гиперправиле всех метапонятий на протопонятия, выводимые из метапонятий с помощью метаправил.

Для рассмотренного ранеее примера метаправила (1) и (2) позволяют вывести из метапонятия ПОНЯТИЕ такие протопонятия, как **идентификатор** и **значение**. Заменяя в гиперправиле ПОНЯТИЕ этими протопонятиями, можно получить следующие БНФ:

значение последовательность: **значение;**

значение последовательность, значение

идентификатор последовательность: **идентификатор;**

идентификатор последовательность, идентификатор

Металингвистическими переменными полученных БНФ являются **значение последовательность**, **значение**, **идентификатор последовательность**, **идентификатор**.

Описанный метод подстановки используется в W-грамматиках для порождения бесконечного множества БНФ, позволяющих описывать синтаксис и семантику языка.

2.3.2. Операционная семантика

В основе операционной модели определения динамической семантики [31, 37, 39] лежит описание того, как программы, составленные на некотором языке программирования, выполняются на виртуальном (или реальном) компьютере. Обычно виртуальный компьютер определяется как автомат (процессор), но гораздо более сложный по сравнению с моделями автоматов, используемых для реализации синтаксического анализа. Процессор может выполнять конечное множество формально определенных *операций*, которые изменяют его состояние. *Состояние* процессора соответствует состояниям программы при ее выполнении и определяется множеством значений, записанных в его память. Начиная с исходного состояния, процессор в соответствии с определяющими его правилами последовательно переходит к следующим состояниям, пока не достигнет конечного.

Семантика каждой конструкции языка программирования определяется *изменениями*, которые произошли *в состоянии процессора* после выполнения данной конструкции. Например, операционное определение семантики

оператора цикла некоторого языка программирования можно представить в следующем виде:

Оператор цикла

```
for id = Выражение1
       to Выражение2
       step Выражение3
do
       ТелоЦикла
next id
```

Операционная семантика

```
    id := Выражение1;
m1: if id > Выражение2
       then goto m2;
    ТелоЦикла
    id := id + Выражение3;
    goto m1;
m2:
```

Использование операционного метода для полного описания семантики языка программирования требует создания двух компонентов (рис. 2.2):

- *транслятора*, необходимого для преобразования операторов языка в операторы виртуального процессора;
- программно моделируемого *интерпретатора* команд виртуального процессора.

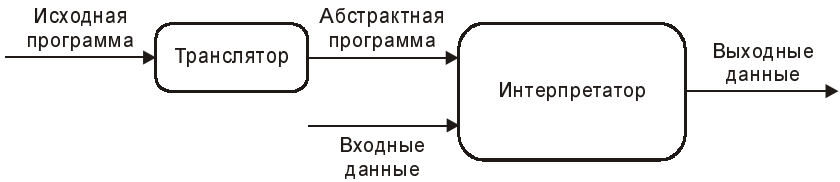


Рис. 2.2. Схематическое представление операционного метода описания семантики

Одним из первых метаязыков, предназначенных для описания операционной семантики языков программирования, является Венский метаязык (Vienna Definition Language), разработанный в венских лабораториях фирмы IBM для формального описания языка PL/1. В настоящее время операционный подход общепринят и используется во многих руководствах и стандартах языков программирования.

2.3.3. Аксиоматическая семантика

Аксиоматическая семантика была создана в процессе разработки метода доказательства правильности программ, заключающегося в доказательстве правильности вычислений, выполняемых программой в соответствии с ее спецификацией. Для доказательства правильности программы каждый оператор сопровождается предшествующим (*предусловием*) и последующим (*постусловием*) логическими выражениями, устанавливающими ограничения, накладываемые на значения переменных в программе.

Семантику каждой синтаксической конструкции языка определяют как конечное множество аксиом или правил вывода, которое можно использовать для вывода результатов выполнения этой конструкции. Считая, что значения входных переменных удовлетворяют некоторым ограничениям, аксиомы и правила вывода могут быть использованы для вывода ограничений на значения переменных после выполнения каждого оператора программы. Примеры использования аксиоматической семантики для описания семантики простейших языковых конструкций (оператор присваивания, последовательность операторов, оператор ветвления, цикл с предусловием) приведены в [39].

Аксиоматическая семантика является мощным инструментом для анализа программ. Однако ее использование для описания семантики языков программирования весьма ограничено из-за большой сложности определения аксиом и правил логического вывода для большинства конструкций языков программирования.

2.3.4. Денотационная семантика

Денотационная семантика — это формальная аппликативная модель для описания языков программирования. Основной концепцией денотационной семантики является определение для каждой сущности языка некоторого математического объекта и некоторой функции, отображающей экземпляры этой сущности в экземпляры этого объекта.

Рассмотрим использование денотационной семантики на примере определения двоичных чисел. Синтаксис двоичных чисел можно описать следующими БНФ:

<двоичное число> ::= **0**
| **1**
| <двоичное число> **0**
| <двоичное число> **1**

Объектами в данном примере являются десятичные числа. Для описания двоичного числа с использованием денотационной семантики и приведенных ранее синтаксических правил его фактическое значение связывается с каждым правилом, содержащим в своей правой части один основной символ определяемого языка. При этом значащие объекты (числа **0** и **1**) должны связываться с первыми двумя правилами, а другие два правила являются, по существу, правилами вычислений, поскольку они объединяют основной символ языка (**0** или **1**), с которым может ассоциироваться объект, с металингвистической переменной, которая представляет собой некоторую синтаксическую конструкцию.

Пусть область определения семантических значений объектов представляет собой множество неотрицательных десятичных целых чисел *N*. Эти объекты могут быть связаны с двоичными числами с помощью функции *M*, которая

отображает синтаксические объекты в объекты множества *N*. Функция *M* может быть определена следующим образом:

$M(\mathbf{0}) = 0$

$M(\mathbf{1}) = 1$

$M(\langle\text{двоичное число}\rangle\mathbf{0}) = 2 \times M(\langle\text{двоичное число}\rangle)$

$M(\langle\text{двоичное число}\rangle\mathbf{1}) = 2 \times M(\langle\text{двоичное число}\rangle) + 1$

Контрольные вопросы

1. Определите понятия "синтаксис" и "семантика" языка.
2. Что такое "метаязык"?
3. Какие расширения БНФ используются наиболее часто?
4. Определите наиболее характерные типы контекстных условий.
5. Определите понятия "статическая семантика" и "динамическая семантика".
6. Опишите основную концепцию W-грамматик и расскажите, как они используются для задания синтаксиса и семантики языков программирования?
7. Определите основную концепцию операционной семантики.
8. Опишите подход к определению семантики языка программирования с использованием аксиоматической семантики.
9. Определите основную концепцию денотационной семантики.
10. Чем отличается операционная семантика от денотационной?

Упражнения

1. Опишите синтаксис раздела описаний переменных и массивов типа `real`, `integer` и `char` языка Pascal.
2. Разработайте описание синтаксиса условного оператора **`if`** языка Pascal. Условия в этом операторе задаются логическими выражениями, составленными из простых переменных булевского типа и логических операций И, ИЛИ, НЕ, а в качестве операторов можно использовать только операторы присваивания, правой частью которых являются логические выражения. Приоритет операций обычный.
3. Задайте синтаксис оператора цикла **`for`** языка Pascal. В качестве операторов в теле цикла можно использовать только операторы присваивания, правой частью которых являются арифметические выражения, составленные из простых переменных и констант целого типа, круглых скобок и знаков арифметических операций: сложения, вычитания, умножения и деления. Приоритет операций обычный.
4. Опишите синтаксис оператора цикла **`while`** некоторого подмножества языка Pascal. Условия в этом операторе задаются с помощью отношений, а

в качестве операторов в теле цикла можно использовать только операторы присваивания, правой частью которых являются арифметические выражения, составленные из простых переменных и констант вещественного типа, круглых скобок и знаков арифметических операций: сложения, вычитания, умножения и деления. Приоритет операций обычный.

5. Определите синтаксис оператора цикла **`repeat`** языка Pascal. Условия в этом операторе задаются логическими выражениями, составленными из простых переменных булевского типа и логических операций И, ИЛИ, НЕ, а в качестве операторов в теле цикла можно использовать только операторы присваивания, правой частью которых являются логические выражения. Приоритет логических операций обычный.
6. Задайте синтаксис раздела описаний переменных и массивов типа `double`, `int` и `char` языка C.
7. Определите синтаксис условного оператора **`if`** языка C, в котором условия задаются с помощью отношений, а в качестве операторов, выполняемых в зависимости от значений условий, можно использовать только операторы присваивания. Правой частью операторов присваивания являются арифметические выражения, составленные из простых переменных и констант целого типа, круглых скобок и знаков арифметических операций: сложения, вычитания, умножения и деления. Приоритет операций обычный.
8. Опишите синтаксис условного оператора присваивания языка C. В правой части такого оператора для задания условий следует использовать отношения и арифметические выражения целого типа, составленные из простых переменных и констант целого типа, круглых скобок и знаков арифметических операций: сложения, вычитания, умножения и деления. Приоритет операций обычный.
9. Разработайте описание синтаксиса оператора цикла **`for`** языка C. В качестве операторов в теле цикла можно использовать только операторы присваивания, правой частью которых являются арифметические выражения целого типа, составленные из простых переменных и констант целого типа, круглых скобок и знаков арифметических операций: сложения, вычитания, умножения и деления. Приоритет операций обычный.
10. Определите синтаксис описания структур и переменных структурного типа в языке C.
11. Опишите синтаксис оператора цикла **`while`** некоторого подмножества языка C. Условия в этом операторе задаются с помощью отношений, а в качестве операторов в теле цикла можно использовать только операторы присваивания, правой частью которых являются арифметические выражения, составленные из простых переменных и констант целого типа, круглых скобок и знаков арифметических операций: сложения, вычитания, умножения и деления. Приоритет операций обычный.
12. Задайте синтаксис оператора цикла **`do ... while`** языка C. Условия в этом операторе представляют собой логические выражения, составлен-

ные из простых переменных булевского типа и логических операций И, ИЛИ, НЕ, а в качестве операторов в теле цикла можно использовать только операторы присваивания, правой частью которых являются логические выражения. Приоритет операций обычный.

13. Определите синтаксис задания перечислимого типа и описания переменных перечислимого типа в языке Pascal.

14. Опишите синтаксис оператора варианта **case** языка Pascal. В качестве операторов, среди которых производится выбор для исполнения, используйте только операторы присваивания, правой частью которых являются арифметические выражения, составленные из простых переменных и констант целого типа, круглых скобок и знаков арифметических операций: сложения, вычитания, умножения и деления. Приоритет операций обычный.

15. Задайте описание синтаксиса оператора цикла **repeat** языка Pascal. Условия в этом операторе задаются с помощью отношений, а в качестве операторов в теле цикла можно использовать только операторы присваивания, Правой частью операторов присваивания являются арифметические выражения, составленные из простых переменных и констант целого типа, круглых скобок и знаков арифметических операций: сложения, вычитания, умножения и деления. Приоритет операций обычный.

16. Определите синтаксис условного оператора **if** языка Pascal, в котором условия задаются с помощью отношений, а в качестве операторов, выполняемых в зависимости от значений условий, можно использовать только операторы присваивания. Правой частью которых являются арифметические выражения, составленные из простых переменных и констант целого типа, круглых скобок и знаков арифметических операций: сложения, вычитания, умножения и деления. Приоритет операций обычный.

17. Задайте синтаксис определения регулярного типа и описания переменных с индексами в языке Pascal.

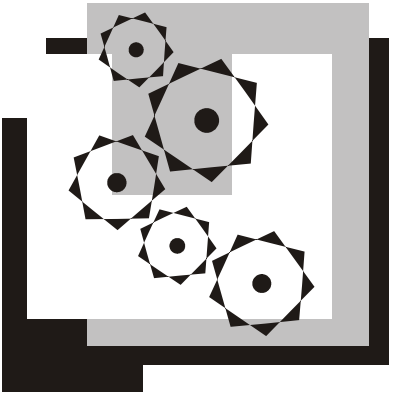
18. Задайте синтаксис определения строкового типа и описания строк в языке Pascal.

19. Разработайте определение синтаксиса описания процедур в языке Pascal, при этом операторы, составляющие тело процедуры, описывать не нужно.

20. Определите синтаксис описания комбинированного типа и переменных комбинированного типа в языке Pascal.

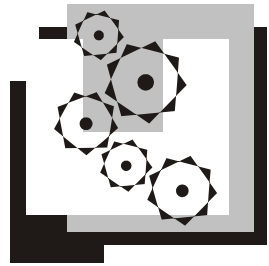
21. Задайте синтаксис описания перечислимого типа и переменных перечислимого типа в языке C.

22. Опишите синтаксис оператора условного ветвления **switch** языка C. В качестве операторов, подлежащих исполнению при совпадении выражения с case-константой, можно использовать только операторы присваивания, правой частью которых являются арифметические выражения, составленные из простых переменных и констант целого типа, круглых скобок и знаков арифметических операций: сложения, вычитания, умножения и деления. Приоритет операций обычный.



Часть II

Формальные грамматики и распознающие автоматы



Глава 3

Формальные грамматики и языки

3.1. Способы определения формальных языков

Тексты на любом языке (естественном, формальном, языке программирования) представляют собой цепочки символов некоторого алфавита. Однако не любая цепочка, составленная из символов заданного алфавита, *имеет смысл* в данном языке. Поэтому вопросы, связанные со способами определения "правильных" цепочек языка, являются основными при проектировании транслятора.

Назовем непустое множество элементов Σ *алфавитом*, а элементы этого множества — *символами*. *Цепочками* (*цепочками над алфавитом* Σ) называются конечные последовательности символов алфавита, представляющие собой размещения элементов из Σ с повторениями.

Пусть $\Sigma = \{a, b, c\}$. Тогда запись *bacbacbac* изображает цепочку над алфавитом Σ, первым элементом которой является символ *b* из алфавита Σ, вторым элементом — символ *a* из алфавита Σ и т. д.

Характеристикой цепочки является ее длина. *Длиной цепочки* называется число символов, образующих цепочку. Длина цепочки *bacbacbac* равна 9 и обозначается как $|bacbacbac|$. Цепочка, не содержащая ни одного символа, называется *пустой*. Пустую цепочку будем обозначать символом ε.

Приведенное определение не описывает механизма построения цепочек и не определяет операции над ними, которые можно использовать при их анализе и синтезе.

Полугруппой называется упорядоченная пара $(S, \bullet)$, где символ 'S' обозначает непустое множество, а точка '$\bullet$' — ассоциативную бинарную операцию, т. е. функцию отображения $(s_1, s_2) \rightarrow s_1 \bullet s_2$ из декартового произведения $S \times S$ в множество S, такую, что для любых элементов $s_1, s_2, s_3 \in S$ выполняется соотношение $(s_1 \bullet s_2) \bullet s_3 = s_1 \bullet (s_2 \bullet s_3)$. Обычно вместо $(S, \bullet)$ пишут сокращенно S, а вместо $s_1 \bullet s_2$ — $s_1 s_2$.

Мощность множества S называется *порядком* полугруппы S, и обозначается $|S|$.

Элемент $e \in S$ называется единицей тогда и только тогда, когда для любого элемента $s \in S$ выполняется равенство $se = s = es$. Элемент $e \in S$ называется нулем тогда и только тогда, когда для любого элемента $s \in S$ выполняется $es = e = es$.

Пусть множество A — непустое. Тогда множество всех конечных упорядоченных последовательностей элементов из A с групповой операцией, задаваемой конкатенацией последовательностей $(a_1, a_2, \ldots, a_n) \bullet (b_1, b_2, \ldots, b_m) = (a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m)$, называется *свободной некоммутативной полугруппой* A^*, *порожденной* A.

Например, если $\Sigma = \{a, b, c\}$, то свободная полугруппа, порожденная множеством Σ, будет содержать все конечные последовательности, составленные из элементов a, b и c, т. е. $\Sigma^* = \{\varepsilon, a, b, c, aa, ab, ac, aaa, \ldots\}$.

Языком L называется подмножество *свободной некоммутативной полугруппы* A^*, или $L \subseteq A^*$.

Данное определение языка имеет существенный недостаток: оно *неконструктивно*. Это означает, что для произвольной цепочки $w \in \Sigma^*$ нельзя определить, принадлежит ли цепочка w языку L.

Рассмотрим конструктивные способы определения языков.

Если язык состоит из конечного числа цепочек, то проблемы определения языка не существует: достаточно *перечислить* все цепочки языка. Но конечные языки встречаются крайне редко и только в теоретических исследованиях. Все языки программирования являются бесконечными, поэтому возникает проблема описания бесконечного языка конечными средствами. Рассмотрим возможные способы описания бесконечных языков:

1. *Словесное описание.* Язык определяется путем неформального описания общих свойств цепочек.

   Примеры словесного описания:

   - множество двоичных записей простых чисел: {1, 10, 11, 101, 111, 1011, ...};

- множество цепочек, состоящих из двух равных непустых групп нулей, разделенных единицей: {010, 00100, 0001000, 000010000, ...}.

2. *Алгебраическое описание.* Язык определяется как результат применения алгебраических операций к некоторому базисному семейству множеств цепочек.

   Примеры алгебраического описания:

   - $L_1 = \{0^n1^m \mid 0 \le n \le m\}$, $L_2 = \{0^n1^n \mid n \ge 1\}$ — базовые множества цепочек над словарем $\Sigma = \{0, 1\}$, $L = L_1 \cap L_2$ — множество цепочек, принадлежащих пересечению множеств L_1 и L_2;
   - $L_1 = \{0^n10^m \mid n > 0,\ m > 0\}$, $L_2 = \{1^k01^l \mid k > 0,\ l > 0\}$ — базовые множества цепочек над словарем $\Sigma = \{0, 1\}$. $L = L_1 \bullet L_2$ — язык, содержащий цепочки, префиксом которых являются цепочки из L_1, а суффиксом — цепочки из L_2.

   Словесное и алгебраическое описания позволяют задать ограниченное число простых языков, не имеющих практического применения.

3. *Порождающие правила.* При таком определении языка задается множество правил, называемых правилами вывода или порождающими правилами, которое порождает цепочки над Σ, принадлежащие языку L. Порождающие правила составляют основу формальных грамматик и БНФ, рассмотренных в *разд. 2.1.*

   Например, язык $L = L_1 \bullet L_2$, где $L_1 = \{0^n10^m \mid n > 0,\ m > 0\}$, $L_2 = \{1^k01^l \mid k > 0,\ l > 0\}$, может порождаться с помощью следующей совокупности БНФ:

   <цепочка> ::= <префикс> <суффикс>

   <префикс> ::= <последовательность нулей> 1 <последовательность нулей>

   <последовательность нулей> ::= 0 | 0 <последовательность нулей>

   <суффикс> ::= <последовательность единиц> 0 <последовательность единиц>

   <последовательность единиц> ::= 1 | 1 <последовательность единиц>

   С помощью приведенных порождающих правил (правил замены металингвистической переменной из левой части БНФ цепочкой из ее правой части) можно получить, например, следующую цепочку (символ, заменяемый на каждом шаге вывода, выделен полужирным шрифтом):

   <**цепочка**> $\Rightarrow$ <**префикс**> <суффикс> $\Rightarrow$

   <**последовательность нулей**> 1 <последовательность нулей> <суффикс> $\Rightarrow$

   0 <**последовательность нулей**> 1 <последовательность нулей> <суффикс> $\Rightarrow$

001 <**последовательность нулей**> <суффикс> ⇒

0010 <**суффикс**> ⇒

0010 <**последовательность единиц**> 0 <последовательность единиц> ⇒

001010 <**последовательность единиц**> ⇒

0010101 <**последовательность единиц**> ⇒ 00101011

4. *Распознающее устройство.* В этом случае определяется некоторое устройство (автомат), которое, принимая на входе цепочку из Σ^* и начиная работу из начального состояния, переходит в заключительное состояние тогда и только тогда, когда цепочка принадлежит языку. В этом случае говорят, что цепочка допускается автоматом.

   Рассмотрим автомат, распознающий язык $L = L_1 \bullet L_2$, где $L_1 = \{0^n 10^m \mid n > 0,\ m > 0\}$, $L_2 = \{1^k 01^l \mid k > 0,\ l > 0\}$. На рис. 3.1 приведена диаграмма переходов автомата, который для цепочки 00101011, начиная в начальном состоянии S_0, выполнит следующую последовательность переходов: $S_0 \Rightarrow S_1 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow S_3 \Rightarrow S_4 \Rightarrow S_5 \Rightarrow S_6 \Rightarrow S_6$. Автомат, прочитав всю цепочку, окажется в конечном состоянии S_6, следовательно, цепочка 00101011 принадлежит языку.

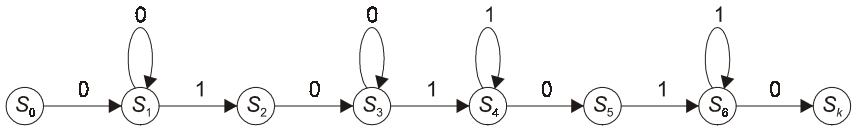


Рис. 3.1. Диаграмма перехода автомата, распознающего язык *L*

Остановимся более подробно на описании синтаксиса языков программирования с помощью формальных грамматик и их использовании при проектировании трансляторов.

3.2. Формальные грамматики

Формальные грамматики — наиболее широко используемый математический аппарат для описания синтаксиса языков программирования. Это связано с простотой и естественностью описания языка и его структуры, с одной стороны, и наличием автоматических средств, позволяющих преобразовать грамматику в программу синтаксического анализа, с другой стороны.

Формальная грамматика [2, 3, 9, 29] определяется с помощью двух словарей и конечного множества правил, описывающих процесс генерации цепочек языка.

Первый словарь, называемый *терминальным словарем*, состоит из основных символов языка, которые в теории формальных грамматик называются *терминальными символами* (*терминалами*). В дальнейшем терминалы будем

обозначать строчными латинскими буквами *a*, *b*, *c*, ... , *t* (иногда с подстрочными индексами), терминальный словарь — символом Σ, а цепочки терминальных символов — строчными латинскими буквами *u*, *v*, *w*, *x*, *y*, *z*.

Второй словарь называют *нетерминальным словарем* грамматики и включают в него вспомогательные символы, используемые для вывода цепочек языка. Символы, входящие в нетерминальный словарь, будем называть *нетерминальными символами* (*нетерминалами*) и обозначать их прописными латинскими буквами *A*, *B*, ... , *T*, а нетерминальный словарь будем обозначать буквой N. Словари терминальных и нетерминальных символов не должны пересекаться, т. е. $\Sigma \cap N = \varnothing$.

В словаре N выделяется нетерминал, называемый *начальным символом грамматики* и обозначаемый по умолчанию *S*. Начальный символ грамматики соответствует наиболее общей из описываемых конструкций языка (для языка программирования начальный символ грамматики соответствует конструкции "программа").

Терминальный и нетерминальный словари образуют полный словарь грамматики $V = \Sigma \cup N$. Одиночные символы этого словаря, которые могут быть как терминалами, так и нетерминалами, будем обозначать прописными латинскими буквами *U*, *V*, *W*, *X*, *Y*, *Z*. (в некоторых случаях с подстрочными индексами), а цепочки символов полного словаря — строчными греческими буквами.

Множество *правил вывода P* представляет собой конечное множество правил, которые описывают процесс порождения цепочек языка. Правило — это элемент множества $V^*NV^* \times V^*$. Далее правило вывода вида (α, β) будем записывать как $\alpha \rightarrow \beta$, считая, что символ $(\rightarrow)$ не входит в словарь V. Правило вывода $\alpha \rightarrow \beta$ рассматривается как правило формальной замены α на β, применимое к любой цепочке, содержащей цепочку α в качестве подцепочки.

Пусть, например, $\Sigma = \{a, b\}$, $N = \{S\}$, основной символ грамматики — *S* и множество правил вывода $P = \{S \rightarrow ab, S \rightarrow aSb\}$. Тогда для цепочки *S* можно применить правило $S \rightarrow ab$ и получить цепочку *ab*, состоящую из терминальных символов. К такой цепочке невозможно применить ни одного правила грамматики. Если к исходной цепочке *S* применить правило $S \rightarrow aSb$, то получим цепочку *aSb*. К новой цепочке также можно применить правило $S \rightarrow ab$ или правило $S \rightarrow aSb$. В первом случае получаем цепочку *aabb*, а во втором — цепочку *aaSbb* и т. д.

Грамматикой называется четверка $G = (N, \Sigma, P, S)$, где N — нетерминальный словарь, Σ — терминальный словарь, *P* — конечное подмножество множества $V^*NV^* \times V^*$, каждый элемент которого (α, β) называется правилом вывода и записывается в виде $\alpha \rightarrow \beta$, $S \in N$ — начальный символ грамматики.

Грамматика $G = (N, \Sigma, P, S)$ рекурсивно определяет *выводимые цепочки*:

1. S — выводимая цепочка;
2. Если $\alpha\beta\gamma$ — выводимая цепочка и $\beta \to \delta$ содержится в P, то $\alpha\delta\gamma$ — также *выводимая цепочка*.

Язык, порождаемый грамматикой G, обозначается $L(G)$. $L(G)$ — это множество терминальных цепочек, порождаемых грамматикой G.

Определим *отношение выводимости* для грамматики $G = (N, \Sigma, P, S)$.

1. Отношение *непосредственного вывода* $\Rightarrow_G$ определяется на множестве V^* следующим образом: если $\gamma_1\varphi\gamma_2 \in V^*$ и $\varphi \to \psi \in P$, то $\gamma_1\varphi\gamma_2 \Rightarrow_G \gamma_1\psi\gamma_2$.
2. Транзитивное замыкание отношения $\Rightarrow_G$ будем обозначать как $\Rightarrow^+_G$. Если имеет место отношение $\alpha \Rightarrow^+_G \beta$, то говорят, что цепочка β выводима из цепочки α *нетривиальным способом*.
3. Транзитивное и рефлексивное замыкание отношения $\Rightarrow_G$ будем обозначать как $\Rightarrow^*_G$. Если имеет место отношение $\alpha \Rightarrow^*_G \beta$, то говорят, что цепочка β выводима из цепочки α.
4. k-ую степень отношения $\Rightarrow_G$ будем обозначать как $\Rightarrow^k_G$. $\alpha \Rightarrow^k_G \beta$, если существует последовательность $\omega_0, \omega_1, \ldots, \omega_k$, состоящая из $k + 1$ цепочек (не обязательно различных), для которых $\alpha = \omega_0$, $\omega_{i-1} \Rightarrow \omega_i$ при $1 \le i \le k$ и $\omega_k = \beta$. Эта последовательность называется *выводом длины* k цепочки β из цепочки α в грамматике G.

   Очевидно, что $\alpha \Rightarrow^+_G \beta$ тогда и только тогда, когда $\alpha \Rightarrow^k_G \beta$ для некоторого $k > 0$, и $\alpha \Rightarrow^*_G \beta$ тогда и только тогда, когда $\alpha \Rightarrow^+_G \beta$ или $\alpha = \beta$.

Цепочка ω называется *сентенциальной формой*, если она выводима из начального символа грамматики, т. е. существует вывод $S \Rightarrow^* \omega$.

Предложением называется сентенциальная форма, состоящая только из терминалов.

Множество предложений грамматики G называется *языком, порождаемым грамматикой* G, и обозначается $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^+ w\}$.

Пример 3.1

Пусть задана грамматика $G = (\{S\}, \{a, b, c\}, P, S)$, у которой множество правил $P = \{S \to aSa, S \to bSb, S \to c\}$.

В этой грамматике возможны, в частности, следующие выводы:

$S \Rightarrow bSb \Rightarrow bcb$

$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabcbaa$

$S \Rightarrow bSb \Rightarrow baSab \Rightarrow baaSaab \Rightarrow baacaab$

Грамматика порождает язык $L(G) = \{xcx^r \mid x \in \{a, b\}^*\}$, е x^r — обращение цепочки x. □

Формальные грамматики являются частным случаем исчислений математической логики. Объектами, к которым в исчислении, представленном порождающей грамматикой, применяются правила вывода, являются цепочки символов над полным словарем грамматики. Единственная аксиома исчисления — цепочка, состоящая из начального символа грамматики, а правилами вывода являются правила из множества P.

Замечание

При получении вывода некоторой цепочки в грамматике на каждом шаге можно использовать любое правило из множества P, применяемое в данный момент, т. е. порядок применения правил вывода в грамматике произволен. Этим формальные грамматики существенно отличаются от алгоритмов, в которых всегда строго определена последовательность действий.

3.3. Классификация формальных грамматик

В определении грамматики, приведенном в *разд. 3.2*, на правила вывода $\alpha \rightarrow \beta$ не накладывалось никаких ограничений, за исключением того, что цепочка α должна содержать хотя бы один нетерминал. Для описания языков программирования и изучения их свойств наибольший интерес представляют грамматики, правила вывода которых удовлетворяют определенным требованиям.

В работах по теории формальных грамматик [3, 6, 9, 34, 35, 48] наиболее часто за основу принимается классификация Хомского, который выделил четыре класса формальных грамматик в зависимости от вида их правил.

- *Грамматика типа 0* или *грамматика общего вида* — это грамматика $G = (N, \Sigma, P, S)$, у которой правила вывода имеют вид $\alpha \rightarrow \beta$, где $\alpha \in V^*NV^*$, а $\beta \in V^*$. Грамматики общего вида не представляют практического интереса, т. к. они слишком свободны, чтобы их можно было использовать для описания синтаксиса языков программирования, и вместе с тем не настолько мощны, чтобы ими можно было пользоваться для задания синтаксиса естественных языков.

  Примером грамматики типа 0 является грамматика $G = (\{A, S\}, \{0, 1\}, P, S)$, где $P = \{S \rightarrow 0A1, 0A \rightarrow 00A1, A \rightarrow \varepsilon\}$.

К грамматикам типа 0 приближаются грамматики с фразовой структурой [7], в которых цепочка из левой части правила вывода $\alpha \rightarrow \beta$ должна состоять только из нетерминалов.

- *Грамматика типа 1* или *контекстно-зависимая грамматика* (*КЗ-грамматика*) — это грамматика $G = (N, \Sigma, P, S)$, у которой правила вывода имеют вид $\gamma_1 A \gamma_2 \rightarrow \gamma_1 \beta \gamma_2$, где $\gamma_1, \gamma_2 \in V^*$, $A \in N$, а $\beta \in V^+$. Цепочки γ_1 и γ_2 называются соответственно левым и правым контекстом. Они задают условия вывода: для любой цепочки замена нетерминала A цепочкой β возможна только в контексте γ_1, γ_2. Языки, порождаемые КЗ-грамматиками, называются *контекстно-зависимыми языками* (*КЗ-языками*).

  Можно доказать [9], что класс языков, порождаемых КЗ-грамматиками, совпадает с классом языков, порождаемых неукорачивающими грамматиками. *Неукорачивающей* называют грамматику, в которой для всех правил вывода $\alpha \rightarrow \beta$ выполняется условие $|\alpha| \le |\beta|$. Для неукорачивающей грамматики существенным является то, что в любом выводе длина цепочек может только возрастать. Это свойство позволяет доказать [6], что язык, порождаемый неукорачивающей грамматикой, *легко распознаваем*. Это значит, что существует алгоритм, который за конечное число шагов определяет, принадлежит произвольная терминальная цепочка языку или нет.

  Примером неукорачивающей грамматики является $G = (\{B, C, S\}, \{a, b, c\}, P, S)$, где $P = \{S \rightarrow aSBC,\ S \rightarrow abC,\ CB \rightarrow BC,\ bB \rightarrow bb,\ bC \rightarrow bc,\ cC \rightarrow cc\}$.

- *Грамматика типа 2* или *контекстно-свободная грамматика* (*КС-грамматика*) — это грамматика $G = (N, \Sigma, P, S)$, у которой правила вывода имеют вид $A \rightarrow \beta$, где $A \in N$, а $\beta \in V^*$. Языки, порождаемые КС-грамматиками, называются *контекстно-свободными языками* (*КС-языками*).

  В общем случае КС-языки не входят в класс КЗ-языков, т. к. КС-грамматики являются укорачивающими грамматиками. В дальнейшем будет показано, что по любой укорачивающей КС-грамматике можно построить неукорачивающую КС-грамматику, которая порождает тот же язык, что и исходная КС-грамматика, за исключением пустой цепочки. Таким образом, языки, порождаемые КС-грамматиками, можно считать легко распознаваемыми, т. к. нетрудно дополнительно проверить, входит ли пустая цепочка в язык.

  Запись правил вывода в КС-грамматике по сути совпадает с формой записи металингвистических формул в нотации БНФ. Действительно, нетерминалу из левой части правила вывода КС-грамматики соответствует металингвистическая переменная из левой части металингвистической формулы. Отношению ($\rightarrow$) — металингвистическая связка (::=), а цепочке терминалов и нетерминалов из правой части правила

вывода — цепочка металингвистических переменных и основных символов языка из правой части металингвистической формулы.

Пример КС-грамматики выглядит следующим образом: $G = (\{E, T, P\}, \{i, +, *, (,)\}, P, E)$, где $P = \{E \to E + T,\ E \to T,\ T \to T * P,\ T \to P,\ P \to i,\ P \to (E)\}$.

- *Грамматика типа 3* или *автоматная* (*регулярная*) *грамматика* — это грамматика $G = (N, \Sigma, P, S)$, у которой правила вывода имеют вид $A \to aB$ или $A \to a$, где $A, B \in N$, $a \in \Sigma \cup \{\varepsilon\}$. Языки, порождаемые автоматными грамматиками, являются подмножеством КС-языков и называются *автоматными* (*регулярными*).

  Примером автоматной грамматики является грамматика $G = (\{A, S\}, \{0, 1\}, P, S)$, где $P = \{S \to 0A, A \to 1A, A \to 0A, A \to 0, A \to 1\}$.

3.4. Выводы и деревья выводов

Остановимся более подробно на способах представления выводов цепочек в КС-грамматиках, используя *размеченный вывод*. В размеченном выводе каждый шаг вывода отмечается порядковым номером применяемого правила к сентенциальной форме.

Рассмотрим грамматику G_0, порождающую скобочные арифметические выражения некоторого языка программирования.

(1) $E \to E + T$

(2) $E \to T$

(3) $T \to T * P$

(4) $T \to P$

(5) $P \to i$

(6) $P \to (E)$

Построим несколько размеченных выводов предложения $i + i * i$ в этой грамматике, отличающиеся порядком применения правил вывода:

1. $E \Rightarrow_{(1)} E + T \Rightarrow_{(2)} T + T \Rightarrow_{(4)} P + T \Rightarrow_{(5)} i + T \Rightarrow_{(3)} i + T * P \Rightarrow_{(4)} i + P * P \Rightarrow_{(5)} i + i * P \Rightarrow_{(5)} i + i * i.$
2. $E \Rightarrow_{(1)} E + T \Rightarrow_{(3)} E + T * P \Rightarrow_{(5)} E + T * i \Rightarrow_{(4)} E + P * i \Rightarrow_{(5)} E + i * i \Rightarrow_{(2)} T + i * i \Rightarrow_{(4)} P + i * i \Rightarrow_{(5)} i + i * i.$
3. $E \Rightarrow_{(1)} E + T \Rightarrow_{(3)} E + T * P \Rightarrow_{(2)} T + T * P \Rightarrow_{(5)} T + T * i \Rightarrow_{(4)} P + T * i \Rightarrow_{(4)} P + P * i \Rightarrow_{(5)} i + P * i \Rightarrow_{(5)} i + i * i.$

В первом случае на каждом шаге вывода заменялся *самый левый* нетерминал сентенциальной формы, во втором случае — *самый правый*, а в третьем выводе выбор нетерминала для замены был произвольным. Все рассмотренные

выводы эквивалентны в том смысле, что в них применяются одни и те же правила в одних и тех же местах, но в различном порядке.

При представлении вывода в виде последовательности сентенциальных форм приходится многократно выписывать неизменяющиеся части цепочек (контекст, в котором применяются правила вывода). Другим недостатком такого представления вывода является то, что сентенциальные формы не содержат никакой информации о синтаксической структуре выводимой цепочки. Для КС-грамматик существует удобное графическое представление класса выводов, которое называется *деревом вывода* или *синтаксическим деревом*.

Дерево вывода в КС-грамматике $G = (N, \Sigma, P, S)$ — это размеченное упорядоченное дерево, каждая вершина которого отмечена символом из множества $\Sigma \cup N \cup \{\varepsilon\}$. Если внутренняя вершина дерева помечена символом A, а ее прямые потомки помечены символами $X_1, X_2, \ldots, X_n$, то в грамматике существует правило $A \to X_1, X_2, \ldots, X_n$. Рассмотрим пример.

Пример 3.2

Рассмотрим левый вывод в грамматике G_0: $E \Rightarrow_{(1)} E + T \Rightarrow_{(2)} T + T \Rightarrow_{(4)} P + T \Rightarrow_{(5)} i + T \Rightarrow_{(3)} i + T * P \Rightarrow_{(4)} i + P * P \Rightarrow_{(5)} i + i * P \Rightarrow_{(5)} i + i * i$ и будем интерпретировать подстановки, выполняемые в нем, как *шаги построения дерева вывода*. Процесс построения дерева приведен на рис. 3.2—3.5, где рамкой отмечен куст дерева, представляющий собой правило грамматики, использованное на данном шаге построения.

В начале вывода (рис. 3.2, *а*) сентенциальная форма состоит из основного символа грамматики E, которому соответствует дерево с одной вершиной (корнем), помеченной символом E. На первом шаге вывода к символу E применяется правило грамматики номер (1), и он заменяется правой частью этого правила $E + T$. Соответствующий шаг построения дерева состоит в том, что к корню дерева (вершине, помеченной основным символом грамматики E) добавляются прямые потомки — вершины, помеченные символами 'E', '+' и 'T'. Полученное на данном шаге дерево приведено на рис. 3.2, *б*.

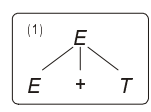
	Цепочка до подстановки	Правило подстановки	Цепочка после подстановки	Дерево вывода
а)	E			E
б)	E	(1) $E \to E + T$	$E + T$	

Рис. 3.2. Первый (*а*) и второй (*б*) шаги построения дерева вывода цепочки $i + i * i$

На рис. 3.3, *а* символ 'E' заменяется правой частью второго правила грамматики. Заметим, что этот символ является меткой листа дерева, полученного на предыдущем шаге. Шаги построения дерева, приведенные на рис. 3.3, *б* и рис. 3.3, *в* отражают применение правил вывода грамматики номер (4) и (5) соответственно. Если выписать слева направо листья дерева, построенного на каждом шаге вывода, получится соответствующая сентенциальная форма.

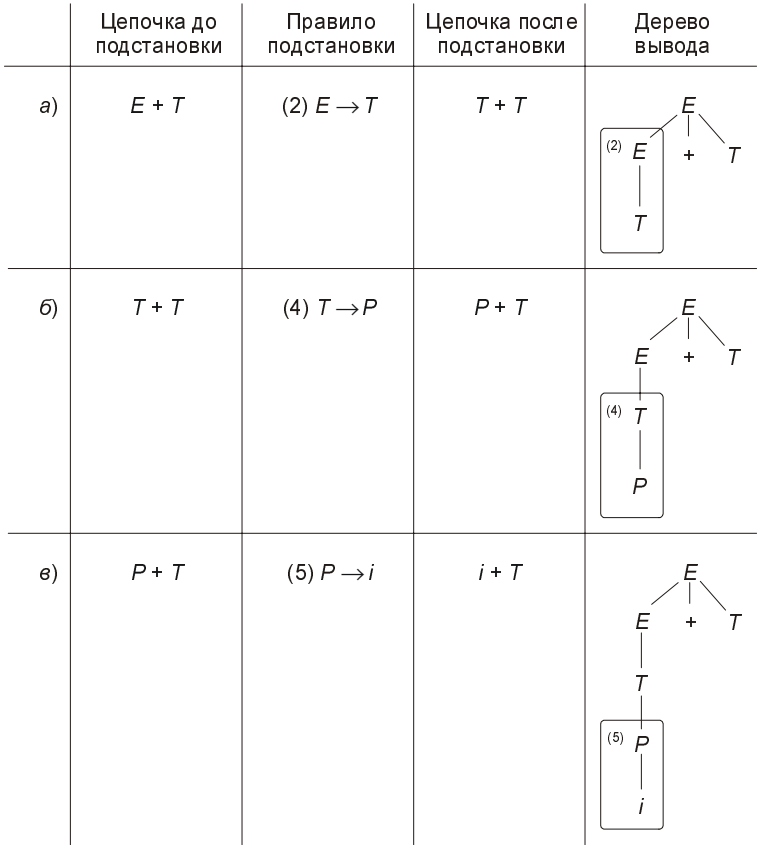


Рис. 3.3. Третий (*а*), четвертый (*б*) и пятый (*в*) шаги построения дерева вывода цепочки $i + i * i$

На рис. 3.4 изображено выполнение трех очередных шагов построения дерева вывода.

Окончательный вид дерева вывода цепочки $i + i * i$ приведен на рис. 3.5. Заметим, что если выписать слева направо листья дерева, построенного на последнем шаге вывода, то получится цепочка $i + i * i$, принадлежащая языку $L(G_0)$. □

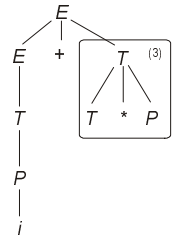
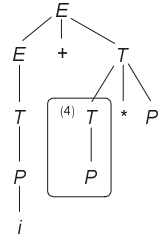
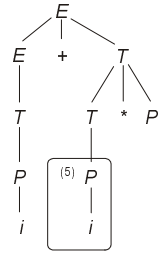
	Цепочка до подстановки	Правило подстановки	Цепочка после подстановки	Дерево вывода
а)	$i + T$	(3) $T \to T * P$	$i + T * P$	
б)	$i + T * P$	(4) $T \to P$	$i + P * P$	
в)	$i + P * P$	(5) $P \to i$	$i + i * P$	

Рис. 3.4. Шестой (*а*), седьмой (*б*) и восьмой (*в*) шаги построения дерева вывода цепочки $i + i * i$

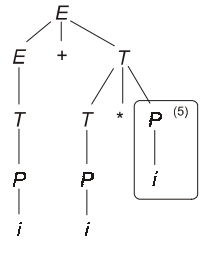
Цепочка до подстановки	Правило подстановки	Цепочка после подстановки	Дерево вывода
$i + i * P$	(5) $P \to i$	$i + i * i$	

Рис. 3.5. Дерево вывода цепочки $i + i * i$

Синтаксическое дерево отражает, какие правила были использованы при выводе цепочки и к каким вхождениям нетерминальных символов они были применены. Единственное ограничение заключается в том, что при выполнении очередного шага правила грамматики должны применяться к каждой внутренней вершине дерева раньше, чем к вершинам, расположенным ниже.

Дерево вывода не несет информации о том, в каком порядке эти правила были использованы при выводе, поэтому может существовать множество выводов, соответствующих одному дереву. Дерево, приведенное на рис. 3.5, соответствует не только рассматриваемому левому выводу, но и правому, и произвольному выводам цепочки $i + i * i$, упомянутым ранее.

Приведем формальное определение дерева вывода [3, 6].

Помеченное упорядоченное дерево D называется *деревом вывода* (*синтаксическим деревом*, *деревом разбора*) в КС-грамматике $G = (N, \Sigma, P, S)$, если выполняются следующие условия:

1. Корень дерева D помечен основным символом грамматики S.
2. Если $D_1, D_2, \ldots, D_k$ — поддеревья, являющиеся прямыми потомками корня дерева S (рис. 3.6), корни которых, в свою очередь, помечены соответственно символами $X_1, X_2, \ldots, X_k$, то в грамматике G существует правило $S \rightarrow X_1, X_2, \ldots, X_k$.
3. Если $X_i \in N$, то поддерево D_i должно быть деревом вывода в грамматике $G^i = (N, \Sigma, P, X_i)$, где X_i — начальный символ грамматики.
4. Если $X_i \in \Sigma$, то поддерево D_i состоит из одной вершины, помеченной терминалом X_i.
5. Если корень дерева имеет единственного потомка, помеченного ε, то этот потомок образует дерево, состоящее из единственной вершины, и в множестве правил грамматики имеется правило $S \rightarrow \varepsilon$.

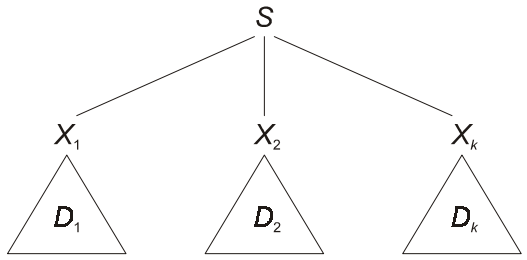


Рис. 3.6. Построение дерева вывода из поддеревьев

Пример 3.3

🗁 На рис. 3.7 изображены деревья выводов цепочек в грамматике G, имеющей следующие правила:

(1) $S \to bAb$ (4) $A \to a$

(2) $S \to \varepsilon$ (5) $B \to Aad$

(3) $A \to cB$

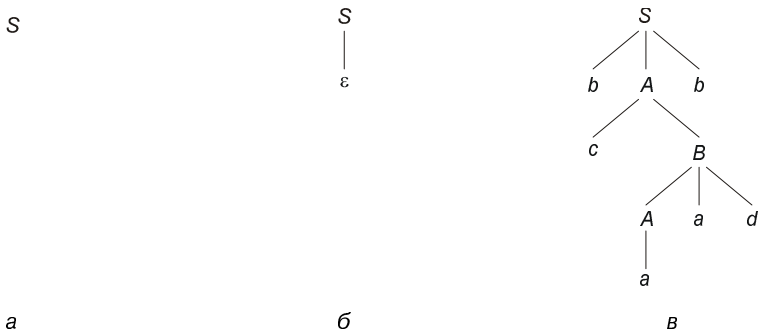


Рис. 3.7. Деревья выводов некоторых цепочек, порождаемых грамматикой G

На рис. 3.7, *а* приведено дерево, состоящее только из корня, соответствующее началу любого вывода в грамматике, на рис. 3.7, *б* — дерево при выводе пустой цепочки ε, а на рис. 3.7, *в* — дерево вывода цепочки *bcaadb*. 🗀

Кроной дерева вывода называется цепочка, полученная выписыванием слева направо меток листьев.

Примеры крон деревьев, изображенных на рис. 3.7 следующие: корень дерева S, пустая цепочка ε и цепочка *bcaadb*.

Возникает вопрос, можно ли для каждого вывода цепочки α, выводимой в грамматике G, построить дерево вывода с кроной α и обратно?

Введем ряд дополнительных понятий, связанных с синтаксическим деревом вывода.

Сечением D дерева вывода в КС-грамматике $G = (N, \Sigma, P, S)$ называется такое множество C его вершин, что никакие две вершины из C не лежат на одном пути в D и ни одну вершину дерева D нельзя добавить к C, не нарушив предыдущего свойства.

Кроной сечения дерева *D* назовем цепочку, полученную конкатенацией (слева направо) меток вершин, образующих это сечение.

По определению, множество вершин, состоящее из одного корня, является сечением. Множество листьев дерева вывода также образует сечение. На рис. 3.8 изображено дерево вывода цепочки *abababbba* в грамматике $G = (\{A, S\}, \{a, b\}, \{S \to aSbS, S \to bSaS, S \to \varepsilon\}, S)$. Множество его вершин, обведенных кружками, представляет собой сечение *abaSbaSbabS*.

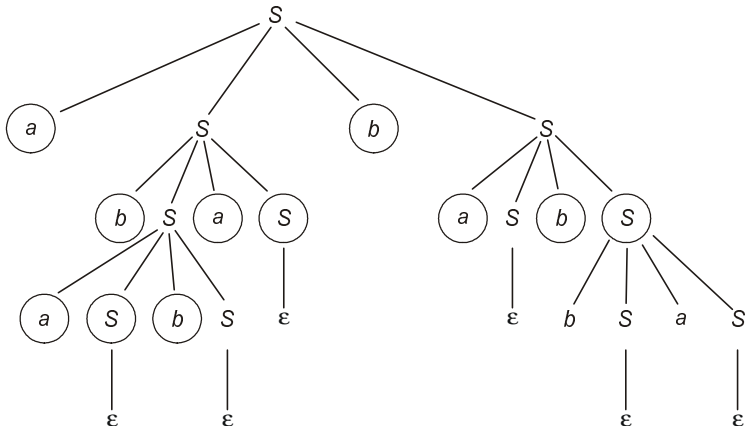


Рис. 3.8. Дерево вывода цепочки *abababbba*

Утверждение

Если $\alpha_0, \alpha_1, \ldots, \alpha_n$ — вывод цепочки α_n из S в КС-грамматике $G = (N, \Sigma, P, S)$, то в этой грамматике можно построить дерево вывода D, для которого α_n — крона , а цепочки $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$ — некоторые из крон сечений.

Доказательство

□ Пусть D_0 — дерево, состоящее из одной вершины, которая помечена основным символом грамматики S. Построим последовательность деревьев выводов D_i $(0 \le i \le n)$ так, чтобы цепочки α_i представляли собой кроны деревьев D_i.

Предположим, что на некотором шаге вывода $\alpha_i = \beta_i A \gamma_i$ и после применения правила грамматики $A \to X_1 X_2 \ldots X_k$ к вхождению нетерминала A мы получим цепочку $\alpha_{i+1} = \beta_i X_1 X_2 \ldots X_k \gamma_i$. Тогда дерево D_{i+1} (рис. 3.9, *б*) получается из дерева D_i (рис. 3.9, *а*) добавлением к листу, отмеченному нетерминалом A, k прямых потомков, которые помечаются символами $X_1, X_2, \ldots, X_k$ (рис. 3.9, *а*). Очевидно, что кроной дерева D_{i+1} будет цепочка α_{i+1}, а дерево D_n будет искомым деревом вывода D. ■

Рис. 3.9. Шаги построения дерева вывода

Пусть D — дерево вывода в КС-грамматике $G = (N, \Sigma, P, S)$ с кроной α. Очевидно, что существует хотя бы одна последовательность $C_0, C_1, \ldots, C_n$ сечений дерева D, обладающая следующими свойствами:

- сечение C_0 содержит только корень дерева D;
- сечение C_{i+1} для $0 \le i < n$ получается из C_i заменой одной нетерминальной вершины ее прямыми потомками;
- сечение C_n — крона дерева D.

Другими словами, если α_i — крона сечения N_i, то $\alpha_0, \alpha_1, \ldots, \alpha_n$ — вывод цепочки α_n из α_0 в грамматике G.

Из доказанных утверждений следует, что в КС-грамматике $G = (N, \Sigma, P, S)$ существует вывод $S \Rightarrow^* \alpha$ тогда и только тогда, когда в грамматике G существует дерево вывода с кроной α.

Если сечение C_{i+1} получается из сечения C_i заменой *самой левой нетерминальной вершины* в C_i ее прямыми потомками, то соответствующий вывод $\alpha_0, \alpha_1, \ldots, \alpha_n$ называется *левым выводом* цепочки α_n из α_0 в грамматике G.

Если сечение C_{i+1} получается из сечения C_i заменой *самой правой нетерминальной вершины* в C_i ее прямыми потомками, то соответствующий вывод $\alpha_0, \alpha_1, \ldots, \alpha_n$ называется *правым выводом* цепочки α_n из α_0 в грамматике G.

Замечание

Заметим, что если $S = \alpha_0, \alpha_1, \ldots, \alpha_n$, где $\alpha_0 = S$ и $\alpha_n = w$ — левый вывод терминальной цепочки w, то каждая промежуточная цепочка α_i $(0 \le i < n)$ имеет вид $x_i A_i \beta_i$, где $x_i \in \Sigma^*$, $A_i \in N$ и $\beta_i \in (N \cup \Sigma)^*$.

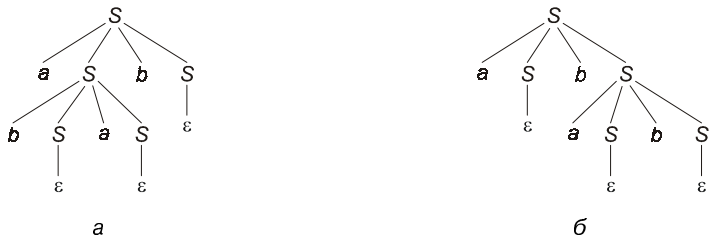
Цепочка α называется *левовыводимой* в грамматике $G = (N, \Sigma, P, S)$, если существует левый вывод $S = \alpha_0, \alpha_1, \ldots, \alpha_n = \alpha$. Будем записывать левый вывод в виде $S \Rightarrow^*_{Gl} \alpha$ или $S \Rightarrow^*_{l} \alpha$, если из контекста понятно, какая грамматика G имеется в виду.

Цепочка α называется *правовыводимой* в грамматике $G = (N, \Sigma, P, S)$, если существует правый вывод $S = \alpha_0, \alpha_1, \ldots, \alpha_n = \alpha$. Будем записывать правый вывод в виде $S \Rightarrow^*_{Gr} \alpha$ или $S \Rightarrow^*_r \alpha$, если из контекста понятно, какая грамматика G имеется в виду.

При выполнении синтаксического анализа КС-языков используются только левые и правые выводы.

Замечание

Если задан вывод $S \Rightarrow^* \alpha$ в КС-грамматике, то это не означает, что существует единственное дерево вывода в этой грамматике с кроной α. Существуют КС-грамматики, у которых может быть несколько деревьев выводов с одной и той же кроной. Например, в КС-грамматике $G = (\{A, S\}, \{a, b\}, \{S \to aSbS, S \to bSaS, S \to \varepsilon\}, S)$ существует вывод $S \Rightarrow^* abab$, для которого можно построить два дерева вывода, приведенных на рис. 3.10.



а *б*

Рис. 3.10. Два дерева вывода цепочки *abab*

3.5. Неоднозначность грамматик

Если грамматика используется для описания языка программирования или разработки языкового процессора, она должна быть однозначной. В противном случае пользователи языка программирования или разработчики языкового процессора могут интерпретировать семантику некоторых конструкций языка по-разному. Неоднозначность грамматик, описывающих синтаксис языков программирования, связана с тем, что языки программирования не являются КС-языками. КС-грамматики позволяют описать только часть синтаксических правил языка, а остальные его свойства (семантика, контекстные условия) описываются при помощи средств, рассмотренных в *гл. 2*.

КС-грамматика G называется *неоднозначной*, если существует хотя бы одна цепочка $w \in L(G)$, которая является кроной двух или более деревьев выводов, или, что то же самое, некоторая цепочка $w \in L(G)$ имеет два или более разных левых (правых) выводов.

Пример 3.4

🗁 Рассмотрим грамматику, описывающую арифметические выражения некоторого языка программирования. Правила грамматики имеют следующий вид:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow i$

Эта грамматика неоднозначна, т. к. для цепочки $i + i * i$ можно построить два дерева вывода (рис. 3.11). Другим недостатком данной грамматики является невозможность задания в ней старшинства операций. 🗀

Рис. 3.11. Деревья выводов цепочки $i + i * i$

Пример 3.5

🗁 Условный оператор, включенный во многие языки программирования, часто описывается при помощи грамматики с правилами:

$S \rightarrow$ **if** b **then** S **else** S

$S \rightarrow$ **if** b **then** S

$S \rightarrow a$

где b — логическое выражение, а a — безусловный оператор. Эта грамматика неоднозначна, т. к. в ней возможны два левых вывода цепочки **if** b **then if** b **then** a **else** a:

1. $S \Rightarrow$ **if** b **then** S **else** $S \Rightarrow$ **if** b **then if** b **then** S **else** $S \Rightarrow$ **if** b **then if** b **then** a **else** $S \Rightarrow$ **if** b **then if** b **then** a **else** a.
2. $S \Rightarrow$ **if** b **then** $S \Rightarrow$ **if** b **then if** b **then** S **else** $S \Rightarrow$ **if** b **then if** b **then** a **else** $S \Rightarrow$ **if** b **then if** b **then** a **else** a.

Два различных дерева вывода цепочки **if** b **then if** b **then** a **else** a приведены на рис. 3.12. 🗀

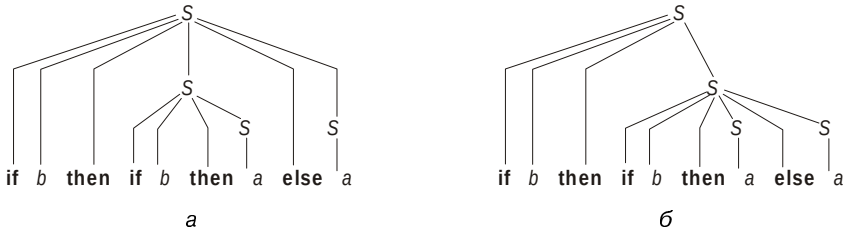


Рис. 3.12. Деревья вывода цепочки **if** *b* **then if** *b* **then** *a* **else** *a*

Наличие двух различных деревьев вывода предполагает две различные интерпретации цепочки **if** *b* **then if** *b* **then** *a* **else** *a*: первая из них — это **if** *b* **then** (**if** *b* **then** *a*) **else** *a*, а вторая — **if** *b* **then** (**if** *b* **then** *a* **else** *a*). В описании языка эту неоднозначность преодолевают достаточно просто, добавляя в семантику фразу типа "... ключевое слово **else** ассоциируется с ближайшим слева ключевым словом **if** ...". При реализации языкового процессора такой подход требует дополнительной семантической обработки входной программы, поэтому обычно грамматику преобразуют, например, следующим образом:

(1) $S_1 \rightarrow$ **if** *b* **then** S_1

(2) $S_1 \rightarrow$ **if** *b* **then** S_2 **else** S_1

(3) $S_1 \rightarrow a$

(4) $S_2 \rightarrow$ **if** *b* **then** S_2 **else** S_2

(5) $S_2 \rightarrow a$

Рассмотрим левый вывод цепочки **if** *b* **then if** *b* **then** *a* **else** *a* в этой грамматике:

$S_1 \Rightarrow_{(1)}$ **if** *b* **then** $S_1 \Rightarrow_{(2)}$ **if** *b* **then if** *b* **then** S_2 **else** $S_1 \Rightarrow_{(5)}$ **if** *b* **then if** *b* **then** *a* **else** $S_1 \Rightarrow_{(3)}$ **if** *b* **then if** *b* **then** *a* **else** *a*.

В связи с тем, что в преобразованной грамматике ключевому слову **else** предшествует только нетерминал S_2 или терминал *a*, существует единственный вывод цепочки **if** *b* **then if** *b* **then** *a* **else** *a*, причем ее интерпретация соответствует принятой в языках программирования (рис. 3.13).

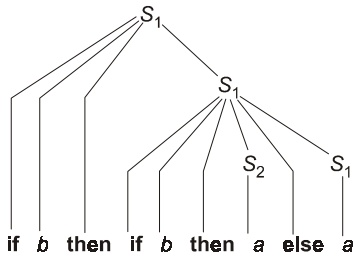


Рис. 3.13. Дерево вывода цепочки **if** *b* **then if** *b* **then** *a* **else** *a*

Ранее были рассмотрены неоднозначные грамматики. В некоторых случаях, изменяя правила грамматики, можно получить однозначную грамматику, порождающую тот же самый язык, в других случаях это сделать невозможно. В общем случае ответить на вопрос, является ли заданная КС-грамматика однозначной или нет, невозможно, т. к. *проблема однозначности произвольной КС-грамматики алгоритмически не разрешима* [3, 6, 9]. Это означает, что нельзя составить алгоритм, который за конечное число шагов определял бы, однозначна КС-грамматика или нет. Однако для некоторых больших подклассов КС-грамматик доказано, что они однозначны. Например, в [6] доказано, что все *детерминированные КС-языки* (*см. разд. 5.4*) определяются однозначными КС-грамматиками.

КС-язык называется *неоднозначным* (*существенно неоднозначным*), если не существует порождающей его однозначной КС-грамматики.

Примером неоднозначного языка является язык $L = \{a^i b^j c^i \mid i = j$ или $j = i\}$ (доказательство его неоднозначности приведено в [3]).

Хотя нет общего алгоритма, определяющего, является ли произвольная КС-грамматика однозначной, доказано, что если КС-грамматика содержит определенного вида группы правил, то такая КС-грамматика является неоднозначной. Приведем наиболее часто встречающиеся группы таких правил при условии, что $A \in N$, $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$:

- $A \to AA$ и $A \to \alpha$;
- $A \to A\alpha A$ и $A \to \beta$;
- $A \to \alpha A$, $A \to A\beta$ и $\to \gamma$;
- $A \to \alpha A$, $A \to \alpha A \beta A$ и $A \to \gamma$.

Например, грамматика из примера 3.4 содержит правила $E \to E + E$ и $E \to E$, которые относятся к первой группе правил, а грамматика из примера 3.5 — правило $S \to$ **if** b **then** S **else** S, т. е. правила вида $A \to \alpha A$, $A \to \alpha A \beta A$ из четвертой группы, в результате чего эти грамматики неоднозначны.

Замечание

Приведенное условие является *необходимым условием* неоднозначности КС-грамматик.

3.6. Непустые, конечные и бесконечные языки

КС-грамматики позволяют описывать формальные языки и языки программирования, поэтому важно обсудить следующие алгоритмические проблемы:

- проблема пустоты КС-языка;

- проблема бесконечности КС-языка;
- проблема принадлежности произвольной цепочки языку, порождаемому заданной КС-грамматикой.

3.6.1. Непустые языки

Очевидно, что необходимым и достаточным условием непустоты языка, порождаемого КС-грамматикой $G = (N, \Sigma, P, S)$, является существование в ней вывода $S \Rightarrow^* x$, где цепочка $x \in \Sigma^+$. Проблема существования такого вывода разрешима (*см. разд. 3.7.1*).

3.6.2. Бесконечные языки

В связи с тем, что множество нетерминальных символов КС-грамматики $G = (N, \Sigma, P, S)$ конечно, необходимым условием бесконечности языка $L(G)$ является неограниченность длин выводимых из S терминальных цепочек. Из этого условия следует, что длины сентенциальных форм также не должны быть ограничены.

Решение проблемы бесконечности можно разбить на две части:

1. Нахождение необходимых и достаточных условий бесконечности множества порождаемых грамматикой $G = (N, \Sigma, P, S)$ сентенциальных форм.
2. Нахождение дополнительных условий бесконечности множества порождаемых грамматикой терминальных цепочек или бесконечности языка $L(G)$.

Для решения первой задачи КС-грамматике $G = (N, \Sigma, P, S)$ удобно сопоставить граф, построенный по следующим правилам:

- каждому нетерминальному символу A грамматики сопоставить вершину графа и пометить ее символом A;
- каждому терминальному символу t_i ($i = 1, 2, \ldots, n$), входящему в правило грамматики вида $A \to t_1 t_2 \ldots t_n$, сопоставить вершину графа и пометить ее символом t_i;
- если правило $A \to \alpha B \beta \in P$, то добавить в граф дугу AB;
- для терминального правила $A \to t_1 t_2 \ldots t_n$ добавить в граф дуги At_1, At_2, ... , At_n.

Пример 3.6

На рис. 3.14 приведен граф для грамматики G_0 со следующими правилами:

(1) $E \to E + T$ (4) $T \to P$

(2) $E \to T$ (5) $P \to i$

(3) $T \to T * P$ (6) $P \to (E)$

Граф содержит вершины, помеченные нетерминальными символами E, T, P и терминальным символом i, входящим в единственное терминальное правило грамматики G_0. Заметим, что граф содержит *петли* и *циклы*. □

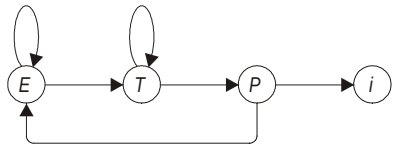


Рис. 3.14. Граф, построенный по грамматике G_0

Очевидно, что если граф, соответствующий КС-грамматике $G = (N, \Sigma, P, S)$, содержит цикл, который начинается и заканчивается в вершине A, то в такой грамматике существует вывод $A \Rightarrow \alpha A \beta$. Для исключения тривиального (и ничего не дающего при выводе) случая $A \Rightarrow^* A$ положим $|\alpha| + |\beta| \neq 0$. Поскольку в грамматике G должен существовать вывод $S \Rightarrow^* \gamma A \delta$, то, учитывая цикл, получим, что множество сентенциальных форм, порождаемых грамматикой, содержит бесконечное число цепочек вида $\gamma \alpha^n A \beta^n \delta$.

Если граф не имеет циклов, то правило $A \rightarrow \alpha B \beta$ не должно содержать в правой части нетерминала A, правило $B \rightarrow \gamma C \delta$ не должно содержать символов A и B и т. д. При выводе в такой грамматике нетерминальные символы будут исчезать из сентенциальных форм и (через конечное число шагов) исчезнут полностью. Очевидно, что множество сентенциальных форм, порождаемое такой грамматикой, не может быть бесконечным. Значит, для того чтобы множество сентенциальных форм, порождаемых грамматикой $G = (N, \Sigma, P, S)$, было бесконечным, необходимо и достаточно, чтобы для некоторого нетерминального символа A существовал вывод $A \Rightarrow \alpha A \beta$, для которого $|\alpha| + |\beta| \neq 0$.

Можно доказать, что проблема существования такого нетерминального символа алгоритмически разрешима. Вторая задача решается также просто, поэтому проблема бесконечности КС-языков *алгоритмически разрешима.*

3.6.3. Проблема принадлежности

Эту проблему неформально можно сформулировать следующим образом: "*Принадлежит ли цепочка* $w \in \Sigma^*$ *языку* $L(G)$?" Ответить на этот вопрос можно, если рассмотреть *все возможные* сентенциальные формы грамматики G, имеющие длину $|w|$. Если ни одна из них не соответствует выводу цепочки w, то $w \notin L(G)$, в противном случае $w \in L(G)$. Такой путь решения проблемы принадлежности весьма неэффективен. Другой проблемой, имеющей практическое значение, является проблема *выводимости*, т. е. построения реального синтаксического дерева заданной строки w в грамматике G. Пути решения этих проблем подробно рассматриваются *в ч. III книги.*

3.7. Эквивалентные преобразования КС-грамматик

Практика показывает, что решение проблемы принадлежности и проблемы выводимости возможно только при наложении определенных ограничений на правила КС-грамматики.

Рассматриваемые далее алгоритмы преобразования КС-грамматик позволяют получить грамматики, эквивалентные исходным.

Эквивалентными называются грамматики, порождающие один и тот же язык.

Доказано [12], что *проблема эквивалентности* произвольных КС-грамматик является алгоритмически не разрешимой. Однако имеется ряд полезных эквивалентных преобразований КС-грамматик, приводящих к упрощению разбора предложений этого языка.

Замечание

При преобразованиях грамматики нужно помнить, что грамматика определяет синтаксическую структуру языка, поэтому любые ее преобразования приводят к изменению этой структуры.

Конечно, приведенные алгоритмы не исчерпывают множества возможных преобразований грамматик. Существует достаточно много других алгоритмов, которые позволяют преобразовать грамматики к заданному классу грамматик [3, 25].

3.7.1. Удаление бесполезных символов

При разработке грамматики языка могут возникнуть следующие ошибки:

- в грамматике имеются нетерминальные символы, которые не участвуют в выводе терминальных цепочек;
- в грамматике существуют символы, недоступные из начального символа грамматики.

Грамматика в этом случае должна быть преобразована.

Нетерминальный символ $A \in N$ называется *производящим*, если из него можно вывести терминальную цепочку, т. е. если существует вывод $A \Rightarrow^{+} x$, где $x \in \Sigma^{+}$. В противном случае символ называется *непроизводящим*.

Символ $X \in \Sigma \cup N$ называется *недостижимым* в КС-грамматике G, если он не может появиться ни в одной из сентенциальных форм.

Символ $X \in \Sigma \cup N$ называется *бесполезным* в КС-грамматике G, если он непроизводящий или недостижимый.

Процедура удаления бесполезных символов из КС-грамматики заключается в удалении сначала непроизводящих, а затем недостижимых символов.

Алгоритм 3.1. *Определение множества производящих нетерминальных символов КС-грамматики*

Вход: КС-грамматика $G = (N, \Sigma, P, S)$.

Выход: Множество производящих нетерминальных символов $N_p = \{A \mid A \Rightarrow^+ x, A \in N, x \in \Sigma^+\}$.

Описание алгоритма:

□ Построить рекурсивно множества производящих нетерминалов $N^0{}_p$, $N^1{}_p$, ... , $N^i{}_p$, ... следующим образом:

1. Положить $N^0{}_p = \varnothing$, $i = 1$.
2. Положить $N^i{}_p = N^{i-1}{}_p \cup \{A \mid A \to \alpha \in P, \alpha \in (N^{i-1}{}_p \cup \Sigma)^+\}$
3. Если $N^i{}_p \neq N^{i-1}{}_p$, положить $i = i + 1$ и перейти к шагу 2.
4. Положить $N_p = N^i{}_p$.

■

При построении множества $N^i{}_p$ используется следующее свойство производящих нетерминалов: если все символы цепочки из правой части правила вывода являются производящими (терминалы или нетерминалы, принадлежащие множеству $N^{i-1}{}_p$ производящих символов, построенному на предыдущем шаге алгоритма), то нетерминал в левой части правила вывода также должен быть производящим. Поскольку $N_p \subseteq N$, то число повторений шага 2 алгоритма не превышает $n + 1$, где n — число нетерминальных символов грамматики G.

Докажем, что алгоритм 3.1 корректно определяет множество производящих нетерминалов КС-грамматики. Доказательство проведем с помощью индукции по i.

Необходимость: Если $A \in N^i{}_p$, то $A \Rightarrow^+ x$ для некоторой цепочки $x \in \Sigma^+$.

Доказательство

□ При $i = 0$ доказательство не требуется, т. к. $N^0{}_p = \varnothing$.

Предположим, что утверждение истинно для некоторого i. Рассмотрим $A \in N^{i+1}{}_p$. Если $A \in N^i{}_p$, то шаг индукции тривиален. Если $A \in N^{i+1}{}_p \setminus N^i{}_p$, то существует правило $A \to X_1\ X_2\ ...\ X_k$, где для всех $i \le j \le k$ $X_j \in N^i{}_p$ или $X_j \in \Sigma$. Таким образом, для каждого X_j можно найти такую цепочку $x_j \in \Sigma^+$, что $X_j \Rightarrow^* x_j$: если $X_j \in \Sigma$, то $x_j = X_j$, иначе существование x_j следует из истинности предположения индукции. Последнее утверждение справедливо для всех $i \le j \le k$, поэтому шаг индукции выполнен. Окончательно имеем:

$$A \Rightarrow X_1\ X_2\ ...\ X_k \Rightarrow^* x_1 X_2\ ...\ X_k \Rightarrow^* ... \Rightarrow^* x_1 x_2\ ...\ x_k,$$

что и требовалось доказать. ■

Достаточность: Если $A \Rightarrow^+ x$ для некоторой цепочки $x \in \Sigma^+$, то $A \in N_p$.

Доказательство

□ Из определения множества N_p следует, что если $N_p = N_p{}^{i-1}$, то $N_p = N_p{}^{i+1} = N_p{}^{i+2} = ...$. Поэтому достаточно показать, что $A \in N^i{}_p$ для некоторого i. Индукцией по m докажем, что если $A \Rightarrow^m x$, то $A \in N^i{}_p$ для некоторого i. При $m = 1$ имеет место тривиальный случай, и $i = 1$. Допустим, что утверждение истинно для некоторого m. Рассмотрим вывод $A \Rightarrow^{m+1} x$. Пусть $A \Rightarrow X_1\ X_2\ ...\ X_k \Rightarrow^m x$, где $x = x_1\ ...\ x_k$ и $X_j \Rightarrow^{mj} x_j$ для всех $i \le j \le k$ и $m_j \le m$. Согласно утверждению индукции, если $X_j \in N$, то $X_j \in N^{ij}{}_p$ для некоторого i_j. Если $X_j \in \Sigma$, то $X_j \in N^0{}_p$ ($i_j = 0$). Пусть $i = max\ (i_1, ...\ , i_k)$. Тогда, по определению, $A \in N^i{}_p$ и шаг индукции окончен. ■

Замечание

Из доказательства следует, что для КС-грамматики $G = (N, \Sigma, P, S)$ *проблема пустоты языка* $L(G)$ *разрешима*, т. е. если $S \in N_p$, то $L(G) \ne \varnothing$.

Множество достижимых символов КС-грамматики можно получить с помощью следующего алгоритма.

Алгоритм 3.2. *Определение множества достижимых символов КС-грамматики*

Вход: КС-грамматика $G = (N, \Sigma, P, S)$.

Выход: Множество достижимых символов $N_r = \{X \mid S \Rightarrow^* \alpha X \beta,\ X \in (\Sigma \cup N),\ \alpha, \beta \in (\Sigma \cup N)^*\}$.

Описание алгоритма:

□ Построить рекурсивно множества достижимых символов $N^0{}_r$, $N^1{}_r$, ... , $N^i{}_r$, ... , следующим образом:

1. Положить $N^0{}_r = \{S\}$, $i = 1$.
2. Положить $N^i{}_r = N^{i-1}{}_r \cup \{X \mid A \to \alpha X \beta \in R$ и $A \in N^{i-1}{}_r\}$.

3. Если $N^i_r \neq N^{i-1}_r$, положить $i = i + 1$ и перейти к шагу 2.
4. Положить $N_r = N^i_r$.

■

Процедура построения достижимых символов основана на следующем свойстве: если нетерминал в левой части правила грамматики является достижимым, то достижимы и все символы правой части этого правила.

Так как $N_r \subseteq \Sigma \cup N$, то число повторений шага 2 алгоритма не превышает $m + n$, где m — число терминалов, а n — число нетерминалов грамматики G.

Доказательство корректности данного алгоритма рекомендуется доказать самостоятельно с помощью индукции в качестве упражнения.

Алгоритм удаления бесполезных символов из КС-грамматики заключается в удалении сначала непроизводящих, а затем недостижимых символов.

Алгоритм 3.3. *Устранение бесполезных символов*

Вход: КС-грамматика $G = (N, \Sigma, P, S)$, для которой $L(G) \neq \varnothing$.

Выход: КС-грамматика $G = (N', \Sigma', P', S)$, у которой $L(G) \neq \varnothing$ и в $\Sigma' \cup N'$ нет бесполезных символов.

Описание алгоритма:

□

1. Построить множество N_p производящих нетерминалов грамматики G.
2. Положить $G_1 = (N_p, \Sigma, P_1, S)$, где P_1 состоит из правил множества P, содержащих только символы из $\Sigma \cup N_p$.
3. Построить множество N_r достижимых символов грамматики G_1.
4. Положить $G' = (N', \Sigma', P', S)$, где $\Sigma' = \Sigma \cap N_r$, $N' = N_r \cap N$, а P' состоит из правил множества P_1, содержащих только символы из множества N_r.

■

Заметим, что для исключения из КС-грамматики бесполезных символов алгоритмы 3.1 и 3.2 следует применять в порядке, указанном в алгоритме 3.3, т. е. вначале исключить непроизводящие нетерминалы, а затем недостижимые символы. Если изменить порядок применения этих алгоритмов, то преобразованная КС-грамматика может содержать бесполезные символы.

Пример 3.7

🗁 Требуется исключить из грамматики $G = (N, \Sigma, P, S)$, где $N = \{A, B, C, S\}$, $\Sigma = \{a, b, c\}$, $P = \{S \to aC, S \to A, A \to cAB, B \to b, C \to a\}$, бесполезные символы.

После выполнения шага 1 алгоритма 3.3 множество $N_p = \{B, C, S\}$.

После исключения непроизводящих нетерминалов получим новую грамматику $G_1 = (\{B, C, S\}, \{a, b, c\}, P, S)$, где $P = \{S \to aC, B \to b, C \to a\}$.

Множество достижимых символов грамматики G_1 $N_r = \{a, C, S\}$.

После исключения множества недостижимых символов из грамматики G_1 получим грамматику $G' = (\{C, S\}, \{a\}, \{S \to aC, C \to a\}, S)$, $L(G') = \{aa\}$.

Если применить к исходной грамматике сначала алгоритм 3.2, то получим, что *все символы* достижимы. Множество производящих символов грамматики $G_1 = \{B, C, S\}$. После исключения непроизводящих символов из грамматики G_1 будем иметь $G' = (\{B, C, S\}, \{a, b, c\}, P', S)$, где $P' = \{S \to aC, B \to b, C \to a\}$, $L(G') = L(G) = \{aa\}$, но грамматика G' содержит бесполезные символы B и b. □

3.7.2. Преобразование КС-грамматики с ε-правилами в эквивалентную неукорачивающую КС-грамматику

Некоторые методы синтаксического анализа не применимы к КС-грамматикам, содержащим ε-правила, поэтому на практике часто бывает удобно исключить из нее такие правила. В случае, когда язык содержит пустую цепочку, избавиться от ε-правил и не изменить порождаемый язык невозможно. Однако можно преобразовать КС-грамматику G таким образом, чтобы полученная грамматика G' была без ε-правил и порождала язык, "почти эквивалентный" исходной грамматике, т. е. чтобы выполнялось условие $L(G') = L(G) \setminus \{\varepsilon\}$.

КС-грамматика $G = (N, \Sigma, P, S)$ называется *неукорачивающей КС-грамматикой* (НКС-грамматикой, КС-грамматикой без ε-правил) при условии, что:

- либо множество P не содержит ε-правил;
- либо имеется только одно ε-правило $S \to \varepsilon$ и S не встречается в правых частях остальных правил вывода.

Нетерминальный символ A, порождающий пустую цепочку, называется *укорачивающим* нетерминалом.

Алгоритм преобразования КС-грамматики с ε-правилами в эквивалентную НКС-грамматику основан на использовании множества укорачивающих нетерминалов, которое может быть построено с помощью следующего алгоритма.

Алгоритм 3.4. *Построение множества укорачивающих нетерминалов*

Вход: КС-грамматика $G = (N, \Sigma, P, S)$.

Выход: Множество укорачивающих нетерминалов: $N_\varepsilon = \{A \mid A \Rightarrow^+ \varepsilon, A \in N\}$.

Описание алгоритма:

□ Требуется построить рекурсивно множества укорачивающих нетерминалов N^0_ε, N^1_ε, ... , N^i_ε, ... следующим образом:

1. Положить $N^0_\varepsilon = \{A \mid A \to \varepsilon, A \in N\}$, $i = 1$.
2. Положить $N^i_\varepsilon = N^{i-1}_\varepsilon \cup \{A \mid A \to \alpha \in P \text{ и } \alpha \in (N^{i-1}_\varepsilon)^+\}$.
3. Если $N^i_\varepsilon \neq N^{i-1}_\varepsilon$, положить $i = i + 1$ и перейти к шагу 2.
4. Положить $N_\varepsilon = N^i_\varepsilon$.

■

Так как $N_\varepsilon \subseteq N$, число повторений шага 2 алгоритма не превышает $n + 1$, где n — число нетерминалов грамматики G.

Замечание

При определении множеств N^i_ε можно рассматривать только правила вывода, не содержащие в правых частях терминалы.

Алгоритм 3.5. *Преобразование КС-грамматики с ε-правилами в эквивалентную НКС-грамматику*

Вход: КС-грамматика $G = (N, \Sigma, P, S)$.

Выход: НКС-грамматика $G' = (N', \Sigma, P', S')$, порождающая язык $L(G') = L(G)$.

Описание алгоритма:

□ Построить множество N_ε укорачивающих нетерминалов.

Построить P' следующим образом:

1. Положить $P' = \varnothing$.
2. Если $A \to \alpha_0 B_1 \alpha_1 B_2 \alpha_2 \ldots B_k \alpha_k \in P$, где $k \geq 0$, $B_j \in N_\varepsilon$ $(1 \leq j \leq k)$ и ни один из символов цепочек $\alpha_i \in (\Sigma \cup N)^*$ $(0 \leq i \leq k)$ не содержит символов из N_ε, то включить в P' все правила вывода вида $A \to \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \ldots X_k \alpha_k$, где $X_j = B_j$ или $X = \varepsilon$. Если все цепочки $\alpha_i = \varepsilon$, то правило $A \to \varepsilon$ не включать в P'.
3. Если $S \in N_\varepsilon$, то положить $N' = N \cup \{S'\}$ и добавить в P' два правила: $S' \to S$ и $S' \to \varepsilon$. В противном случае положить $N' = N$ и $S' = S$.
4. Положить $G' = (N', \Sigma, P', S')$.

■

Замечание

Если правая часть правила вывода КС-грамматики с ε-правилами содержит k нетерминалов $B_j \in N_\varepsilon$, то в множество P' включается 2^k правил, соответствующих всевозможным способам исключения символа B_j из правой части исходного правила.

Докажем, что алгоритм 3.5 строит НКС-грамматику, эквивалентную исходной КС-грамматике. Непосредственно из множества P' видно, что грамматика G' является неукорачивающей грамматикой. Для доказательства того, что что $L(G) = L(G')$, докажем вспомогательное утверждение: $A \Rightarrow^*_{G'} x$ *для любых* $A \in N$ *и* $x \in \Sigma^+$ *тогда и только тогда, когда* $A \Rightarrow^*_G x$.

Необходимость: Если $A \Rightarrow^*_{G'} x$, то $A \Rightarrow^*_G x$ для любых $A \in N$ и $x \in \Sigma^+$.

Доказательство

□ Если $B \to \beta \in P'$, то $B \Rightarrow^*_G \beta$. Отсюда следует, что любой вывод, корректный в грамматике G', будет корректным и в грамматике G. ■

Достаточность: Если $A \Rightarrow^*_G x$, то $A \Rightarrow^*_{G'} x$ для любых $A \in N$ и $x \in \Sigma^+$.

Доказательство

□ Докажем это утверждение методом индукция по числу шагов вывода n.

При $n = 0$ получится тривиальный случай. Если $n = 1$, то P содержит одно правило вывода, например $A \to x$. Поскольку $x \neq \varepsilon$, то это правило принадлежит и множеству P' грамматики G'. Предположим, что утверждение индукции справедливо для всех $n \le m$ шагов вывода.

Рассмотрим вывод $A \Rightarrow^{m+1} x$. Очевидно, что множество P должно содержать правило вывода $A \to X_1 \ldots X_k$, где $X_i \in (\Sigma \cup N)$. Разобьем строку x на подстроки $x_1 \ldots x_k$ таким образом, что если $X_i \in N$, то $X_i \Rightarrow^*_G x_i$, а если $X_i \in \Sigma$, то $X_i = x_i$ и, следовательно, $X_i \Rightarrow^*_G x_i$. Некоторые подстроки x_i могут оказаться пустыми, т. к. грамматика G имеет ε-правила. Пусть индексы $i \le i_1 < i_2 < \ldots < i_r \le k$ такие, что $x_{i_1}, x_{i_2}, \ldots, x_{i_r}$ — непустые строки из последовательности подстрок $x_1 \ldots x_k$. Тогда $x = x_{i_1} x_{i_2} \ldots x_{i_r}$ и $X_{i_j} \Rightarrow^*_G x_{i_j}$, где $j = 1, \ldots, r$. Каждый из таких выводов состоит из числа шагов вывода, не превышающих k, и для него справедливо утверждение, что $X_i \Rightarrow^*_{G'} x_{i_j}$. Поскольку правило вывода $A \to X_{i_1} \ldots X_{i_k}$, то $A \Rightarrow^{k+1}_{G'} x$, откуда по индукции получаем, что исходное утверждение верно для любого числа шагов вывода. ■

Если в качестве нетерминала A взять начальный символ грамматики S, то из доказанного утверждения следует, что $S \Rightarrow^*_G x$ тогда и только тогда, когда $S \Rightarrow^*_{G'} x$, а следовательно, $x \in L(G)$ тогда и только тогда, когда $x \in L(G')$ для любой цепочки $x \in \Sigma^+$. Очевидно, что $\varepsilon \in L(G)$ тогда и только тогда, когда $\varepsilon \in L(G')$. Следовательно, $L(G) = L(G')$.

Пример 3.8

🗁 Преобразовать КС-грамматику $G = (N, \Sigma, P, S)$, где $N = \{A, S\}$, $\Sigma = \{b, c\}$, $P = \{S \to cA,\ S \to \varepsilon,\ A \to cA,\ A \to bA,\ A \to \varepsilon\}$, в эквивалентную НКС-грамматику.

Используя алгоритм 3.4, получаем: $N^0_\varepsilon = \{S, A\}$, $N^1_\varepsilon = \{S, A\}$, значит $N_\varepsilon = N^1_\varepsilon = \{S, A\}$.

Далее, выполняя шаги алгоритма 3.5, имеем:

Шаг 1. $P' = \varnothing$.

Шаг 2. Рассмотрим правило $S \to cA$. Для него в новое множество правил грамматики P' нужно включить правила $S \to cA$ и $S \to c$. Для правила $A \to cA$ требуется добавить в множество P' правила $A \to cA$ и $A \to c$, а для правила $A \to bA$ — правила $A \to bA$ и $A \to b$. На данном шаге алгоритма получаем $P' = \{S \to cA, S \to c, A \to cA, A \to c, A \to bA, A \to b\}$

Шаг 3. В рассматриваемой грамматике $S \in N_\varepsilon$, поэтому в множество нетерминальных символов добавляем новый нетерминал S', а в множество правил P' — два правила: $S' \to S$ и $S' \to \varepsilon$.

Окончательно имеем НКС-грамматику $G' = (\{S', S, A\}, \{b, c\}, \{S' \to S, S' \to \varepsilon, S \to cA, S \to c, A \to cA, A \to c, A \to bA, A \to b\}, S')$, которая эквивалентна исходной грамматике. 🗀

3.7.3. Исключение цепных правил

В КС-грамматиках, описывающих синтаксис языков программирования, часто встречаются правила, правая часть которых состоит из одного нетерминального символа (*цепные* правила). Во многих случаях введение таких правил позволяет упростить формальное описание синтаксических конструкций и их семантическую интерпретацию. Классическим примером КС-грамматики с цепными правилами является грамматика G_0, порождающая скобочные арифметические выражения. В этой грамматике цепные правила позволяют задать старшинство операций (см. рис. 3.11). В некоторых случаях, наличие цепных правил усложняет синтаксический анализ и возникает необходимость их исключения из грамматики.

✍ Правило вывода вида $A \to B$ называется *цепным правилом*.

Алгоритм 3.6. *Исключение цепных правил*

Вход: НКС-грамматика $G = (N, \Sigma, P, S)$.

Выход: Эквивалентная НКС-грамматика $G' = (N, \Sigma, P', S)$ без цепных правил.

***Описание алгоритма*:**

□

1. Для каждого $A \in N$ построить множество $N_A = \{B \mid A \Rightarrow^* B\}$ следующим образом:
   1.1. Положить $N^0_A = \{A\}$ и $i = 1$.
   1.2. Положить $N^i_A = N^{i-1}_A \cup \{C \mid B \to C \in P$ и $B \in N^{i-1}_A\}$.
   1.3. Если $N^i_A \neq N^{i-1}_A$, то положить $N_A = N^i_A$.
2. Положить $P' = \varnothing$.
3. Если $B \to \alpha \in P$ и не является цепным правилом, положить $P' = P' \cup \{A \to \alpha\}$ для всех таких A, что $B \in N_A$.
4. Положить $G' = (N, \Sigma, P', S)$.

■

Поскольку $N_A \subseteq N$ и множество P конечно, алгоритм исключает цепные правила за конечное число шагов.

Докажем корректность этого алгоритма.

Доказательство

□ Непосредственно из множества P' видно, что грамматика G' не содержит цепных правил. Теперь необходимо доказать, что $L(G') = L(G)$.

1. Докажем, что $L(G') \subseteq L(G)$.

   Пусть $w \in L(G')$. Тогда в грамматике G' существует вывод $S \Rightarrow \alpha_0 \Rightarrow \alpha_1 \ldots \Rightarrow \alpha_n = w$. Если при выводе $\alpha_i \Rightarrow \alpha_{i+1}$ применяется правило $A \to \beta$, то существует нетерминальный символ B (возможно $B = A$), такой что $A \Rightarrow^*_G B$ и $B \Rightarrow_G \beta$, т. е. $A \Rightarrow^*_G \beta$ и $\alpha_i \Rightarrow^*_G \alpha_{i+1}$. Отсюда следует, что $S \Rightarrow^*_G w$ и $w \in L(G)$, а значит $L(G') \subseteq L(G)$.

2. Докажем, что $L(G) \subseteq L(G')$.

   Пусть $w \in L(G)$ и левый вывод цепочки w в грамматике G — это $S \Rightarrow_l \alpha_0 \Rightarrow_l \alpha_1 \ldots \Rightarrow_l \alpha_n = w$.

   Можно найти последовательность индексов $i_1, i_2, \ldots, i_k$, состоящую в точности из тех же индексов j, для которых при выводе $\alpha_{j-1} \Rightarrow_l \alpha_j$ применяется не цепное правило. (Например, $i_k = n$, так как вывод терминальной цепочки не может заканчиваться цепным правилом). При левом выводе последовательность цепных правил заменяет символ, занимающий одну и ту же позицию в левовыводимых цепочках, из которых состоит соответствующая часть вывода. Таким образом, $w \in L(G')$, а значит $L(G) = L(G')$. ■

Пример 3.9

🗁 Требуется исключить цепные правила из КС-грамматики G_0 с правилами P:

(1) $E \to E + T$ (4) $T \to P$

(2) $E \to T$ (5) $P \to i$

(3) $T \to T * P$ (6) $P \to (E)$

Шаг 1. После выполнения шага 1 алгоритма 3.6 получим: $N_E = \{E, T, P\}$, $N_T = \{T, P\}$, $N_P = \{P\}$.

Шаг 2. $P' = \varnothing$.

Шаг 3. Выберем первый нетерминал E из множества N_E. Формируя множество правил, левая часть которых — нетерминальный символ E, а правые части — это правые части нецепных правил исходной грамматики, в левой части которых находятся символы из множества N_E, получаем: $\{E \to E + T, E \to T * P, E \to (E), E \to i\}$. Включаем эти правила в P' и получаем $P' = \{E \to E + T, E \to T * P, E \to (E), E \to i\}$. Таким же образом рассматриваем остальные символы из N_E, затем символы множеств N_T и N_P. Окончательно получаем $G' = (N, \Sigma, P', S)$, где $P' = \{E \to E + T, E \to T * P, E \to i, E \to (E), T \to T * P, T \to (E), T \to i, P \to (E), P \to i\}$. 🗀

КС-грамматика $G = (N, \Sigma, P, S)$ называется *грамматикой без циклов*, если в ней нет выводов $A \to^+ A$ для $A \in N$.

КС-грамматика $G = (N, \Sigma, P, S)$ называется *приведенной*, если она без циклов, без ε-правил и без бесполезных символов.

Замечание

Если применить к КС-грамматике алгоритмы устранения недостижимых символов, устранения бесполезных символов, преобразования в грамматику без ε-правил и устранения цепных правил, то получим приведенную КС-грамматику. Можно сформулировать очевидное утверждение: если L — контекстно-свободный язык, то $L = L(G)$ для некоторой приведенной КС-грамматики G.

3.7.4. Удаление произвольного правила вывода

Рассмотрим еще одно полезное преобразование грамматик — удаление произвольного правила вывода. Такое преобразование часто используется при приведении грамматики к заданному классу.

Назовем *A-правилом* КС-грамматики G правило вывода вида $A \to \alpha$, где $A \in N$, $\alpha \in (\Sigma \cup N)^*$.

Алгоритм 3.7. *Удаление произвольного правила из КС-грамматики*

Вход: КС-грамматика $G = (N, \Sigma, P, S)$ и A-правило вывода вида $A \to \alpha B \beta \in P$, где $A, B \in N$ и цепочки $\alpha, \beta \in (\Sigma \cup N)^*$.

Выход: КС-грамматика $G' = (N, \Sigma, P', S)$, такая, что $L(G') = L(G)$ и $A \to \alpha B\beta \notin P'$.

Описание алгоритма:

□

1. Пусть $B \to \gamma_1$, $B \to \gamma_2$, ... , $B \to \gamma_k$ — все B-правила грамматики G. Построить P' следующим образом: $P' = (P \setminus \{A \to \alpha B\beta\}) \cup \{A \to \alpha\gamma_1\beta \mid \alpha\gamma_2\beta \mid ... \mid \alpha\gamma_k\beta\}$
2. Положить $G' = (N, \Sigma, P', S)$.

■

Пример 3.10

🗁 Удалить правило $P \to (E)$ из КС-грамматики G_0 с правилами:

(1) $E \to E + T$ (4) $T \to P$

(2) $E \to T$ (5) $P \to i$

(3) $T \to T * P$ (6) $P \to (E)$

E-правила грамматики G будут выглядеть следующим образом: $E \to E + T$ и $E \to T$. Из множества правил необходимо удалить заданное правило $P \to (E)$ и добавить P-правила $P \to (E + T)$ и $P \to (T)$. Новое множество правил грамматики P' имеет вид: $\{E \to E + T,\ E \to T,\ T \to T * P,\ T \to P,\ P \to i,\ P \to (E + T),\ P \to (T)\}$. 🗀

3.7.5. Устранение левой рекурсии

Рассмотрим преобразования грамматик, полезные при нисходящем синтаксическом анализе. Известно (*см. гл. 6*), что грамматики, используемые для таких методов анализа, не должны быть леворекурсивными.

Нетерминал A КС-грамматики $G = (N, \Sigma, P, S)$ называется *рекурсивным,* если для некоторых α и β существует вывод $A \Rightarrow^{*} \alpha A\beta$. Если $\alpha = \varepsilon$, то A называется *леворекурсивным нетерминальным символом.* Аналогично, если $\beta = \varepsilon$, то A называется *праворекурсивным нетерминальным символом.*

Грамматика, имеющая хотя бы один леворекурсивный нетерминал, называется *леворекурсивной.* Аналогично определяется *праворекурсивная* грамматика.

Грамматика, в которой все нетерминалы, возможно, кроме начального символа, рекурсивные, называется *рекурсивной грамматикой.*

Покажем, что *любой КС-язык определяется хотя бы одной не леворекурсивной грамматикой.*

Утверждение

Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика, в которой $A \to A\alpha_1$, $A \to A\alpha_2$, ... , $A \to A\alpha_m$, $A \to \beta_1$, $A \to \beta_2$, ... , $A \to \beta_n$ — все A-правила грамматики, причем ни одна из цепочек β_i $(1 \le i \le n)$ не начинается с нетерминального символа A.

Если $G' = (N \cup \{A\}, \Sigma, P', S)$, где A' — новый нетерминальный символ, а в P' исходные A-правила заменены правилами $A \to \beta_1$, $A \to \beta_2$, ... , $A \to \beta_n$, $A \to \beta_1 A'$, $A \to \beta_2 A'$, ... , $A \to \beta_n A'$, $A' \to \alpha_1$, $A' \to \alpha_2$, ... , $A' \to \alpha_m$, $A' \to \alpha_1 A'$, $A' \to \alpha_2 A'$, ... , $A' \to \alpha_m A'$, то $L(G') = L(G)$.

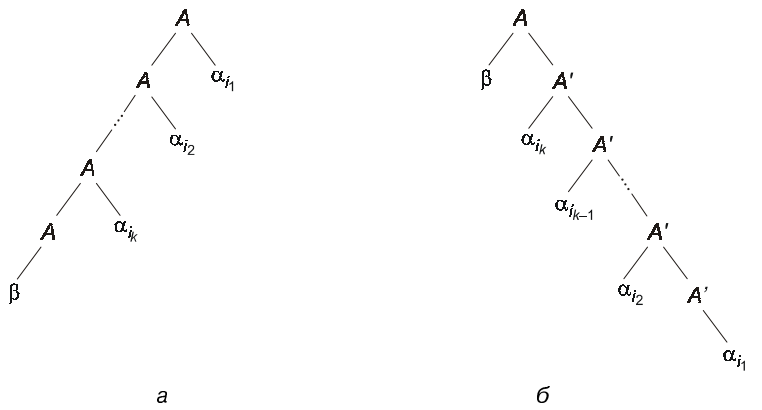
Доказательство

□ Докажем вначале, что $L(G') \subseteq L(G)$. Для этого рассмотрим цепочки, которые можно получить в грамматике G из нетерминала A с помощью применения A-правил только к самому левому нетерминальному символу и которые образуют регулярное множество $(\beta_1 + \beta_2 + ... + \beta_n)(\alpha_1 + \alpha_2 + ... + \alpha_m)^*$. Это в точности те цепочки, которые можно получить в G' из A с помощью правых выводов, применив один раз A-правило и несколько раз A'-правила. Все шаги вывода в грамматике G, в которых не используются A-правила, можно непосредственно сделать в G', т. к. остальные правила в G и G' одни и те же. Отсюда можно заключить, что $L(G') \subseteq L(G)$.

Для доказательства обратного включения $L(G') \supseteq L(G)$ в грамматике G' нужно взять правый вывод и рассматривать последовательность шагов, состоящую из одного применения A-правила и нескольких применений A'-правил.

Таким образом, $L(G) = L(G')$. ■

На рис. 3.15 показано, как действует описанное преобразование на деревья выводов: рис. 3.15, *а* содержит часть дерева в грамматике G, а рис. 3.15, *б* — соответствующую часть дерева в грамматике G'.



а *б*

Рис. 3.15. Деревья выводов в грамматике G (*а*) и G' (*б*)

Пример 3.11

🗁 Требуется исключить левую рекурсию из грамматики G_0 с правилами:

(1) $E \to E + T$ (4) $T \to P$

(2) $E \to T$ (5) $P \to i$

(3) $T \to T * P$ (6) $P \to (E)$

Применив к E-правилам $E \to E + T$ и $E \to T$ описанное преобразование, получим $\alpha = +T$ и $\beta = T$. В соответствии с этим преобразованием, новые E-правила будут иметь вид: $E \to T$, $E \to TE'$, и в грамматику необходимо добавить правило для нового нетерминала E': $E' \to +T$.

После рассмотрения всех правил исходной грамматики G получим правила новой грамматики G':

$E \to T$, $E \to TE'$, $E' \to +T$, $E' \to +TE'$, $T \to P$, $T \to PT'$, $T' \to *P$, $T' \to *PT'$, $P \to (E + T)$, $P \to (E)$, $P \to i$. 🗀

Теперь приведем алгоритм, в основе которого лежит приведенное утверждение и который позволяет устранить левую рекурсию из приведенной КС-грамматики.

Алгоритм 3.8. *Устранение прямой левой рекурсии*

Вход: Приведенная КС-грамматика $G = (N, \Sigma, P, S)$.

Выход: Эквивалентная КС-грамматика $G' = (N', \Sigma, P', S)$ без левой рекурсии.

Описание алгоритма:

□

1. Пусть множество нетерминалов грамматики $N = \{A_1, A_2, \ldots, A_n\}$. Преобразуем грамматику таким образом, чтобы в правиле $A_i \to \alpha$ цепочка α начиналась либо с терминала, либо с такого A_j, что $j > i$. Установим $i = 1$.
2. Пусть множество A_i-правил имеет вид $A_i \to A_i\alpha_1$, $A_i \to A_i\alpha_2$, ... , $A_i \to A_i\alpha_m$, $A_i \to \beta_1$, $A_i \to \beta_2$, ... , $A_i \to \beta_m$, где ни одна из цепочек β_j не начинается с A_k, если $k \le i$. Заменим A_i-правила правилами $A_i \to \beta_1$, $A_i \to \beta_2$, ... , $A_i \to \beta_n$, $A_i \to \beta_1 A'_i$, $A_i \to \beta_2 A'_i$, ... , $A_i \to \beta_n A'_i$ и $A'_i \to \alpha_1$, $A'_i \to \alpha_2$, ... , $A'_i \to \alpha_m$, $A'_i \to \alpha_1 A'_i$, $A'_i \to \alpha_2 A'_i$, ... , $A'_i \to \alpha_m A'_i$, где A'_i — новый нетерминальный символ. После выполнения такой замены правые части всех A_i-правил начинаются с терминального символа или с нетерминала A_k для некоторого $k > i$.
3. Если $i = n$, то преобразование завершено, иначе положить $i = i + 1$ и $j = 1$.

4. Заменить каждое правило $A_i \to A_j\alpha$ правилами $A_i \to \beta_1\alpha$, $A_i \to \beta_2\alpha$, ... , $A_i \to \beta_m\alpha$, где $A_j \to \beta_1$, $A_j \to \beta_2$, ... , $A_j \to \beta_m$ — все A_j-правила грамматики. Правая часть A_j-правила начинается с терминального символа или с нетерминала A_k для $k > j$, поэтому правая часть каждого построенного A_i-правила обладает этим же свойством.
5. Если $j = i - 1$, то перейти к шагу 2 алгоритма, иначе положить $j = j + 1$ и перейти к шагу 4.

■

Основываясь на алгоритме 3.8, можно доказать следующее утверждение:

Каждый КС-язык определяется не леворекурсивной грамматикой.

Схема доказательства приведена в [3].

Пусть G — приведенная грамматика, порождающая язык L. При применении к ней алгоритма 3.8 используются только преобразования, описанные в алгоритме 3.7 и в доказанном ранее утверждении, поэтому результирующая грамматика G' порождает язык L.

Рассмотрим два утверждения:

1. После выполнения шага 2 алгоритма 3.8 правая часть каждого A_i-правила начинается с терминала или с такого A_k, что $k > i$.
2. После выполнения шага 4 алгоритма 3.8 правая часть каждого A_i-правила начинается с терминала или с такого A_k, что $k > j$.

Докажем, что утверждение (1) истинно для всех $i \le n$, а утверждение (2) истинно для всех таких пар (i, j), что $i \le n$ и $i > j$. Значения этих параметров в ходе работы алгоритма 3.8 на шаге 2 и шаге 4 будут соответственно следующими:

$$1, (2, 1), 2, \ldots , i - 1, (i, 1), \ldots , (i, j), \ldots , (i, i - 1), i, \ldots , (n, n - 1), n.$$

Проведем доказательство с помощью индукции по значениям параметров в приведенном порядке.

Доказательство

□ Так как на шаге 2 алгоритма цепочки β_1, ... , β_p не могут начинаться с A_1, то для $i = 1$ утверждение (1) истинно.

Предположим, что утверждения (1) и (2) истинны для значений параметров, предшествующих (i, j). Поскольку $j < i$, то по предположению индукции утверждение (1) истинно для j. Поэтому на шаге 4 алгоритма каждая из цепочек β_1, ... , β_m начинается с терминала или с такого нетерминала A_k, что $k > j$. Так как после завершения шага 4 алгоритма цепочки β_1, ... , β_m становятся префиксами правых частей A_i-правил, то (1) истинно для (i, j).

Аналогично доказывается, что если утверждения (1) и (2) истинны для значений параметров, предшествующих i, то (1) истинно для i.

Из утверждения (1) следует, что нетерминалы A_1, ... , A_n не являются леворекурсивными, т. к. если $A \Rightarrow^+_l A_k\alpha$ для некоторого α, то $k > i$. Теперь нужно показать, что нетерминалы A'_i, появляющиеся на шаге 2 алгоритма, не могут быть леворекурсивными. Это непосредственно следует из того, что грамматика G приведенная. На шаге 2 все цепочки α_1, ... , α_n не пустые, и потому A'_i не может быть первым символом правой части правила. ■

Пример 3.12

🗁 Устраним левую рекурсию в грамматике G, определяемой множеством правил $P = \{S \to AB,\ S \to a,\ A \to BS,\ A \to Sb,\ B \to SA,\ B \to BB,\ B \to a\}$.

Положим $A_1 = S$, $A_2 = A$ и $A_3 = B$ и выполним алгоритм 3.8 по шагам.

Шаг 1. $i = 1$.

Шаг 2. Правила вывода вида $A_1 \to A_1\alpha$ (в обозначениях исходной грамматики $S \to S\alpha$) в грамматике нет. Шаг не выполняется.

Шаг 3. $i = 2, j = 1$.

Шаг 4. Найдем все правила вида $A_2 \to A_1\alpha$ (для нашего примера это $A \to S\alpha$). Такое правило в грамматике только одно — это $A \to Sb$. Для выполнения замены этого правила нужно найти такие правила, в правой части которых стоит символ A_1 (т. е. S).

После замены правила $A \to Sb$ получим новое множество правил $P' = \{S \to AB,\ S \to a,\ A \to BS,\ A \to Abb,\ A \to ab,\ B \to SA,\ B \to BB,\ B \to a\}$.

Шаг 5. $j = i - 1$. Переходим к шагу 2.

Шаг 2. $i = 2$. Правила $A \to BS$, $A \to Abb$, $A \to ab$ заменяем правилами $A \to BS$, $A \to ab$, $A \to BSA'$, $A \to abA'$ и $A' \to Bb$, $A' \to BbA'$. Получаем следующее множество правил: $P' = \{S \to AB,\ S \to a,\ A \to BS,\ A \to ab,\ A \to BSA',\ A \to abA',\ A' \to Bb,\ A' \to BbA',\ B \to SA,\ B \to BB,\ B \to a\}$.

Шаг 3. $i = 3, j = 1$.

Шаг 4. Используя правила $S \to AB$, $S \to a$, заменяем правило $B \to SA$ правилами $B \to ABA$, $B \to aA$. Получаем новое множество правил грамматики: $P' = \{S \to AB,\ S \to a,\ A \to BS,\ A \to ab,\ A \to BSA',\ A \to abA',\ A' \to Bb,\ A' \to BbA',\ B \to ABA,\ B \to aA\}$.

Шаг 5. $j \neq i - 1, j = 2$. Переходим к шагу 4.

Шаг 4. ($i = 3$, $j = 2$). Используя правила $A \to BS$, $A \to ab$, $A \to BSA'$, $A \to abA'$, заменяем правило $B \to ABA$ правилами $B \to aA$, $B \to BSBA$, $B \to abBA$, $B \to BSA'BA$, $B \to abA'BA$ и получаем новое множество правил грамматики $P' = \{S \to AB,\ S \to a,\ A \to BS,\ A \to ab,\ A \to BSA',\ A \to abA',\ A' \to Bb,\ A' \to BbA',\ B \to aA,\ B \to BSBA,\ B \to abBA,\ B \to BSA'BA,\ B \to abA'BA\}$.

Шаг 5. $j = i - 1$. Переходим к шагу 2.

Шаг 2. $i = 3$. Правила $B \to BSBA$, $B \to BSA'BA$, $B \to aA$, $B \to abBA$, $B \to abA'BA$ заменяем правилами $B \to ab$, $B \to abBA$, $B \to abA'BA$, $B \to abB'$, $B \to abBAB'$, $B \to abA'BAB'$, $B' \to SBA$, $B' \to SA'BA$, $B' \to SBAB'$, $B' \to SA'BAB'$.

Шаг 3. $i = n$, выход из алгоритма с окончательным множеством правил грамматики $P' = \{S \to AB,\ S \to a,\ A \to BS,\ A \to ab,\ A \to BSA',\ A \to abA',\ A' \to Bb,\ A' \to BbA',\ B \to ab,\ B \to abBA,\ B \to abA'BA,\ B \to abB',\ B \to abBAB',\ B \to abA'BAB',\ B' \to SBA,\ B' \to SA'BA,\ B' \to SBAB',\ B' \to SA'BAB'\}$. □

3.7.6. Левая факторизация

Рассмотрим преобразование, которое называется *левой факторизацией* и которое позволяет исключить из КС-грамматики правила, имеющие в своих правых частях одинаковые префиксы.

Алгоритм 3.9. *Левая факторизация грамматики*

Вход: Приведенная КС-грамматика $G = (N, \Sigma, P, S)$.

Выход: Эквивалентная левофакторизованная КС-грамматика $G' = (N', \Sigma, P', S)$.

Описание алгоритма:

□

1. Пусть множество A-правил имеет вид: $A \to \alpha\beta_1$, $A \to \alpha\beta_2$, ... , $A \to \alpha\beta_m$, $A \to \gamma_1$, $A \to \gamma_2$, ... , $A \to \gamma_p$, где ни одна из цепочек γ_j не начинается с α. Заменим A-правила правилами $A \to \alpha A'$, $A \to \gamma_1$, $A \to \gamma_2$, ... , $A \to \gamma_p$ и $A' \to \beta_1$, $A' \to \beta_2$, ... , $A' \to \beta_m$, где A' — новый нетерминальный символ.
2. Выполнять преобразование до тех пор, пока в грамматике остаются правила с одинаковыми префиксами.

■

Пример 3.13

Выполнить левую факторизацию КС-грамматики $G = (\{S, A\}, \{a, b\}, \{S \to abSa, S \to aaAb, S \to b, A \to baAb, A \to b\}, S)$.

S-правило содержит три альтернативы. Две первые из них имеют одинаковый префикс a, поэтому S-правило нужно заменить на следующие правила: $S \to aS'$, $S \to b$ и $S' \to bSa$, $S \to aAb$.

Обе альтернативы A-правила имеют общий префикс b и должны быть заменены правилами $A \to bA'$, $A' \to aAb$, $A' \to \varepsilon$. Грамматика после левой факторизации имеет следующий вид: $G' = (\{S, S', A, A'\}, \{a, b\}, \{S \to aS', S \to b, S' \to bSa, S \to aAb, A \to bA', A' \to aAb, A' \to \varepsilon\}, S)$. □

3.8. Нормальная форма Хомского

Существуют формы КС-грамматик (нормальные формы), которые позволяют упростить рассмотрение их свойств.

КС-грамматика $G = (N, \Sigma, P, S)$ называется грамматикой в *нормальной форме Хомского*, если каждое правило из P имеет один из следующих видов:

1. $A \to BC$, где A, B и $C \in N$;
2. $A \to a$, где $a \in \Sigma$;
3. $S \to \varepsilon$, если $\varepsilon \in L(G)$, причем S не встречается в правых частях правил.

Замечание

Иногда нормальную форму Хомского называют *бинарной нормальной формой* КС-грамматики. Это объясняется тем, что деревья выводов в такой грамматике являются бинарными.

Рассмотрим алгоритм, который позволяет преобразовать произвольную КС-грамматику в нормальную форму Хомского.

Алгоритм 3.10. *Преобразование КС-грамматики к нормальной форме Хомского*

Вход: Приведенная КС-грамматика $G = (N, \Sigma, P, S)$, не содержащая цепных правил.

Выход: КС-грамматика в нормальной форме Хомского.

Описание алгоритма:

□

1. Если начальный символ грамматики S входит в правые части правил, то нужно пополнить грамматику новым начальным символом S' и включить в нее правило $S' \to S$.
2. Если правило $S \to \varepsilon$ принадлежит P, то включить его в P'.
3. Включить в P' все правила из P, которые имеют вид $A \to BC$ и $A \to a$.
4. Для каждого правила из P вида $A \to X_1 \ldots X_k$, где $k > 2$, в множество нетерминальных символов включить $(k - 1)$ нетерминальных символов вида: $\langle X_2 \ldots X_k\rangle$, $\langle X_3 \ldots X_k\rangle$, ... , $\langle X_{k-2} X_{k-1} X_k\rangle$, $\langle X_{k-1} X_k\rangle$, а в множество правил P' — правила:

$$
\begin{aligned}
A &\to X'_1\langle X_2 \ldots X_k\rangle \\
\langle X_2 \ldots X_k\rangle &\to X'_2\langle X_3 \ldots X_k\rangle \\
&\ldots \\
\langle X_{k-2}X_{k-1}X_k\rangle &\to X'_{k-2}\langle X_{k-1}X_k\rangle \\
\langle X_{k-1}X_k\rangle &\to X'_{k-1}X'_k.
\end{aligned}
$$

Символы X'_i определяются следующим образом:

- если $X_i \in N$, то $X'_i = X_i$;
- если $X_i \in \Sigma$ ($X_i = a$), то X'_i — новый нетерминальный символ $<a>$.

5. Для каждого правила из P вида $A \to X_1X_2$, где хотя бы один из символов X_1 и X_2 принадлежит Σ, включить в P' правило $A \to X'_1 X'_2$.
6. Для каждого нетерминального символа вида $<a>$, введенного на шагах 4 и 5, включить в P' правило $<a> \to a$.
7. Результирующей грамматикой будет грамматика $G = (N', \Sigma, P', S')$, в которой $N' = N \cup \{<a>\}$.

■

Для того чтобы показать, что $L(G) = L(G')$, необходимо применить алгоритм удаления произвольного правила грамматики к каждому правилу грамматики G', в правую часть которого входит $<a>$, а затем использовать этот алгоритм для правил с нетерминальными символами вида $<X_i \ldots X_j>$.

Пример 3.14

🗁 Преобразовать приведенную КС-грамматику $G = (\{S, A, B\}, \{a, b\}, \{S \to A, S \to ABA, A \to aA, A \to a, A \to B, B \to bB, B \to b\}, S)$ к нормальной форме Хомского.

Перед преобразованием грамматики G необходимо удалить из нее цепные правила. Используя алгоритм 3.6, вычисляем множества $N_S = \{S, A, B\}$, $N_A = \{A, B\}$, $N_B = \{B\}$, а затем получаем множества правил грамматики $P' = \{S \to ABA, S \to aA, S \to a, A \to aA, A \to a, B \to bB, B \to b\}$.

Теперь переходим к выполнению алгоритма 3.10.

Шаг 1. Основной символ грамматики S не входит в правые части правил. Пополнение грамматики не производится.

Шаг 2. $P'' = \varnothing$. Грамматика не содержит правил вида $S \to \varepsilon$. P'' не изменяется.

Шаг 3. Включаем в P'' все правила из P вида $A \to BC$ и $A \to a$ и получаем: $P'' = \{S \to a, A \to a, B \to b\}$.

Шаг 4. В множестве правил P' имеется только одно правило, правая часть которого содержит более двух символов — это $S \to ABA$. Для этого правила в множество P'' необходимо включить правила $S \to A<BA>$ и $<BA> \to BA$, где $<BA>$ — новый нетерминальный символ. Теперь $P'' = \{S \to a, A \to a, B \to b, S \to A<BA>, <BA> \to BA\}$.

Шаг 5. Для каждого правила из P' $S \to aA$, $A \to aA$ и $B \to bB$ необходимо включить в P'' соответственно правила $S \to <a>A$, $A \to <a>A$ и $B \to <b>B$.

Шаг 6. Дополним P'' правилами $<a> \to a$ и $<b> \to b$.

В результате получим грамматику $G' = (N', \Sigma, P'', S)$, в которой $N' = N \cup \{BA>, <a>, <b>\}$, $P'' = \{S \to A<BA>, S \to <a>A, S \to a, A \to <a>A, A \to a, B \to <b>B, B \to b, <BA> \to BA, <a> \to a, <b> \to b\}$. □

3.9. Нормальная форма Грейбах

Покажем, что для каждого КС-языка можно найти грамматику, в которой все правые части правил начинаются с терминальных символов. Построение такой грамматики основано на устранении левой рекурсии.

КС-грамматика $G = (N, \Sigma, P, S)$ называется *грамматикой в нормальной форме Грейбах*, если в ней нет ε-правил и каждое правило из P, отличное от $S \to \varepsilon$, имеет вид $A \to a\alpha$, где $a \in \Sigma$ и $\alpha \in N^*$.

Если грамматика не леворекурсивная, то на множестве ее нетерминалов можно определить естественный частичный порядок $\boldsymbol{R}$ — такое отношение на N, что $A\boldsymbol{R}B$ тогда и только тогда, когда $A \Rightarrow^+ B\alpha$ для некоторого α. Это отношение можно расширить до линейного порядка "меньше" (<), обладающего таким свойством, что если $A \to B\alpha \in P$, то $A < B$ [3].

Приведем алгоритм [3], позволяющий вложить частичный порядок на множестве нетерминальных символов в линейный порядок.

Алгоритм 3.11. *Топологическая сортировка*

Вход: Частичный порядок $\boldsymbol{R}$ на конечном множестве A.

Выход: Линейный порядок $\boldsymbol{R}'$ на множестве A, для которого $\boldsymbol{R} \subseteq \boldsymbol{R}'$.

Описание алгоритма:

□ Пусть заданное множество $A = \{a_1, a_2, \ldots, a_n\}$ — это конечное множество, состоящее из n элементов. Тогда линейный порядок можно представить в виде списка элементов множества A: $a_1, a_2, \ldots, a_n$, такого, что $a_i \boldsymbol{R}' a_j$, где $i < j$. Для построения списка нужно выполнить следующие действия:

1. Установить $i = 1$, $A_i = A$, $\boldsymbol{R}_i = \boldsymbol{R}$.
2. Если множество $A_i \neq \varnothing$, то выбрать из этого множества элемент a_i, обладающий следующим свойством: $a\boldsymbol{R}_i a_i$ для всех $a \in A_i$, и выполнить шаг 3. Если $A_i = \varnothing$, то перейти к шагу 4 алгоритма.
3. Положить $A_{i+1} = A_i - \{a_i\}$ и $\boldsymbol{R}_{i+1} = \boldsymbol{R}_i \cap (A_{i+1} \times A_{i+1})$, установить $i = i + 1$ и перейти к шагу 2 алгоритма.
4. Искомый линейный порядок определяется списком $a_1, a_2, \ldots, a_{i-1}$.

■

Пример 3.15

🗁 Используем алгоритм 3.11 для определения линейного порядка на множестве нетерминалов для нелеворекурсивной грамматики с правилами $P' = \{E \to T,\ E \to TE',\ E' \to +T,\ E' \to +TE',\ T \to P,\ T \to PT',\ T' \to *P,\ T' \to *PT',\ P \to (E),\ P \to i\}$.

На множестве нетерминальных символов этой грамматики $N = \{E, E', T, T', P\}$ существует частичный порядок $\boldsymbol{R} = \{(E, T), (E, P), (T, P)\}$. Применим алгоритм 3.11 к $\boldsymbol{R}$.

Шаг 1. $i = 1$, $A_1 = N = \{E, E', T, T', P\}$, $\boldsymbol{R}_1 = \boldsymbol{R} = \{(E, T), (E, P), (T, P)\}$.

Шаг 2. $A_1 \neq \varnothing$. Символ из A_1, обладающий заданным свойством, — это символ $a_1 = E'$. Очевидно, что $(E, E') \notin \boldsymbol{R}_1$, $(E', E') \notin \boldsymbol{R}_1$, $(T, E') \notin \boldsymbol{R}_1$, $(T', E') \notin \boldsymbol{R}_1$, $(P, E') \notin \boldsymbol{R}_1$.

Шаг 3. $A_2 = \{E, T, T', P\}$, $\boldsymbol{R}_2 = \boldsymbol{R}_1$, $i = 2$. Переходим к шагу 2.

Шаг 2. $A_2 \neq \varnothing$. Символ из A_2, обладающий заданным свойством, — это символ $a_2 = E$.

Шаг 3. $A_3 = \{T, T', P\}$, $\boldsymbol{R}_3 = \{(T, P)\}$, $i = 3$. Переходим к шагу 2.

Шаг 2. $A_3 \neq \varnothing$. Символ из A_3, обладающий заданным свойством, — это символ $a_3 = T'$.

Шаг 3. $A_4 = \{T, P\}$, $\boldsymbol{R}_4 = \{(T, P)\}$, $i = 4$. Переходим к шагу 2.

Шаг 2. $A_4 \neq \varnothing$. Символ из A_4, обладающий заданным свойством, — это символ $a_4 = T$.

Шаг 3. $A_5 = \{P\}$, $\boldsymbol{R}_5 = \varnothing$, $i = 5$. Переходим к шагу 2.

Шаг 2. $A_5 = \varnothing$. Переходим к шагу 4.

Шаг 4. Линейный порядок: E', E, T', T.

Окончательно для исходной грамматики имеем $E' < E < T' < T < P$. 🗀

Рассмотрим алгоритм преобразования КС-грамматики в нормальную форму Грейбах.

Алгоритм 3.12. *Преобразование к нормальной форме Грейбах*

Вход: Нелеворекурсивная приведенная КС-грамматика $G = (N, \Sigma, P, S)$.

Выход: Эквивалентная КС-грамматика G' в нормальной форме Грейбах.

Описание алгоритма:

□

1. Построить такой линейный порядок ($<$) на N, что каждое A-правило начинается либо с терминала, либо с такого нетерминала B, что $A < B$. Упорядочить $N = \{A_1, A_2, \ldots, A_n\}$ так, что $A_1 < A_2 < \ldots < A_n$.

2. Положить $i = n - 1$.
3. Если $i = 0$, то перейти к шагу 5 алгоритма. В противном случае заменить каждое правило вида $A_i \to A_j\alpha$, где $j > i$, правилами $A_i \to \beta_1\alpha$, $A_i \to \beta_2\alpha$, ... , $A_i \to \beta_m\alpha$, где $A_j \to \beta_1$, $A_j \to \beta_2$, ... , $A_j \to \beta_m$ — все A_j-правила.
4. Положить $i = i - 1$ и перейти к шагу 3.
5. Теперь правая часть каждого правила начинается терминальным символом. В каждом правиле $A \to aX_1 \ldots X_k$ заменить $X_j \in \Sigma$ новым нетерминалом X'_j.
6. Для новых нетерминалов X'_j, введенных на 5-ом шаге, добавить правила $X'_j \to X_j$.

■

Пример 3.16

🗁 Преобразовать в нормальную форму Грейбах грамматику G с правилами $\{E \to T,\ E \to TE',\ E' \to +T,\ E' \to +TE',\ T \to P,\ T \to PT',\ T' \to *P,\ T' \to *PT',\ P \to (E),\ P \to i\}$.

Шаг 1. Упорядочим нетерминалы следующим образом: $E' < E < T' < T < P$ (*см. пример 1.15*).

Шаг 2. $i = n - 1 = 4$.

Шаг 3. $i \neq 0$. В грамматике есть два правила с нетерминалом T в левых частях: $T \to P$ и $T \to PT'$. Их правые части начинаются нетерминалами, поэтому правила должны быть заменены с использованием P-правил $P \to (E)$ и $P \to i$. После замены правила исходной грамматики будут иметь вид: $\{E \to T,\ E \to TE',\ E' \to +T,\ E' \to +TE',\ T \to (E),\ T \to i,\ T \to (E)T',\ T \to iT',\ T' \to *P,\ T' \to *PT',\ P \to (E),\ P \to i\}$.

Шаг 4. $i = i - 1 = 3$.

Шаг 3. $i \neq 0$. В грамматике есть два правила с нетерминалом T' в левых частях: $T' \to *P$ и $T' \to *PT'$. Их правые части начинаются с терминалов и не должны заменяться.

Шаг 4. $i = i - 1 = 2$.

Шаг 3. $i \neq 0$. В грамматике есть два правила с нетерминалом E в левых частях: $E \to T$ и $E \to TE'$. Их правые части начинаются нетерминалами, поэтому эти правила должны быть заменены с использованием T-правил $T \to (E)$, $T \to i$, $T \to (E)T'$, $T \to iT'$. После замены правила исходной грамматики примут следующий вид: $\{E \to (E),\ E \to i,\ E \to (E)T',\ E \to iT',\ E \to (E)E',\ E \to iE',\ E \to (E)T'E',\ E \to iT'E',\ E' \to +T,\ E' \to +TE',\ T \to (E),\ T \to i,\ T \to (E)T',\ T \to iT',\ T' \to *P,\ T' \to *PT',\ P \to (E),\ P \to i\}$.

Шаг 4. $i = i - 1 = 1$.

Шаг 3. $i \neq 0$. В грамматике есть два правила с нетерминалом E' в левых частях. Правые части этих правил начинаются терминалами и не должны заменяться.

Шаг 4. $i = i - 1 = 0$.

Шаг 3. $i = 0$. Переход к шагу 5 алгоритма.

Шаг 5. В грамматике есть только один терминал, стоящий на третьей (или более) позиции правой части правила. Обозначим его символом A. Тогда правила грамматики принимают вид: $\{E \to (EA,\ E \to i,\ E \to (EAT',\ E \to iT',\ E \to (EAE',\ E \to iE',\ E \to (EAT'E',\ E \to iT'E',\ E' \to +T,\ E' \to +TE',\ T \to (EA,\ T \to i,\ T \to (EAT',\ T \to iT',\ T' \to *P,\ T' \to *PT',\ P \to (EA,\ P \to i\}$.

Шаг 6. Добавим в грамматику правило для нового нетерминала A. Окончательно правила грамматики принимают вид: $\{E \to (EA,\ E \to i,\ E \to (EAT',\ E \to iT',\ E \to (EAE',\ E \to iE',\ E \to (EAT'E',\ E \to iT'E',\ E' \to +T,\ E' \to +TE',\ T \to (EA,\ T \to i,\ T \to (EAT',\ T \to iT',\ T' \to *P,\ T' \to *PT',\ P \to (EA,\ P \to i,\ A \to)\}$. □

Можно доказать, что если L — КС-язык, то $L = L(G)$ для некоторой грамматики G в нормальной форме Грейбах.

Схема доказательства этого утверждения приведена в [3]. Для доказательства индукцией по $n - i$ (т. е. по i, но в обратном порядке, начиная с $i = n - 1$ и заканчивая $i = 1$) можно показать, что после выполнения шага 3 алгоритма 3.12 правая часть каждого A_i-правила начинается терминалом. Ключевой момент в этом доказательстве — это использование линейного порядка ($<$). После шага 5 грамматика преобразуется к нормальной форме Грейбах и порождаемый ею язык не изменится (см. алгоритм 3.7).

Недостатком рассмотренного алгоритма преобразования грамматики к нормальной форме Грейбах является то, что в результате получается *много новых правил*. Можно применить метод, в котором грамматика будет иметь меньше правил, но, возможно, *больше нетерминалов*.

Опишем метод преобразования грамматик в терминах систем определяющих уравнений.

Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика, в которой нет ε-правил и нет цепных правил.

Пусть Δ и Σ — два непересекающихся алфавита. *Системой определяющих уравнений* в алфавитах Δ и Σ называется система уравнений вида:

$$A = \alpha_1 + \alpha_2 + \ldots + \alpha_k,$$

где $A \in \Delta$ и $\alpha_i \in (\Delta \cup \Sigma)^*$. Если $k = 0$, то уравнение имеет вид $A = \varnothing$. Для каждого $A \in \Delta$ в системе есть одно уравнение.

Решением системы определяющих уравнений называется такое отображение f множества Δ в $P(\Sigma^*)$, что, если подставить $f(A)$ в каждое уравнение вместо каждого $A \in \Delta$, уравнения станут равенствами. Решение f называется *наименьшей неподвижной точкой*, если $f(A) \subseteq g(A)$ для любого решения g и любого $A \in \Delta$.

Чтобы получить КС-грамматику, соответствующую системе определяющих уравнений, надо для каждого уравнения $A = \alpha_1 + \alpha_2 + ... + \alpha_k$ построить правила $A \to \alpha_1$, $A \to \alpha_2$, ... , $A \to \alpha_k$. Нетерминалами будут символы алфавита Δ. Очевидно, что это соответствие взаимно однозначно.

Пример 3.17

Множество правил $\{A \to AaB,\ A \to BB,\ A \to b,\ B \to aA,\ B \to Baa,\ B \to Bd,\ B \to c\}$ можно представить в виде системы уравнений:

$$A = AaB + BB + b$$

$$B = aA + BAa + Bd + c,$$

где A и B — неизвестные, представляющие множества.

Приведем несколько полезных утверждений о системах определяющих уравнений.

Утверждение 1. *Наименьшая неподвижная точка* системы определяющих уравнений в алфавитах Δ и Σ единственна и имеет вид $f(A) = \{w \mid A \Rightarrow^*_G w$ и $w \in \Sigma^*\}$, где G — соответствующая КС-грамматика.

При определении метода преобразования грамматик в нормальную форму Грейбах удобно пользоваться матричным представлением систем определяющих уравнений.

Пусть $\Delta = \{A_1, A_2, ... , A_n\}$. Матричное уравнение $\mathbf{A} = \mathbf{AR} + \mathbf{B}$ представляет собой n уравнений. Здесь $\mathbf{A}$ — вектор-строка $[A_1, A_2, ... , A_n]$, $\mathbf{R}$ — матрица порядка n, элементами которой служат регулярные выражения, $\mathbf{B}$ — вектор-строка, состоящая из n регулярных выражений.

Замечание

В качестве скалярного умножения будем использовать конкатенацию, а в качестве сложения — операцию объединения. Сложение и умножение векторов и матриц определяются обычным образом.

Элементом матрицы $\mathbf{R}$, расположенным в i-ой строке и j-ом столбце, будет регулярное выражение $\alpha_1 + ... + \alpha_k$, если $A_i\alpha_1$, ... , $A_i\alpha_k$ — все члены уравнения

для A_j, первым символом которых является A_i. В качестве j-ого компонента вектора **B** возьмем сумму тех членов уравнения для A_j, которые начинаются с символа из множества Σ. Таким образом, B_j и R_{ij} — такие выражения, что уравнение для A_j (в КС-грамматике ему соответствует множество A_j-правил) можно записать в виде $A_j = A_1R_{1j} + A_2R_{2j} + ... + A_nR_{nj} + B_j$, где B_j — сумма выражений, начинающихся терминалами.

Пример 3.18

🗁 Матричное представление системы определяющих уравнений из примера 3.17 имеет вид:

$$\begin{bmatrix} A & B \end{bmatrix} = \begin{bmatrix} A & B \end{bmatrix} \begin{bmatrix} aB & \varnothing \\ B & aA+d \end{bmatrix} + \begin{bmatrix} b & aA+c \end{bmatrix}.$$

🗀

Для матричного уравнения **A** = **AR** + **B** найдем такую эквивалентную систему определяющих уравнений, что все правые части соответствующих ей правил начинаются терминальными символами. Этот переход основан на следующем утверждении.

Пусть **A** = **AR** + **B** — система определяющих уравнений. Тогда ее наименьшей неподвижной точкой будет **A** = **BR***, где **R*** = **I** + **R** + **R**^{2} + **R**^{3} + ... , **I** — единичная матрица (на ее главной диагонали стоят значения ε), **R**^{2} = **RR**, **R**^{3} = **RRR** и т. д.

Если положить **R**$^+$ = **RR***, то наименьшую неподвижную точку матричного уравнения **A** = **AR** + **B** можно записать в виде **A** = **B**(**R**$^+$ + **I**) = **BR**$^+$ + **BI** = **BR**$^+$ + **B**. Эта система не является системой определяющих уравнений, т. к. элементы матрицы **R**$^+$ могут быть бесконечными суммами членов исходных уравнений, поэтому для нее нельзя найти соответствующую грамматику.

Заменим **R**$^+$ новой матрицей неизвестных **Q**, каждым элементом q_{ij} которой является новый символ. Так как **R**$^+$ = **RR**$^+$ + **R**, то можно получить систему определяющих уравнений **Q** = **RQ** + **R** с неизвестными q_{ij}. Если **Q** и **R** — матрицы порядка n, то система состоит из n^2 уравнений.

Утверждение 2. Пусть **A** = **AR** + **B** — система определяющих уравнений в алфавитах Δ и Σ. Пусть **Q** — матрица того же порядка, что и **R**, причем все ее элементы — различные новые символы. Тогда система определяющих уравнений, состоящая из **A** = **BQ** + **B** и **Q** = **RQ** + **R**, имеет наименьшую неподвижную точку, которая совпадает на Δ с наименьшей неподвижной точкой системы **A** = **AR** + **B**.

Теперь можно определить другой алгоритм преобразования приведенной грамматики к нормальной форме Грейбах.

Алгоритм 3.13. *Преобразование к нормальной форме Грейбах*

Вход: Приведенная КС-грамматика $G = (N, \Sigma, P, S)$ без правил вида $S \to \varepsilon$.

Выход: Эквивалентная КС-грамматика $G' = (N', \Sigma, P', S)$ в нормальной форме Грейбах.

Описание алгоритма:

□

1. Построить по грамматике G систему определяющих уравнений $\mathbf{A} = \mathbf{AR} + \mathbf{B}$ в алфавитах N и Σ.
2. Пусть **Q** — матрица порядка n, состоящая из новых символов, и мощность множества N равна n. Построить новую систему определяющих уравнений

   $$\mathbf{A} = \mathbf{BQ} + \mathbf{B},\ \mathbf{Q} = \mathbf{RQ} + \mathbf{R} \ldots$$

   и соответствующую грамматику G_1. В векторе **B** каждый компонент, отличный от ∅, начинается с терминального символа, поэтому для $A \in N$ все A-правила грамматики G_1 будут начинаться терминалами.
3. Грамматика G приведенная, поэтому символ ε не входит в матрицу **R,** т. е. для элементов q_{ij} матрицы **Q** все соответствующие q-правила грамматики G_1 начинаются символами из $N \cup \Sigma$. Заменить во всех правых частях q-правил, начинающихся с нетерминала A, этот нетерминал правыми частями всех A-правил. В результате получится грамматика, у которой все правые части правил начинаются с терминальных символов.
4. Если в правой части некоторого правила терминал a встречается не на первом месте, то заменить его новым терминалом a' и добавить правило $a' \to a$. Полученную грамматику обозначить как G'.

■

Грамматика G является приведенной и не содержит правила $S \to \varepsilon$. Поскольку ε не является компонентом вектора **B** и матрицы **R**, то G' является грамматикой в нормальной форме Грейбах. Из алгоритма 3.7 и приведенных ранее утверждений 1 и 2 следует, что $L(G') = L(G)$, а значит, алгоритм 3.13 строит грамматику G', эквивалентную исходной грамматике G.

Пример 3.19

🗁 Преобразуем грамматику G с правилами $\{S \to AB,\ A \to aA,\ A \to a,\ B \to bB,\ B \to b\}$ в нормальную форму Грейбах.

Система определяющих уравнений этой грамматики выглядит следующим образом:

$$S = AB$$

$$A = aA + a$$

$$B = bB + b$$

Шаг 1. Система определяющих уравнений $\mathbf{A} = \mathbf{AR} + \mathbf{B}$ в алфавитах N и Σ имеет вид:

$$\begin{bmatrix} S & A & B \end{bmatrix} = \begin{bmatrix} S & A & B \end{bmatrix} \begin{bmatrix} \varnothing & \varnothing & \varnothing \\ B & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing \end{bmatrix} \begin{bmatrix} \varnothing & aA + a & bB + b \end{bmatrix}. \quad (3.1)$$

Замечание

По определению $R_{ij} = \alpha_1 + \ldots + \alpha_k$, если $A_i\alpha_1, \ldots, A_i\alpha_k$ — члены уравнения для A_j, первым символом которых является A_i.

В нашем примере:

- $R_{11} = \varnothing$, т. к. уравнение, левой частью которого является $A_1 = S$, не содержит членов, начинающихся с S;
- в уравнении для $A_1 = S$ есть член, начинающийся с $A_2 = A$. В элемент матрицы помещается суффикс найденного члена, который в данном случае равен B, т. е. $R_{21} = B$.

Замечание

Для проверки правильности построения матрицы **R** рекомендуется произвести вычисления в правой части уравнения и построить соответствующие правила грамматики, которые должны совпадать с исходными правилами.

Для рассматриваемого примера:

$$\begin{bmatrix} S & A & B \end{bmatrix} = \begin{bmatrix} S & A & B \end{bmatrix} \begin{bmatrix} \varnothing & \varnothing & \varnothing \\ B & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing \end{bmatrix} \begin{bmatrix} \varnothing & aA + a & bB + b \end{bmatrix},$$

$$\begin{bmatrix} S & A & B \end{bmatrix} = \begin{bmatrix} AB & \varnothing & \varnothing \end{bmatrix} + \begin{bmatrix} \varnothing & aA + a & bB + b \end{bmatrix},$$

$$\begin{bmatrix} S & A & B \end{bmatrix} = \begin{bmatrix} AB & aA + a & bB + b \end{bmatrix}.$$

Последнее уравнение соответствует правилам КС-грамматики $S \to AB$, $A \to aA$, $A \to a$, $B \to bB$, $B \to b$, которые совпадают с исходными правилами.

Шаг 2. Перепишем построенную систему в соответствии с алгоритмом 3.13 в виде **A** = **BQ** + **B**:

$$\begin{bmatrix} S & A & B \end{bmatrix} = \begin{bmatrix} \varnothing & aA + a & bB + b \end{bmatrix} \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} + \begin{bmatrix} \varnothing & aA + a & bB + b \end{bmatrix} \quad (3.2)$$

и добавим систему вида **Q** = **RQ** + **R**:

$$\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} = \begin{bmatrix} \varnothing & \varnothing & \varnothing \\ B & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing \end{bmatrix} \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} + \begin{bmatrix} \varnothing & \varnothing & \varnothing \\ B & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing \end{bmatrix}. \quad (3.3)$$

После преобразований уравнения (3.2) получим

$$\begin{bmatrix} S & A & B \end{bmatrix} = \begin{bmatrix} Y_1 & Y_2 & Y_3 \end{bmatrix},$$

где

$Y_1 = aAX_{21} + aX_{21} + bBX_{31} + bX_{31}$

$Y_2 = aAX_{22} + aX_{22} + bBX_{32} + bX_{32} + aA + a$

$Y_3 = aAX_{23} + aX_{23} + bBX_{33} + bX_{33} + bB + b$

Это уравнение соответствует следующим правилам КС-грамматики:

$S \to aAX_{21}$, $S \to aX_{21}$, $S \to bBX_{31}$, $S \to bX_{31}$,

$A \to aAX_{22}$, $A \to aX_{22}$, $A \to bBX_{32}$, $A \to bX_{32}$, $A \to aA$, $A \to a$,

$B \to aAX_{21}$, $B \to aX_{21}$, $B \to bBX_{31}$, $B \to bX_{31}$, $B \to bB$, $B \to b$.

После преобразований уравнения (1.3) получим:

$$\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} = \begin{bmatrix} \varnothing & \varnothing & \varnothing \\ BX_{11} + B & BX_{12} & BX_{13} \\ \varnothing & \varnothing & \varnothing \end{bmatrix},$$

что соответствует правилам КС-грамматики:

$X_{21} \to BX_{11}$, $X_{21} \to B$,

$X_{22} \to BX_{12}$,

$X_{23} \to BX_{13}$.

Объединив эти две группы правил, получим грамматику со следующими правилами:

$S \to aAX_{21}$, $S \to aX_{21}$, $S \to bBX_{31}$, $S \to bX_{31}$,

$A \to aAX_{22}$, $A \to aX_{22}$, $A \to bBX_{32}$, $A \to bX_{32}$, $A \to aA$, $A \to a$,

$B \to aAX_{21}$, $B \to aX_{21}$, $B \to bBX_{31}$, $B \to bX_{31}$, $B \to bB$, $B \to b$,

$X_{21} \to BX_{11}$, $X_{21} \to B$,

$X_{22} \to BX_{12}$,

$X_{23} \to BX_{13}$,

которая содержит бесполезные символы. Удалив их с помощью алгоритма 3.3, получим грамматику с правилами:

$S \to aAX_{21}$, $S \to aX_{21}$, $A \to aA$, $A \to a$, $B \to bB$, $B \to b$, $X_{21} \to B$.

Шаг 3. В грамматике есть правило, правая часть которого начинается с нетерминала. Оно должно быть заменено с использованием B-правил: $B \to bB$, $B \to b$. После замены правила грамматики примут вид:

$S \to aAX_{21}$, $S \to aX_{21}$, $A \to aA$, $A \to a$, $B \to bB$, $B \to b$, $X_{21} \to bB$, $X_{21} \to b$.

Шаг 4. Не выполняется.

Можно заметить, что исходная и полученная грамматики определяют один и тот же язык $L = \{a^n b^m \mid n > 0, m > 0\}$.

Преобразуем грамматику, рассмотренную в примере 3.15, при помощи второго алгоритма преобразования в нормальную форму Грейбах.

Пример 3.20

🗁 Правила исходной грамматики имеют следующий вид:

$E \to T$, $E \to TE'$, $E' \to +T$, $E' \to +TE'$, $T \to P$, $T \to PT'$, $T' \to *P$, $T' \to *PT'$, $P \to (E)$, $P \to i$.

Для того чтобы терминальные символы грамматики не пересекались с операциями, используемыми для записи уравнений, выполним следующую замену символов: символ '+' заменен символом 'a', символ '*' — символом 'b', символ ' (' — символом 'c', а символ ') ' — символом 'd'. После замены правила грамматики имеют следующий вид:

$E \to T$, $E \to TE'$,

$E' \to aT$, $E' \to aTE'$,

$T \to P$, $T \to PT'$,

$T' \to bP$, $T' \to bPT'$,

$P \to cEd$, $P \to i$.

Система определяющих уравнений этой грамматики будет выглядеть следующим образом:

$E = T + TE'$

$E' = aT + aTE'$

$T = P + PT'$

$T' = bP + bPT'$

$P = cEd + i$

Матрицы, из которых будут строиться системы определяющих уравнений, следующие:

$$\mathbf{A} = \begin{bmatrix} E & E' & T & T' & P \end{bmatrix},$$

$$\mathbf{R} = \begin{bmatrix} \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varepsilon + E' & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varepsilon + T' & \varnothing & \varnothing \end{bmatrix},$$

$$\mathbf{B} = \begin{bmatrix} \varnothing & aT + aTE' & \varnothing & bP + bPT' & cEd + i \end{bmatrix},$$

$$\mathbf{Q} = \begin{bmatrix} X_{11} & X_{12} & X_{13} & X_{14} & X_{15} \\ X_{21} & X_{22} & X_{23} & X_{24} & X_{25} \\ X_{31} & X_{32} & X_{33} & X_{34} & X_{35} \\ X_{41} & X_{42} & X_{43} & X_{44} & X_{45} \\ X_{51} & X_{52} & X_{53} & X_{54} & X_{55} \end{bmatrix}.$$

Шаг 1. Построим систему определяющих уравнений $\mathbf{A} = \mathbf{AR} + \mathbf{B}$.

Шаг 2. Построим систему определяющих уравнений $\mathbf{A} = \mathbf{BQ} + \mathbf{B}$. После выполнения действий в правой части этого уравнения оно принимает вид:

$$\begin{bmatrix} E & E' & T & T' & P \end{bmatrix} = \begin{bmatrix} Y_1 & Y_2 & Y_3 & Y_4 & Y_5 \end{bmatrix},$$

где

$$Y_1 = aTX_{21} + aTE'X_{21} + bPX_{41} + bPT'X_{41} + cEdX_{51} + iX_{51},$$

$$Y_2 = aTX_{22} + aTE'X_{22} + bPX_{42} + bPT'X_{42} + cEdX_{52} + iX_{52} + aT + aTE',$$

$$Y_3 = aTX_{23} + aTE'X_{23} + bPX_{43} + bPT'X_{43} + cEdX_{53} + iX_{53},$$

$$Y_4 = aTX_{24} + aTE'X_{24} + bPX_{44} + bPT'X_{44} + cEdX_{54} + iX_{54} + bP + bPT',$$

$$Y_5 = aTX_{25} + aTE'X_{25} + bPX_{45} + bPT'X_{45} + cEdX_{55} + iX_{55} + cEd + i.$$

Правила КС-грамматики, соответствующие этой системе уравнений, выглядят следующим образом:

$E \to aTX_{21}$, $E \to aTE'X_{21}$, $E \to bPX_{41}$, $E \to bPT'X_{41}$, $E \to cEdX_{51}$, $E \to iX_{51}$,

$E' \to aTX_{22}$, $E' \to aTE'X_{22}$, $E' \to bPX_{42}$, $E' \to bPT'X_{42}$, $E' \to cEdX_{52}$, $E' \to iX_{52}$, $E' \to aT$, $E' \to aTE'$,

$T \to aTX_{23}$, $T \to aTE'X_{23}$, $T \to bPX_{43}$, $T \to bPT'X_{43}$, $T \to cEdX_{53}$, $T \to iX_{53}$,

$T' \to aTX_{24}$, $T' \to aTE'X_{24}$, $T' \to bPX_{44}$, $T' \to bPT'X_{44}$, $T' \to cEdX_{54}$, $T' \to iX_{54}$, $T' \to bP$, $T' \to bPT'$,

$P \to aTX_{25}$, $P \to aTE'X_{25}$, $P \to bPX_{45}$, $P \to bPT'X_{45}$, $P \to cEdX_{55}$, $P \to iX_{55}$, $P \to cEd$, $P \to i$.

Построим систему определяющих уравнений $\mathbf{Q} = \mathbf{RQ} + \mathbf{R}$. После выполнения преобразований она имеет вид:

$$\begin{bmatrix} X_{11} & X_{12} & X_{13} & X_{14} & X_{15} \\ X_{21} & X_{22} & X_{23} & X_{24} & X_{25} \\ X_{31} & X_{32} & X_{33} & X_{34} & X_{35} \\ X_{41} & X_{42} & X_{43} & X_{44} & X_{45} \\ X_{51} & X_{52} & X_{53} & X_{54} & X_{55} \end{bmatrix} = \begin{bmatrix} Y_{11} & Y_{12} & Y_{13} & Y_{14} & Y_{15} \\ Y_{21} & Y_{22} & Y_{23} & Y_{24} & Y_{25} \\ Y_{31} & Y_{32} & Y_{33} & Y_{34} & Y_{35} \\ Y_{41} & Y_{42} & Y_{43} & Y_{44} & Y_{45} \\ Y_{51} & Y_{52} & Y_{53} & Y_{54} & Y_{55} \end{bmatrix},$$

где

$$Y_{31} = X_{11} + E'X_{11} + E',$$

$Y_{32} = X_{12} + E'X_{12}$,

$Y_{33} = X_{13} + E'X_{13}$,

$Y_{34} = X_{14} + E'X_{14}$,

$Y_{35} = X_{15} + E'X_{15}$,

$Y_{51} = X_{31} + T'X_{31}$,

$Y_{52} = X_{32} + T'X_{32}$,

$Y_{53} = X_{33} + T'X_{33} + T'$,

$Y_{54} = X_{34} + T'X_{34}$,

$Y_{55} = X_{35} + T'X_{35}$.

Правила КС-грамматики, соответствующие этой системе уравнений, следующие:

$X_{31} \to X_{11}$, $X_{31} \to E'X_{11}$, $X_{31} \to E'$,

$X_{32} \to X_{12}$, $X_{32} \to E'X_{12}$,

$X_{33} \to X_{13}$, $X_{33} \to E'X_{13}$,

$X_{34} \to X_{14}$, $X_{34} \to E'X_{14}$,

$X_{35} \to X_{15}$, $X_{35} \to E'X_{15}$,

$X_{51} \to X_{31}$, $X_{51} \to T'X_{31}$,

$X_{52} \to X_{32}$, $X_{52} \to T'X_{32}$,

$X_{53} \to X_{33}$, $X_{53} \to T'X_{33}$, $X_{53} \to T'$,

$X_{54} \to X_{34}$, $X_{54} \to T'X_{34}$,

$X_{55} \to X_{35}$, $X_{55} \to T'X_{35}$.

Объединив два множества правил, получим КС-грамматику:

$E \to aTX_{21}$, $E \to aTE'X_{21}$, $E \to bPX_{41}$, $E \to bPT'X_{41}$, $E \to cEdX_{51}$, $E \to iX_{51}$,

$E' \to aTX_{22}$, $E' \to aTE'X_{22}$, $E' \to bPX_{42}$, $E' \to bPT'X_{42}$, $E' \to cEdX_{52}$, $E' \to iX_{52}$, $E' \to aT$,

$E' \to aTE'$,

$T \to aTX_{23}$, $T \to aTE'X_{23}$, $T \to bPX_{43}$, $T \to bPT'X_{43}$, $T \to cEdX_{53}$, $T \to iX_{53}$,

$T' \to aTX_{24}$, $T' \to aTE'X_{24}$, $T' \to bPX_{44}$, $T' \to bPT'X_{44}$, $T' \to cEdX_{54}$, $T' \to iX_{54}$, $T' \to bP$,

$T' \to bPT'$,

$P \to aTX_{25}$, $P \to aTE'X_{25}$, $P \to bPX_{45}$, $P \to bPT'X_{45}$, $P \to cEdX_{55}$, $P \to iX_{55}$, $P \to cEd$,

$P \to i$,

$X_{31} \to X_{11}$, $X_{31} \to E'X_{11}$, $X_{31} \to E'$,

$X_{32} \to X_{12}$, $X_{32} \to E'X_{12}$,

$X_{33} \to X_{13}$, $X_{33} \to E'X_{13}$,

$X_{34} \to X_{14}$, $X_{34} \to E'X_{14}$,

$X_{35} \to X_{15}$, $X_{35} \to E'X_{15}$,

$X_{51} \to X_{31}$, $X_{51} \to T'X_{31}$,

$X_{52} \to X_{32}$, $X_{52} \to T'X_{32}$,

$X_{53} \to X_{33}$, $X_{53} \to T'X_{33}$, $X_{53} \to T'$,

$X_{54} \to X_{34}$, $X_{54} \to T'X_{34}$,

$X_{55} \to X_{35}$, $X_{55} \to T'X_{35}$,

которая содержит бесполезные символы. Удалив бесполезные символы при помощи алгоритма 3.3, получим грамматику:

$E \to cEdX_{51}$, $E \to iX_{51}$,

$E' \to aT$, $E' \to aTE'$,

$T \to cEdX_{53}$, $T \to iX_{53}$,

$T' \to bP$, $T' \to bPT'$,

$P \to cEd$, $P \to i$,

$X_{31} \to E'$,

$X_{51} \to X_{31}$, $X_{51} \to T'X_{31}$,

$X_{53} \to X_{33}$, $X_{53} \to T'X_{33}$, $X_{53} \to T'$.

Шаг 3. В грамматике есть правила, правая часть которых начинается с нетерминала. После замены в соответствии с шагом 3 алгоритма правила грамматики принимают вид:

$E \to cEdX_{51}$, $E \to iX_{51}$,

$E' \to aT$, $E' \to aTE'$,

$T \to cEdX_{53}$, $T \to iX_{53}$,

$T' \to bP$, $T' \to bPT'$,

$P \to cEd$, $P \to i$,

$X_{31} \to aT$, $X_{31} \to aTE'$,

$X_{51} \to aT$, $X_{51} \to aTE'$, $X_{51} \to bPX_{31}$, $X_{51} \to bPT'X_{31}$,

$X_{53} \to bP$, $X_{53} \to bPTaT$, $X_{53} \to aTE'$.

Шаг 4. В правой части некоторых правил грамматики терминалы встречаются не на первом месте, поэтому заменим их новыми нетерминалами и добавим в грамматику соответствующие правила. Выполнив обратное

преобразование символов: символ 'a' — на символ '+', 'b' — на символ '*', 'c' — на символ '(', 'd' — на символ ')', получим следующие правила грамматики:

$E \to (EDB,\ E \to iB,$

$E' \to +T,\ E' \to +TE',$

$T \to (EDC,\ T \to iC,$

$T' \to *P,\ T' \to *PT',$

$P \to (ED,\ P \to i,$

$A \to +T,\ A \to +TE',$

$B \to +T,\ B \to +TE',\ B \to *PA,\ B \to *PT'A,$

$C \to *P,\ C \to *PT,\ C \to +T,\ C \to +TE',$

$D \to).$

□

Сравнение результатов преобразований грамматики приведено в табл. 3.1.

Таблица 3.1. *Результаты преобразований грамматик*

Номер алгоритма	Исходная грамматика		Преобразованная грамматика	
	Число правил грамматики	Число нетерминалов	Число правил грамматики	Число нетерминалов
3.12	10	5	19	6
3.13	10	5	20	9

Из табл. 3.1 видно, что алгоритм 3.12, незначительно увеличивая число нетерминальных символов, увеличил число правил в преобразованной грамматике практически вдвое. Алгоритм 3.13, как и предполагалось, увеличил число нетерминалов по сравнению с исходной грамматикой в 2 раза. Видимо, особенности рассмотренной грамматики таковы, что второе свойство алгоритма 3.13 (незначительное увеличение числа правил преобразованной грамматики) проиллюстрировать не удалось.

3.10. Свойства замкнутости КС-языков

По определению, множество *замкнуто* относительно некоторой операции, если результат ее применения к любому элементу множества (для унарной

операции) или к любой паре элементов (для бинарной операции) содержится в этом множестве.

Свойства замкнутости используются, в основном, в теоретических исследованиях: они позволяют упростить доказательства принадлежности языков к классу КС-языков.

Часто при проектировании языковых процессоров грамматику, описывающую язык, приходится разбивать на части (подграмматики), что позволяет использовать различные алгоритмы синтаксического анализа для разных конструкций языка.

Пусть $\boldsymbol{L}$ — класс языков и язык $L \subseteq \Sigma^*$ принадлежит $\boldsymbol{L}$. Допустим, что $\Sigma = \{a_1, \ldots, a_n\}$ и языки $L_{a_1}, \ldots, L_{a_n}$ принадлежат $\boldsymbol{L}$. Класс языков $\boldsymbol{L}$ *замкнут относительно подстановки*, если для любого набора языков $L, L_{a_1}, \ldots, L_{a_n}$ язык

$$L' = \{x_1 \ldots x_k \mid a_{j1} \ldots a_{jk} \in L, x_1 \in L_{a_{j1}}, \ldots, x_k \in L_{a_{jk}}\}$$

принадлежит $\boldsymbol{L}$.

Пример 3.21

Если язык $L = \{0^n 1 0^n \mid n > 0\}$, а языки $L_0 = \{a^m b^m \mid m > 0\}$ и $L_1 = \{c^2\}$, то результатом подстановки L_0 и L_1 в L является язык:

$$L' = \{a^{m1} b^{m1} a^{m2} b^{m2} \ldots a^{mn} b^{mn} cc a^{m1} b^{m1} a^{m2} b^{m2} \ldots a^{mn} b^{mn} \mid n > 0,\ m > 0\},$$

т. к. вместо каждого нуля цепочки языка L, стоящего на i-ой позиции, необходимо подставить цепочку языка L_0, имеющую вид $a^{mi} b^{mi}$, а вместо 1 — цепочку cc.

Пусть имеется n КС-языков $L_1, L_2, \ldots, L_n$, которые порождаются КС-грамматиками, и некоторый КС-язык L, порождаемый грамматикой $G = (N, \{a_1, \ldots, a_n\}, P, S)$. Преобразуем эти грамматики так, чтобы попарные пересечения их нетерминальных словарей были бы пустыми. Свяжем каждый терминальный символ a_i грамматики G с начальным символом грамматики G_i $(1 \le i \le n)$, порождающей язык L_i. Объединение грамматик $G_1, \ldots, G_n$ является КС-грамматикой, порождающей язык, получаемый из языков $L_1, L_2, \ldots, L_n$ с помощью операции *подстановки* языков $L_1, L_2, \ldots, L_n$ в L.

Докажем далее, что класс КС-языков замкнут относительно подстановки.

Утверждение

Пусть $L \subseteq \Sigma^*$ — КС-язык, $\Sigma = \{a_1, \ldots, a_n\}$, $L_a \subseteq \Sigma^*{}_a$ — КС-язык для каждого $a \in \Sigma$ и L' — результат подстановки языков L_a вместо a в цепочки языка L.

Доказательство

□ Пусть $G = \{N, \Sigma, P, S\}$ — КС-грамматика языка L, и $G_a = \{\Sigma_a, N_a, P_a, a'\}$ — КС-грамматика языка L_a. Предполагая, что N и N_a попарно не пересекаются, построим грамматику $G' = \{N', \Sigma', P', S\}$ следующим образом:

1. $N' = \bigcup_{a \in \Sigma} N_a \cup N$.
2. $\Sigma' = \bigcup_{a \in \Sigma} \Sigma_a$.
3. $P' = \{A \to h(\alpha) \mid A \to \alpha \in P\} \cup \bigcup_{a \in \Sigma} P_a$, где $h(A)$ — такой гомоморфизм, определенный на $\Sigma \cup N$, что $h(A) = A$ для всех $A \in N$ и $h(a) = a'$ для всех $a \in \Sigma$. Это означает, что множество правил грамматики G' включает в себя все правила грамматик G_a, а также правила грамматики G, в которых все терминалы a заменены нетерминалами a'.

Пусть имеется цепочка $a_{i1} \ldots a_{jk} \in L$ и $x_i \in L_{a_{ji}}$ для $1 \le i \le k$. Тогда в грамматике G' существует вывод $S \Rightarrow^* a'_{j1} \ldots a'_{jk} \Rightarrow^* x_1 a'_{j2} \ldots a'_{jk} \Rightarrow \ldots \Rightarrow^* x_1 \ldots x_k$. Следовательно, $L' \subseteq L(G')$.

Допустим теперь, что некоторая цепочка $w \in L(G')$. Рассмотрим дерево вывода этой цепочки D. По условию, множества нетерминалов N и N_a не пересекаются, поэтому каждый лист дерева с меткой, отличной от ε, имеет одного предка, помеченного меткой a', такой, что $a \in \Sigma$. Если удалить из дерева D каждую вершину, у которой есть предок, отличный от нее и помеченный a', где $a \in \Sigma$, то получим дерево вывода D' с кроной $a'_{j1} \ldots a'_{jk}$, где $a_{j1} \ldots a_{jk} \in L$. Если x_i — крона поддерева дерева D, над которым доминирует i-ый лист дерева D', то $w = x_1 \ldots x_k$ и $x_i \in L_{a_{ji}}$. Таким образом, $L(G') = L'(G)$. ■

Как следствие, верно утверждение: класс КС-языков замкнут относительно объединения, конкатенации, итерации, позитивной итерации и гомоморфизма.

Контрольные вопросы

1. Дайте определения цепочки и языка. Какие операции можно выполнять над цепочками символов и над языками?
2. Перечислите и приведите примеры различных способов определения бесконечных языков.
3. Что такое синтаксис и семантика языка? Какие средства существуют для описания синтаксиса и семантики?
4. Дайте определение формальной грамматики.

5. Каким образом можно классифицировать формальные грамматики?
6. Дайте определение формального языка. Можно ли считать языки программирования формальными?
7. Дайте определение отношения выводимости. Какие виды выводов наиболее часто используются?
8. Определите связь между выводом и деревом вывода.

Упражнения

1. Преобразуйте КС-грамматику $G = (N, \Sigma, P, S)$ в эквивалентную грамматику, не содержащую бесполезных символов:

   1.1. $S \to b$, $S \to C$, $S \to cCB$, $A \to e$, $A \to Ab$, $B \to Bb$, $B \to cB$, $C \to Ca$, $C \to Bf$, $C \to d$;

   1.2. $S \to aC$, $S \to bA$, $A \to cAB$, $B \to aC$, $C \to bA$, $C \to d$;

   1.3. $S \to aABC$, $S \to aE$, $A \to SCD$, $A \to c$, $B \to bFD$, $C \to aE$, $D \to aD$, $B \to b$, $E \to aCE$, $E \to a$, $F \to AB$;

   1.4. $S \to aA$, $A \to aA$, $A \to b$, $A \to cC$, $B \to a$, $B \to cB$, $C \to bAC$.

2. Преобразуйте исходную КС-грамматику $G = (N, \Sigma, P, S)$ в эквивалентную НКС-грамматику:

   2.1. $S \to AB$, $A \to SA$, $A \to BB$, $A \to bB$, $B \to b$, $B \to aA$, $B \to \varepsilon$;

   2.2. $S \to Aa$, $S \to bB$, $A \to cAdA$, $A \to a$, $A \to \varepsilon$, $B \to cBdd$, $B \to \varepsilon$;

   2.3. $S \to bA$, $A \to bA$, $A \to aB$, $A \to \varepsilon$, $B \to bB$, $B \to \varepsilon$;

   2.4. $S \to aAB$, $S \to bA$, $S \to \varepsilon$, $A \to aAB$, $A \to \varepsilon$, $B \to bB$, $B \to c$;

   2.5. $S \to aAB$, $S \to bBS$, $S \to \varepsilon$, $A \to cBS$, $A \to \varepsilon$, $B \to dB$, $B \to \varepsilon$.

3. Преобразуйте КС-грамматику $G = (N, \Sigma, P, S)$ в эквивалентную КС-грамматику, не содержащую правил с одинаковой правой частью:

   3.1. $S \to AC$, $A \to B$, $A \to AaB$, $B \to i$, $C \to D$, $C \to DaC$, $D \to i$;

   3.2. $S \to 1A$, $S \to B0$, $A \to 1A$, $A \to C$, $B \to B0$, $B \to C$, $C \to 1C0$, $C \to \varepsilon$;

   3.3. $S \to SS$, $S \to 1A0$, $A \to 1A0$, $A \to \varepsilon$;

   3.4. $S \to aC$, $S \to bA$, $A \to cAB$, $B \to aC$, $C \to bA$, $C \to d$;

   3.5. $S \to abSa$, $S \to aaAb$, $S \to b$, $A \to baAb$, $A \to b$.

3. Преобразуйте НКС-грамматику $G = (N, \Sigma, P, S)$ в эквивалентную КС-грамматику, не содержащую цепных правил:

   4.1. $S \to LA$, $S \to LB$, $L \to P{:=}$, $L \to Q{:=}$, $P \to i$, $A \to F$, $Q \to i$, $B \to F$, $F \to Q(i)$;

   4.2. $S \to AC$, $A \to B$, $A \to AaB$, $B \to i$, $C \to D$, $C \to DaC$, $D \to i$;

4.3. $S \to 1A$, $S \to B0$, $A \to 1A$, $A \to C$, $B \to B0$, $B \to C$, $C \to 1C0$, $C \to \varepsilon$;

4.4. $S \to T+P$, $S \to T$, $T \to T*P$, $T \to P$, $P \to C$, $C \to |C$, $C \to |$;

4.5. $S \to A$, $S \to B$, $A \to 1A0$, $A \to 1a0$, $B \to 1B00$, $B \to 1b00$.

4. Исключите левую рекурсию из КС-грамматики $G = (N, \Sigma, \underline{P}, S)$:

5.1. $S \to Ba$, $S \to Ab$, $A \to Sa$, $A \to AAb$, $A \to a$, $B \to Sb$, $B \to BBa$, $B \to b$;

5.2. $S \to AB$, $S \to a$, $A \to BS$, $A \to Sb$, $B \to SA$, $B \to BB$, $B \to a$;

5.3. $S \to Ab$, $A \to Sa$, $A \to cB$, $B \to bS$, $B \to c$;

5.4. $S \to AB$, $A \to SA$, $A \to BB$, $A \to bB$, $B \to b$, $B \to aA$, $B \to \varepsilon$;

5.5. $S \to SaA$, $S \to AA$, $S \to b$, $A \to ASa$, $A \to Ad$, $A \to c$.

4. Преобразуйте в нормальную форму Хомского КС-грамматики $G = (N, \Sigma, P, S)$:

6.1. $S \to AB$, $A \to SA$, $A \to BB$, $A \to bB$, $B \to b$, $B \to aA$, $B \to \varepsilon$;

6.2. $S \to Aa$, $S \to bB$, $A \to cAdA$, $A \to a$, $A \to \varepsilon$, $B \to cBdd$, $B \to \varepsilon$;

6.3. $S \to SS$, $S \to 1A0$, $A \to 1A0$, $A \to \varepsilon$;

6.4. $S \to aC$, $S \to bA$, $A \to cAB$, $B \to aC$, $C \to bA$, $C \to d$.

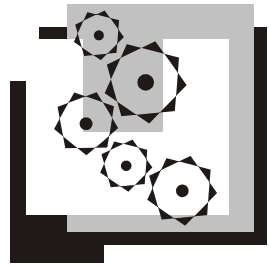
5. Преобразуйте в нормальную форму Грейбах КС-грамматики $G = (N, \Sigma, P, S)$:

7.1. $S \to A$, $S \to B$, $A \to 1A0$, $A \to 1a0$, $B \to 1B00$, $B \to 1b00$;

7.2. $S \to Aa$, $S \to bB$, $A \to cAdA$, $A \to a$, $A \to \varepsilon$, $B \to cBdd$, $B \to \varepsilon$;

7.3. $S \to abSa$, $S \to aaAb$, $S \to b$, $A \to baAb$, $A \to b$;

7.4. $S \to AB$, $A \to SA$, $A \to BB$, $A \to bB$, $B \to b$, $B \to aA$, $B \to \varepsilon$.



Глава 4

Конечные автоматы и преобразователи

Определение языка конечными средствами возможно не только при помощи формальных грамматик, но также путем использования математической модели устройства, распознающего предложения языка. Остановимся более подробно на описании синтаксиса языков с помощью простейших распознающих устройств автоматов [2, 6, 38, 46] и их использовании при проектировании языковых процессоров.

4.1. Распознающий автомат

Распознаватель (рис. 4.1) состоит из трех частей: входной ленты, устройства управления с конечной памятью и вспомогательной памяти.

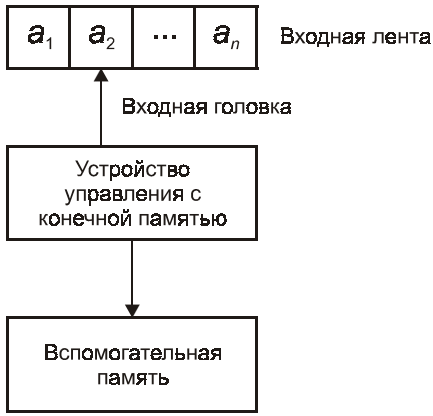


Рис. 4.1. Модель распознавателя

Входную ленту можно рассматривать как линейную последовательность клеток, каждая из которых содержит один символ конечного входного алфавита Σ.

Самую левую и самую правую позиции входной ленты могут занимать особые символы, не принадлежащие входному алфавиту, называемые соответственно *начальным* и *концевым маркером*.

Входная головка в каждый момент времени читает (или обозревает) один символ входной ленты, называемый *текущим*. Распознаватель работает по тактам. За один *такт работы* распознавателя входная головка может сдвинуться на одну клетку влево (вправо) или остаться неподвижной.

Вспомогательная память распознавателя может быть нескольких типов в зависимости от языка, который может быть определен с его помощью. Во вспомогательной памяти хранится информация, построенная из символов некоторого конечного алфавита Г. Поведение вспомогательной памяти можно задать с помощью двух функций:

- функции *доступа к памяти* f;
- функции *преобразования памяти* g.

Например, для вспомогательной памяти типа *магазин* функция доступа f — это отображение вида:

$$f: \quad \Gamma^+ \to \Gamma,$$

в котором $f(z_1, \ldots, z_n) = z_1$ (в магазине доступен только верхний символ — *вершина магазина*), а функция преобразования памяти g — отображение вида:

$$g: \quad \Gamma^* \times \Gamma^* \to \Gamma^*,$$

в котором $g(z_1, \ldots, z_n, Y_1, \ldots, Y_m) = Y_1 \ldots Y_m z_2 \ldots z_n$ (верхний символ магазина заменяется цепочкой конечной длины из символов, принадлежащих алфавиту Г).

Устройство управления представляет собой множество состояний вместе с отображением, которое описывает изменение состояний в зависимости от текущего входного символа и информации, извлеченной из вспомогательной памяти. Устройство управления определяет также направление движения входной ленты и преобразование вспомогательной памяти. Другими словами, под устройством управления понимается программа, управляющая поведением распознавателя.

Работу распознавателя удобно описывать в терминах *конфигураций*, каждая из которых определяет текущее состояние распознавателя: состояние устройства управления, содержимое входной ленты и положение входной головки, содержание вспомогательной памяти. Переход из одной конфигурации в другую (*такт работы распознавателя*) выполняется следующим образом:

- читается входной символ и с помощью функции доступа определяется содержимое вспомогательной памяти. Входной символ, содержимое вспомогательной памяти и текущее состояние устройства управления определяют действия, которые необходимо выполнить в данном такте;

- сдвигается (или остается неподвижной) входная головка;
- во вспомогательную память помещается некоторая информация;
- изменяется состояние устройства управления.

Из множества конфигураций выделим две: начальную и конечную.

Конфигурация называется н*ачальной*, если устройство управления находится в заданном начальном состоянии, вспомогательная память имеет заданное начальное состояние, а входная головка читает самый левый символ входной ленты.

Конфигурация называется *заключительной*, если устройство управления находится в одном из множества заключительных состояний, входная головка обозревает самый правый символ входной ленты (концевой маркер) и, возможно, вспомогательная память удовлетворяет некоторым условиям.

Говорят, что распознаватель *допускает входную цепочку w*, если, начиная с начальной конфигурации, в которой цепочка *w* записана на входной ленте, распознаватель может выполнить последовательность тактов, завершающуюся конечной конфигурацией.

Языком, *определяемым* (*допускаемым*, *распознаваемым*) распознавателем, называется множество цепочек, которое он допускает.

Для каждого класса грамматик из иерархии Хомского существует естественный класс распознавателей, допускающих такой же класс языков:

- язык, определяемый грамматикой общего вида, допускается машиной Тьюринга;
- контекстно-зависимый язык допускается двусторонним недетерминированным линейно ограниченным автоматом;
- контекстно-свободный язык допускается односторонним недетерминированным автоматом с магазинной памятью;
- автоматный язык допускается односторонним недетерминированным конечным автоматом.

4.2. Конечный автомат

Конечный автомат является простейшим распознавателем, у которого отсутствует вспомогательная память (рис. 4.2).

Рассмотрим *односторонний* (входная головка автомата сдвигается только вправо) *недетерминированный конечный автомат* (НКА). В качестве дополнительного требования будем считать, что входная головка сдвигается на *каждом такте* работы автомата. Это не сужает класс языков, допускаемых таким автоматом, но упрощает его описание.

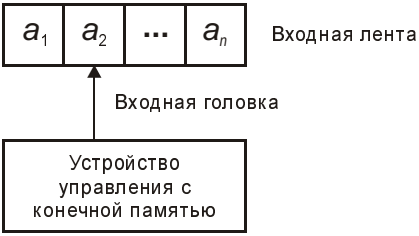


Рис. 4.2. Модель конечного автомата

Недетерминированным конечным автоматом называется пятерка объектов $K = (Q, \Sigma, \delta, q_0, F)$, где Q — конечное *множество состояний устройства управления*, Σ — конечный *алфавит входных символов*, δ — *функция переходов*, которая задается отображением $\delta: Q \times \Sigma \to P(Q)$, в котором $P(Q)$ есть конечное множество подмножеств множества Q, $q_0 \in Q$ — *начальное состояние* устройства управления, $F \subseteq Q$ — *множество заключительных состояний*.

Такт работы НКА определяется текущим состоянием устройства управления и текущим входным символом. На каждом такте НКА переходит из одной конфигурации в другую. При этом входная головка сдвигается на один символ вправо и производится изменение состояния устройства управления.

Формально конфигурацией автомата называется пара $(q, w) \in Q \times \Sigma^*$. Конфигурация (q_0, w) называется н*ачальной*, а (q, ε), где $q \in F$, — *заключительной*.

Такт работы конечного автомата представляется бинарным отношением $(\vdash)$, определенным на множестве конфигураций. Если $q' \in \delta(q, a)$, то $(q, aw) \vdash (q', w)$ для всех $w \in \Sigma^*$.

Пусть $\{C\}$ — множество конфигураций. Тогда:

- $C \vdash^0 C'$ означает, что $C = C'$;
- $C_0 \vdash^k C_k$ для $k \geq 1$ говорит, что существуют такие конфигурации $C_1, \ldots, C_{k-1}$, в которых $C_i \vdash C_{i+1}$ для всех $0 \leq i < k$;
- $C \vdash^+ C'$ означает, что $C \vdash^k C'$ для некоторого $k \geq 1$, а $C \vdash C'$ означает, что $C \vdash^k C'$ для некоторого $k \geq 0$.

Отношения $(\vdash^+)$ и $(\vdash^*)$ являются *транзитивным* и *рефлексивно-транзитивным* замыканием отношения $(\vdash)$ соответственно.

Говорят, что конечный автомат $K = (Q, \Sigma, \delta, q_0, F)$ *допускает* (*распознает*) входную цепочку $w \in \Sigma$, если $(q_0, w) \vdash (q, \varepsilon)$ для некоторого $q \in F$.

Языком, *определяемым* (*допускаемым*, *распознаваемым*) конечным автоматом K, называется множество входных цепочек, допускаемых этим автоматом, т. е.

$$L(K) = \{w \mid w \in \Sigma \text{ и } (q_0, w) \vdash^* (q, \varepsilon) \text{ для некоторого } q \in F\}.$$

Пример 4.1

🗁 Рассмотрим конечный автомат $K = (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \delta, q_0, \{q_3\})$, в котором функция переходов δ определяется следующим образом:

1) $\delta(q_0, a) = \{q_1\}$; 2) $\delta(q_0, b) = \{q_0\}$;
3) $\delta(q_1, a) = \{q_1\}$; 4) $\delta(q_1, b) = \{q_2\}$;
5) $\delta(q_2, a) = \{q_3\}$; 6) $\delta(q_2, b) = \{q_0\}$;
7) $\delta(q_3, a) = \{q_3\}$; 8) $\delta(q_3, b) = \{q_3\}$.

Этот конечный автомат допускает все цепочки, состоящие из символов a и b и содержащие хотя бы одну подцепочку aba. Для входной цепочки $baababa$ автомат K выполнит следующую последовательность тактов (цифра под знаком отношения ($\vdash$) означает номер используемой на данном шаге функции переходов):

$$(q_0, baababa) \vdash_2 (q_0, aababa) \vdash_1$$
$$(q_1, ababa) \vdash_3 (q_1, baba) \vdash_4$$
$$(q_2, aba) \vdash_5 (q_3, ba) \vdash_8$$
$$(q_3, a) \vdash_7 (q_3, \varepsilon).$$

Таким образом, цепочка $baababa$ принадлежит языку $L(K)$. 🗀

Пример 4.2

🗁 Определим конечный автомат, допускающий цепочки из 0 и 1, содержащие подцепочку 11.

Примерами цепочек, допускаемых этим автоматом, являются цепочки 11, 011001, 10111.

$K = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$. Здесь q_0 — начальное состояние, в котором автомат находится до тех пор, пока на вход не поступит символ 1, q_1 — состояние, соответствующее приходу первой единицы, а q_2 — заключительное состояние, соответствующее тому, что на вход поступила цепочка, допускаемая автоматом. Функция переходов δ определяется следующим образом:

1) $\delta(q_0, 0) = \{q_0\}$; 2) $\delta(q_0, 1) = \{q_1\}$;
3) $\delta(q_1, 0) = \{q_0\}$; 4) $\delta(q_1, 1) = \{q_2\}$;
5) $\delta(q_2, 0) = \{q_2\}$; 6) $\delta(q_2, 1) = \{q_2\}$.

Для входной цепочки 10111 автомат K выполнит следующую последовательность тактов:

$$(q_0, 10111) \vdash (q_1, 0111) \vdash (q_0, 111) \vdash$$
$$(q_1, 11) \vdash (q_2, 1) \vdash (q_2, \varepsilon).$$

🗀

4.3. Способы задания конечных автоматов

Определение конечного автомата как пятерки объектов с детальным описанием функции переходов слишком громоздко и неинформативно. Существует два более удобных способа описания автоматов:

- *таблица переходов*;
- *диаграмма* (граф) переходов.

Таблица переходов представляет собой обычное табличное представление функции, которая двум аргументам ставит в соответствие одно значение. Строки таблицы отмечены состояниями, а столбцы — входными символами. Начальное состояние находится всегда в первой строке, а конечные состояния — отмечены символом (*). В табл. 4.1 приведена таблица переходов конечного автомата из примера 4.1.

Таблица 4.1. *Таблица переходов для конечного автомата из примера 4.1*

	a	**b**
q_0	$\{q_1\}$	$\{q_0\}$
q_1	$\{q_1\}$	$\{q_2\}$
q_2	$\{q_3\}$	$\{q_0\}$
q_3^*	$\{q_3\}$	$\{q_3\}$

Диаграммой переходов НКА называется неупорядоченный граф, удовлетворяющий следующим условиям:

- каждому состоянию q соответствует некоторая вершина, отмеченная его именем;
- диаграмма переходов содержит дугу из состояния p в состояние q, отмеченную символом a, если $p \in \delta(q, a)$. Если существует несколько входных символов, переводящих автомат из состояния p в состояние q, то диаграмма переходов может содержать одну дугу, отмеченную списком этих символов;
- вершины, соответствующие заключительным состояниям (состояниям из множества F), отмечаются двойным кружком, остальные состояния — одинарным.

На рис. 4.3 приведена диаграмма переходов для конечного автомата из примера 4.1.

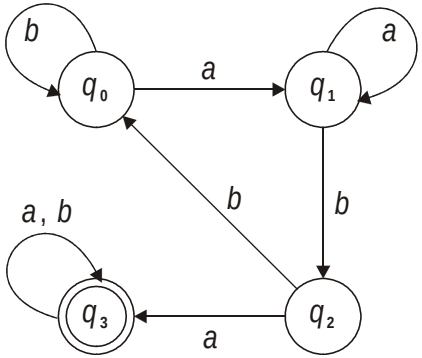


Рис. 4.3. Диаграмма переходов для конечного автомата из примера 4.1

Пример 4.3

🗁 На рис. 4.4 приведена диаграмма переходов конечного автомата $K = (\{q_0, q_1, q_2, q_3, q_4\}, \{a, b, c\}, q_0, \delta, F)$, допускающего цепочки, состоящие из символов алфавита $\{a, b, c\}$, которые заканчиваются символом, ранее встречающимся в этой цепочке. Например, $acac \in L(K)$, а цепочка $abbaac \notin L(K)$.

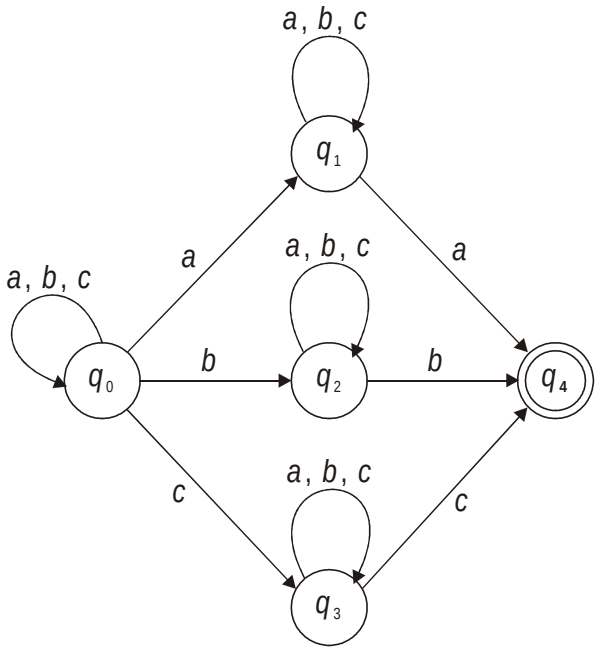


Рис. 4.4. Диаграмма переходов для конечного автомата из примера 4.3

На рис. 4.5 изображены последовательности состояний, в которые может перейти данный НКА при распознавании цепочки *acac*. Поскольку $q_4 \in F$, то $acac \in L(K)$, а автомат выполнит следующую последовательность тактов:

$$(q_0, acac) \vdash (q_0, cac) \vdash (q_3, ac) \vdash (q_3, c) \vdash (q_4, \varepsilon).$$

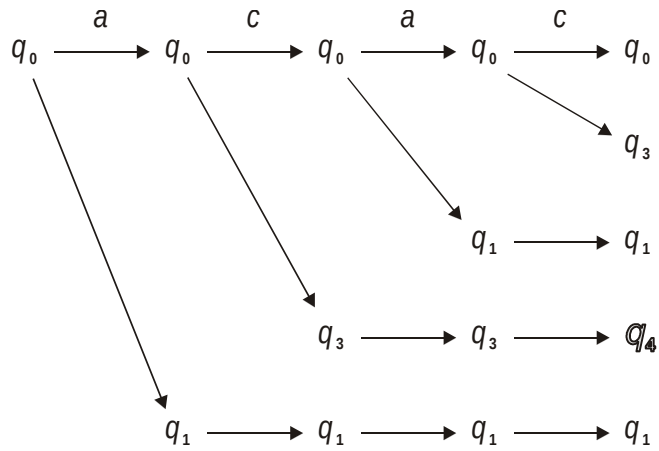


Рис. 4.5. Состояния, в которые может перейти НКА из примера 4.3 при обработке цепочки *acac*

4.4. Детерминированные конечные автоматы

Как следует из определения НКА, автомат, находясь в некоторой конфигурации, при поступлении на вход одного и того же символа может перейти в несколько состояний из заданного множества состояний. На практике, в связи со сложностью моделирования работы НКА, часто используются детерминированные конечные автоматы (ДКА).

Детерминированным конечным автоматом называется конечный автомат $K = (Q, \Sigma, \delta, q_0, F)$, если множество $\delta(q, a)$ содержит не более одного состояния для любых $q \in Q$ и $a \in \Sigma$.

Если $\delta(q, a)$ для всех $q \in Q$ и $a \in \Sigma$ содержит *одно состояние*, то автомат K называется *полностью определенным* конечным автоматом.

Конечные автоматы из примеров 4.1 и 4.2 — детерминированные и полностью определенные, а конечный автомат из примера 4.3 — недетерминированный.

В дальнейшем конечным автоматом будем называть полностью определенный детерминированный конечный автомат.

Можно доказать [3, 38], что если $L = L(K)$ для некоторого недетерминированного конечного автомата, то можно построить такой детерминированный конечный автомат K', что $L = L(K')$. Рассмотрим алгоритм построения детерминированного конечного автомата по недетерминированному конечному автомату.

Алгоритм 4.1. *Построение ДКА по НКА*

Вход: НКА $K = (Q, \Sigma, \delta, q_0, F)$.

Выход: ДКА $K' = (Q', \Sigma, \delta', q_0', F')$, у которого $L(K') = L(K)$.

Описание алгоритма:

□

1. Входные алфавиты автоматов K и K' совпадают.
2. Множество Q' содержит множество всех подмножеств множества Q (*булеан* множества Q). Если $|Q| = n$, то $|Q'| = 2^n$. Обычно не все состояния из Q' достижимы из начального состояния (состояние q достижимо из начального состояния q_0', если имеется путь из состояния q_0' в состояние q). Недостижимые состояния можно исключить из множества Q', поэтому фактически число состояний ДКА может быть гораздо меньше, чем 2^n.
3. $q_0' = q_0$.
4. F' есть множество подмножеств S множества Q, для которых $S \cap F \neq \varnothing$, т. е. F' состоит из всех множеств состояний Q недетерминированного конечного автомата K, содержащих хотя бы одно заключительное состояние.
5. Для каждого множества $S \subseteq Q$ и каждого входного символа a из Σ выполняется:

$$\delta'(S,a) = \bigcup_{p \text{ из } S} \delta(p,a),$$

т. е. чтобы найти $\delta'(S, a)$ необходимо для каждого состояния p из S найти состояния, в которые можно попасть из p для входного символа a, а затем взять объединение множеств найденных состояний по всем состояниям p.

■

Пример 4.4

🗁 Построим ДКА K', допускающий язык, определяемый НКА из примера 4.3. Исходный НКА K содержит 5 состояний, следовательно, ДКА K' должен

содержать не более 32 состояний. Рассматривая только достижимые состояния, получим функцию переходов ДКА, представленную в виде таблицы переходов (табл. 4.2).

Таблица 4.2. *Таблица переходов ДКА из примере 4.4*

	a	**b**	**c**
$p_0 = \{q_0\}$	p_1	p_2	p_3
$p_1 = \{q_0, q_1\}$	p_4	p_5	p_6
$p_2 = \{q_0, q_2\}$	p_5	p_7	p_8
$p_3 = \{q_0, q_3\}$	p_6	p_8	p_9
$p_4 = \{q_0, q_1, q_4\}$	p_4	p_5	p_6
$p_5 = \{q_0, q_1, q_2\}$	p_{10}	p_{10}	p_{11}
$p_6 = \{q_0, q_1, q_3\}$	p_{12}	p_{11}	p_{12}
$p_7 = \{q_0, q_2, q_4\}$	p_5	p_7	p_8
$p_8 = \{q_0, q_2, q_3\}$	p_{11}	p_{13}	p_{13}
$p_9 = \{q_0, q_3, q_4\}$	p_6	p_8	p_9
$p_{10} = \{q_0, q_1, q_2, q_4\}$	p_{10}	p_{11}	p_{11}
$p_{11} = \{q_0, q_1, q_2, q_3\}$	p_{14}	p_{14}	p_{14}
$p_{12} = \{q_0, q_1, q_3, q_4\}$	p_{12}	p_{11}	p_{12}
$p_{13} = \{q_0, q_2, q_3, q_4\}$	p_{11}	p_{13}	p_{13}
$p_{14} = \{q_0, q_1, q_2, q_3, q_4\}$	$\varnothing$	$\varnothing$	$\varnothing$

Окончательно получим следующий ДКА $K' = (\{p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}\}, \{a, b, c\}, p_0, \delta, \{p_4, p_7, p_9, p_{10}, p_{12}, p_{13}, p_{14}\})$. □

4.5. Автоматные грамматики и конечные автоматы

Между конечными автоматами и автоматными грамматиками существует тесная связь: класс языков, допускаемых конечными автоматами, совпадает с классом языков, порождаемых автоматными грамматиками.

Утверждение 1

Пусть $G = (\mathrm{N}, \Sigma, P, S)$ — автоматная грамматика. Тогда существует такой конечный автомат $K = (Q, \Sigma, \delta, q_0, F)$, что $L(K) = L(G)$. Конечный автомат строится следующим образом:

1. Входные алфавиты конечного автомата и автоматной грамматики совпадают.
2. $Q = \mathrm{N} \cup \{E\}$, где E — заключительное состояние конечного автомата.
3. $q_0 = S$.
4. Если $S \to \varepsilon \in P$, то $F = \{S, E\}$, в противном случае $F = \{E\}$.
5. Функция переходов конечного автомата определяется следующим образом:
   - $E \in \delta(B, a)$, если $B \to a \in P$, $B \in \mathrm{N}$, $a \in \Sigma$;
   - если $B \to aC \in P$, то $C \in \delta(B, a)$;
   - $\delta(E, a) = \varnothing$ для всех $a \in \Sigma$.

Замечание

Построенный конечный автомат в общем случае является недетерминированным.

Пример 4.5

Построим конечный автомат $K = (Q, \Sigma, \delta, q_0, F)$ по автоматной грамматике $G = (\{S, A, B, C\}, \{a, b\}, P, S)$, где $P = \{S \to aA, S \to bB, A \to aA, A \to \varepsilon, A \to bB, B \to aC, C \to aC, C \to \varepsilon\}$.

1. Исключим ε-правила из грамматики G.

   $G' = (\{S, A, B, C\}, \{a, b\}, S, P')$, где $P' = \{S \to aA, S \to a, S \to bB, A \to aA, A \to a, A \to bB, B \to \mathrm{aC}, B \to a, C \to aC, C \to a\}$.

2. $K = (\{S, A, B, C, E\}, \{a, b\}, \delta, S, \{E\})$, где

   $\delta(S, a) = \{A, E\}$; $\delta(S, b) = \{B\}$;

   $\delta(A, a) = \{A, E\}$; $\delta(A, b) = \{B\}$;

   $\delta(B, a) = \{C, E\}$; $\delta(B, b) = \varnothing$;

   $\delta(C, a) = \{C, E\}$; $\delta(C, b) = \varnothing$;

   $\delta(E, a) = \varnothing$; $\delta(E, b) = \varnothing$.

Утверждение 2

Пусть $K = (Q, \Sigma, \delta, q_0, F)$ — конечный автомат. Тогда существует такая автоматная грамматика $G = (\mathrm{N}, \Sigma, P, S)$, что $L(G) = L(K)$. Автоматная грамматика определяется следующим образом:

1. Входные алфавиты автоматной грамматики и конечного автомата совпадают.

2. $N = Q$.
3. $S = q_0$.
4. Правила вывода автоматной грамматики определяются следующим образом:
   - $B \to aC \in P$, если $\delta(B, a) = C$, где $B, C \in Q$, $a \in \Sigma$;
   - $B \to \varepsilon \in P$, если $B \in F$.

Пример 4.6

Построить автоматную грамматику $G = (N, \Sigma, P, S)$ по конечному автомату $K = (\{S_0, S_1, S_2, S_3\}, \{a, b\}, \delta, S_0, \{S_3\})$, где

$\delta(S_0, a) = \{S_1\}$; $\delta(S_0, b) = \{S_2\}$;

$\delta(S_1, a) = \{S_1\}$; $\delta(S_1, b) = \{S_3\}$;

$\delta(S_2, a) = \{S_3\}$; $\delta(S_2, b) = \{S_2\}$;

$\delta(S_3, a) = \varnothing$; $\delta(S_3, b) = \varnothing$.

$G = (\{S_0, S_1, S_2, S_3\}, \{a, b\}, P, S_0)$, где $P = \{S_0 \to aS_1, S_0 \to bS_2, S_1 \to aS_1, S_1 \to bS_3, S_2 \to aS_3, S_2 \to bS_2, S_3 \to \varepsilon\}$.

После исключения ε-правил и бесполезных символов получим:

$G = (\{S_0, S_1, S_2\}, \{a, b\}, P, S_0)$, где $P = \{S_0 \to aS_1, S_0 \to bS_2, S_1 \to aS_1, S_1 \to b, S_2 \to a, S_2 \to bS_2\}$. □

4.6. Решение проблемы принадлежности для конечных автоматов

Проблема принадлежности для конечных автоматов может быть *разрешима* с помощью следующего алгоритма.

Алгоритм 4.2. *Решение проблемы принадлежности*

Вход: Конечный автомат $K = (Q, \Sigma, \delta, q_0, F)$ и цепочка $w \in \Sigma$.

Выход: Ответ "ДА", если $w \in L(K)$; ответ "НЕТ", если $w \notin L(K)$.

Описание алгоритма:

□ Пусть $w = a_1 a_2 \ldots a_n$. Найти последовательность состояний $q_1 = \delta(q_0, a_1)$, $q_2 = \delta(q_1, a_2)$, ... , $q_n = \delta(q_{n-1}, a_n)$. Если $q_n \in F$, то выдать ответ "ДА"; в противном случае выдать ответ "НЕТ". ■

Рассмотрим временную и емкостную сложности этого алгоритма. Естественными мерами сложности в данном случае служат число шагов и число

ячеек памяти, требуемых компьютеру с произвольным доступом к памяти для хранения целых чисел.

Для определения емкостной сложности алгоритма предположим, что именами состояний q_0, q_1, ... , q_i, ... и именами входных символов a_1, a_2, ... , a_j, ... являются целые двоичные числа. Если в качестве машины предполагается использовать обычный компьютер с фон-неймановской архитектурой, то для хранения функции переходов δ можно построить такой двухмерный массив, что в ячейке (i, j) будет находиться (q_i, a_j). Таким образом, емкостная сложность алгоритма определяется размером двухмерного массива.

Легко видеть, что время работы алгоритма линейно зависит от длины цепочки w. Общее время работы алгоритма будет зависеть от времени, необходимого для построения описания автомата (двухмерного массива), которое пропорционально длине описания, и времени выполнения самого алгоритма, которое пропорционально $|w|$.

4.7. Решение проблемы пустоты языка для конечных автоматов

Проблема пустоты языка для конечных автоматов *разрешима* с помощью следующего алгоритма.

Алгоритм 4.3. *Решение проблемы пустоты*

Вход: Конечный автомат $K = (Q, \Sigma, \delta, q_0, F)$.

Выход: Ответ "ДА", если $L(K) \neq \varnothing$; ответ "НЕТ" в противном случае.

Описание алгоритма:

□ Определить множество состояний, достижимых из состояния q_0. Если это множество содержит какое-нибудь заключительное состояние, то выдать ответ "ДА"; в противном случае — "НЕТ". ■

4.8. Решение проблемы эквивалентности для конечных автоматов

Проблема эквивалентности для конечных автоматов *разрешима*. Прежде чем рассмотреть алгоритм, решающий эту проблему, дадим ряд определений и докажем вспомогательную лемму.

Пусть $K = (Q, \Sigma, \delta, q_0, F)$ — конечный автомат, а q_1 и q_2 — его различные состояния. Говорят, что цепочка $x \in \Sigma$ *различает* состояния q_1 и q_2, если $(q_1, x) \vdash (q_3, \varepsilon)$, $(q_2, x) \vdash (q_4, \varepsilon)$ и одно из состояний q_3 и q_4 принадлежит F, а другое нет.

Говорят, что состояния q_1 и q_2 *k-неразличимы*, и пишут $q_1 \equiv^k q_2$, если не существует такой цепочки x, различающей состояния q_1 и q_2, у которой $|x| \le k$.

Говорят, что состояния q_1 и q_2 *неразличимы*, и пишут $q_1 \equiv q_2$, если они k-неразличимы для любого $k \ge 0$.

Если два состояния можно различить, то это можно сделать с помощью входной цепочки, длина которой меньше числа состояний конечного автомата. Докажем это.

Лемма 4.1

Пусть $K = (Q, \Sigma, \delta, q_0, F)$ — конечный автомат с n состояниями. Состояния q_1 и q_2 неразличимы тогда и только тогда, когда они $(n-2)$-неразличимы.

Доказательство

□ Необходимость условия тривиальна.

Достаточность условия тривиальна в тех случаях, когда множество F содержит 0 или n элементов.

Рассмотрим случай, когда число элементов множества F отлично от 0 или n. Покажем, что

$$\equiv \subseteq \equiv^{n-2} \subseteq \equiv^{n-3} \subseteq \ldots \subseteq \equiv^2 \subseteq \equiv^1 \subseteq \equiv^0.$$

Для любых состояний q_1 и q_2 справедливо:

1. $q_1 \equiv^0 q_2$ тогда и только тогда, когда q_1 и q_2 одновременно либо принадлежат, либо не принадлежат множеству F.
2. $q_1 \equiv^k q_2$ тогда и только тогда, когда $q_1 \equiv^{k-1} q_2$ и $\delta(q_1, a) \equiv^{k-1} \delta(q_2, a)$ для всех $a \in \Sigma$.

Отношение $(\equiv^0)$ разбивает множество состояний Q на два класса: F и $(Q - F)$. Если $\equiv^{k+1} \ne \equiv^k$, то отношение $(\equiv^{k+1})$ тоньше $(\equiv^k)$, т. е. в нем по крайней мере на один класс эквивалентности больше, чем в $(\equiv^k)$. Так как каждое из множеств F и $(Q - F)$ содержит не более $(n - 1)$ элементов, то можно получить не более $(n - 2)$ последовательных утончений отношения $(\equiv^0)$. Если $\equiv^{k+1} = \equiv^k$ для некоторого k, то в соответствии с утверждением (2) $\equiv^{k+1} = \equiv^{k+2} = \ldots$. Следовательно, отношение $(\equiv)$ — это первое из отношений $(\equiv^k)$, для которых $\equiv^{k+1} = \equiv^k$. ■

Алгоритм 4.4. *Решение проблемы эквивалентности для конечных автоматов*

Вход: Два детерминированных полностью определенных конечных автомата $K_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$ и $K_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$, таких, что $Q_1 \cap Q_2 = \varnothing$.

Выход: Ответ "ДА", если $L(K_1) = L(K_2)$; "НЕТ" в противном случае.

Описание алгоритма:

□ Построить конечный автомат

$$K = (Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, \delta_1 \cup \delta_2, q_1, F_1 \cup F_2).$$

С помощью доказанной ранее леммы определить, различимы ли два состояния q_1 и q_2. Если состояния q_1 и q_2 различимы, то выдать ответ "НЕТ"; в противном случае выдать "ДА". ■

4.9. Конечные преобразователи

Конечный автомат является простой формальной моделью устройства, распознающего автоматные (регулярные) языки, т. е. с помощью конечного автомата можно легко определить, принадлежит или нет заданная входная цепочка языку, допускаемому этим автоматом. Использование такой модели для решения реальных задач не эффективно, т. к. очень часто полученный результат (принадлежит или не принадлежит анализируемая цепочка языку) не может удовлетворить пользователя. Например, использование конечного автомата при реализации языкового процессора или его блока не представляется возможным, т. к. языковый процессор (и каждый его блок) выполняет *перевод* исходной программы (входной цепочки) в объектную программу и, следовательно, реализует некоторое отношение на множестве входных и выходных цепочек языкового процессора.

Устройство, построенное на основе конечного автомата и выполняющее перевод входных цепочек в выходные, получило название *конечного преобразователя*. Модель конечного преобразователя приведена на рис. 4.6. Конечный преобразователь на каждом такте выдает на *выходную ленту* цепочку конечной длины (возможно, пустую).

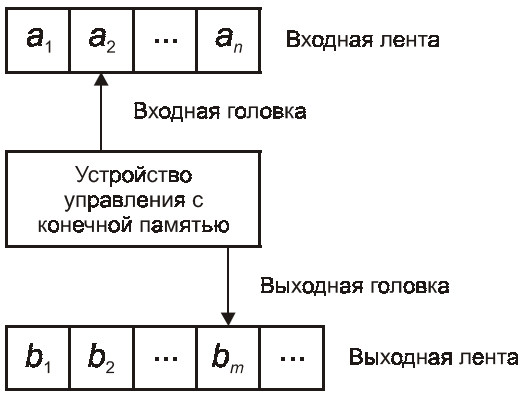


Рис. 4.6. Модель конечного преобразователя

Недетерминированным конечным преобразователем называется шестерка объектов $M = (Q, \Sigma, \Delta, \delta, q_0, F)$, где Q — конечное *множество состояний управляющего устройства*, Σ — конечный *алфавит входных символов*, Δ — конечный *алфавит выходных символов*, δ — *функция переходов и выходов*, которая задается отображением $\delta: Q \times \Sigma \to P(Q) \times \Delta$, $q_0 \in Q$ — *начальное состояние* устройства управления, $F \subseteq Q$ — *множество заключительных состояний*.

Конфигурация конечного преобразователя — это тройка $(q, x, y) \in Q \times \Sigma \times \Delta$, где $q \in Q$ — текущее состояние конечного преобразователя, $x \in \Sigma$ — необработанная часть входной цепочки, а $y \in \Delta$ — часть выходной цепочки конечного преобразователя, построенная к текущему моменту времени.

Такт работы конечного преобразователя представляется бинарным отношением ($\vdash$), которое определяется на множестве конфигураций конечного преобразователя следующим образом: для всех $q \in Q$, $a \in \Sigma$, $x \in \Sigma$ и $y \in \Delta$, таких, что $\delta(q, a)$ содержит (r, z), $(q, ax, y) \vdash (r, x, yz)$. Отношения ($\vdash^k$), ($\vdash^+$) и ($\vdash^*$) определяются аналогично соответствующим отношениям конечного автомата.

Цепочка $y \in \Delta$ называется *выходом* для цепочки $x \in \Sigma$, если $(q_0, x, \varepsilon) \vdash^* (q, \varepsilon, y)$ для некоторого $q \in F$.

Переводом, определяемым конечным преобразователем M, называется множество пар цепочек

$$\tau(M) = \{(x, y) \mid (q_0, x, \varepsilon) \vdash (q, \varepsilon, y) \text{ для некоторого } q \in F\}.$$

Перевод, определяемый конечным преобразователем, называется *регулярным переводом* или *конечным преобразованием*.

Пример 4.7

Рассмотрим конечный преобразователь, который распознает действительные числа без знака и выдает значение 1 для чисел вида *dd...d.* , значение 2 — для чисел вида *.dd...d* и значение 3 — для чисел вида *dd...d.dd...d*, где *d* — любая цифра.

Конечный преобразователь имеет 5 состояний: q_0 — начальное состояние, q_1 — состояние, соответствующее целой части действительного числа, q_2 — состояние, соответствующее дробной части действительного числа вида *dd...d.dd...d*, q_3 — состояние, соответствующее дробной части действительного числа, начинающегося десятичной точкой, а q_4 — заключительное состояние, в котором формируется результирующая цепочка.

$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{d, .\}, \{1, 2, 3\}, \delta, q_0, \{q_4\})$, где функция переходов и выходов имеет следующий вид:

$\delta(q_0, d) = \{(q_1, \varepsilon)\}$;

$\delta(q_0, .) = \{(q_3, \varepsilon)\}$;

$\delta(q_1, d) = \{(q_1, \varepsilon)\}$;

$\delta(q_1, .) = \{(q_2, \varepsilon), \{(q_4, 1)\}$;

$\delta(q_2, d) = \{(q_2, \varepsilon), \{(q_4, 3)\}$;

$\delta(q_3, d) = \{(q_3, \varepsilon), \{(q_4, 2)\}$.

Граф переходов и выходов построенного конечного преобразователя приведен на рис. 4.7. □

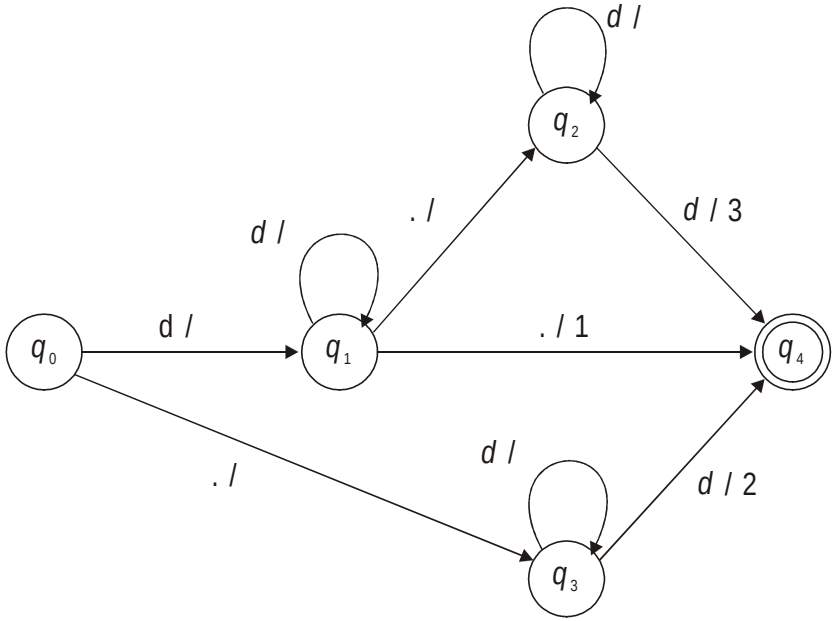


Рис. 4.7. Граф переходов и выходов конечного преобразователя из примера 4.7

Для входной цепочки 234.17 построенный конечный преобразователь выполнит следующую последовательность тактов:

$(q_0, 234.17, \varepsilon) \vdash (q_1, 34.17, \varepsilon) \vdash (q_1, 4.17, \varepsilon) \vdash (q_1, .17, \varepsilon) \vdash (q_2, 17, \varepsilon) \vdash (q_2, 7, \varepsilon) \vdash \vdash (q_4, \varepsilon, 3)$ — заключительная конфигурация.

Контрольные вопросы

1. Дайте определение распознавателя. Что представляют собой входная лента, устройство управления и вспомогательная память?
2. Какое соответствие имеется между классами грамматик из иерархии Хомского и классами распознавателей, допускающих такой же класс языков?

3. Дайте определение недетерминированного конечного автомата.
4. Дайте определение детерминированного конечного автомата.
5. Опишите алгоритм построения детерминированного конечного автомата, эквивалентного исходному недетерминированному конечному автомату.
6. Опишите алгоритм построения недетерминированного конечного автомата по автоматной грамматике.
7. Опишите алгоритм определения автоматной грамматики для недетерминированного конечного автомата.
8. Приведите решение проблемы принадлежности для конечных автоматов.
9. Приведите решение проблемы пустоты языка для конечных автоматов.
10. Приведите решение проблемы эквивалентности для конечных автоматов.
11. Дайте определение недетерминированного конечного преобразователя. Что представляют собой входная лента, устройство управления и выходная лента конечного преобразователя?
12. Дайте определение перевода, определяемого конечным преобразователем.

Упражнения

1. Построить детерминированный конечный автомат по автоматной грамматике $G = (N, \Sigma, P, S)$. Определить язык, допускаемый конечным автоматом.

   1.1. $N = \{S, A, B, C\}$, $\Sigma = \{a, b\}$, $P = \{S \to aA, S \to bB, A \to aA, A \to \varepsilon, A \to bB, B \to aC, C \to aC, C \to \varepsilon\}$.

   1.2. $N = \{S, A, B, C\}$, $\Sigma = \{a, b\}$, $P = \{S \to aA, S \to bS, A \to aA, A \to bB, B \to bS, B \to aC, C \to aC, C \to bC, C \to \varepsilon\}$.

   1.3. $N = \{S, A, B, C, D\}$, $\Sigma = \{0, 1\}$, $P = \{S \to 1A, A \to 0S, A \to 1B, B \to 0C, C \to \varepsilon, C \to 0C, C \to 1D, D \to \varepsilon, D \to 0D\}$.

   1.4. $N = \{S, A, B\}$, $\Sigma = \{0, 1\}$, $P = \{S \to \varepsilon, S \to 1S, S \to 0A, A \to 1S, A \to 0B, B \to \varepsilon, B \to 0B, B \to 1B\}$.

   1.5. $N = \{S, A\}$, $\Sigma = \{a, b\}$, $P = \{S \to aA, S \to bA, A \to aB, A \to bB, B \to aB, B \to a, B \to bB, B \to b\}$.

   1.6. $N = \{S, A, B, C\}$, $\Sigma = \{a, b\}$, $P = \{S \to aB, A \to bB, B \to aB, B \to aC, B \to bB, B \to bC, C \to \varepsilon\}$.

   1.7. $N = \{S, A, B\}$, $\Sigma = \{0, 1\}$, $P = \{S \to 0A, S \to 1B, A \to 0S, A \to 0, B \to 1S, B \to 1\}$.

1.8. $N = \{S, A, C, D\}$, $\Sigma = \{+, -, d, .\}$, $P = \{S \to +A, S \to -A, S \to d, S \to dC, A \to dC, A \to d, C \to dC, C \to d, C \to .D, D \to d, D \to dD\}$.

1.9. $N = \{S, A, B, C\}$, $\Sigma = \{a, b, c\}$, $P = \{S \to \varepsilon, S \to aA, A \to bB, A \to bS, A \to c, B \to bC, C \to bA\}$.

1.10. $N = \{S, A, B\}$, $\Sigma = \{0, 1\}$, $P = \{S \to 0A, S \to 0, S \to 1B, A \to 0C, B \to 1S, B \to 1, C \to 0A, C \to 0\}$.

2. Построить детерминированный конечный преобразователь:

2.1. Определить детерминированный конечный преобразователь, преобразующий последовательность действительных чисел без знака в формате с фиксированной точкой (число не может начинаться и заканчиваться десятичной точкой) в последовательность целых чисел, полученную из входной последовательности путем отбрасывания дробной части (разделитель между элементами последовательности — запятая, последовательность заканчивается символом '#').

2.2. Определить конечный преобразователь, осуществляющий перевод последовательности целых и вещественных чисел в формате с фиксированной точкой со знаком или без знака в цепочку символов 0 и 1 по следующему правилу: целому числу соответствует значение 0, а вещественному — значение 1. Вещественное число не должно начинаться или заканчиваться десятичной точкой. Числа разделяются запятой, признаком конца последовательности является символ '#'.

2.3. Определить конечный преобразователь, осуществляющий перевод последовательности составных имен (идентификаторов, разделенных символом '.') в последовательность цепочек 1^n, где n — число простых идентификаторов, входящих в составной идентификатор.

2.4. Определить детерминированный конечный преобразователь, преобразующий последовательность целых чисел без знака в цепочку 1^n, где n — количество четных чисел. Целые числа разделяются запятой, последовательность чисел заканчивается символом '#'.

2.5. Определить детерминированный конечный преобразователь, преобразующий последовательность целых чисел без знака в цепочку 1^n, где n — количество нечетных чисел. Целые числа разделяются запятой, последовательность чисел заканчивается символом '#'.

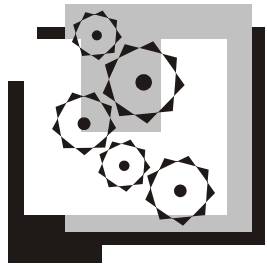
2.6. Определить детерминированный конечный преобразователь, преобразующий последовательность целых чисел без знака в цепочку 1^n, где n — количество чисел, кратных числу 5. Целые числа разделяются запятой, последовательность чисел заканчивается символом '#'.

2.7. Определить детерминированный конечный преобразователь, преобразующий последовательность целых чисел без знака в цепочку 1^n, где n — количество чисел, кратных числу 10. Целые числа разделяются запятой, последовательность чисел заканчивается символом '#'.

2.8. Определить детерминированный конечный преобразователь, преобразующий множество цепочек, в которых четное количество нулей ограничено произвольным числом единиц, во множество цепочек $\{1^n$, где n — число групп с четным количеством нулей$\}$. Цепочки разделяются запятой, последовательность цепочек заканчивается символом '#'.

2.9. Целые числа задаются последовательностью символов '/' (количество символов '/' определяет значение числа). Определить детерминированный конечный преобразователь, преобразующий последовательность таких чисел в цепочку a^n, где n — количество четных чисел. Числа разделяются запятой, последовательность чисел заканчивается точкой.

2.10. Целые числа задаются последовательностью символов '/' (количество символов '/' определяет значение числа). Определить детерминированный конечный преобразователь, преобразующий последовательность таких чисел в цепочку a^n, где n — количество нечетных чисел. Числа разделяются запятой, последовательность чисел заканчивается точкой.



Глава 5

Автоматы и преобразователи с магазинной памятью

5.1. Определение автомата с магазинной памятью

Автоматы с магазинной памятью представляют собой математическую модель синтаксических анализаторов контекстно-свободных языков. В общем случае автомат с магазинной памятью — это односторонний недетерминированный распознаватель, вспомогательная память которого организована по принципу магазина (в каждый момент времени доступен только верхний символ магазина) и имеет потенциально неограниченную длину. Будем представлять магазин в виде цепочки символов, причем верхним элементом магазина будем считать самый левый символ цепочки. Модель автомата с магазинной памятью приведена на рис. 5.1.

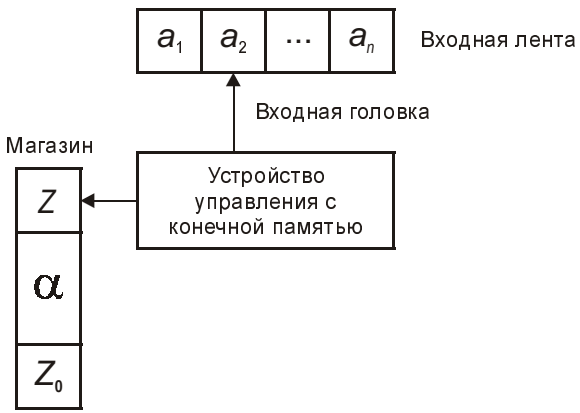


Рис. 5.1. Модель автомата с магазинной памятью

Автоматом с магазинной памятью (*МП-автоматом*) называется семерка объектов $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, где Q — конечное *множество состояний устройства управления*, Σ — конечный *алфавит входных символов*, δ — *функция переходов*, которая задает отображение множества $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ в множество конечных подмножеств множества $Q \times \Gamma^*$, $q_0 \in Q$ — *начальное состояние* устройства управления, Z_0 — *начальный символ* (*дно*) *магазина*, $F \subseteq Q$ — *множество заключительных состояний*.

Конфигурацией МП-автомата P называется тройка $(q, x, \alpha) \in Q \times \Sigma^* \times \Gamma^*$, где q — текущее состояние управляющего устройства, x — необработанная часть входной цепочки (первый символ цепочки x находится под входной головкой; если $x = \varepsilon$, то считается, что вся входная цепочка прочитана), α — содержимое магазина (самый левый символ цепочки α считается верхним символом магазина; если $\alpha = \varepsilon$, то магазин считается пустым).

Такт работы МП-автомата будем описывать бинарным отношением ($\vdash$), определенным на множестве конфигураций. Будем писать $(q, aw, Z) \vdash (q', w, \gamma\alpha)$, если

$(q', \gamma) \in \delta(q, a, Z)$, где $q, q' \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, $w \in \Sigma^*$, $Z \in \Gamma$ и $\alpha, \gamma \in \Gamma^*$.

Если $a \neq \varepsilon$ и входная цепочка прочитана не вся, то запись $(q, aw, A\alpha) \vdash (q', w, \gamma\alpha)$ означает, что МП-автомат в состоянии q, обозревая символ a входной цепочки и имея символ A в верхушке магазина, может перейти в новое состояние q', сдвинуть входную головку на один символ вправо и заменить верхний символ магазина A цепочкой магазинных символов γ. Если $\gamma = \varepsilon$, то верхний символ удаляется из магазина.

Если $a = \varepsilon$, то текущий входной символ в этом такте, называемом *ε-тактом*, не принимается во внимание и входная головка остается неподвижной. При этом состояние устройства управления и содержимое магазина могут измениться.

ε-такты могут выполняться и в том случае, когда вся входная цепочка прочитана, но если магазин пуст, то очередной такт работы МП-автомата невозможен по определению.

Так же, как и для конечных автоматов, можно определить транзитивное ($\vdash^+$) и рефлексивно-транзитивное ($\vdash^*$) замыкания отношения ($\vdash$).

Начальной конфигурацией МП-автомата называется конфигурация вида (q_0, w, Z_0), где устройство управления находится в начальном состоянии, на входной ленте записана цепочка $w \in \Sigma^*$, которую необходимо распознать, а магазин содержит только начальный символ Z_0.

Заключительной конфигурацией МП-автомата называется конфигурация вида (q, ε, α), где $q \in F$ — одно из заключительных состояний устройства управления, входная цепочка прочитана до конца, а в магазине записана некоторая, заранее определенная цепочка $\alpha \in \Gamma^*$.

Говорят, что цепочка $w \in \Sigma^*$ *допускается* МП-автоматом $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, если $(q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha)$ для некоторых $q \in F$ и $\alpha \in \Gamma^*$. Языком, *определяемым* (*допускаемым*, *распознаваемым*) МП-автоматом P, называется множество входных цепочек, допускаемых этим автоматом, т. е.

$$L(P) = \{w \mid w \in \Sigma^* \text{ и } (q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha) \text{ для некоторых } q \in F \text{ и } \alpha \in \Gamma^*\}.$$

Пример 5.1

🗁 Определим МП-автомат P, допускающий язык $L = \{a^n b^n \mid n \geq 0\}$.

МП-автомат имеет следующий вид:

$$P = (\{q_0, q_1, q_2\}, \{a, b\}, \{Z, a\}, \delta, q_0, Z, \{q_2\}),$$

в котором функция переходов δ определяется следующим образом:

1) $\delta(q_0, a, Z) = \{(q_1, aZ)\}$;
2) $\delta(q_1, a, a) = \{(q_1, aa)\}$;
3) $\delta(q_1, b, a) = \{(q_2, \varepsilon)\}$;
4) $\delta(q_2, b, a) = \{(q_2, \varepsilon)\}$;
5) $\delta(q_2, \varepsilon, Z) = \{(q_0, \varepsilon)\}$.

Работа этого МП-автомата заключается в том, что он вначале копирует префикс входной цепочки, состоящей из символов a, затем удаляет из магазина по одному символу a на каждый символ b, который читается с входной ленты. Например, для входной цепочки $aaabbb$ МП-автомат выполнит следующую последовательность тактов:

$$\begin{aligned}
&(q_0, aaabbb, Z) \vdash_1 (q_1, aabbb, aZ) \vdash_2\\
&(q_1, abbb, aaZ) \vdash_2 (q_1, bbb, aaaZ) \vdash_3\\
&(q_2, bb, aaZ) \vdash_4 (q_2, b, aZ) \vdash_4\\
&(q_2, \varepsilon, Z) \vdash_5 (q_0, \varepsilon, \varepsilon).
\end{aligned}$$

🗀

Пример 5.2

🗁 Определим МП-автомат P, допускающий язык $L = \{ww^R \mid w \in \{a, b\}^+\}$, где w^R —цепочка, реверсивная цепочке w.

МП-автомат имеет следующий вид:

$P = (\{q_0, q_1, q_2\}, \{a, b\}, \{Z, a, b\}, \delta, q_0, Z, \{q_2\})$, где

1) $\delta(q_0, a, Z) = \{(q_0, aZ)\}$;
2) $\delta(q_0, b, Z) = \{(q_0, bZ)\}$;
3) $\delta(q_0, a, a) = \{(q_0, aa), (q_1, \varepsilon)\}$;
4) $\delta(q_0, a, b) = \{(q_0, ab)\}$;
5) $\delta(q_0, b, a) = \{(q_0, ba)\}$;
6) $\delta(q_0, b, b) = \{(q_0, bb), (q_1, \varepsilon)\}$;
7) $\delta(q_1, a, a) = \{(q_1, \varepsilon)\}$;
8) $\delta(q_1, b, b) = \{(q_1, \varepsilon)\}$;
9) $\delta(q_1, \varepsilon, Z) = \{(q_2, \varepsilon)\}$.

МП-автомат сначала копирует в магазине некоторую часть входной цепочки в соответствии с функциями переходов (1), (2), (4) и (5) и в зависимости от первых элементов множеств (3) и (6). Так как P — недетерминированный МП-автомат, то когда верхний символ магазина совпадает с текущим входным символом, он может в любой момент предположить, что в магазине записана цепочка w, перейти в состояние q_1, используя вторые элементы множеств (3) и (6), и начать сравнивать цепочку в магазине с оставшейся частью входной цепочки (функции переходов (7) и (8)). Если МП-автомат обнаруживает несовпадение очередных символов, он осуществляет возврат в точку, где применялись функции перехода из альтернативного множества значений, и процесс анализа возобновляется. Если какой-либо выбор тактов приведет к тому, что символ Z окажется верхним и единственным символом магазина, то в соответствии с функцией перехода (9) этот символ удаляется из магазина и автомат переходит в заключительную конфигурацию (q_2, ε).

Таким образом, для любой конфигурации вида $(q_0, aw, a\alpha)$ МП-автомат P может сделать один из двух тактов: либо поместить в магазин еще один символ a, либо удалить из магазина верхний символ. Например, для входной цепочки $abba$ МП-автомат P может среди прочих выполнить следующие последовательности тактов:

(1) $(q_0, abba, Z) \vdash (q_0, bba, aZ) \vdash$
$(q_0, ba, baZ) \vdash (q_0, a, bbaZ) \vdash$
$(q_0, \varepsilon, abbaZ) \vdash \vdash$

(2) $(q_0, abba, Z) \vdash (q_0, bba, aZ) \vdash$
$(q_0, ba, baZ) \vdash (q_1, a, aZ) \vdash$
$(q_1, \varepsilon, Z) \vdash (q_2, \varepsilon, \varepsilon).$

Поскольку последовательность (2) заканчивается заключительной конфигурацией $(q_2, \varepsilon, \varepsilon)$, то МП-автомат P допускает входную цепочку $abba$. □

5.2. Расширенные МП-автоматы

Для начала докажем следующее вспомогательное утверждение: "То, что происходит с верхним символом магазина, не зависит от того, что находится в магазине под этим символом".

Утверждение 1

Пусть $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — МП-автомат. Если $(q, w, A) \vdash^n (q', \varepsilon, \varepsilon)$, то $(q, w, A\alpha) \vdash^n (q', \varepsilon, \alpha)$ для всех $A \in \Gamma$ и $\alpha \in \Gamma^*$.

Доказательство

□ Выполним доказательство с помощью индукции по n. Для $n = 1$ утверждение верно. Допустим, что оно верно для всех $1 \le n < n'$. Пусть $(q, w, A) \vdash^{n'} (q', \varepsilon, \varepsilon)$. Тогда имеет место следующая последовательность тактов:

$$\begin{aligned}(q, w, A) &\vdash (q_1, w_1, X_1 \dots X_k)\\ &\vdash^{n_1} (q_2, w_2, X_2 \dots X_k)\\ &\dots\\ &\vdash^{n_{k-1}} (q_k, w_k, X_k)\\ &\vdash^{n_k} (q', \varepsilon, \varepsilon),\end{aligned}$$

где $k \ge 1$ и $n_i < n'$ для $1 \le i \le k$.

Рассмотрим, как МП-автомат выполняет приведенную ранее последовательность тактов. Из конфигурации $(q_1, w_1, X_1 \dots X_k)$ МП-автомат сделает ряд тактов, в результате которых длина магазинной цепочки впервые станет равной $k - 1$. Так как ни один из символов цепочки $X_2 \dots X_k$ не был верхним символом магазина, они так и останутся в нем, при этом n_1 — число выполненных к этому моменту тактов. Затем, к некоторому моменту времени, МП-автомат выполнит еще n_2 тактов и длина магазина впервые станет равной $k - 2$. МП-автомат будет продолжать работу до тех пор, пока магазин не станет пустым.

Тогда для любых $\alpha \in \Gamma^*$ также возможна последовательность тактов:

$$\begin{aligned}(q, w, A\alpha) &\vdash (q_1, w_1, X_1 \dots X_k\alpha)\\ &\vdash^{n_1} (q_2, w_2, X_2 \dots X_k\alpha)\\ &\dots\\ &\vdash^{n_{k-1}} (q_k, w_k, X_k\alpha)\\ &\vdash^{n_k} (q', \varepsilon, \alpha).\end{aligned}$$

■

Расширим определение МП-автомата, разрешив ему за один такт работы заменять цепочку символов конечной длины, расположенную в верхушке магазина, другой цепочкой конечной длины.

Расширенным автоматом с магазинной памятью называется семерка объектов $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, где δ — *функция переходов*, которая задает отображение множества $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^*$ в множество конечных подмножеств множества $Q \times \Gamma^*$, а все остальные объекты такие же, как и у МП-автомата.

Для расширенных МП-автоматов будем считать, что верхним элементом магазина является самый правый символ цепочки.

Конфигурация расширенного МП-автомата определяется так же, как для МП-автомата, и мы можем записать $(q, aw, \gamma\alpha) \vdash (q', w, \gamma\beta)$, если $(q', \beta) \in \delta(q, a, \alpha)$, где $q, q' \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, $w \in \Sigma^*$ и $\alpha, \beta, \gamma \in \Gamma^*$. Это означает, что если $a \neq \varepsilon$, то расширенный МП-автомат при выполнении такта сдвигает входную головку на один символ вправо и заменяет цепочку α, находящуюся в верхушке магазина, на цепочку β.

Замечание

Расширенный МП-автомат может выполнять такты и тогда, когда магазин пуст.

Пример 5.3

Определим расширенный МП-автомат P, распознающий язык $L = \{ww^R \mid w \in \{a, b\}^+\}$, где w^R — цепочка, реверсивная цепочке w.

$P = (\{q_0, q_1\}, \{a, b\}, \{S, Z, a, b\}, \delta, q_0, Z, \{q_1\})$, где

1) $\delta(q_0, a, \varepsilon) = \{(q_0, a)\}$;
2) $\delta(q_0, b, \varepsilon) = \{(q_0, b)\}$;
3) $\delta(q_0, \varepsilon, \varepsilon) = \{(q_0, S)\}$;
4) $\delta(q_0, \varepsilon, aSa) = \{(q_0, S)\}$;
5) $\delta(q_0, \varepsilon, bSb) = \{(q_0, S)\}$;
6) $\delta(q_0, \varepsilon, SZ) = \{(q_1, \varepsilon)\}$.

Для входной цепочки *aabbaa* расширенный МП-автомат P может выполнить, например, следующую последовательность тактов:

$$
\begin{aligned}
&(q_0, aabbaa, Z) \vdash (q_0, abbaa, Za) \vdash\\
&(q_0, bbaa, Zaa) \vdash (q_0, baa, Zaab) \vdash\\
&(q_0, baa, ZaabS) \vdash (q_0, aa, ZaabSb) \vdash\\
&(q_0, aa, ZaaS) \vdash (q_0, a, ZaaSa) \vdash\\
&(q_0, a, ZaS) \vdash (q_0, \varepsilon, ZaSa) \vdash\\
&(q_0, \varepsilon, ZS) \vdash (q_1, \varepsilon, \varepsilon).
\end{aligned}
$$

Работа расширенного МП-автомата P состоит в том, что он сначала записывает в магазин префикс входной цепочки. Дойдя до ее предполагаемой середины, автомат записывает в магазин символ S — маркер середины цепочки. Далее он помещает в магазин очередной входной символ и заменяет в магазине подцепочку *aSa* или *bSb* на *S*. Описанный процесс продолжается до тех пор, пока не будет достигнут конец входной цепочки. Если при этом в магазине находится цепочка ZS (верхний символ магазина справа), то автомат находится в заключительной конфигурации и $abba \in L(P)$.

Покажем, что язык L определяется МП-автоматом тогда и только тогда, когда он определяется расширенным МП-автоматом. Необходимость этого условия очевидна; докажем достаточность.

Утверждение 2

Пусть $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — расширенный МП-автомат. Тогда существует такой МП-автомат P_1, что $L(P_1) = L(P)$.

Доказательство

□ Положим $m = max\{|\alpha| \mid \delta(q, a, \alpha) \neq \varnothing\}$ для некоторых $q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$. Определим МП-автомат P_1, который будет моделировать работу МП-автомата P.

Выделим в конечной памяти устройства управления автомата P_1 часть памяти длины m (буфер), в которой будут храниться m верхних символов магазина расширенного МП-автомата P. Если в некотором такте расширенный МП-автомат P заменяет k верхних символов магазина цепочкой из l символов, то эта цепочка заменит k первых символов в буфере МП-автомата P_1. Если $l < k$, то P_1 выполнит $(k - l)$ вспомогательных ε-тактов, во время которых $(k - l)$ верхних символов магазина перепишутся в буфер МП-автомата P_1. Если $l > k$, то $(l - k)$ символов перепишутся из буфера в магазин.

Определим формально МП-автомат P_1 как:

$P_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, q_1, Z_1, F_1)$, где

1. $Q_1 = \{[q, \alpha] \mid q \in Q, \alpha \in \Gamma_1^*, 0 \leq |\alpha| \leq m\}$;
2. $\Gamma_1 = \Gamma \cup \{Z_1\}$;
3. δ_1 определяется следующим образом:
   - 3.1. Допустим, что $(r, Y_1 \ldots Y_l) \in \delta(q, a, X_1 \ldots X_k)$. Тогда:
     - если $l \geq k$, то для всех $Z \in \Gamma_1$ и $\alpha \in \Gamma_1^*$, $|\alpha| = m - k$ $\delta_1([q, X_1 \ldots X_k\alpha], a, Z)$ содержит $([r, \beta], \gamma Z)$, где $\beta\gamma = Y_1 \ldots Y_l\alpha$ и $|\beta| = m$;
     - если $l < k$, то для всех $Z \in \Gamma_1$, $\alpha \in \Gamma_1^*$, $|\alpha| = m - k$, $\delta_1([q, X_1 \ldots X_k\alpha], a, Z)$ содержит $([r, Y_1 \ldots Y_l aZ], \varepsilon)$.
   - 3.2. Для всех $q \in Q$, $Z \in \Gamma_1$, $\alpha \in \Gamma_1^*$, $|\alpha| < m$, $\delta_1([q, a], \varepsilon, Z) = \{([q, aZ], \varepsilon)\}$.
4. $q_{01} = [q_0, Z_0 Z_1^{m-1}]$. В начальном состоянии МП-автомата P_1 в буфере записана цепочка $Z_0 Z_1^{m-1}$, при этом символы Z_1 используются как специальные маркеры, отмечающие дно магазина.
5. $F_1 = \{[q, \alpha] \mid q \in F, \alpha \in \Gamma_1^*\}$

Непосредственно применяя правила, определяющие МП-автомат P_1, можно показать, что для расширенного МП-автомата P имеет место отношение:

$$(q, aw, X_1 \ldots X_k X_{k+1} \ldots X_n) \vdash (r, w, Y_1 \ldots Y_l X_{k+1} \ldots X_n)$$

тогда и только тогда, когда для МП-автомата P_1 выполняется отношение

$$([q, \alpha], aw, \beta) \vdash^+ ([r, \alpha''], w, \beta'),$$

где:

1. $\alpha\beta = X_1 \ldots X_n Z_1^m$;
2. $\alpha'\beta' = Y_1 \ldots Y_l X_{k+1} \ldots X_n Z_1^m$;
3. $|\alpha| = |\beta| = m$;
4. Между двумя указанными конфигурациями МП-автомата P_1 нет ни одной конфигурации, состояние которой имело бы вторую компоненту (буфер) длиной m.

Таким образом, для расширенного МП-автомата P $(q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha)$ для некоторых $q \in F$ и $\alpha \in \Gamma_1^*$ тогда и только тогда, когда для МП-автомата P_1 справедливо:

$$([q_0, Z_0 Z_1^{m-1}], w, Z_1) \vdash^* (([q, \beta], \varepsilon, \gamma),$$

где $|\beta| = m$ и $\beta\gamma = \alpha Z_1^m$. Отсюда $L(P_1) = L(P)$. ■

В общем случае МП-автомат или расширенный МП-автомат могут закончить работу, если в магазине записана некоторая, заранее заданная, цепочка. Для практического использования более удобно иметь автоматы, которые завершают свою работу при условии, что в магазине пусто.

Пусть $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — МП-автомат или расширенный МП-автомат. Автомат P *допускает цепочку* $w \in \Sigma^*$ *опустошением* магазина, если $(q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)$ для некоторого $q \in F$.

Множество цепочек, допускаемых автоматом опустошением магазина, будем обозначать $L_\varepsilon(P)$. Докажем следующее утверждение.

Утверждение 3

Если $L = L(P)$ для некоторого автомата $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, то можно построить такой автомат с магазинной памятью P', что $L_\varepsilon(P) = L$.

Доказательство

□ Определим автомат P' так, чтобы он моделировал работу автомата P, причем каждый раз, когда P переходит в заключительное состояние, P' решает, продолжить моделирование работы автомата P или перейти в специальное состояние q_ε, которое опустошит магазин. При этом возможно, что для некоторой цепочки w автомат P выполнит последовательность тактов, приводящую к опустошению магазина, а его устройство управления при этом окажется не в заключительном состоянии. Чтобы автомат P' в этом случае не мог допустить цепочку, в магазин записывается специальный

символ (маркер), отмечающий дно магазина, который автомат P' может удалить только в состоянии q_ε.

Построим автомат $P' = (Q', \Sigma, \Gamma', \delta', q_0', Z_0', F')$, такой, что:

1. $Q' = Q \cup \{q_\varepsilon, q_0'\}$, где q_ε — состояние, опустошающее магазин, а q_0' — начальное состояние автомата P'.
2. $\Gamma' = \Gamma \cup \{Z_0'\}$, где Z_0' — маркер дна магазина.
3. $F' = \{q_\varepsilon\}$.
4. δ' определяется следующим образом:
   - 4.1. На первом такте автомат P' записывает в магазин символ Z_0 и переходит в начальное состояние q_0 автомата P, т. е. $\delta'(q_0', \varepsilon, Z_0') = \{(q_0, Z_0Z_0')\}$.
   - 4.2. Если $(r, \gamma) \in \delta(q, a, Z)$, то $\delta'(q, a, Z)$ содержит (r, γ) для всех $a \in \Sigma \cup \{\varepsilon\}$, $q \in Q$, $Z \in \Gamma$.
   - 4.3. Для всех $q \in Q$, $Z \in \Gamma \cup \{Z_0'\}$ множество $\delta'(q, \varepsilon, Z)$ содержит пару $(q_\varepsilon, \varepsilon)$.
   - 4.4. Для всех $Z \in \Gamma \cup \{Z_0'\}$ $\delta'(q, \varepsilon, Z) = \{(q_\varepsilon, \varepsilon)\}$.

МП-автомат P' выполнит для цепочки w следующую последовательность тактов:

$$\begin{aligned}(q_0', w, Z_0') &\vdash (q_0, w, Z_0Z_0')\\ &\vdash^{n} (q, \varepsilon, Y_1Y_2 \ldots Y_mZ_0')\\ &\vdash (q_\varepsilon, \varepsilon, Y_2 \ldots Y_mZ_0')\\ &\vdash^{r-1} (q_\varepsilon, \varepsilon, \varepsilon), \text{ где } Y_r = Z_0'\end{aligned}$$

тогда и только тогда, когда для автомата P выполняется отношение:

$$(q_0, w, Z_0) \vdash^{n} (q, \varepsilon, Y_1 \ldots Y_r) \text{ для } q \in F \text{ и } Y_1 \ldots Y_{r-1} \in \Gamma^*.$$

Следовательно, $L_\varepsilon(P) = L$. ■

Справедливо и обратное утверждение, которое мы приведем без доказательства.

Утверждение 4

Если $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — МП-автомат или расширенный МП-автомат, допускающий цепочки опустошением магазина, то можно построить такой автомат P', что $L(P') = L_\varepsilon(P)$.

5.3. Эквивалентность МП-автоматов и КС-грамматик

Покажем, что контекстно-свободный язык допускается односторонним недетерминированным автоматом с магазинной памятью [3, 37].

Лемма 5.1

Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. По грамматике G можно построить такой МП-автомат P, что $L(P) = L(G)$.

Пусть $P = (\{q\}, \Sigma, \Sigma \cup N, \delta, q, S, \{q\})$, где δ определяется следующим образом:

1. Если $A \to \alpha$ — правило вывода грамматики G, то $\delta(q, \varepsilon, A)$ содержит (q, α).
2. $\delta(q, a, a) = \{(q, \varepsilon)\}$ для всех $a \in \Sigma$.

Докажем, что $A \Rightarrow^m w$ тогда и только тогда, когда $(q, w, A) \vdash^n (q, \varepsilon, \varepsilon)$ для некоторых $m, n \geq 1$.

Доказательство

Необходимость: □ Необходимость условия доказывается индукцией по m.

Допустим, что $A \Rightarrow^m w$. Если $m = 1$ и $w = a_1\ a_2\ \ldots\ a_k$ $(k \geq 0)$, то

$$(q, a_1\ a_2\ \ldots\ a_k, A) \vdash (q, a_1\ a_2\ \ldots\ a_k, a_1\ a_2\ \ldots\ a_k) \vdash^k (q, \varepsilon, \varepsilon).$$

Предположим, что $A \Rightarrow^m w$ для некоторого $m > 1$. Пусть $w = x_1\ x_2\ \ldots\ x_k$. Тогда первый шаг соответствующего вывода в грамматике G должен иметь вид $A \Rightarrow X_1\ X_2\ \ldots\ X_k$, где $X_i \Rightarrow^{m_i} x_i$ для некоторого $m_i < m$ $(1 \leq i \leq k)$. Если $X_i \in N$, то по предположению индукции $(q, x_i, X_i) \vdash^* (q, \varepsilon, \varepsilon)$. Если $X_i = x_i$ и $x_i \in \Sigma$, то $(q, x_i, X_i) \vdash (q, \varepsilon, \varepsilon)$. Объединив последовательности тактов для k символов цепочки w, получим $(q, w, A) \vdash^+ (q, \varepsilon, \varepsilon)$. ■

Достаточность: □ Докажем индукцией по n, что если $(q, w, A) \vdash^n (q, \varepsilon, \varepsilon)$, то $A \Rightarrow^+ w$.

Если $n = 1$, то $w = \varepsilon$ и $A \to \varepsilon$ — правило вывода грамматики G. Предположим, что утверждение верно для всех $n' < n$. Тогда первый такт, выполненный МП-автоматом P для цепочки $w = x_1\ x_2\ \ldots\ x_k$, должен иметь вид $(q, w, A) \vdash (q, w, X_1\ X_2\ \ldots\ X_k)$ и, в соответствии с утверждением 1, $(q, x_i, X_i) \vdash^{n_i} (q, \varepsilon, \varepsilon)$ для $1 \leq i \leq k$ и $w = x_1\ x_2\ \ldots\ x_k$. Следовательно, $A \to X_1\ X_2\ \ldots\ X_k$ — правило вывода грамматики G, и по предположению индукции $X_i \Rightarrow^+ x_i$ для $X_i \in N$. Если $X_i \in \Sigma$, то $X_i \Rightarrow^0 x_i$. Таким образом,

$$\begin{aligned} A &\Rightarrow X_1\ X_2\ \ldots\ X_k \\ &\Rightarrow^* x_1\ X_2\ \ldots\ X_k \\ &\ldots \end{aligned}$$

$\Rightarrow^*$ $x_1\ x_2\ \ldots\ x_{k-1}\ X_k$

$\Rightarrow^*$ $x_1\ x_2\ \ldots\ x_{k-1}\ x_k = w$ — вывод цепочки w в грамматике G. ■

Из доказанного в данной лемме утверждения, в частности, следует, что $S \Rightarrow^+ w$ тогда и только тогда, когда $(q, w, S) \vdash^+ (q, \varepsilon, \varepsilon)$. Следовательно, $L_\varepsilon(P) = L(G)$.

Пример 5.4

🗁 Определим МП-автомат, распознающий скобочные арифметические выражения, построенные из идентификаторов i и знаков операций сложения и умножения. Приоритет операций обычный.

КС-грамматика G, определяющая данные арифметические выражения, имеет следующий вид: $G = (\{E, T, M\}, \{i, +, *, (,)\}, \{E \to E + T, \quad E \to T, T \to T * M, T \to M, M \to (E), M \to i\}, E)$.

Используя лемму 5.1, получим:

$P = (\{q\}, \{i, +, *, (,)\}, \{i, +, *, (,), E, T, M\}, \delta, q, E, \{q\})$,

в котором функция переходов δ определяется следующим образом:

1) $\delta(q, \varepsilon, E) = \{(q, E + T), \{(q, T)\}$;
2) $\delta(q, \varepsilon, T) = \{(q, T * M), \{(q, M)\}$;
3) $\delta(q, \varepsilon, M) = \{(q, (E)), \{(q, i)\}$;
4) $\delta(q, a, a) = \{(q, \varepsilon)\}$ для всех $a \in \{i, +, *, (,)\}$.

При анализе входной цепочки $i * (i + i)$ МП-автомат P может выполнить следующую последовательность тактов:

$$
\begin{aligned}
(q, i * (i + i), E) &\vdash (q, i * (i + i), T) \\
&\vdash (q, i * (i + i), T * M) \\
&\vdash (q, i * (i + i), i * M) \\
&\vdash (q, * (i + i), * M) \\
&\vdash (q, (i + i), M) \\
&\vdash (q, (i + i), (E)) \\
&\vdash (q, i + i), E)) \\
&\vdash (q, i + i), E + T)) \\
&\vdash (q, i + i), T + T))
\end{aligned}
$$

$\vdash \quad (q, i + i), M + T))$

$\vdash \quad (q, i + i), i + T))$

$\vdash \quad (q, + i), + T))$

$\vdash \quad (q, i), T))$

$\vdash \quad (q, i), M))$

$\vdash \quad (q, i), I))$

$\vdash \quad (q,),))$

$\vdash \quad (q, \varepsilon, \varepsilon).$

□

Последовательность тактов, которую выполняет МП-автомат P, соответствует левому выводу цепочки $i * (i + i)$ в грамматике G:

$$E \Rightarrow \quad T \Rightarrow \quad T * M \Rightarrow \quad M * M \Rightarrow \quad i * M \Rightarrow \quad i * (E) \Rightarrow \quad i * (E + T) \Rightarrow$$
$$i * (T + T) \Rightarrow i * (M + T) \Rightarrow i * (i + T) \Rightarrow i * (i + M) \Rightarrow i * (i + i).$$

Тип синтаксического анализатора для цепочек, порождаемых грамматикой G, который можно построить на основе МП-автомата, полученного в соответствии с леммой 5.1, принято называть *нисходящим* (*предсказывающим*) *анализатором*, т. к. дерево вывода в этом случае строится сверху вниз, а каждый такт, в котором происходит замена верхнего символа магазина цепочкой символов, являющихся правой частью правила вывода грамматики, можно интерпретировать как *предсказание* очередного шага левого вывода.

Можно построить расширенный МП-автомат, который будет строить дерево вывода снизу вверх, моделируя в обратном порядке правые выводы в соответствующей КС-грамматике. Синтаксические анализаторы, построенные на основе расширенных МП-автоматов, называют *восходящими анализаторами*.

Рассмотрим правый вывод цепочки $i + i * i$, порождаемый грамматикой арифметических выражений из примера 5.4:

$$E \Rightarrow \quad E + T \Rightarrow \quad E + T * M \Rightarrow \quad E + T * i \Rightarrow \quad E + M * i \Rightarrow \quad E + i * i \Rightarrow$$
$$T + i * i \Rightarrow M + i * i \Rightarrow i + i * i.$$

Запишем обращение этого вывода:

$$i + i * i \Leftarrow \quad M + i * i \Leftarrow \quad T + i * i \Leftarrow \quad E + i * i \Leftarrow \quad E + M * i \Leftarrow \quad E + T * i \Leftarrow$$
$$E + T * M \Leftarrow E + T \Leftarrow E.$$

Если считать, что переход от сентенциальной формы $E + T * M$ к сентенциальной форме $E + T$ происходит с использованием в обратном направлении правила вывода $T \rightarrow T * M$, то можно сказать, что цепочка $E + T * M$ *свертывается слева* к цепочке $E + T$, причем это единственно

возможная левая свертка. Аналогичным образом можно правовыводимую цепочку $i + i * i$ свернуть слева к цепочке $M + i * i$ с помощью правила вывода $M \to i$, цепочку $M + i * i$ к цепочке $T + i * i$ с помощью правила $T \to M$ и т. д.

Определим формально левую свертку.

Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика, и $S \Rightarrow_r^* \alpha Aw \Rightarrow_r \alpha\beta w \Rightarrow_r^* xw$ — правый вывод в ней. Будем говорить, что правовыводимую цепочку $\alpha\beta w$ можно свернуть слева к правовыводимой цепочке αAw с помощью правила вывода $A \to \beta$. Это вхождение цепочки β в цепочку $\alpha\beta w$ называется *основой цепочки* $\alpha\beta w$.

Очевидно, что основа цепочки — это ее любая подцепочка, которая является правой частью некоторого правила вывода, причем после замены ее левой частью этого правила также получится правовыводимая цепочка.

Пример 5.5

Пусть задана КС-грамматика с правилами вывода $\{S \to Ac,\ S \to Bd,\ A \to aAb,\ A \to ab,\ B \to aBbb,\ B \to abb\}$, порождающая язык $\{a^n b^n c \mid n \geq 1\} \cup \{a^n b^{2n} d \mid n \geq 1\}$.

Рассмотрим цепочку *aabbbbd*. Подцепочка *abb* является основой этой цепочки, т. к. *abb* — правая часть правила вывода $B \to abb$ и *aBbbd* — правовыводимая цепочка. Подцепочка *ab* также является правой частью правила вывода $A \to ab$, но она не является основой, так как цепочка *aAbbbd* не является правовыводимой.

Если грамматика однозначна, то порождаемые ею цепочки имеют не более одного правого вывода, а значит, не более одной основы и одного основывающего правила.

Замечание

Цепочка может не иметь основы, если эта цепочка не правовыводимая в данной грамматике.

Основу правовыводимой цепочки можно также определить в терминах деревьев вывода.

Основа — это крона самого левого поддерева глубиной 1 некоторого дерева вывода правовыводимой цепочки.

Дерево глубиной 1, состоящее из вершины и ее прямых потомков, которые являются листьями, называется *кустом*.

Дерево вывода цепочки *aabbbbd* в грамматике из примера 5.5 приведено на рис. 5.2, *а*. Самый левый куст дерева состоит из левой вершины, помеченной символом *B*, которая является его корнем, и кроны *aab*. Если удалить листья самого левого куста, то останется дерево, изображенное на рис. 5.2, *б*. Крона *aBbbd* этого дерева — результат левой свертки цепочки *aabbbbd*, а его основа — крона поддерева с корнем, обозначенным символом *B*. Снова удалив основу, получим дерево, приведенное на рис. 5.2, *в*, а затем — дерево, состоящее из одного корня *S* (рис. 5.2, *г*).

Описанный процесс свертки деревьев называется *отсечением основ*.

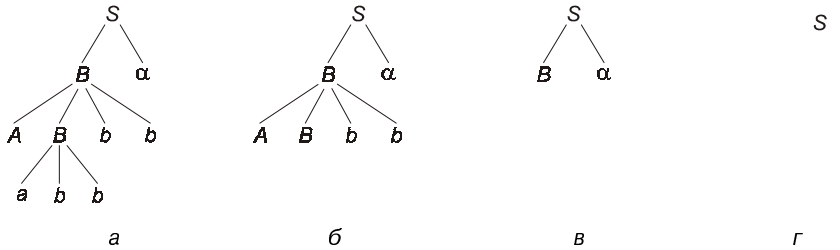


Рис. 5.2. Отсечение основ

По КС-грамматике *G* можно построить эквивалентный расширенный МП-автомат *P*, работа которого заключается в отсечении основ.

Лемма 5.2

Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. По грамматике *G* можно построить такой расширенный МП-автомат *P*, что $L(P) = L(G)$.

Пусть $P = (\{q, r\}, \Sigma, \Sigma \cup N \cup \{\perp\}, \delta, q, \perp, \{r\})$, где δ определяется как:

1. $\delta(q, a, \varepsilon) = \{(q, a)\}$ для всех $a \in \Sigma$ (символы с входной ленты переносятся в магазин).
2. Если $A \to \alpha$ — правило вывода грамматики *G*, то $\delta(q, \varepsilon, a)$ содержит (q, A).
3. $\delta(q, \varepsilon, \perp S) = \{(r, \varepsilon)\}$.

Покажем, что расширенный МП-автомат *P* строит правовыводимые цепочки грамматики *G*, начиная с терминальной цепочки, записанной на входной ленте, и заканчивая цепочкой *S*.

Доказательство

□ Индукцией по *n* докажем, что:

$$S \Rightarrow_r^* \alpha A w \Rightarrow_r^n xy \text{ влечет } (q, xy, \perp) \vdash^* (q, y, \perp\alpha A). \qquad (5.1)$$

При $n = 0$ шаг индукции тривиален: автомат P не делает ни одного такта. Предположим, что утверждение (5.1) верно для всех $n_i < n$, т. е. можно написать

$$\alpha A y \Rightarrow_r \alpha\beta y \Rightarrow_r^{n-1} xy.$$

Допустим, что цепочка $\alpha\beta$ состоит только из терминалов. Тогда $\alpha\beta = x$ и

$$(q, xy, \perp) \vdash^* (q, y, \perp\alpha\beta) \vdash (q, y, \perp\alpha A).$$

Если цепочка $\alpha\beta$ содержит нетерминалы, то можно написать $\alpha\beta = \gamma Bz$, где B — самый правый нетерминал. С учетом (5.1) справедливо, что из $S \Rightarrow_r^* \gamma Bzy \Rightarrow_r^{n-1} xy$ следует $(q, xy, \perp) \vdash^* (q, zy, \perp\gamma B)$ и $(q, zy, \perp\gamma B) \vdash^* (q, y, \perp\gamma Bz) \vdash (q, y, \perp\alpha A)$. Следовательно, утверждение (5.1) верно для всех n. Так как $(q, \varepsilon, \perp S) \vdash (r, \varepsilon, \varepsilon)$, то $L(G) \subseteq L(P)$.

Для того чтобы доказать справедливость обратного включения $L(P) \subseteq L(G)$ и, следовательно, равенство $L(G) = L(P)$, докажем, что

$$\text{из } (q, xy, \perp) \vdash^n (q, y, \perp\alpha A) \text{ следует } S \Rightarrow_r^* \alpha A y \Rightarrow_r^n xy. \qquad (5.2)$$

При $n = 0$ шаг индукции выполняется автоматически. Предположим, что утверждение (5.2) верно для всех $n < m$. Если верхний символ магазина автомата P является нетерминалом, то последний такт был выполнен автоматом по второму правилу определения функции δ, и можно написать:

$$(q, xy, \perp) \vdash^{m-1} (q, y, \perp\alpha\beta) \vdash (q, y, \perp\alpha A),$$

где $A \to \beta$ — правило грамматики G.

Если цепочка $\alpha\beta$ содержит нетерминал, то по предположению индукции $\alpha By \Rightarrow^* xy$, откуда $\alpha Ay \Rightarrow \alpha\beta y \Rightarrow^* xy$, что и утверждалось ранее.

В качестве частного случая утверждения (5.2) получаем, что из $(q, w, \perp) \vdash^* (q, \varepsilon, \perp S)$ следует $S \Rightarrow^* w$. Так как расширенный МП-автомат P допускает цепочку w тогда и только тогда, когда $(q, w, \perp) \vdash^* (q, \varepsilon, \perp S) \vdash (r, \varepsilon, \varepsilon)$, то отсюда следует, что $L(P) \subseteq L(G)$. Таким образом, $L(G) = L(P)$. ■

Заметим, что сразу после свертки правовыводимая цепочка вида αAx представлена в расширенном МП-автомате P так, что αA находится в магазине, а цепочка x представляет собой непрочитанную часть цепочки на входной ленте. Автомат P может продолжать переносить символы цепочки x в магазин до тех пор, пока в верхней части магазина не образуется основа и автомат не выполнит следующую свертку.

Пример 5.6

🗁 Определим расширенный МП-автомат с магазинной памятью, распознающий арифметические выражения из примера 5.3, следующим образом:

$$P = (\{q, r\}, \{i, +, *, (,)\}, \{i, +, *, (,), E, T, M\}, \delta, q, \perp, \{r\}),$$

в котором функция переходов δ определяется следующим образом:

$\delta(q, a, \varepsilon) = \{(q, a)\}$ для всех $a \in \Sigma$,

$\delta(q, \varepsilon, E + T) = \{(q, E)\}$,

$\delta(q, \varepsilon, T) = \{(q, E)\}$,

$\delta(q, \varepsilon, T * M) = \{(q, T)\}$,

$\delta(q, \varepsilon, M) = \{(q, T)\}$,

$\delta(q, \varepsilon, i) = \{(q, M)\}$,

$\delta(q, \varepsilon, (E)) = \{(q, M)\}$,

$\delta(q, \varepsilon, \perp E) = \{(r, \varepsilon)\}$.

При анализе входной цепочки $i + i * i$ МП-автомат P может выполнить следующую последовательность тактов:

$$
\begin{aligned}
(q, i + i * i, \perp) &\vdash (q, + i * i, \perp i) \\
&\vdash (q, + i * i, \perp M) \\
&\vdash (q, + i * i, \perp T) \\
&\vdash (q, + i * i, \perp E) \\
&\vdash (q, i * i, \perp E +) \\
&\vdash (q, * i, \perp E + i) \\
&\vdash (q, * i, \perp E + M) \\
&\vdash (q, * i, \perp E + T) \\
&\vdash (q, i, \perp E + T *) \\
&\vdash (q, \varepsilon, \perp E + T * i) \\
&\vdash (q, \varepsilon, \perp E + T * M) \\
&\vdash (q, \varepsilon, \perp E + T) \\
&\vdash (q, \varepsilon, \perp E) \\
&\vdash (q, \varepsilon, \varepsilon).
\end{aligned}
$$

□

Замечание

При анализе входной цепочки $i + i * i$ расширенный МП-автомат P может выполнить большое число различных последовательностей тактов, но только приведенная ранее — единственная последовательность, которая переводит начальную конфигурацию в заключительную.

Лемма 5.3

Пусть $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — МП-автомат. Можно построить такую КС-грамматику $G = (N, \Sigma, P, S)$, что $L(G) = L(P)$.

Доказательство

□ Построим грамматику G так, чтобы левый вывод цепочки w в ней прямо соответствовал последовательности тактов, которую выполнит МП-автомат P при обработке цепочки w.

Нетерминальные символы будем обозначать в виде $[qZr]$, где $q, r \in Q$, а $Z \in \Gamma$.

Пусть $G = (N, \Sigma, P, S)$, где $N = \{[qZr] \mid q, r \in Q, Z \in \Gamma\} \cup \{S\}$, а правила вывода определяются следующим образом:

1. Если $\delta(q, a, Z)$ содержит $(r, X_1 \ldots X_k)$, где $k \geq 1$, добавим к множеству правил грамматики G все правила вида

$$[qZs_k] \rightarrow a[rX_1s_1]\,[s_1X_2s_2] \ldots [s_{k-1}X_ks_k]$$

   для каждой последовательности состояний $s_1, s_2, \ldots s_k$ из Q.

2. Если $\delta(q, a, Z)$ содержит (r, ε), добавить к P правило $[qZr] \rightarrow a$.

3. Для каждого $q \in Q$ добавить к P правила вида $S \rightarrow [q_0Z_0q]$.

Индукцией по m и n легко доказать [3], что для любых $q, r \in Q$ и $Z \in \Gamma$ $[qZr] \Rightarrow^m w$ тогда и только тогда, когда $(q, w, Z) \vdash^n (r, \varepsilon, \varepsilon)$.

Из последнего утверждения следует, что $S \Rightarrow [q_0Z_0q] \Rightarrow^+ w$ тогда и только тогда, когда $(q_0, w, Z_0) \vdash^+ (q, \varepsilon, \varepsilon)$ для $q \in Q$. Следовательно, $L_\varepsilon(P) = L(G)$. ■

5.4. Детерминированные МП-автоматы

В *разд. 5.3* было показано, что для любой КС-грамматики G можно построить недетерминированный МП-автомат, распознающий язык $L(G)$, причем требование недетерминированности является существенным. На практике недетерминированные МП-автоматы из-за сложности реализации фактически не используются, уступив место детерминированным МП-автоматам, которые в каждой конфигурации могут сделать не более одного очередного такта.

Детерминированным автоматом с магазинной памятью (*ДМП-автоматом*) называется МП-автомат $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, у которого для каждых $q \in Q$ и $Z \in \Gamma$ выполняется одно из двух условий:

1. $\delta(q, a, Z)$ содержит не более одного элемента для каждого $a \in \Sigma$ и $\delta(q, \varepsilon, Z) = \varnothing$.
2. $\delta(q, a, Z) = \varnothing$ для всех $a \in \Sigma$, и $\delta(q, \varepsilon, Z)$ содержит не более одного элемента.

Эти ограничения позволяют выбрать в любой конфигурации не более одного такта для продолжения работы ДМП-автомата, что позволяет достаточно просто моделировать его работу в практических приложениях.

Замечание

Поскольку множество $\delta(q, a, Z)$ для ДМП-автомата содержит не более одного элемента, будем записывать $\delta(q, a, Z) = (q', \alpha)$ вместо $\delta(q, a, Z) = \{(q', \alpha)\}$.

Язык, допускаемый ДМП-автоматом, называется *детерминированным КС-языком*. Класс детерминированных КС-языков является подмножеством КС-языков, т. е. существуют КС-языки, которые нельзя определить ДМП-автоматами.

Пример 5.7

Построить ДМП-автомат, распознающий язык $L = \{wcw^R \mid w \in \{a, b\}^+\}$, где w^R — цепочка, реверсивная цепочке w.

$P = (\{q_0, q_1, q_2\}, \{a, b, c\}, \{Z, a, b\}, \delta, q_0, Z, \{q_2\})$, где

1) $\delta(q_0, a, Z) = (q_0, aZ)$;
2) $\delta(q_0, a, a) = (q_0, aa)$;
3) $\delta(q_0, a, b) = (q_0, ab)$;
4) $\delta(q_0, b, Z) = (q_0, bZ)$;
5) $\delta(q_0, b, a) = (q_0, ba)$;
6) $\delta(q_0, b, b) = (q_0, bb)$;
7) $\delta(q_0, c, a) = (q_1, a)$;
8) $\delta(q_0, c, b) = (q_1, b)$;
9) $\delta(q_1, a, a) = (q_1, \varepsilon)$;
10) $\delta(q_1, a, b) = (q_1, \varepsilon)$;
11) $\delta(q_1, b, a) = (q_1, \varepsilon)$;
12) $\delta(q_1, b, b) = (q_1, \varepsilon)$;
13) $\delta(q_1, \varepsilon, Z) = (q_2, \varepsilon)$.

Этот ДМП-автомат, находясь в состоянии q_0, читает символы входной цепочки до тех пор, пока не встретит символ c, отмечающий ее середину. Когда автомат достигнет символа c, он перейдет в состояние q_1 и начнет сравнивать считываемые символы входной цепочки с символами, находящимися в магазине. Если символы совпадают, то автомат завершает свою работу в состоянии q_2.

Для входной цепочки *aabacabaa* расширенный МП-автомат P может выполнить, например, следующую последовательность тактов:

$$
\begin{aligned}
&(q_0, aabacabaa, Z) \vdash (q_0, abacabaa, aZ) \vdash \\
&(q_0, bacabaa, aaZ) \vdash (q_0, acabaa, baaZ) \vdash \\
&(q_0, cabaa, abaaZ) \vdash (q_1, abaa, abaaZ) \vdash \\
&(q_1, baa, baaZ) \vdash (q_1, aa, aaZ) \vdash \\
&(q_1, a, aZ) \vdash (q_1, \varepsilon, Z) \vdash \\
&(q_2, \varepsilon, Z).
\end{aligned}
$$

□

Расширенный МП-автомат $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ называется *детерминированным*, при выполнении следующих условий:

- $\delta(q, a, \gamma)$ содержит не более одного элемента для всех $q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$ и $\gamma \in \Gamma^*$;
- если $\delta(q, a, \alpha) \neq \varnothing$, $\delta(q, a, \beta) \neq \varnothing$ и $\alpha \neq \beta$, то ни одна из цепочек α и β не является суффиксом другой;
- если $\delta(q, a, \alpha) \neq \varnothing$, $\delta(q, \varepsilon, \beta) \neq \varnothing$, то ни одна из цепочек α и β не является суффиксом другой.

Пример 5.8

Расширенный МП-автомат P из примера 5.6 является недетерминированным. Первое условие для этого автомата выполняется, т. к. функция перехода содержит по одному элементу для всех терминальных символов. Третье условие также выполняется. Рассмотрим второе условие. Выбрав пару непустых функций $\delta(q, \varepsilon, E + T) = \{(q, E)\}$ и $\delta(q, \varepsilon, T) = \{(q, E)\}$, видим, что цепочка T является суффиксом цепочки $E + T$, и, следовательно, расширенный МП-автомат P является недетерминированным.

В *разд. 5.2* было доказано, что если P — расширенный МП-автомат, то можно построить такой МП-автомат P', что $L(P') = L(P)$. МП-автомат P' будет детерминированным тогда и только тогда, когда исходный расширенный МП-автомат P был детерминированным.

Из определения МП-автомата P (обычного или расширенного) следует, что автомат может завершить свою работу по анализу входной цепочки $w \notin L(P)$, не дочитав ее до конца. При построении синтаксического анализатора желательно иметь ДМП-автомат, который считывает всю входную цепочку, даже если она не принадлежит языку, допускаемому автоматом. Известно [3, 37], что построение такого автомата возможно.

Модифицируем сначала ДМП-автомат таким образом, чтобы для любой конфигурации, в которой часть входной цепочки осталась непрочитанной, был *всегда возможен очередной такт*.

Лемма 5.4

Пусть $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — ДМП-автомат. Можно построить такой эквивалентный ДМП-автомат $P' = (Q', \Sigma, \Gamma', \delta', q_0', Z_0', F')$, что:

1. Для всех $a \in \Sigma$, $q \in Q'$ и $Z \in \Gamma'$ выполняется одно из двух условий:
   - $\delta'(q, a, Z)$ содержит только один элемент и $\delta'(q, \varepsilon, Z) = \varnothing$;
   - $\delta'(q, a, Z) = \varnothing$ и $\delta'(q, \varepsilon, Z)$ содержит только один элемент.

Выполнение любого из этих условий говорит о том, что МП-автомат является детерминированным и всегда выполняет очередной такт.

2. Если $\delta'(q, a, Z_0') = (r, \gamma)$ для некоторого $a \in \Sigma \cup \{\varepsilon\}$, то $\gamma = \alpha Z_0'$ для некоторой цепочки $\alpha \in \Gamma^*$.

   Выполнение этого условия позволит не стирать символ Z_0', играющий роль маркера дна магазина, предотвращая тем самым полное опустошение магазина.

Доказательство

□ Пусть $\Gamma' = \Gamma \cup \{Z_0'\}$ и $Q' = \{q_0', q_\varepsilon\} \cup Q$, а δ' определяется следующими правилами:

1. $\delta'(q_0', \varepsilon, Z_0') = (q_0, Z_0 Z_0')$;
2. Для всех $q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$ и $Z \in \Gamma$, таких, что $\delta(q, a, Z) \neq \varnothing$, положить $\delta'(q, a, Z) = \delta(q, a, Z)$;
3. Если $\delta(q, \varepsilon, Z) = \varnothing$ и $\delta(q, a, Z) = \varnothing$ для некоторых $a \in \Sigma$ и $Z \in \Gamma$, положить $\delta'(q, a, Z) = (q_\varepsilon, Z)$;
4. Для всех $Z \in \Gamma'$ и $a \in \Sigma$ положить $\delta'(q_\varepsilon, a, Z) = (q_\varepsilon, Z)$.

■

Этот ДМП-автомат работает следующим образом:

- в начальном состоянии q_0' записывает в магазин символы $Z_0 Z_0'$ и переходит в состояние q_0;
- начиная с состояния q_0, автомат P' моделирует работу автомата P до тех пор, пока очередной такт не становится невозможным;
- если очередной такт работы автомата P невозможен, то автомат P' переходит в состояние q_ε и остается в нем до тех пор, пока не будет прочитана вся входная цепочка.

Другая проблема, которая может возникнуть при построении требуемого ДМП-автомата, заключается в том, что ДМП-автомат может, начиная с некоторой конфигурации, выполнить бесконечное число ε-тактов, не прочитав больше ни одного входного символа (т. е. попасть в зацикливающую конфигурацию).

Конфигурация (q, w, α) ДМП-автомата называется *зацикливающей*, если для любого $i \geq 1$ найдется такая конфигурация (p_i, w, β_i), что $|\beta_i| \geq |\alpha|$ и

$$(q, w, \alpha) \vdash (p_1, w, \beta_1) \vdash (p_2, w, \beta_2) \vdash \ldots$$

Это значит, что конфигурация является зацикливающей, если ДМП-автомат P может делать бесконечное число тактов, не уменьшая число символов в магазине. При этом содержимое магазина может бесконечно расти или циклически совпадать с несколькими различными цепочками.

Если ДМП-автомат P может перейти в зацикливающую конфигурацию, то существуют незацикливающие конфигурации, из которых после ряда ε-тактов, укорачивающих цепочку в магазине, он переходит в зацикливающую конфигурацию. Если ДМП-автомат P попадает в зацикливающую конфигурацию в середине входной цепочки, то он больше не будет ее использовать, даже если удовлетворяет условию леммы 5.4.

Рассмотрим, каким образом можно преобразовать ДМП-автомат P в эквивалентный ему ДМП-автомат P', который никогда не попадает в зацикливающую конфигурацию.

Алгоритм 5.1. *Обнаружение зацикливающих конфигураций*

Вход: ДМП-автомат $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$.

Выход:

1. $C_1 = \{(q, A) \mid (q, \varepsilon, A)$ — зацикливающая конфигурация и не существует такого заключительного состояния $r \in F$, что $(q, \varepsilon, A) \vdash^* (r, \varepsilon, \alpha)$ для некоторой цепочки $\alpha \in \Gamma^*\}$.
2. $C_2 = \{(q, A) \mid (q, \varepsilon, A)$ — зацикливающая конфигурация и $(q, \varepsilon, A) \vdash^* (r, \varepsilon, \alpha)$ для некоторых $r \in F$ и $\alpha \in \Gamma^*\}$.

Описание алгоритма:

□ Пусть $|Q| = n_1$, $|\Gamma| = n_2$ и l — длина самой длинной цепочки, которую ДМП-автомат P может записать в магазин за один такт. Тогда длина самой длинной из возможных цепочек, которую P может записать в магазин, равна $n_1 n_2 l$. Обозначим через n_3 максимальное число ε-тактов, которое может сделать ДМП-автомат P, не зацикливаясь.

$n_3 = n_1$, если $n_2 = 1$, и $n_3 = \dfrac{n_1(n_2^{\,n_1 n_2 l+1} - n_2)}{n_2 - 1}$, если $n_2 \neq 1$.

Для определения зацикливающих конфигураций необходимо выполнить следующие действия:

1. Моделируя работу ДМП-автомата P для всех $q \in Q$ и $A \in \Gamma$, определить, существуют ли такие $r \in Q$ и $\alpha \in \Gamma^*$, что $(q, \varepsilon, A) \vdash^{n_3} (r, \varepsilon, \alpha)$. Если существуют, то (q, ε, A) — зацикливающая конфигурация.
2. Если (q, ε, A) — зацикливающая конфигурация, то, моделируя работу автомата P, определить, существует ли такое состояние $r \in F$, что $(q, \varepsilon, A) \vdash^j (r, \varepsilon, \alpha)$ для некоторого $0 \le j \le n_3$. Если да, то включить (q, A) в C_2, в противном случае включить (q, A) в C_1.

■

В работе [3] доказывается, что если автомат P, начиная с конфигурации (q, ε, A), может достичь заключительной конфигурации, то это должно произойти за n_3 или меньшее число тактов.

ДМП-автомат $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ называется *дочитывающим*, если для каждой цепочки $w \in \Sigma^*$ существуют такие $p \in Q$ и $\alpha \in \Gamma^*$, что $(q_0, w, Z_0) \vdash^* (p, \varepsilon, \alpha)$.

По определению, ДМП-автомат является дочитывающим, если он читает до конца *любую* входную цепочку. В [3, 37] предлагается способ, позволяющий по заданному ДМП-автомату P построить эквивалентный ему дочитывающий ДМП-автомат P'.

Лемма 5.5

Пусть $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — ДМП-автомат. Можно построить такой эквивалентный ДМП-автомат $P' = (Q \cup \{p, r\}, \Sigma, \Gamma, \delta', q_0, Z_0, F \cup \{p\})$, где p и r — новые состояния, а δ' определяется по следующим правилам:

1. Для всех $q \in Q$, $a \in \Sigma$ и $Z \in \Gamma$ положить $\delta'(q, a, Z) = \delta(q, a, Z)$.
2. Для всех $q \in Q$ и $Z \in \Gamma$ положить $\delta'(q, \varepsilon, Z) = \delta(q, \varepsilon, Z)$, если (q, ε, Z) не является зацикливающей конфигурацией.
3. Для всех $(q, Z) \in C_1$ положить $\delta'(q, \varepsilon, Z) = (r, Z)$, где C_1 — множество, построенное с помощью алгоритма 5.1.
4. Для всех $(q, Z) \in C_2$ положить $\delta'(q, \varepsilon, Z) = (p, Z)$, где C_2 — множество, построенное с помощью алгоритма 5.1.
5. Для всех $a \in \Sigma$ и $Z \in \Gamma$ положить $\delta'(p, a, Z) = (r, Z)$ и $\delta'(r, a, Z) = (r, Z)$.

Автомат P' построен таким образом, что он моделирует работу автомата P до тех пор, пока P не попадает в зацикливающую конфигурацию. Тогда автомат P' в очередном такте перейдет в состояние p или r в зависимости от того, встречается в циклической последовательности конфигураций заключительное состояние или нет. Затем P' переходит в состояние r и остается в нем, пока не дочитает входную цепочку до конца. Если автомат P не попадает в зацикливающую конфигурацию, то через конечное число тактов он должен сделать не ε-такт или попасть в конфигурацию, укорачивающую цепочку в магазине. Так как глубина магазина конечна, то после нескольких повторений второй ситуации автомат P начнет читать символы из входной цепочки или перейдет в зацикливающую конфигурацию.

5.5. Преобразователи с магазинной памятью

Преобразователи с магазинной памятью (МП-преобразователи) строятся на основе автоматов с магазинной памятью путем добавления к ним выходной ленты и возможности записи на нее цепочек конечной длины (рис. 5.3).

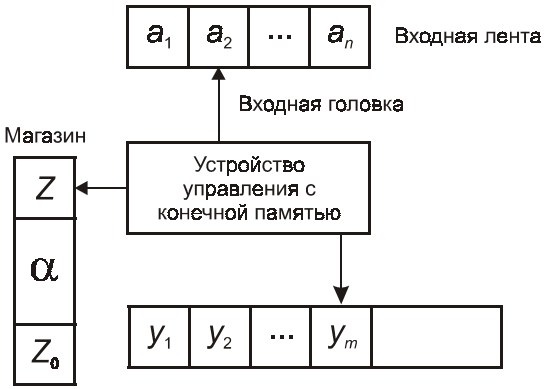


Рис. 5.3. Преобразователь с магазинной памятью

Преобразователем с магазинной памятью называется восьмерка объектов $D = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$, где все символы, обозначающие объекты, имеют тот же смысл, что и в определении МП-автомата, за исключением того, что Δ — конечный *выходной алфавит*, а δ — функция переходов и выходов, представляющая собой отображение множества $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ в множество конечных подмножеств множества $Q \times \Gamma^* \times \Delta^*$.

Если $D = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$ — МП-преобразователь, то МП-автомат $P = (Q, \Sigma, \Gamma, \delta', q_0, Z_0, F)$, где $\delta'(q, a, Z)$ содержит (r, γ) тогда и только тогда, когда $\delta(q, a, Z)$ содержит (r, γ, y) для некоторого $y \in \Delta^*$, называется МП-автоматом, *лежащим в основе* МП-преобразователя D.

Работу МП-преобразователя будем описывать в терминах конфигураций. *Конфигурацией МП-преобразователя* называется четверка (q, x, α, y), где q — текущее состояние МП-преобразователя, x — необработанная часть входной цепочки, α — цепочка в магазине (вершина слева), а y — цепочка на выходной ленте, выданная к настоящему времени.

Если $(r, \alpha, z) \in \delta(q, a, Z)$, то будем писать $(q, ax, Z\gamma, y) \vdash (r, x, \alpha\gamma, yz)$ для любых $x \in \Sigma^*$, $\gamma \in \Gamma^*$ и $y \in \Delta^*$.

На множестве конфигураций выделим *начальную конфигурацию* $(q_0, x, Z_0, \varepsilon)$ и *заключительную конфигурацию* $(q, \varepsilon, \alpha, y)$, где $q \in F$ и $\alpha \in \Gamma^*$.

Цепочка y называется *выходом* для цепочки x, если $(q_0, x, Z_0, \varepsilon) \vdash^* (q, \varepsilon, \alpha, y)$ для некоторых $q \in F$ и $\alpha \in \Gamma^*$.

Переводом (*преобразованием*), определяемым МП-преобразователем D, называется множество $\tau(D) = \{(x, y) \mid (q_0, x, Z_0, \varepsilon) \vdash^* (q, \varepsilon, \alpha, y)$ для некоторых $q \in F$ и $\alpha \in \Gamma^*\}$.

Аналогично МП-автоматам, можно определить преобразование входной цепочки x в выходную цепочку y *опустошением магазина*, если имеет место $(q_0, x, Z_0, \varepsilon) \vdash^* (q, \varepsilon, \varepsilon, y)$ для некоторого $q \in F$.

Переводом, определяемым МП-преобразователем D *опустошением магазина*, называется множество $\tau_\varepsilon(D) = \{(x, y) \mid (q_0, x, Z_0, \varepsilon) \vdash^* (q, \varepsilon, \varepsilon, y)$ для некоторого $q \in F\}$.

Пример 5.9

Пусть МП-преобразователь, который переводит арифметическое выражение, записанное в инфиксной форме, в эквивалентную префиксную польскую запись, выглядит следующим образом:

$$D = (\{q\}, \{i, +, *, (,)\}, \{i, +, *, (,), E, T, P\}, \{i, +, *\}, \delta, q, E, \{q\}),$$

где δ определяется формулами:

$$\delta(q, \varepsilon, E) = \{(q, E + T, +), (q, T, \varepsilon)\},$$

$$\delta(q, \varepsilon, T) = \{(q, T * P, *), (q, P, \varepsilon)\},$$

$$\delta(q, \varepsilon, P) = \{(q, (E), \varepsilon), (q, i, i)\},$$

$$\delta(q, a, a) = \{(q, \varepsilon, \varepsilon)\} \text{ для всех } a \in \{i, +, *, (,)\}.$$

Для входной цепочки $i * (i + i)$ МП-преобразователь D среди других может выполнить следующую последовательность тактов, опустошая магазин:

$$\begin{array}{llll}
(q, & i * (i + i), & E, & \varepsilon) \vdash (q, \; i * (i + i), \; T, \; \varepsilon) \vdash \\
(q, & i * (i + i), & T * P, & *) \vdash (q, \; i * (i + i), \; P * P, \; *) \vdash \\
(q, & i * (i + i), & i * P, & * i) \vdash (q, \; * (i + i), \; * P, \; * i) \vdash \\
(q, & (i + i), & P, & * i) \vdash (q, \; (i + i), \; (E), \; * i) \vdash \\
(q, & i + i), & E), & * i) \vdash (q, \; i + i), \; E + T), \; * i +) \vdash \\
(q, & i + i), & T + T), & * i +) \vdash (q, \; (i + i), \; P + T), \; * i +) \vdash \\
(q, & i + i), & i + T), & * i + i) \vdash (q, \; +i), \; + T), \; * i + i) \vdash \\
(q, & i), & T), & * i + i) \vdash (q, \; i), \; P), \; * i + i) \vdash \\
(q, & i), & i), & * i + i) \vdash (q, \;), \;), \; * i + ii) \vdash \\
(q, & \varepsilon, & \varepsilon, & * i + ii) .
\end{array}$$

МП-преобразователь моделирует нисходящий перевод. Моделирование восходящего перевода можно получить с помощью расширенного преобразователя с магазинной памятью.

МП-преобразователь $D = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$ *называется расширенным*, если все символы, обозначающие объекты, имеют тот же смысл, что и в определении МП-преобразователя, за исключением того, что δ — это отображение множества $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^*$ в множество конечных подмножеств множества $Q \times \Gamma^* \times \Delta^*$.

Пример 5.10

🗁 Пусть расширенный МП-преобразователь, который переводит арифметическое выражение, записанное в инфиксной форме, в эквивалентную постфиксную польскую запись, выглядит следующим образом:

$$D = (\{q, r\}, \{i, +, *, (,)\}, \{i, +, *, (,), E, T, P\}, \{i, +, *\}, \delta, q, \perp, \{r\}),$$

где δ определяется равенствами:

$$\delta(q, a, \varepsilon) = \{(q, a, \varepsilon)\} \text{ для всех } a \in \{i, +, *, (,)\},$$

$$\delta(q, \varepsilon, E + T) = \{(q, E, +)\},$$

$$\delta(q, \varepsilon, T) = \{(q, E, \varepsilon)\},$$

$$\delta(q, \varepsilon, T * P) = \{(q, T, *)\},$$

$$\delta(q, \varepsilon, P) = \{(q, T, \varepsilon)\},$$

$$\delta(q, \varepsilon, i) = \{(q, P, i)\},$$

$$\delta(q, \varepsilon, (E)) = \{(q, P, \varepsilon)\},$$

$$\delta(q, \varepsilon, \perp E) = \{(r, \varepsilon, \varepsilon)\}.$$

Для входной цепочки $i + i * i$ расширенный МП-преобразователь D, опустошая магазин, среди прочих может выполнить следующую последовательность тактов:

$$\begin{array}{llrr|llrr}
(q, & i + i * i, & \perp, & \varepsilon) \vdash & (q, & + i * i, & \perp i, & \varepsilon) \vdash \\
(q, & + i * i, & \perp P, & i) \vdash & (q, & i * i, & \perp T, & i) \vdash \\
(q, & + i * i, & \perp E, & i) \vdash & (q, & i * i, & \perp E +, & i) \vdash \\
(q, & * i, & \perp E + i, & i) \vdash & (q, & * i, & \perp E + P, & ii) \vdash \\
(q, & * i, & \perp E + T, & ii) \vdash & (q, & i, & \perp E + T *, & ii) \vdash \\
(q, & \varepsilon, & \perp E + T * i, & ii) \vdash & (q, & \varepsilon, & \perp E + T * P, & iii) \vdash \\
(q, & \varepsilon, & \perp E + T, & iii *) \vdash & (q, & \varepsilon, & \perp E, & iii * +) \vdash \\
(r, & \varepsilon, & \varepsilon, & iii * +) . & & & &
\end{array}$$

🗀

Для реализации языковых процессоров желательно использовать только детерминированные модели перевода. Определим детерминированный МП-преобразователь и детерминированный расширенный МП-преобразователь.

Детерминированным преобразователем с магазинной памятью (ДМП-преобразователем) называется МП-преобразователь $D = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$, у которого:

- для всех $q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$ и $Z \in \Gamma$ множество $\delta(q, a, Z)$ содержит не более одного элемента;
- если $\delta(q, \varepsilon, Z) \neq \varnothing$, то $\delta(q, a, Z) = \varnothing$ для всех $a \in \Sigma$.

Пример 5.11

🗁 Пусть детерминированный МП-преобразователь, который переводит арифметическое выражение, записанное в польской префиксной записи, в польскую постфиксную запись, определяется как:

$$D = (\{q\}, \{i, +, *\}, \{E, +, *\}, \{i, +, *\}, \delta, q, E, \{q\}),$$

где δ имеет вид:

$$\delta(q, i, E) = \{(q, \varepsilon, i)\},$$

$$\delta(q, +, E) = \{(q, EE+, \varepsilon)\},$$

$$\delta(q, *, E) = \{(q, EE*, \varepsilon)\},$$

$$\delta(q, \varepsilon, +) = \{(q, \varepsilon, +)\},$$

$$\delta(q, \varepsilon, *) = \{(q, \varepsilon, *)\}.$$

Для входной цепочки $* i + ii$ МП-преобразователь D выполнит следующую последовательность тактов, опустошая магазин:

$$(q, * i + ii, E, \varepsilon) \vdash (q, i + ii, EE*, \varepsilon) \vdash$$
$$(q, + ii, E*, i) \vdash (q, ii, EE+*, i) \vdash$$
$$(q, i, E+*, ii) \vdash (q, \varepsilon, +*, iii) \vdash$$
$$(q, \varepsilon, *, iii+) \vdash (q, \varepsilon, \varepsilon, iii+*).$$

🗀

Расширенный МП-преобразователь $D = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$ называется *детерминированным расширенным преобразователем с магазинной памятью* (*расширенным ДМП-преобразователем*), если выполняются следующие условия:

- для всех $q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$и $\gamma \in \Gamma^*$ множество $\delta(q, a, \gamma)$ содержит не более одного элемента;
- если $\delta(q, a, \alpha) \neq \varnothing$, $\delta(q, a, \beta) \neq \varnothing$ и $\alpha \neq \beta$, то ни одна из цепочек α и β не является суффиксом другой (верхушка магазина справа);
- если $\delta(q, a, \alpha) \neq \varnothing$, $\delta(q, \varepsilon, \beta) \neq \varnothing$ и $\alpha \neq \beta$, то ни одна из цепочек α и β не является суффиксом другой.

Замечание

Если в основе МП-преобразователя (расширенного МП-преобразователя) лежит детерминированный МП-автомат (детерминированный расширенный МП-автомат), то МП-преобразователь (расширенный МП-преобразователь) может быть и недетермированным.

Контрольные вопросы

1. Дайте определение недетерминированного автомата с магазинной памятью и языка, допускаемого МП-автоматом.
2. Определите такие понятия, как конфигурация МП-автомата, такт работы МП-автомата, ε-такт.
3. Дайте определение недетерминированного расширенного автомата с магазинной памятью. Что представляют собой входная лента, устройство управления и вспомогательная память расширенного МП-автомата?
4. Определите такие понятия, как конфигурация расширенного МП-автомата, такт работы МП-автомата, ε-такт.
5. Опишите процесс построения МП-автомата по расширенному МП-автомату.
6. Дайте определение МП-автомата, допускающего входные цепочки опустошением магазина.
7. Опишите процесс построения МП-автомата, допускающего входные цепочки опустошением магазина.
8. Опишите алгоритм построения недетерминированного МП-автомата по КС-грамматике.
9. Опишите алгоритм построения недетерминированного расширенного МП-автомата по КС-грамматике.
10. Опишите алгоритм построения КС-грамматики по недетерминированному МП-автомату.
11. Дайте определение детерминированного МП-автомата.
12. Дайте определение детерминированного расширенного МП-автомата.
13. Докажите лемму о существовании ДМП-автомата, для которого всегда возможен очередной такт.
14. Дайте определение зацикливающей конфигурации.
15. Опишите алгоритм построения множества зацикливающих конфигураций.
16. Дайте определение дочитывающего ДМП-автомата.
17. Опишите алгоритм, позволяющий по заданному ДМП-автомату построить эквивалентный ему дочитывающий ДМП-автомат.
18. Дайте определение недетерминированного МП-преобразователя и недетерминированного расширенного МП-преобразователя.
19. Дайте определение детерминированного МП-преобразователя и детерминированного расширенного МП-преобразователя.

20. Будет ли МП-преобразователь детерминированным, если он построен на основе ДМП-автомата?
21. Дайте определение перевода, определяемого МП-преобразователем.

Упражнения

1. Постройте МП-автомат P и расширенный МП-автомат P' по КС-грамматике $G = (\mathrm{N}, \Sigma, P, S)$.
   1.1. $\mathrm{N} = \{S, L, B\}$, $\Sigma = \{i, =, *\}$, $\mathrm{R} = \{S \to L = B,\ S \to B,\ L \to *B,\ L \to i,\ B \to L\}$.
   1.2. $\mathrm{N} = \{S, T, P\}$, $\Sigma = \{i, \neg, \wedge, \vee, (,)\}$, $P = \{S \to S \vee T,\ S \to T,\ T \to T \wedge P,\ T \to P,\ P \to \neg P,\ P \to i,\ P \to (S)\}$.
   1.3. $\mathrm{N} = \{S, A, B, C, P, Q\}$, $\Sigma = \{m, p, q, x, y\}$, $P = \{S \to AB,\ S \to PQx,\ A \to xy,\ A \to m,\ B \to bC,\ C \to bC,\ C \to \varepsilon,\ P \to pP,\ P \to \varepsilon,\ Q \to qQ,\ Q \to \varepsilon\}$.
   1.4. $\mathrm{N} = \{S, A, B\}$, $\Sigma = \{a, b, c, d\}$, $P = \{S \to Aa,\ S \to bB,\ A \to cAdA,\ A \to a,\ A \to \varepsilon,\ B \to cBdd,\ B \to \varepsilon\}$.
   1.5. $\mathrm{N} = \{S, T, P, C\}$, $\Sigma = \{+, \times, /, .\}$, $P = \{S \to S + T,\ S \to T,\ T \to T \times P,\ T \to P,\ P \to .C.,\ C \to /C,\ C \to /\}$.
   1.6. $\mathrm{N} = \{S, D, R, X, Y\}$, $\Sigma = \{begin, end, d, r, ;, ,\}$, $\mathrm{R} = \{S \to begin\ D;R\ end,\ D \to dX,\ X \to ,dX,\ X \to \varepsilon,\ R \to rY,\ Y \to ,rY,\ Y \to \varepsilon\}$.
   1.7. $\mathrm{N} = \{S, A, L\}$, $\Sigma = \{a, b, (,)\}$, $P = \{S \to bAb,\ A \to (L,\ A \to a,\ L \to Aa)\}$.
   1.8. $\mathrm{N} = \{S, A\}$, $\Sigma = \{a, b, c\}$, $P = \{S \to SaA,\ S \to AA,\ S \to b,\ A \to ASa,\ A \to Ad,\ A \to c\}$.
   1.9. $\mathrm{N} = \{S, A, B\}$, $\Sigma = \{a, b, c, d\}$, $P = \{S \to Ab,\ A \to Sa,\ A \to cB,\ B \to bS,\ B \to c\}$.
   1.10. $\mathrm{N} = \{S, A, B\}$, $\Sigma = \{a, b, c, d\}$, $P = \{S \to dAa,\ S \to \varepsilon,\ A \to aSB,\ A \to dSc,\ B \to b,\ B \to \varepsilon\}$.
2. Постройте ДМП-автомат, распознающий цепочки:
   2.1. из множества $\{a^n b^m c^n \mid n > 0,\ m \geq 0\}$.
   2.2. из множества $\{a^n b^m c^m d^n \mid n > 0,\ m \geq 0\}$.
   2.3. в алфавите $\{0, 1\}$ с одинаковым количеством нулей и единиц.
   2.4. из множества $\{0^n 1^n\ 0^m 1^m \ldots \mid n, m, \ldots > 0\}$.
   2.5. из множества $\{0^n 1^n \mid n > 0\}$.
   2.6. из множества $\{0^n 1^n \mid n > 0\} \cup \{1^n 0^n \mid n > 0\}$.

2.7. из множества $\{0^n10^n \mid n > 0\}$.

2.8. из множества $\{1^{3n+2}0^n \mid n \geq 0\}$.

2.9. из множества $\{0^n1^m \mid n > m > 0\}$.

2.10. из множества $\{0^n1^m \mid n \geq m > 0\}$.

3. Постройте ДМП-преобразователь, осуществляющий перевод:

3.1. произвольной цепочки из множества $\{a^nb^mc^n$, где $n > 0$, $m \geq 0\}$ в цепочку вида 1^{n+m}.

3.2. произвольной цепочки, состоящей из нулей и единиц, в цепочку вида 1^n0^m, где n и m — соответственно число единиц и нулей в данной цепочке.

3.3. произвольной цепочки из множества $\{a^nb^mc^md^n$, где $n > 0$, $m \geq 0\}$ в цепочку вида 1^n0^{m+n}.

3.4. произвольной цепочки из множества $\{0^n1^n\,0^m1^m$ ... , где n, m, ... $> 0\}$ в цепочку вида $1^{n+m+\ldots}$.

3.5. произвольной цепочки из множества $\{0^n1^n$, $n > 0\}$ в цепочку вида a^{2n}.

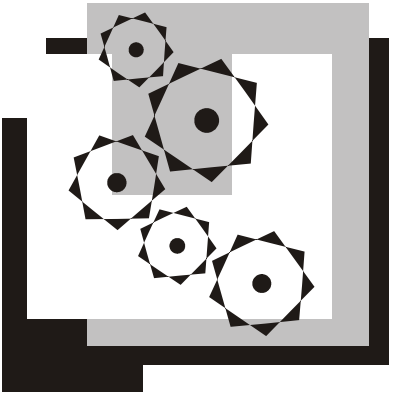
3.6. произвольной цепочки из множества $\{0^n1^n$, $n > 0\} \cup \{1^n0^n$, $n > 0\}$ в цепочку вида $1^{2n}0^{2n}$.

3.7. цепочки из множества $\{1^m0^n$, $m, n > 0$, $m \neq n\}$ в цепочку вида 1^{m-n}, если $m > n$, или в цепочку 1^{n-m}, если $n > m$.

3.8. произвольной цепочки из множества $\{1^{3n+2}0^n$, $n \geq 0\}$ в цепочку вида 1^n0^n.

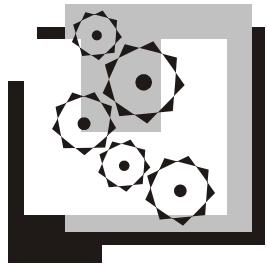
3.9. произвольной цепочки из множества $\{1^n0^m$, $n, m > 0\}$ в цепочку вида 0^n1^{2n}.

3.10. цепочки b_i в цепочку $(b_{i+1})^r$, где b_i — цепочка из нулей и единиц, являющаяся двоичным представлением числа i.



Часть III

Методы синтаксического анализа



Глава 6

Общие методы синтаксического анализа

Практически все методы синтаксического анализа приводят к необходимости построения устройств или *процессоров*, которые при своей работе некоторым образом используют правила КС-грамматики.

Действия таких процессоров можно интерпретировать как распознавание отдельных правил грамматики в дереве вывода. Это объясняет, почему такие процессоры часто называют *синтаксическими анализаторами*, а их работу — *синтаксически управляемой трансляцией.*

Рассматриваемые далее анализаторы просматривают входную цепочку слева направо, используя одну из двух стратегий разбора: нисходящую или восходящую. В нисходящих анализаторах правила грамматики распознаются сверху вниз, т. е. верхние правила раньше нижних, при этом для анализируемой цепочки строится левый вывод. Восходящие анализаторы распознают нижние правила раньше, чем верхние, и строят правый разбор.

В данной главе мы дадим формальные определения этих двух распространенных типов разбора и сопоставим их возможности.

6.1. Определение разбора

Будем считать, что цепочка $w \in L(G)$ для некоторой КС-грамматики G *проанализирована* или *разобрана*, если известно ее дерево вывода в данной грамматике.

Замечание

При анализе входной цепочки с использованием неоднозначной грамматики может быть построено одно, несколько или все деревья выводов входной цепочки.

Обычно компилятор выполняет синтаксический анализ путем моделирования МП-автомата, анализирующего входные цепочки (*см. гл. 5*).

МП-автомат отображает входные цепочки в соответствующие левые выводы (нисходящий анализ) или правые разборы (восходящий анализ). Поэтому мы будем рассматривать проблему разбора как проблему отображения цепочек в их левые или правые выводы.

Пусть заданы КС-грамматика $G = (N, \Sigma, P, S)$, правила которой перенумерованы целыми числами 1, 2, ... , p, и цепочка $\alpha \in (N \cup \Sigma)^*$. Тогда:

- *левым разбором* цепочки α называется последовательность правил, примененных при ее левом выводе из S;
- *правым разбором* цепочки α называется обращение последовательности правил, примененных при правом выводе цепочки α из S.

Левый и правый разборы будем представлять в виде последовательности номеров правил грамматики из множества {1, 2, ... , p}.

Пример 6.1

Рассмотрим анализ цепочки $i * (i + i) \in L(G_0)$. После нумерации правила грамматики G_0 принимают следующий вид:

(1) $E \to E + T$ (4) $T \to P$

(2) $E \to T$ (5) $P \to i$

(3) $T \to T * P$ (6) $P \to (E)$

Построим левый вывод цепочки $i * (i + i)$:

$$E \Rightarrow_{(2)} T \Rightarrow_{(3)} T * P \Rightarrow_{(4)} P * P \Rightarrow_{(5)} i * P \Rightarrow_{(6)} i * (E) \Rightarrow_{(1)} i * (E + T) \Rightarrow_{(2)} i * (T + T) \Rightarrow_{(4)} i * (P + T) \Rightarrow_{(5)} i * (i + T) \Rightarrow_{(4)} i * (i + P) \Rightarrow_{(5)} i * (i + i).$$

По определению, левый разбор цепочки представляет собой последовательность номеров правил грамматики при ее левом выводе. Следовательно, левый разбор цепочки $i * (i + i)$ — это последовательность 23456124545.

Построим правый вывод цепочки $i * (i + i)$:

$$E \Rightarrow_{(2)} T \Rightarrow_{(3)} T * P \Rightarrow_{(6)} T * (E) \Rightarrow_{(1)} T * (E + T) \Rightarrow_{(4)} T * (E + P) \Rightarrow_{(5)} T * (E + i) \Rightarrow_{(2)} T * (T + i) \Rightarrow_{(4)} T * (P + i) \Rightarrow_{(5)} T * (i + i) \Rightarrow_{(4)} P * (i + i) \Rightarrow_{(5)} i * (i + i).$$

Правый разбор цепочки — это обращение последовательности номеров правил при правом выводе данной цепочки, поэтому правый разбор цепочки $i * (i + i)$ — это последовательность 54542541632.

Процесс левого и правого вывода цепочки $i * (i + i)$ в грамматике G_0 приведен на рис. 6.1. Деревья выводов на рисунках деформированы таким образом, чтобы показать последовательность применения правил грамматики при их построении. На рис. 6.1, *а* номера применяемых при выводе правил помещены слева от дерева, а на рис. 6.1, *б* — справа. □

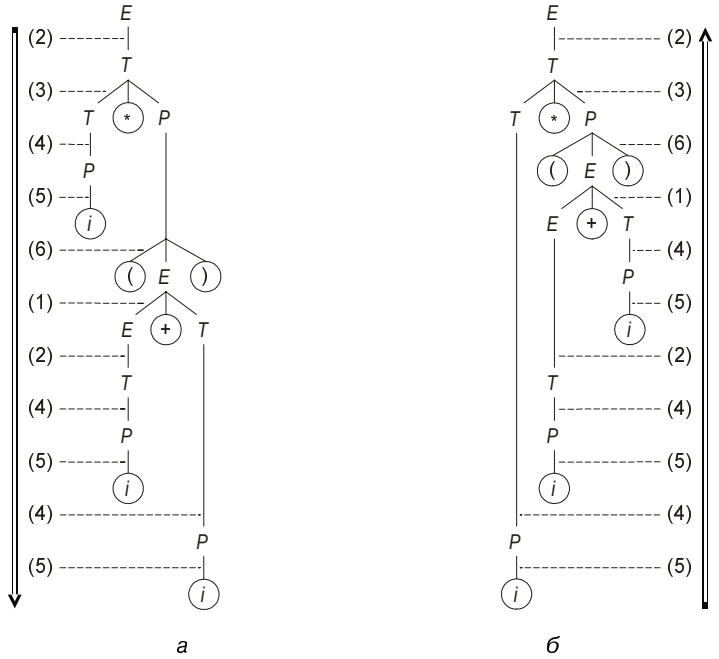


Рис. 6.1. Левый (*а*) и правый (*б*) выводы цепочки $i * (i + i)$

Рис. 6.1 хорошо иллюстрирует, что левый разбор совпадает с левым выводом, а правый разбор представляет собой обращение правого вывода.

6.2. Нисходящий разбор

Рассмотрим выполнение синтаксического анализа КС-грамматик при условии, что анализируемая цепочка рассматривается слева направо (левый анализ).

Если нам известен левый разбор π цепочки $w \in L(G)$, где G — КС-грамматика, то мы можем построить дерево ее разбора следующим образом.

Пометим корень дерева основным символом грамматики S. Пусть $\pi = i_1\ i_2\ ...\ i_n$. Тогда i_1 определяет номер правила, которое надо применить к S. Предположим, что это правило $S \to X_1\ X_2\ ...\ X_{i-1}\ X_i\ X_{i+1}\ ...\ X_k$. Присоединим k прямых потомков к вершине, помеченной S, и пометим их символами X_1, X_2, ... , X_{i-1}, X_i, X_{i+1}, ... , X_k. Если X_i — первый слева нетерминал в цепочке $X_1\ X_2\ ...\ X_{i-1}\ X_i\ X_{i+1}\ ...\ X_k$, то первыми $(i - 1)$-ыми символами цепочки *w должны* быть символы $X_1\ ...\ X_{i-1}$. Следующее примененное при выводе правило грамматики с номером i_2 *должно* иметь в левой части нетерминал X_i. Допустим, например, что это правило $X_i \to Y_1\ Y_2\ ...\ Y_l$, и продолжим построение

дерева разбора цепочки w, применяя процедуру, аналогичную только что использованной, к вершине, помеченной нетерминальным символом X_i. Присоединим l прямых потомков к вершине, обозначенной X_i, и пометим их символами Y_1, Y_2, ... , Y_l (рис. 6.2).

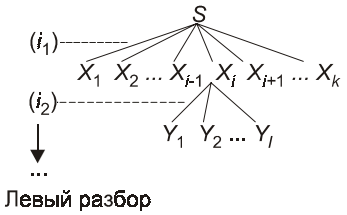


Рис. 6.2. Построение дерева разбора

Продолжая выполнять описанные ранее действия, можно построить все дерево разбора цепочки w, соответствующее левому разбору π.

Если задана КС-грамматика $G = (N, \Sigma, P, S)$, в которой правила перенумерованы от 1 до p, и цепочка $w \in \Sigma^*$, для которой мы хотим построить левый разбор, то можно считать, что *известны корень* и *крона дерева* разбора, и нам остается только построить промежуточные вершины (рис. 6.3).

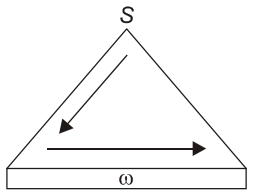


Рис. 6.3. Стратегия левого нисходящего разбора

Стратегия левого нисходящего разбора предлагает заполнять дерево разбора, начиная с корня, и двигаться слева направо, направляясь к кроне.

Известно (*см. гл. 5*), что для любой грамматики G можно построить недетерминированный МП-преобразователь, который может быть *основой синтаксического анализатора.*

Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика, в которой правила перенумерованы числами от 1 до p. *Левым анализатором* M_l называется недетерминированный МП-преобразователь $(\{q\}, \Sigma, N \cup \Sigma, \{1, 2, ... , p\}, \delta, q, S, \{q\})$, где δ определяется следующим образом:

1. Если $A \to \alpha$ — правило из P с номером i, то $(q, \alpha, i) \in \delta(q, \varepsilon, A)$.
2. $\delta(q, a, a) = \{(q, \varepsilon, \varepsilon)\}$ для всех $a \in \Sigma$.

Построенный таким образом анализатор M_l может определить только левый вывод цепочки w, поскольку может выполнять следующие операции:

- *развертывать* нетерминал, расположенный наверху магазина, в соответствии с некоторым правилом из P и одновременно выдавать номер этого правила, используя функцию δ, построенную по правилу (1) определения;
- *сравнивать* текущий входной символ с верхним символом магазина, применяя функцию δ, построенную по правилу (2) определения. 12

Пример 6.2

🗁 Построим левый анализатор для грамматики G_0:

(1) $E \to E + T$ (4) $T \to P$

(2) $E \to T$ (5) $P \to i$

(3) $T \to T * P$ (6) $P \to (E)$.

$M_l = (\{q\}, \{i, +, *, (,)\}, \{i, +, *, (,), E, T, P\}, \{1, 2, 3, 4, 5, 6\}, \delta, q, E, \{q\})$.

Определим функцию δ:

- по правилу (2) определения $\delta(q, a, a) = \{(q, \varepsilon, \varepsilon)\}$ для всех $a \in \Sigma$;
- для правила грамматики (1) $E \to E + T$ по правилу (1) определения в функцию $\delta(q, \varepsilon, E)$ необходимо включить элемент $(q, E + T, 1)$.

Повторив аналогичные действия для всех правил грамматики, получим окончательный вид функции переходов-выходов δ:

$\delta(q, \varepsilon, E) = \{(q, E + T, 1), (q, T, 2)\}$,

$\delta(q, \varepsilon, T) = \{(q, T * P, 3), (q, P, 4)\}$,

$\delta(q, \varepsilon, P) = \{(q, (E), 6), (q, i, 5)\}$.

Для входной цепочки $i + i * i$ построенный анализатор M_l может выполнить такую последовательность тактов:

$(q, i + i * i, E, \varepsilon) \vdash (q, i + i * i, E + T, 1) \vdash (q, i + i * i, T + T, 12)$

$\vdash (q, i + i * i, P + T, 124) \vdash (q, i + i * i, i + T, 1245)$

$\vdash (q, + i * i, + T, 1245) \vdash (q, i * i, T, 1245)$

$\vdash (q, i * i, T * P, 12453) \vdash (q, i * i, P * P, 124534)$

$\vdash (q, i * i, i * P, 1245345) \vdash (q, * i, * P, 1245345)$

$\vdash (q, i, P, 1245345) \vdash (q, i, i, 12453455)$

$\vdash (q, \varepsilon, \varepsilon, 12453455)$.

Левый вывод входной цепочки $i + i * i$ равен 12453455. 🗀

Замечание

По определению, построенный анализатор является недетерминированным. Чтобы воспользоваться таким анализатором на практике, необходимо преобразовать его в детерминированный.

Нисходящий анализ налагает ограничение на КС-грамматику: она *не должна содержать правил с левой рекурсией*.

6.3. Восходящий разбор

В *гл. 5* было показано, что восходящий анализ базируется на понятии *основы*. Пусть имеется КС-грамматика с правилами:

(1) $S \rightarrow (A\ S)$ (3) $A \rightarrow (S\ a\ A)$

(2) $S \rightarrow (b)$ (4) $A \rightarrow (a)$

Рассмотрим правый вывод цепочки ((*a*) (((*b*) *a* (*a*)) (*b*))), дерево вывода которой приведено на рис. 6.4:

$$S \Rightarrow_{(1)} (A\ S) \Rightarrow_{(1)} (A (A\ S)) \Rightarrow_{(2)} (A (A (b)) \Rightarrow_{(3)} (A ((S\ a\ A) (b)) \Rightarrow_{(4)}$$
$$(A ((S\ a (a)) (b))) \Rightarrow_{(2)} (A (((b) a (a)) (b))) \Rightarrow_{(4)}$$
$$((a) (((b) a (a)) (b))).$$

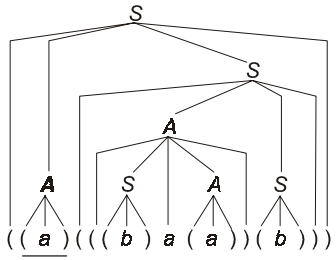


Рис. 6.4. Дерево вывода цепочки ((*a*) (((*b*) *a* (*a*)) (*b*)))

На рис. 6.5, *а* вывод цепочки представлен в другой форме. Для каждого *i*-го шага вывода полужирным шрифтом выделен нетерминал, подлежащий замене. Номер используемого правила указан около стрелки, а в цепочке, приведенной на $(i + 1)$-ом шаге вывода, подчеркнута правая часть этого правила. Например, на 7-ом шаге вывода самый правый (и единственный) нетерминал *A* подлежит замене. Для замены использована правая часть 4-го правила, которая подчеркнута в полученной цепочке.

Вывод цепочки можно рассматривать как процесс построения дерева вывода: замена нетерминального символа правой частью соответствующего правила α приводит к добавлению символов данной правой части α в качестве

непосредственных потомков рассматриваемого нетерминала *A*. На рис. 6.4 выделен шрифтом нетерминал *A*, к которому "подвешивается" подчеркнутая правая часть четвертого правила грамматики.

Выполнение разбора входной цепочки ((*a*) (((*b*) *a* (*a*)) (*b*))) показано на рис. 6.5, *б*. На каждом шаге разбора найденная основа (подчеркнутая цепочка) заменяется (эта операция обозначается символом ($\succ$)) правой частью соответствующего основывающего правила (нетерминалом, записанным на следующем шаге разбора и выделенным полужирным шрифтом).

		номер шага		
S	$\Rightarrow_{(1)}$	-1-	((*a*) (((*b*) *a* (*a*)) (*b*)))	$\succ_{(4)}$
(**A S**)	$\Rightarrow_{(1)}$	-2-	(**A** (((*b*) *a* (*a*)) (*b*)))	$\succ_{(2)}$
(*A* (**A S**))	$\Rightarrow_{(2)}$	-3-	(*A* ((**S** *a* (*a*)) (*b*)))	$\succ_{(4)}$
(*A* (**A** (*b*)))	$\Rightarrow_{(3)}$	-4-	(*A* ((**S** *a* **A**) (*b*)))	$\succ_{(3)}$
(*A* ((**S** *a* **A**) (*b*)))	$\Rightarrow_{(4)}$	-5-	(*A* (**A** (*b*)))	$\succ_{(2)}$
(*A* ((**S** *a* (*a*)) (*b*)))	$\Rightarrow_{(2)}$	-6-	(*A* (**A S**))	$\succ_{(1)}$
(**A** (((*b*) *a* (*a*)) (*b*)))	$\Rightarrow_{(4)}$	-7-	(**A S**)	$\succ_{(1)}$
((*a*) (((*b*) *a* (*a*)) (*b*)))		-8-	**S**	
а			*б*	

Рис. 6.5. Правый вывод и правый разбор цепочки ((*a*) (((*b*) *a* (*a*)) (*b*)))

По определению, правый разбор цепочки *w* в КС-грамматике $G = (N, \Sigma, P, S)$ — это последовательность правил, с помощью которых можно свернуть цепочку *w* к начальному символу грамматики *S*.

Допустим, что мы построили дерево вывода цепочки *w*. Тогда в терминах деревьев выводов правый анализ цепочки *w* представляет собой последовательное определение основ и их удалений (свертки). Такой процесс сводит дерево вывода с кроной *w* к одной вершине, помеченной основным символом грамматики *S*.

Так как в начале восходящего синтаксического анализа цепочки *w* имеется только крона (цепочка *w*), то это равносильно процессу построения дерева вывода, начинающемуся с одной только кроны и идущему от листьев к корню. Поэтому с процессом порождения правого разбора часто ассоциируется термин "восходящий" анализ.

Пример 6.3

🗁 Рассмотрим правый вывод цепочки $i + i * i$ в грамматике G_0:

$$E \Rightarrow_{(1)} E + T \Rightarrow_{(3)} E + T * P \Rightarrow_{(5)} E + T * i \Rightarrow_{(4)} E + P * i \Rightarrow_{(5)}$$
$$E + i * i \Rightarrow_{(2)} T + i * i \Rightarrow_{(4)} P + i * i \Rightarrow_{(5)} i + i * i.$$

Записав в обратном порядке последовательность правил, примененных в выводе 13545245, получаем правый разбор цепочки $i + i * i$, равный 54254531. 🗀

На рис. 6.6 и 6.7 проиллюстрированы процессы свертки и восходящего построения деревьев вывода и разбора для цепочки $i + i * i$.

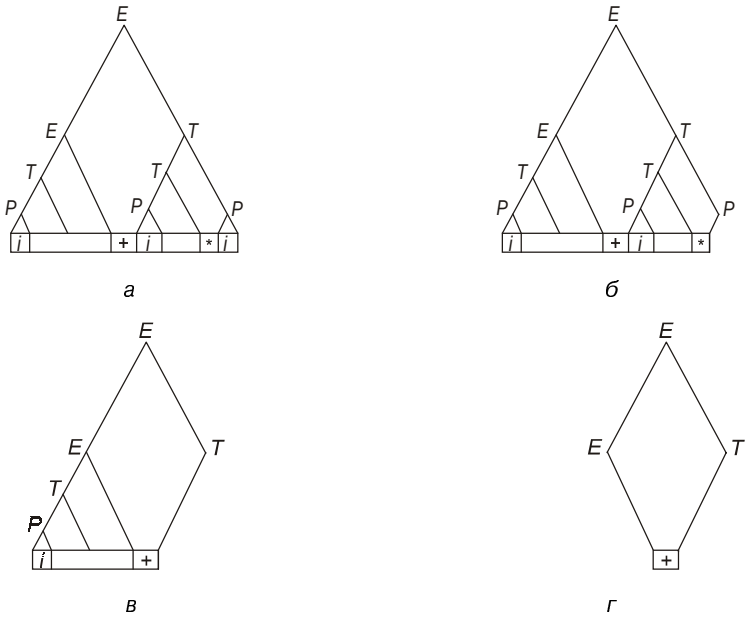


Рис. 6.6. Процесс свертки дерева вывода при восходящем анализе цепочки

На рис. 6.6, *а* изображено дерево вывода цепочки $i + i * i$ (его несколько необычная форма объясняется удобством иллюстрации процессов свертки и синтеза деревьев). Первая основа, определяемая при анализе, — символ i — по правилу грамматики (5) $P \rightarrow i$ должна быть свернута к нетерминалу P. После выполнения свертки дерево имеет вид, приведенный на рис. 6.6, *б*. На рис. 6.6, *в* и *г* показан вид дерева вывода после выполнения четвертой и седьмой (по порядку) свертки. Четвертая свертка была выполнена с использованием правила (3) $T \rightarrow T * P$, а седьмая — правила (2) $E \rightarrow T$.

Рис. 6.7 иллюстрирует процесс построения дерева разбора при восходящем анализе. В начале анализа дерево содержит только крону, равную анализируемой цепочке (рис. 6.7, *а*). После определения первой по порядку основы

на ней может быть построен соответствующий фрагмент дерева (рис. 6.7, *б*). На рис. 6.7, *в* и *г* приведены частичные деревья, построенные после определения четвертой и седьмой основ, соответственно. При совместном рассмотрении рис. 6.6 и рис. 6.7 видно, что при выполнении некоторой свертки удаляемая часть дерева на рис. 6.6 точно соответствует части, включаемой в дерево на рис. 6.7.

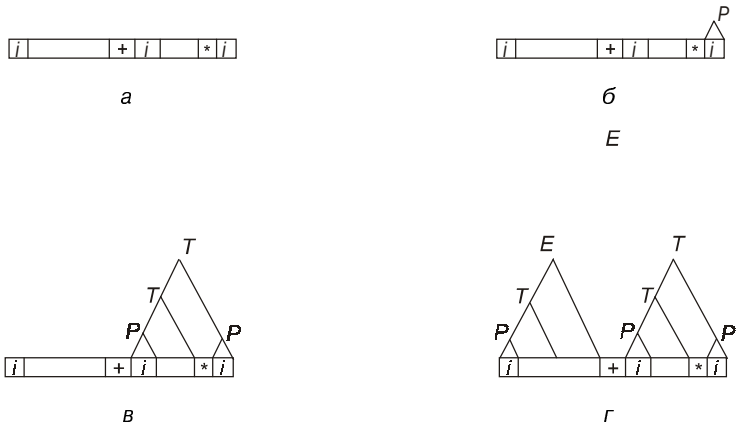


Рис. 6.7. Процесс построения дерева разбора при восходящем анализе цепочки $i + i * i$

Перейдем теперь к формальному определению восходящего анализатора.

Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. Назовем *правым анализатором* расширенный недетерминированный МП-преобразователь $M_r = (\{q\}, \Sigma, \Sigma \cup N \cup \{\perp\}, \{1, \ldots, p\}, \delta, q, \perp, \{q\})$, у которого δ определяется следующим образом:

1. Если $A \to \alpha$ — правило из P с номером i, то $(q, A, i) \in \delta(q, \varepsilon, \alpha)$.
2. $\delta(q, a, \varepsilon) = \{(q, a, \varepsilon)\}$ для всех $a \in \Sigma$.
3. $\delta(q, \varepsilon, \perp S) = \{(q, \varepsilon, \varepsilon)\}$.

Правый анализатор работает следующим образом:

- M_r переносит входные символы в магазин, используя функции δ, построенные по правилу (2) определения;
- когда наверху магазина появляется основа, анализатор может свернуть ее, используя функцию δ, построенную по правилу (1) определения, и выдать номер правила грамматики, использованного при свертке;

- M_r продолжает переносить в магазин входные символы до тех пор, пока в его верхней части не появится новая основа, которая свертывается, после чего на выход подается номер соответствующего правила грамматики;
- преобразователь действует так до тех пор, пока в магазине не останется только начальный символ грамматики с маркером дна магазина. В этом случае, используя правило (3) определения, M_r может перейти в заключительную конфигурацию с опустошением магазина.

Можно доказать [3], что для КС-грамматики $G = (N, \Sigma, P, S)$ преобразователь M_r, построенный в соответствии с его определением, определяет перевод

$$\tau_\varepsilon(M_r) = \{(w, \pi^r) \mid S \Rightarrow^*_r w\}.$$

Пример 6.4

Построим правый анализатор для грамматики G_0:

(1) $E \to E + T$ (4) $T \to P$

(2) $E \to T$ (5) $P \to i$

(3) $T \to T * P$ (6) $P \to (E)$

МП-преобразователь будет выглядеть следующим образом:

$$M_r = (\{q\}, \{i, +, *, (,)\}, \{i, +, *, (,), E, T, P, \bot\}, \{1, 2, 3, 4, 5, 6\}, \delta, q, \bot, \{q\}).$$

Определим функцию δ:

- по правилу (2) определения анализатора включаем в δ-функцию: $\delta(q, a, \varepsilon) = \{(q, a, \varepsilon)\}$ для всех $a \in \Sigma$;
- по правилу (3) определения включаем в δ-функцию: $\delta(q, \varepsilon, \bot S) = \{(q, \varepsilon, \varepsilon)\}$;
- для правила грамматики (1) $E \to E + T$ по правилу определения (1) в функцию $\delta(q, \varepsilon, E + T)$ включаем элемент $(q, E, 1)$.

Повторив аналогичные действия для всех правил грамматики, получим окончательный вид функции переходов-выходов δ:

$\delta(q, \varepsilon, E + T) = \{(q, E, 1)\}$;

$\delta(q, \varepsilon, T) = \{(q, E, 2)\}$;

$\delta(q, \varepsilon, T * P) = \{(q, T, 3)\}$;

$\delta(q, \varepsilon, P) = \{(q, T, 4)\}$;

$\delta(q, \varepsilon, (E)) = \{(q, P, 6)\}$;

$\delta(q, \varepsilon, i) = \{(q, P, 5)\}$;

$\delta(q, a, \varepsilon) = \{(q, a, \varepsilon)\}$ для всех $a \in \Sigma$;

$\delta(q, \varepsilon, \bot E,) = \{(q, \varepsilon, \varepsilon)\}$.

Для входной цепочки $i * (i + i)$ анализатор M_r может среди прочих выполнить следующую последовательность тактов:

$(q, i * (i + i), \perp, \varepsilon) \vdash (q, * (i + i), \perp i, \varepsilon) \vdash (q, * (i + i), \perp P, 5)$
$\vdash (q, * (i + i), \perp T, 54) \vdash (q, (i+ i), \perp T *, 54)$
$\vdash (q, i + i), \perp T * (, 54) \vdash (q, + i), \perp T * (i, 54)$
$\vdash (q, + i), \perp T * (P, 545) \vdash (q, + i), \perp T * (T, 5454)$
$\vdash (q, + i), \perp T* (E, 54542) \vdash (q, i), \perp T * (E +, 54542)$
$\vdash (q,), \perp T * (E + i, 54542) \vdash (q,), \perp T * (E + P, 545425)$
$\vdash (q,), \perp T * (E + T, 5454254) \vdash (q, \varepsilon, \perp T * (E + T), 5454254)$
$\vdash (q, \varepsilon, \perp T * (E), 54542541) \vdash (q, \varepsilon, \perp T * P, 545425416)$
$\vdash (q, \varepsilon, \perp T, 5454254163) \vdash (q, \varepsilon, \perp E, 54542541632)$
$\vdash (q, \varepsilon, \varepsilon, 54542541632).$

Таким образом, для входной цепочки $i * (i + i)$ анализатор M_r выдает ее правый разбор 54542541632. □

Далее в этой главе мы рассмотрим алгоритмы синтаксического анализа, применимые ко всему классу контекстно-свободных языков.

Вначале мы обсудим алгоритмы с полным возвратом. Эти алгоритмы *детерминированно* моделируют недетерминированные анализаторы. Кроме этого, мы рассмотрим табличный алгоритм Эрли.

Замечание

Возвратные алгоритмы разбора весьма неэффективны и их следует избегать в приложениях. Практически для всех языков программирования существуют легко анализируемые грамматики, к которым применимы более эффективные алгоритмы разбора. Общие алгоритмы разбора рассматриваются здесь для того, чтобы пояснить внутренние проблемы, связанные с построением и работой синтаксических анализаторов.

Перед изучением нисходящего анализа с возвратами, при котором для входной цепочки строится левый разбор, и восходящего анализа с возвратами, выдающего правый разбор, рассмотрим (неформально) детерминированное моделирование недетерминированного МП-преобразователя, использующего возвраты.

6.4. Моделирование недетерминированного МП-преобразователя

Все алгоритмы синтаксического разбора независимо от формы их описания моделируют работу некоторого преобразователя с магазинной памятью. В основе нисходящего M_l и восходящего M_r анализаторов лежат недетерминированные МП-преобразователи. Рассмотрим, каким образом

можно детерминированно моделировать работу недетерминированного МП-преобразователя.

Недетерминированный МП-преобразователь P для входной цепочки w может выполнить конечное множество последовательностей тактов, каждая из которых также ограничена по длине. Наиболее простой способ детерминированного моделирования МП-преобразователя P состоит в том, чтобы любым образом линейно упорядочить последовательности тактов, а затем в некотором порядке промоделировать каждую такую последовательность.

Если целью анализа является получение *одного разбора* данной входной цепочки w, то, обнаружив *первую* последовательность тактов, оканчивающуюся заключительной конфигурацией, можно прекратить моделирование P. В противном случае, для получения, например, *всех разборов* входной цепочки, мы должны промоделировать *все последовательности* тактов.

Замечание

Если ни одна последовательность тактов не оканчивается заключительной конфигурацией, то придется выполнить их все.

Обычно последовательности тактов упорядочивают так, чтобы перед моделированием очередной последовательности можно было вернуться по последним тактам, сделанным преобразователем (выполняя их обратное прослеживание), к конфигурации, в которой возможен еще не испытанный альтернативный такт. Если для входа w возможны бесконечные последовательности тактов, осуществить полное прямое моделирование МП-преобразователя P невозможно. Для исключения такой ситуации на преобразователь необходимо наложить некоторые ограничения.

МП-автомат $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ называется *незацикливающимся*, если для каждой цепочки $w \in \Sigma^*$ найдется такая константа k_w, что если $(q_0, w, Z_0) \vdash^m (q, x, \gamma)$, то $m \le k_w$.

МП-преобразователь называется *незацикливающимся*, если лежащий в его основе МП-автомат тоже незацикливающийся.

Можно доказать [3], что левый анализатор не зацикливается тогда и только тогда, когда грамматика G *не леворекурсивна*, а правый анализатор не зацикливается, когда G *не содержит циклов и* ε*-правил.*

Пример 6.5

Построим левый анализатор для грамматики G с правилами:

(1) $S \rightarrow aSbS$

(2) $S \rightarrow aS$

(3) $S \rightarrow c$

Левый анализатор для заданной грамматики — это МП-преобразователь P с функцией переходов-выходов δ, которая определяется следующим образом:

$\delta(q, a, S) = \{(q, aSbS, 1), (q, S, 2)\}$,

$\delta(q, c, S)=\{(q, S, 3)\}$,

$\delta(q, b, b)=\{(q, \varepsilon, \varepsilon)\}$,

$\delta(q, c, c)=\{(q, \varepsilon, \varepsilon)\}$.

Разберем при помощи МП-преобразователя P входную цепочку *aacbac*. На рис. 6.8 и 6.9 изображены соответственно левое и правое поддеревья дерева, представляющего всевозможные последовательности тактов, которые может сделать преобразователь P для этого входа.

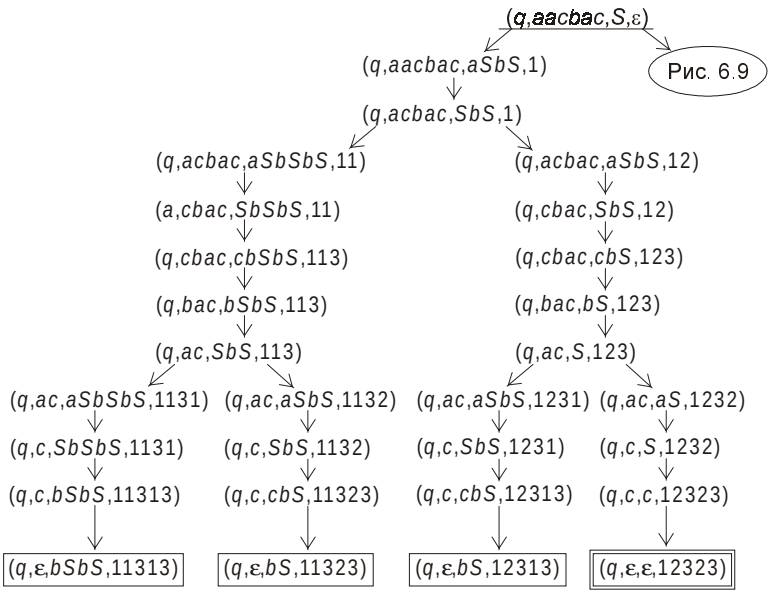


Рис. 6.8. Левое поддерево тактов МП-преобразователя для цепочки *aacbac*

Вершина дерева конфигураций представляет собой начальную конфигурацию (q, *aacbac*, S, ε).

Функции переходов преобразователя P показывают, что из начальной конфигурации можно перейти в две следующие конфигурации: (q, *aacbac*, *aSbS*, 1) или (q, *aacbac*, *aS*, 2). Из конфигурации (q, *aacbac*, *aSbS*, 1) анализатор P детерминированно переходит в конфигурацию (q, *acbac*, *SbS*, 1), а из конфигурации (q, *aacbac*, *aS*, 2) — в конфигурацию (q, *acbac*, S, 2).

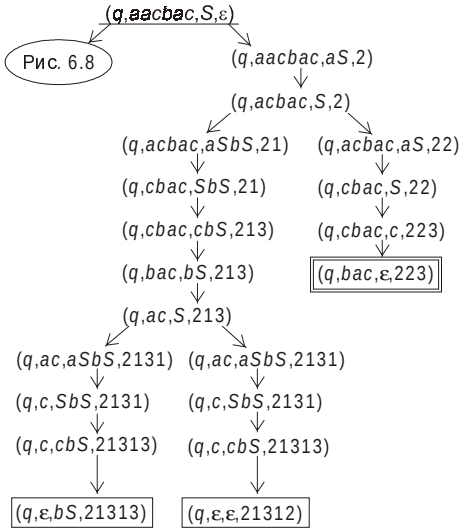


Рис. 6.9. Правое поддерево тактов МП-преобразователя для цепочки *aacbac*

Далее из (*q*, *acbac*, *SbS*, 1) преобразователь *P* может перейти в конфигурацию (*q*, *acbac*, *aSbSbS*, 11) или в (*q*, *acbac*, *aSbS*, 12). Если *P* находится в конфигурации (*q*, *aacbac*, *aS*, 2), то он может перейти в конфигурацию (*q*, *acbac*, *aSbS*, 21) или в (*q*, *acbac*, *aS*, 22) и т. д.

Один из способов нахождения всех разборов данной входной цепочки состоит в определении всех допускающих конфигураций, достижимых из начальной в дереве конфигураций. Это можно сделать, прослеживая всевозможные пути, начинающиеся в начальной вершине и заканчивающиеся в конфигурации, из которой дальнейший переход невозможен. Введем порядок, в котором будут перебираться пути, упорядочив выборы очередных тактов, доступных преобразователю *P*, для каждой комбинации состояния, входного символа и верхнего символа магазина. Пусть в случае, когда применима функция перехода $\delta(q, a, S)$, в качестве первого выбора применяется ее значение (*q*, *aSbS*, 1), а (*q*, *S*, 2) используется в качестве второго выбора.

Определим теперь все допускающие конфигурации для рассматриваемой цепочки, прослеживая всевозможные конфигурации для преобразователя *P*.

Из начальной конфигурации (*q*, *aacbac*, *S*, ε), делая первый выбор, мы переходим в конфигурацию (*q*, *aacbac*, *aSbS*, 1), а затем в конфигурацию (*q*, *acbac*, *SbS*, 1). Далее из конфигурации (*q*, *acbac*, *SbS*, 1), используя $\delta(q, a, S) = (q, aSbS, 1)$, $\delta(q, c, S) = (q, S, 3)$, $\delta(q, b, b) = (q, \varepsilon, \varepsilon)$ и $\delta(q, c, c) = (q, \varepsilon, \varepsilon)$, преобразователь переходит в конфигурацию (*q*, *ac*, *SbS*, 113), после чего детерминированно достигает конфигурации (*q*, ε, *SbS*, 11313). Эта конфигурация является заключительной, но не является *допускающей* (на рисунках допускающие заключительные конфигурации заключены в двойные рамки, а недопускающие — в одинарные).

Чтобы определить, существует ли другая заключительная конфигурация, нужно вернуться по дереву конфигураций, пока не встретится конфигурация, для которой возможен другой, еще не рассмотренный выбор очередного такта. В данном случае это конфигурация (q, ac, SbS, 113).

Для продолжения моделирования мы должны уметь восстановить конфигурацию (q, ac, SbS, 113) по конфигурации (q, ε, SbS, 11313). Возврат должен включать обратный сдвиг входной головки, восстановление предыдущего содержимого магазина и удаление выходных символов, выданных при переходе от конфигурации преобразователя (q, ac, SbS, 113) к конфигурации (q, ε, SbS, 11313).

Восстановив конфигурацию, мы должны сделать новый выбор очередного такта, используя значение $\delta(q, a, S) = (q, S, 2)$. Выполняя дальнейшее моделирование преобразователя, получим заключительную, но не допускающую конфигурацию (q, ε, bS, 12313).

Продолжая моделирование, мы получим *допускающую конфигурацию* (q, ε, ε, 12323) и можем выдать левый разбор входной цепочки 12323.

Если нам нужен только один разбор входной цепочки, то на этом можно остановиться. Но если нас интересуют все разборы, то моделирование нужно продолжить. Вторая допускающая конфигурация — это конфигурация (q, ε, ε, 21323), соответствующая правому разбору 21323.

Если входная цепочка синтаксически неверна, то придется рассмотреть всевозможные последовательности тактов. Если исчерпаны все последовательности тактов, а допускающая конфигурация не обнаружена, то преобразователь должен выдать сообщение об ошибке. □

Приведенные действия по моделированию иллюстрируют характерные черты алгоритмов, которые иногда называют *недетерминированными* (или *переборными*): на некоторых шагах таких алгоритмов допускается выбор из альтернатив и все альтернативы нужно перебрать.

Так как для выполнения синтаксического анализа обычно задается грамматика, а не МП-преобразователь, то рассмотрение восходящего и нисходящего анализа будем выполнять в терминах КС-грамматики, а не левого или правого анализатора для нее. Теперь вместо обхода последовательностей тактов анализатора мы будем *обходить* всевозможные *выводы*, *совместимые с данной входной цепочкой*. Но в любом случае, способ работы соответствующих алгоритмов состоит в последовательном моделировании преобразователя с магазинной памятью.

6.5. Алгоритм нисходящего разбора

Нисходящий синтаксический анализ характеризуется тем, что дерево разбора входной цепочки строится сверху вниз, начиная с корня. Рассмотрим этот процесс.

Пусть имеется некоторая КС-грамматика. Для каждого нетерминального символа грамматики перенумеруем в некотором порядке его альтернативы. Для продвижения по входной цепочке введем *входной указатель*, который указывает на *текущий символ* цепочки. В начальный момент входной указатель ссылается на самый левый символ входной цепочки.

Процесс построения начинается с дерева, состоящего из одной вершины, помеченной основным символом грамматики S. Это начальная *активная вершина*. Затем рекурсивно выполняются следующие действия:

1. Если активная вершина помечена нетерминальным символом A, то выбираем первую альтернативу этого символа (первую правую часть A-правила). Пусть первая альтернатива равна $X_1 \ldots X_k$. Тогда добавляем k прямых потомков вершины A с метками $X_1, \ldots, X_k$ и делаем первую вершину, помеченную символом X_1, активной. Если $k = 0$, то активной становится вершина, расположенная непосредственно справа от A.
2. Если активная вершина помечена терминальным символом a, то сравниваем текущий входной символ с a. Если они совпадают, то активной должна быть вершина, расположенная непосредственно справа от вершины, помеченной a, и требуется сдвинуть входной указатель на один символ вправо.

   Если a не совпадает с текущим входным символом, то необходимо:

   - вернуться к вершине, к которой применялось предыдущее правило;
   - установить входной указатель, в позицию, соответствующую предыдущему правилу;
   - испытать следующую альтернативу.

Если альтернатив больше нет, нужно вернуться к предыдущей вершине, являющейся предком рассматриваемой вершины, и т. д.

Замечание

При построении вывода мы пытаемся сохранить совместимость построенного дерева с входной цепочкой. Если $x\alpha$ — крона дерева, построенного к данному моменту, и цепочка α либо пуста, либо начинается нетерминальным символом, то x — префикс входной цепочки (рис. 6.10).

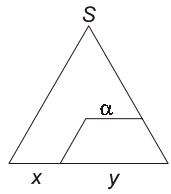


Рис. 6.10. Схема дерева вывода цепочки xy

Пример 6.6

🗁 Рассмотрим анализ цепочки *aacbac* в грамматике, имеющей правила:

(1) $S \rightarrow aSbS$

(2) $S \rightarrow aS$

(3) $S \rightarrow c$

Для удобства на рисунках, иллюстрирующих построение дерева вывода, будем отмечать позицию входного указателя (текущего символа входной цепочки) стрелкой над символом, а активную вершину дерева — стрелкой под символом.

Шаг 1. Упорядочим *S*-правила грамматики следующим образом:

$S \rightarrow aSbS \mid aS \mid c$.

Шаг 2. Построение начинается с дерева, состоящего из одной вершины (корня), помеченной символом *S* (рис. 6.11, *а*).

Шаг 3. Активная вершина помечена нетерминалом *S*. Применяем первую альтернативу *S*-правила *aSbS*, расширяя дерево так, чтобы оно было совместимо с данной входной цепочкой. Добавив к корню *S* прямых потомков *S*, *b* и *S*, получим дерево, изображенное на рис. 6.11, *б*. Активная вершина дерева помечена терминалом *a*.

Шаг 4. Так как активная вершина дерева — терминал *a* и первый символ входной цепочки тоже *a*, то передвинем входной указатель на второй входной символ и сделаем нетерминал *S*, стоящий непосредственно справа от *a*, новым активным символом (рис. 6.11, *в*).

Рекурсивно повторяя шаги 1 и 2, выполняем следующие действия: развернем активный символ *S* с помощью первой альтернативы и получим дерево, изображенное на рис. 6.11, *г*. Так как новый активный символ *a* совпадает со вторым входным символом, то передвинем входной указатель на третий входной символ (рис. 6.11, *д*). Теперь развернем самый левый нетерминал *S* на рис. 6.11, *д*. На этот раз нельзя применить ни первую, ни вторую альтернативу, т. к. в результате получится левовыводимая цепочка, *несовместимая* с входной цепочкой. Поэтому, применив третью альтернативу, придем к рис. 6.11, *е*. Очевидно, что можно передвинуть входной указатель с третьего на четвертый, а затем и на пятый входной символ, т. к. *c* и *b* — следующие два активных символа левовыводимой цепочки, представленной на рис. 6.11, *е*. Результат этих действий приводит к дереву, изображенному на рис. 6.11, *ж*.

Далее с помощью первой альтернативы для *S* развернем самый левый символ *S* на рис. 6.11, *ж* и получим дерево, приведенное на рис. 6.12, *а*.

Пятый входной символ — это символ *a*, поэтому входной указатель можно передвинуть на один символ вправо и изменить активный символ дерева (рис. 6.12, *б*).

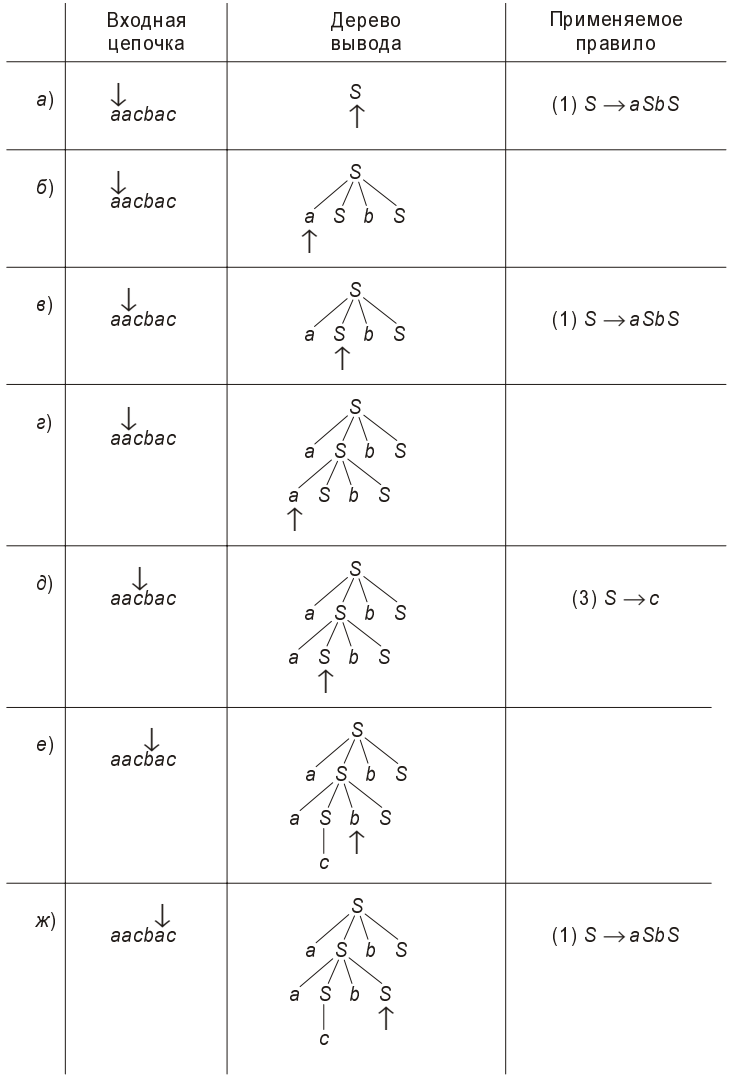


Рис. 6.11. Первые семь шагов построения дерева вывода цепочки *aacbac*

Развернем теперь активный нетерминал *S* (рис. 6.12, *б*). Здесь можно использовать только третью альтернативу. Получаем дерево, приведенное на рис. 6.12, *в*.

Очередной входной символ — *c*. Передвигаем входной указатель на один символ вправо (считаем, что конец входной цепочки обозначается маркером) и изменяем активный символ дерева (рис. 6.12, *г*). Однако дерево на рис. 6.12, *г* порождает дополнительные символы, а именно: *bSbS*, которых нет во входной цепочке. Это означает, что при поиске разбора входной цепочки мы пошли по неверному пути. □

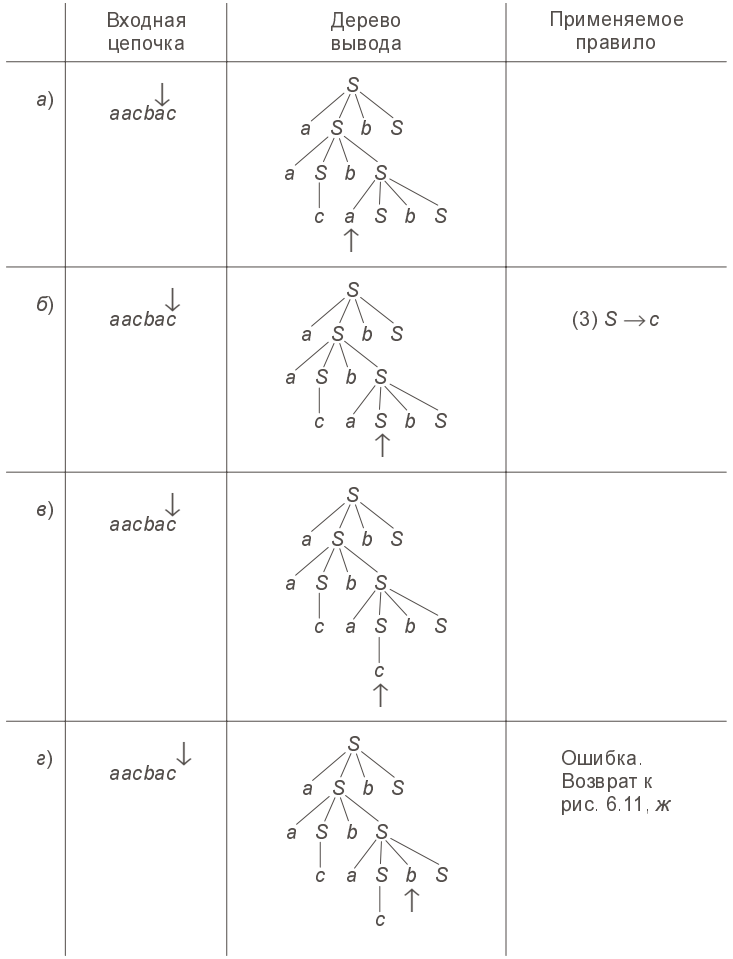


Рис. 6.12. Продолжение построения дерева вывода цепочки *aacbac*

Если сравнить построение дерева вывода с работой МП-преобразователя из примера 6.5, то увидим, что мы прошли через последовательность следующих конфигураций:

$$
\begin{aligned}
(q, aacbac, S, \varepsilon) &\vdash (q, aacbac, aSbS, 1) \vdash (q, acbac, SbS, 1)\\
&\vdash (q, acbac, aSbSbS, 11) \vdash (q, cbac, SbSbS, 11)\\
&\vdash (q, cbac, cbSbS, 113) \vdash (q, bac, bSbS, 113)\\
&\vdash (q, ac, SbS, 113) \vdash (q, ac, aSbSbS, 1131)\\
&\vdash (q, c, SbSbS, 1131) \vdash (q, c, cbSbS, 11313)\\
&\vdash (q, \varepsilon, bSbS, 11313) \vdash (q, \varepsilon, bSbS, 11313).
\end{aligned}
$$

Из последней конфигурации (q, ε, $bSbS$, 11313) *переход невозможен.*

Значит, нам придется искать какую-либо другую левовыводимую цепочку. Для этого мы должны найти (прослеживая построение дерева *в обратном порядке*) дерево, для которого *можно использовать другую альтернативу.*

Мы возвращаемся к дереву, приведенному на рис. 6.11, *ж*, где использовалась первая альтернатива, и переставляем входной указатель на пятую позицию.

Теперь применим вторую альтернативу и получим дерево на рис. 6.13, *а*. Входной указатель можно передвинуть вперед, поскольку порожденный символ *а* совпадает с входным символом *а* (рис. 6.13, *б*).

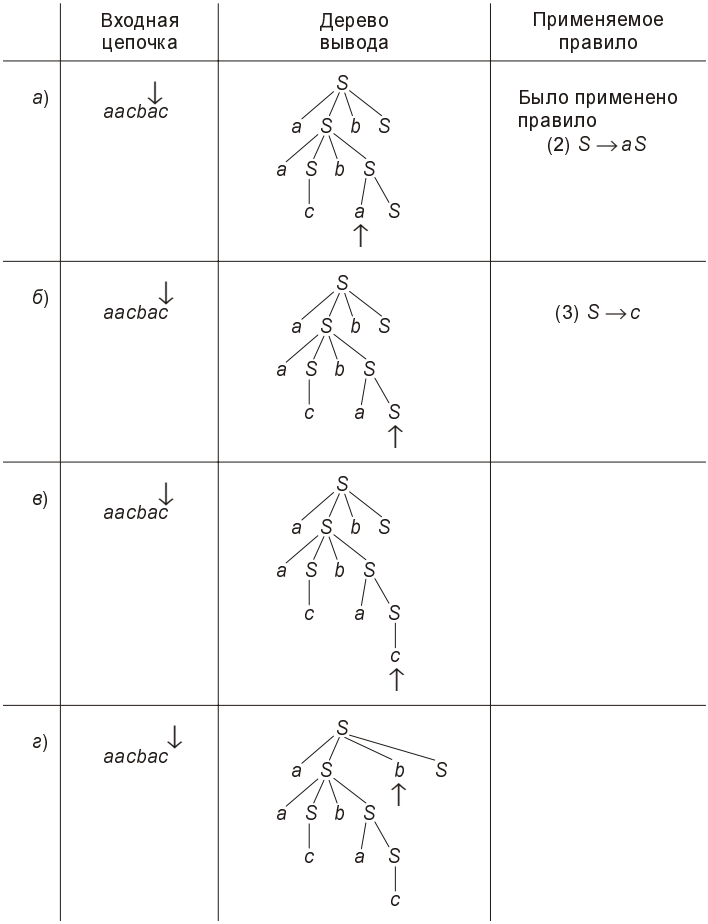


Рис. 6.13. Продолжение построения дерева вывода цепочки *aacbac*

Продолжая этот процесс, получим деревья, приведенные на рис. 6.13, *в* и *г*. Оказывается, что и этот путь построения дерева вывода неверен. Опять

выполняем обратное прослеживание и находим дерево, при построении которого возможно использование другой альтернативы. Таким деревом является дерево, изображенное на рис. 6.11, *б*.

Для продолжения построения, начиная с дерева, изображенного на рис. 6.11, *б*, необходимо передвинуть входной указатель на вторую позицию и применить к активному символу S вторую альтернативу. Процесс построения дерева приведен на рис. 6.14. На рис. 6.14, *е* изображено окончательное дерево вывода входной цепочки, и на этом мы можем остановиться. Для поиска других разборов входной цепочки нужно сделать возврат и использовать другую альтернативу.

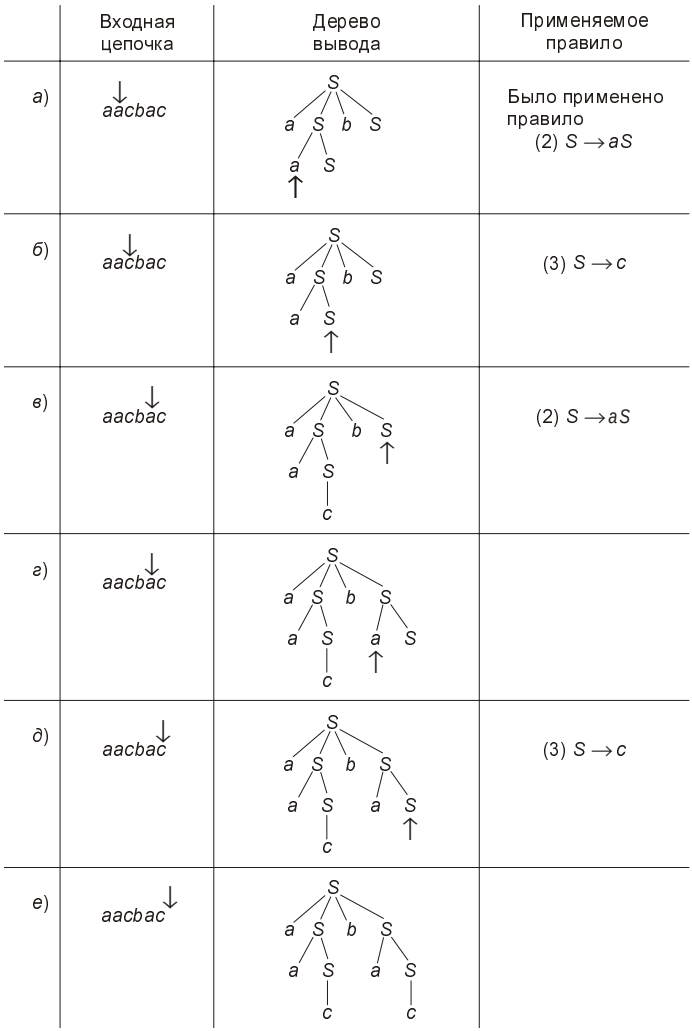


Рис. 6.14. Завершение построения дерева вывода цепочки *aacbac*

Исходная грамматика не леворекурсивна, и мы с помощью возвратов в конце концов исчерпаем все возможности, т. е. окажемся в корне и все альтернативы для S будут испытаны. В этот момент можно остановиться и, если нужный разбор не обнаружен, выдать сообщение о том, что входная цепочка синтаксически неправильна.

Заметим, что если грамматика леворекурсивная, то процесс может никогда не остановиться. Если, например, Aa — первая альтернатива для A, то она будет применяться всякий раз, когда надо развернуть A:

$$A \Rightarrow Aa \Rightarrow Aaa \Rightarrow Aaaa \Rightarrow \ldots$$

По этой причине нисходящий алгоритм разбора можно применить только к грамматикам без левой рекурсии.

Перед описанием алгоритма нисходящего разбора с возвратами введем следующие обозначения:

- В алгоритме будут использованы:
  - магазин L_1 (вершина магазина справа), в котором будет представлена текущая история проделанных выборов альтернатив и входные символы, по которым прошла входная головка алгоритма;
  - магазин L_2 (вершина магазина слева), который служит для представления текущей левовыводимой цепочки. В ней находится символ, помечающий активную вершину дерева вывода;
  - счетчик, в котором хранится текущая позиция входного указателя (предполагается, что входная цепочка длиной n символов заканчивается концевым маркером (#), который стоит на $(n + 1)$-ом месте).
- Упорядочим альтернативы для каждого нетерминала $A \in N$ и обозначим за A_j индекс j-ой альтернативы нетерминала A. Если, например, в грамматике есть A-правило с упорядоченными альтернативами $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$, то A_1 — индекс альтернативы α_1, A_2 — индекс альтернативы α_2, а A_k — индекс последней альтернативы α_k.
- Конфигурация алгоритма — это четверка (s, i, α, β), где:
  - s — состояние алгоритма;
  - i — позиция входного указателя $(1 \le i \le n + 1)$;
  - $\alpha \in (\Sigma \cup J)^*$ — содержимое магазина L_1 (J — множество индексов альтернатив);
  - β — содержимое магазина L_2.

  Начальная конфигурация алгоритма — $(q, 1, \varepsilon, S\perp)$.

Отношение перехода ($\vdash$) определяется на множестве конфигураций.

Если $(s, i, \alpha, \beta) \vdash (s', i', \alpha', \beta')$, то алгоритм из текущей конфигурации (s, i, α, β) переходит в следующую конфигурацию $(s', i', \alpha', \beta')$.

Алгоритм может находиться в одном из трех состояний:

- q — состояние нормальной деятельности;
- b — состояние возврата;
- t — заключительное состояние.

Алгоритм может выполнять следующие типы шагов:

1. *Разрастание дерева*: $(q, i, \alpha, A\beta) \vdash (q, i, \alpha A_1, \gamma_1\beta)$, где $A \to \gamma_1 \in P$ и γ_1 — первая альтернатива нетерминала A.

   Этот шаг соответствует построению куста дерева, у которого корень — самый левый нетерминал дерева и листья — первая альтернатива A-правила.

2. Успешное сравнение входного символа a_i с порожденным символом a: $(q, i, \alpha, a\beta) \vdash (q, i + 1, \alpha a, \beta)$.

   Если i-й входной символ совпадает с очередным порожденным терминальным символом, то он передается из магазина L_2 в магазин L_1 и значение позиции входного указателя увеличивается на единицу.

3. *Успешное завершение*: $(q, n + 1, \alpha, \perp) \vdash (t, n + 1, \alpha, \varepsilon)$.

   Достигнут конец входной цепочки и найдена левовыводимая цепочка, совпадающая с входной. Левый разбор входной цепочки восстанавливается по цепочке α с помощью гомоморфизма h, который определяется следующим образом:

$$h = \begin{cases} h(a) = \varepsilon, \text{ для } a \in \Sigma; \\ h(A_i) = p, \ p - \text{номер правила } A \to \gamma; \\ \gamma - i\text{-ая альтернатива нетерминала } A. \end{cases}$$

4. Неудачное сравнение входного символа a_i с порожденным символом a: $(q, i, \alpha, a\beta) \vdash (b, i, \alpha, a\beta)$.

   Алгоритм переходит в состояние возврата.

5. *Возврат по входу*: $(b, i, \alpha a, \beta) \vdash (b, i - 1, \alpha, a\beta)$ для всех $a \in \Sigma$.

   В состоянии возврата входные символы переносятся из магазина L_1 в магазин L_2.

6. *Испытание очередной* альтернативы: если существует очередная $(j + 1)$-ая альтернатива γ_{j+1} нетерминала A, то $(b, i, \alpha A_j, \beta) \vdash (q, i, \alpha A_{j+1}, \beta)$. При этом γ_j в магазине L_2 заменяется на γ_{j+1}. Возможны два случая:

   6.1. $i = 1$, $A = S$ и S имеет только j альтернатив, поэтому следующая конфигурация невозможна (всевозможные левовыводимые цепочки, совместимые с входной цепочкой w, уже исчерпаны, а ее разбор не найден);

6.2. переход в конфигурацию $(b, i, \alpha, A\beta)$ в оставшихся случаях. Все альтернативы нетерминала A исчерпаны, дальнейший возврат происходит путем удаления A_j из L_1 и замены в L_2 цепочки γ_j на A.

Далее приведем алгоритм нисходящего анализа.

Алгоритм 6.1. *Нисходящий разбор с возвратами*

Вход: Нелеворекурсивная КС-грамматика $G = (\Sigma, N, S, P)$, правила которой пронумерованы числами 1, 2, ... , p, и входная цепочка $w = a_1 a_2 \ldots a_n$.

Выход: Один левый разбор цепочки w, если таковой существует. В противном случае вернуть слово "ошибка".

Описание алгоритма:

□

1. Начиная с начальной конфигурации, пока это возможно, определять последующие конфигурации алгоритма:

$$C_0 \vdash C_1 \vdash \ldots \vdash C_i \vdash \ldots$$

2. Если последняя определенная конфигурация имеет вид $(t, n + 1, \gamma, \varepsilon)$, то в качестве результата выдать $h(\gamma)$ и остановиться.

В противном случае выдать слово "ошибка".

■

Рассмотрим работу алгоритма 6.1 на примере.

Пример 6.7

🗁 Рассмотрим анализ цепочки $i + i * i$ для грамматики G_1, представляющей собой упрощенную грамматику G_0, с правилами:

(1) $E \to E + T$ (4) $T \to P$

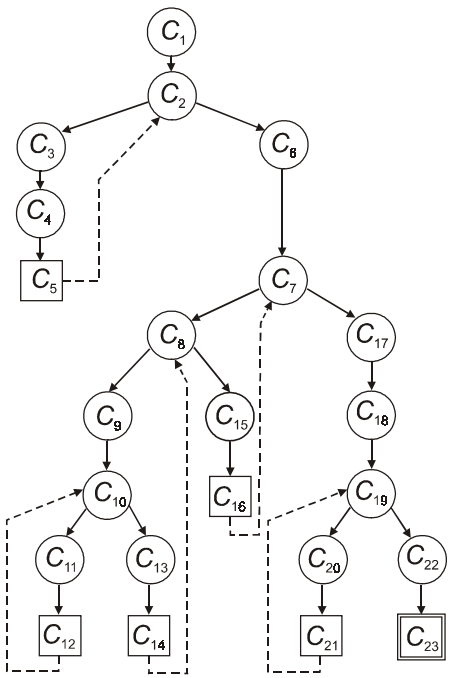
(2) $E \to T$ (5) $P \to i$

(3) $T \to T * P$

Зададим следующие индексы альтернатив:

Индекс	Альтернатива
E_1	$E + T$
E_2	T
T_1	$T * P$
T_2	P
P_1	i

При анализе работы алгоритма удобно сравнивать его работу с действиями МП-преобразователя, выполняющего левый разбор. Дерево конфигураций преобразователя для входа $i + i * i$ приведено на рис. 6.15, где такты работы преобразователя показаны стрелками, а возвраты — пунктирными стрелками.



C_1	(q,	i+i*i,	E,	ε)
C_2	(q,	i+i*i,	T+E,	1)
C_3	(q,	i+i*i,	P*T+E,	13)
C_4	(q,	i+i*i,	i*T+E,	135)
C_5	(q,	+i*i,	*T+E,	135)
C_6	(q,	i+i*i,	P+E,	14)
C_7	(q,	i*i,	E,	145)
C_8	(q,	i*i,	T+E,	1451)
C_9	(q,	i*i,	P*T+E,	14513)
C_{10}	(q,	i,	T+E,	145135)
C_{11}	(q,	i,	P*T+E,	1451353)
C_{12}	(q,	ε,	*T+E,	14513535)
C_{13}	(q,	i,	P+E,	1451354)
C_{14}	(q,	ε,	+E,	14513545)
C_{15}	(q,	i*i,	P+E,	14514)
C_{16}	(q,	i*i,	i+E,	145145)
C_{17}	(q,	i*i,	T,	1452)
C_{18}	(q,	i*i,	P*T,	14523)
C_{19}	(q,	i,	T,	145235)
C_{20}	(q,	i,	P*T,	1452353)
C_{21}	(q,	ε,	*T,	14523535)
C_{22}	(q,	i,	P,	1452354)
C_{23}	(q,	ε,	ε,	14523545)

Рис. 6.15. Работа МП-преобразователя для цепочки $i + i * i$

Выполнение алгоритма 6.1. начинается с конфигурации (q, 1, ε, $E\perp$). Для цепочки $i + i * i$ сначала выполняется шаг алгоритма типа *разрастание дерева*. Выбирается первая альтернатива для нетерминала E, и алгоритм переходит в конфигурацию (q, 1, E_1, $T + E\perp$), которая соответствует конфигурации C_2 преобразователя.

После этого алгоритм выполняет еще один шаг типа *разрастание дерева* для первой альтернативы нетерминала T, переходя в конфигурацию (q, 1, E_1T_1, $P * T + E\perp$), соответствующую конфигурации МП-преобразователя с меткой C_3.

Далее применяется первая (и единственная) альтернатива для P, после чего алгоритм переходит в конфигурацию (q, 1, $E_1T_1P_1$, $i * T + E\perp$), соответствующую конфигурации C_4 преобразователя. Первый входной символ совпадает с порожденным терминалом в верхушке магазина L_1, поэтому входной указатель передвигается на второй символ и алгоритм переходит в конфигурацию (q, 2, $E_1T_1P_1i$, $* T + E\perp$).

Второй символ входной цепочки (+) не совпадает с терминалом, находящимся в вершине магазина L_1 (*), поэтому алгоритм переходит в конфигурацию (b, 2, $E_1T_1P_1i$, * $T + E\perp$), соответствующую конечной (*недопускающей*) конфигурации C_5 преобразователя.

Для продолжения разбора МП-преобразователь должен выполнить возврат к конфигурации C_4, с которой может быть продолжен анализ. Алгоритм также выполняет возврат. Сначала выполняется шаг *возврат по входу*, и алгоритм из конфигурации (b, 2, $E_1T_1P_1i$, * $T + E\perp$) переходит в конфигурацию (b, 1, $E_1T_1P_1$, i * $T + E\perp$), соответствующую конфигурации C_4 преобразователя, а затем выполняется шаг (6.2), переводя алгоритм в конфигурацию (b, 1, E_1T_1, P * $T + E\perp$). Полученная конфигурация соответствует конфигурации C_3 МП-преобразователя, т. е. алгоритм завершил возврат, скорректировав содержимое магазина и входной указатель.

Используем вторую альтернативу для нетерминала T. Алгоритм по правилу (6.1) перейдет в конфигурацию (q, 1, E_1T_2, $P + E\perp$), соответствующую конфигурации C_6 преобразователя.

Далее выполняются следующие шаги алгоритма:

(q, 1, E_1T_2, $P + E\perp$) — первая альтернатива для P;

$\vdash$ (q,1, $E_1T_2P_1$, $i + E\perp$) — $a_i = a$;

$\vdash$ (q, 2, $E_1T_2P_1$ i, $+ E\perp$) — $a_i = a$;

$\vdash$ (q, 3, $E_1T_2P_1$ i +, $E\perp$) — первая альтернатива для E;

$\vdash$ (q, 3, $E_1T_2P_1$ $i + E_1$, $T + E\perp$) — первая альтернатива для T;

$\vdash$ (q, 3, $E_1T_2P_1$ $i + E_1T_1$, P * $T + E\perp$) — первая альтернатива для P;

$\vdash$ (q, 3, $E_1T_2P_1$ $i + E_1T_1P_1$, i * $T + E\perp$) — $a_i = a$;

$\vdash$ (q, 4, $E_1T_2P_1$ $i + E_1T_1P_1$ i, * $T + E\perp$) — $a_i = a$;

$\vdash$ (q, 5, $E_1T_2P_1$ $i + E_1T_1P_1$ i *, $T + E\perp$) — первая альтернатива для T;

$\vdash$ (q, 5, $E_1T_2P_1$ $i + E_1T_1P_1$ i * T_1, P * $T + E\perp$) — первая альтернатива для P;

$\vdash$ (q, 5, $E_1T_2P_1$ $i + E_1T_1P_1$ i * T_1P_1, i * $T + E\perp$) — $a_i = a$;

$\vdash$ (q, 6, $E_1T_2P_1i + E_1T_1P_1$ i * T_1P_1 i, * $T + E\perp$) — $a_i \neq a$.

Алгоритм достиг конфигурации, соответствующей конфигурации C_{12} МП-преобразователя. Это не допускающая конфигурация, поэтому алгоритм выполняет возврат (к конфигурации C_{10}):

(q, 6, $E_1T_2P_1$ $i + E_1T_1P_1$ i * T_1P_1 i, * $T + E\perp$)

$\vdash$ (b, 6, $E_1T_2P_1$ $i + E_1T_1P_1$ i * T_1P_1 i, * $T + E\perp$) — возврат по входу;

$\vdash$ (b, 5, $E_1T_2P_1$ $i + E_1T_1P_1$ i * T_1P_1, i * $T + E\perp$) — шаг (6.2);

$\vdash$ (b, 5, $E_1T_2P_1$ $i + E_1T_1P_1$ i * T_1, P * $T + E\perp$) — шаг (6.1);

$\vdash$ (q, 5, $E_1T_2P_1$ $i + E_1T_1P_1$ i * T_2, $P + E\perp$) — вторая альтернатива для T.

После выполнения возврата алгоритм пробует использовать вторую альтернативу для нетерминала T. Эта альтернатива также не приводит к построению вывода. Алгоритм возвращается к конфигурации, соответствующей конфигурации C_8, и пробует использовать при выводе вторую альтернативу для нетерминала T. Пройдя конфигурации, соответствующие конфигурациям C_{15} и C_{16} МП-преобразователя, алгоритм возвращается к конфигурации, соответствующей конфигурации C_{17} преобразователя. Эти действия алгоритма описываются следующей последовательностью шагов:

$(q, 5, E_1T_2P_1\, i + E_1T_1P_1\, i * T_2, P + E\perp)$

$\vdash (q, 5, E_1T_2P_1\, i + E_1T_1P_1\, i * T_2P_1, i + E\perp)$

$\vdash (q, 6, E_1T_2P_1\, i + E_1T_1P_1\, i * T_2P_1\, i, + E\perp)$

$\vdash (b, 6, E_1T_2P_1\, i + E_1T_1P_1\, i * T_2P_1\, i, + E\perp)$

$\vdash (b, 5, E_1T_2P_1\, i + E_1T_1P_1\, i * T_2P_1, i + E\perp)$

$\vdash (b, 5, E_1T_2P_1\, i + E_1T_1P_1\, i * T_2, P + E\perp)$

$\vdash (b, 5, E_1T_2P_1\, i + E_1T_1P_1\, i *, T + E\perp)$

$\vdash (b, 4, E_1T_2P_1\, i + E_1T_1P_1\, i, * T + E\perp)$

$\vdash (b, 3, E_1T_2P_1\, i + E_1T_1P_1, i * T + E\perp)$

$\vdash (b, 3, E_1T_2P_1\, i + E_1T_1, P * T + E\perp)$

$\vdash (q, 3, E_1T_2P_1\, i + E_1T_2, P + E\perp)$

$\vdash (q, 3, E_1T_2P_1\, i + E_1T_2P_1, i + E\perp)$

$\vdash (q, 4, E_1T_2P_1\, i + E_1T_2P_1\, i, + E\perp)$

$\vdash (b, 4, E_1T_2P_1\, i + E_1T_2P_1\, i, + E\perp)$

$\vdash (b, 3, E_1T_2P_1\, i + E_1T_2P_1, i + E\perp)$

$\vdash (b, 3, E_1T_2P_1\, i + E_1T_2, P + E\perp)$

$\vdash (b, 3, E_1T_2P_1\, i + E_1, T + E\perp)$

$\vdash (q, 3, E_1T_2P_1\, i + E_2, T\perp).$

Теперь алгоритм проходит конфигурации, соответствующие конфигурациям C_{18}, C_{19}, C_{20} и C_{21} преобразователя, возвращается к конфигурации C_{19} и, наконец, переходит в заключительную конфигурацию:

$(q, 3, E_1T_2P_1\, i + E_2, T\perp)$

$\vdash (q, 3, E_1T_2P_1\, i + E_2T_1, P * T\perp)$

$\vdash (q, 3, E_1T_2P_1\, i + E_2T_1P_1, i * T\perp)$

$\vdash (q, 4, E_1T_2P_1\, i + E_2T_1P_1\, i, * T\perp)$

$\vdash (q, 5, E_1T_2P_1\, i + E_2T_1P_1\, i *, T\perp)$

$\vdash (q, 5, E_1T_2P_1\, i + E_2T_1P_1\, i * T_1, P * T\perp)$

$\vdash (q, 5, E_1T_2P_1\ i + E_2T_1P_1\ i * T_1P_1, i * T\perp)$

$\vdash (q, 6, E_1T_2P_1\ i + E_2T_1P_1\ i * T_1P_1\ i, * T\perp)$

$\vdash (b, 6, E_1T_2P_1\ i + E_2T_1P_1\ i * T_1P_1\ i, * T\perp)$

$\vdash (b, 5, E_1T_2P_1\ i + E_2T_1P_1\ i * T_1P_1, i * T\perp)$

$\vdash (b, 5, E_1T_2P_1\ i + E_2T_1P_1\ i * T_1, P * T\perp)$

$\vdash (q, 5, E_1T_2P_1\ i + E_2T_1P_1\ i * T_2, P\perp)$

$\vdash (q, 5, E_1T_2P_1\ i + E_2T_1P_1\ i * T_2P_1, i\perp)$

$\vdash (q, 6, E_1T_2P_1\ i + E_2T_1P_1\ i * T_2P_1i, \perp)$

$\vdash (t, 6, E_1T_2P_1\ i + E_2T_1P_1\ i * T_2P_1i, \varepsilon).$

Работа алгоритма завершена успешно. Учитывая, что гомоморфизм h задается в данном случае как $h(E_1) = 1$, $h(E_2) = 2$, $h(T_1) = 3$, $h(T_2) = 4$, $h(P_1) = 5$, то цепочка $E_1T_2P_1i + E_2T_1P_1i * T_2P_1i$, построенная в магазине L_1, после преобразования в левый разбор будет иметь вид 14523545. □

Можно доказать [3], что алгоритм 6.1 действительно дает левый разбор цепочки w в соответствии с грамматикой G, если он существует. В этой же книге доказаны две теоремы, определяющие временную и емкостную сложности нисходящего анализатора:

1. Существует такая константа c, что алгоритм 6.1 для входной цепочки w длины $n \geq 1$ использует не более cn ячеек, если для каждого символа из обеих магазинных цепочек, входящих в конфигурации, требуется только одна ячейка.
2. Существует такая константа c, что алгоритм 6.1 при обработке входной цепочки w длины $n \geq 1$ делает не более c^n элементарных операций, при условии, что вычисление одного шага алгоритма 6.1 требует фиксированного числа элементарных операций.

Основной недостаток алгоритма нисходящего анализа с возвратами — это выполнение большого числа шагов. Для ускорения работы алгоритма можно выполнить следующее:

- упорядочить правила грамматики так, чтобы наиболее вероятные альтернативы испытывались первыми;
- заглядывать вперед на k входных символов, чтобы определить, надо ли использовать данную альтернативу.

6.6. Алгоритм восходящего разбора

В отличие от алгоритма 6.1, алгоритм восходящего разбора начинает строить дерево с листьев (входных символов) и пытается построить дерево разбора, *поднимаясь* от листьев к корню, перебирая всевозможные обращенные правые выводы, совместимые с входной цепочкой.

Неформально работу анализатора можно описать следующим образом:

1. Анализатор рассматривает цепочку, расположенную в верхней части магазина, и определяет, совпадает ли она с правой частью какого-либо правила грамматики. Если совпадение найдено, то производится свертка, заменяющая верхние символы магазина левой частью соответствующего правила.

   В случае совпадения верхушки магазина с правыми частями нескольких правил грамматики, правила упорядочиваются произвольным образом и применяется первое из них.

2. Если свертка невозможна, то в магазин переносится очередной входной символ и процесс продолжается. В любом случае перед переносом всегда проверяется возможность выполнения свертки.

3. Если анализатор достиг конца цепочки, а свертка все еще невозможна, то он возвращается к последнему шагу, на котором она была сделана. Если при таком возврате возможна другая свертка, то она выполняется и процесс продолжается.

Замечание

Анализатор, основанный на выполнении двух операций: свертки и переноса входного символа в магазин, часто называют алгоритмом типа "перенос-свертка".

Рассмотрим на примере работу восходящего анализатора с возвратами.

Пример 6.8

🗁 Пусть задана грамматика с правилами:

(1) $S \rightarrow AB$

(2) $A \rightarrow ab$

(3) $B \rightarrow aba$

Построим разбор входной цепочки *ababa*, выполняя восходящий анализ.

1. Сначала перенесем в магазин первый символ входной цепочки — символ *a*. Так как свертка пока невозможна, то в магазин помещаем следующий символ — *b*.

   Теперь наверху магазина находится цепочка *ab*, которая является правой частью второго правила. Заменив ее на левую часть этого правила (нетерминал *A*), получим частичное дерево, приведенное на рис. 6.16, *а*.

2. Так как символ *A*, стоящий теперь в вершине магазина не является основой, то перенесем в магазин очередной входной символ — символ *a*. Свертка опять невозможна, поэтому перенесем в магазин символ *b*.

3. В результате магазин содержит цепочку *Aab,* и мы можем свернуть *ab* к *A*. На рис. 6.16, *б* изображено полученное частичное дерево.
4. Переносим в магазин последний входной символ *a* и обнаруживаем, что свертка невозможна. Возвращаемся к последней позиции, в которой была сделана свертка, когда в магазине была цепочка *Aab* (верхний символ магазина находится справа).

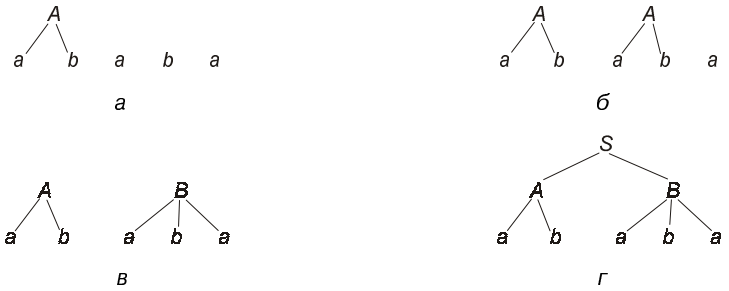


Рис. 6.16. Построение дерева вывода цепочки *ababa*

5. Так как другая свертка здесь невозможна (альтернативы нет), то выполним перенос входного символа. В магазине окажется цепочка *Aaba*. Верхушку магазина *aba*, используя правило (3), можно свернуть к *B*, получив при этом дерево, изображенное на рис. 6.16, *в*. Далее свернем *AB* в *S* и получим завершенное дерево, показанное на рис. 6.16, *г*.

□

Рассмотренный в примере метод выполнения разбора с возвратами можно интерпретировать, как процедуру прослеживания всевозможных последовательностей тактов недетерминированного *правого* анализатора для данной грамматики.

Так же, как и в случае нисходящего разбора, надо избегать ситуаций, в которых число возможных последовательностей тактов бесконечно:

- если грамматика содержит циклы, т. е. выводы вида $A \Rightarrow^+ A$ для некоторого нетерминала A, то число частичных деревьев в этом случае может быть бесконечным;
- если грамматика имеет ε-правила, то можно проделать произвольное число сверток, при которых пустая цепочка "свертывается" к нетерминалу.

Перед описанием алгоритма восходящего анализа приведем используемые обозначения и предварительные действия, необходимые для его работы.

Алгоритм использует два магазина:

- в магазине L_1 будет храниться цепочка терминалов и нетерминалов, из которой выводится часть входной цепочки, расположенная слева от входного указателя;

- в магазине L_2 будет храниться история переносов и сверток, необходимых для получения из входной цепочки содержимого магазина L_1.

Алгоритм описывается в терминах конфигураций, подобных тем, что использовались в алгоритме 6.1. Для конфигурации (s, i, α, β) введем следующие обозначения:

- s — состояние алгоритма;
- i — текущая позиция входного указателя (как и ранее, $(n + 1)$-ый входной символ — это правый концевой маркер ($\perp$));
- α — содержимое магазина L_1 (верх которого расположен справа);
- β — содержимое магазина L_2 (верх которого расположен слева).

Начальная конфигурация алгоритма — $(q, 1, \perp, \varepsilon)$.

Перед работой алгоритма правила исходной грамматики должны быть упорядочены произвольным образом.

Как и раньше, алгоритм может находиться в одном из трех состояний: q, b или t.

Алгоритм 6.2. *Восходящий разбор с возвратами*

Вход: КС-грамматика $G = (\Sigma, N, S, P)$ без циклов и ε-правил. Правила из P перенумерованы числами 1, 2, ... , p. Входная цепочка $w = a_1 a_2 \ldots a_n$ $(n \geq 1)$.

Выход: Один обращенный правый разбор цепочки w, если он существует. В противном случае — слово "ошибка".

Описание алгоритма:

□

1. *Свертка*: $(q, i, \alpha\beta, \gamma) \vdash (q, i, \alpha A, j\gamma)$ при условии, что $A \to \beta$ — правило из P с номером j и β — первая правая часть, которая является суффиксом цепочки $\alpha\beta$. Номер правила записывается в L_2.

   Если сохраняется возможность выполнить свертки, то повторить шаг 1.

   В противном случае перейти к шагу 2.

2. *Перенос*: $(q, i, \alpha, \gamma) \vdash (q, i + 1, \alpha a_i, s\gamma)$.

   Входной символ переносится в магазин L_1, позиция входного указателя увеличивается на 1, и в магазин L_2 записывается символ s, обозначающий перенос.

   Если $i \neq n + 1$, то вернуться к шагу 1.

   Если $i = n + 1$, то перейти к шагу 3.

3. *Допуск*: $(q, n + 1, \perp S, \gamma) \vdash (t, n + 1, \perp S, \gamma)$.

   Выдать $h(\gamma)$ — обращенный правый разбор цепочки w, где h — гомоморфизм, определенный равенствами: $h(s) = \varepsilon$, $h(j) = j$ для всех номеров правил. *Останов.*

Если шаг 3 неприменим, перейти к шагу 4.

4. *Переход в состояние возврата*: $(q, n + 1, \alpha, \gamma) \vdash (b, n + 1, \alpha, \gamma)$ при условии, что $\alpha \neq \perp S$.

5. *Возврат*:

   5.1. $(b, i, \alpha A, j\gamma) \vdash (q, i, \alpha' B, k\gamma)$, если $A \to \beta$ — правило из P с номером j, а следующее правило в упорядочении, правая часть которого является суффиксом цепочки $\alpha\beta$, — это правило $B \to \beta'$ с номером k (заметим, что $\alpha\beta = \alpha'\beta'$).

   Происходит возврат к предыдущей свертке, и делается попытка выполнить свертку с помощью следующей альтернативы.

   Перейти к шагу 1.

   5.2. $(b, n + 1, \alpha A, j\gamma) \vdash (b, n + 1, \alpha\beta, \gamma)$, если $A \to \beta$ — правило из P с номером j и для цепочки $\alpha\beta$ нет никакой другой свертки.

   Если других сверток не существует, необходимо вернуть данную свертку и продолжить возврат, оставляя входной указатель на позиции $n + 1$.

   Перейти к шагу 5.

   5.3. $(b, i, \alpha A, j\gamma) \vdash (q, i + 1, \alpha\beta a, s\gamma)$, если $i \neq n + 1$, $A \to \beta$ — правило из P с номером j и для цепочки $\alpha\beta$ нет никакой другой свертки.

   Произошел возврат к предыдущей свертке и, если других сверток нет, попробовать сделать перенос. Здесь символ $a = a_i$ переносится в магазин L_1, а символ s записывается в магазин L_2.

   Перейти к шагу 1.

   5.4. $(b, i, \alpha a, s\gamma) \vdash (b, i - 1, \alpha, \gamma)$, если наверху магазина L_2 находится символ переноса s.

   В позиции i исчерпаны все альтернативы и нужно вернуть операцию переноса. Входной указатель сдвигается влево, терминальный символ удаляется из L_1, а символ переноса s — из L_2.

   Если этот шаг невыполним, то выдать слово "ошибка".

■

Пример 6.9

🗁 Пусть задана грамматика G_1 с правилами:

(1) $E \to E + T$ (4) $T \to P$

(2) $E \to T$ (5) $P \to i$

(3) $T \to T * P$

Выполним восходящий разбор входной цепочки $i + i * i$.

Начальная конфигурация алгоритма — $(q, 1, \perp, \varepsilon)$.

Шаг 1. В магазине L_1 нет цепочки, суффикс которой совпадал бы с правой частью какого-либо правила грамматики. Переходим к шагу 2.

Шаг 2. Выполняется *перенос*, алгоритм переходит в конфигурацию $(q, 2, \perp i, s)$ и возвращается к шагу 1.

Шаг 1. Суффикс магазинной цепочки $\perp i$, равный i, — это правая часть правила с номером (5). Анализатор выполняет свертку и переходит в конфигурацию $(q, 2, \perp P, 5s)$.

Алгоритм повторяет шаг 1, пока есть возможность выполнять свертку, при этом он проходит следующие конфигурации:

$(q, 2, \perp P, 5s)$

$\vdash (q, 2, \perp T, 45s)$

$\vdash (q, 2, \perp E, 245s)$.

Далее, выполняя шаги 2 и 1, алгоритм проходит конфигурации:

$(q, 2, \perp E, 245s)$

$\vdash (q, 3, \perp E +, s245s)$ — шаг 2;

$\vdash (q, 4, \perp E + i, ss245s)$ — шаг 2;

$\vdash (q, 4, \perp E + P, 5ss245s)$ — шаг 1;

$\vdash (q, 4, \perp E + T, 45ss245s)$ — шаг 1;

$\vdash (q, 5, \perp E, s145ss245s)$ — шаг 1;

$\vdash (q, 5, \perp E *, s145ss245s)$ — шаг 2;

$\vdash (q, 6, \perp E * i, ss145ss245s)$ — шаг 2;

$\vdash (q, 6, \perp E * P, 5ss145ss245s)$ — шаг 1;

$\vdash (q, 6, \perp E * T, 45ss145ss245s)$ — шаг 1;

$\vdash (q, 6, \perp E * E, 245ss145ss245s)$.

В последней конфигурации свертки невозможны.

Алгоритм переходит сначала к шагу 2, а затем к шагу 3, т. к. позиция входного указателя равна $(n + 1)$. Шаг 3 (*допуск*) также неприменим, поэтому алгоритм, переходит в *состояние возврата*, при этом алгоритм оказывается в конфигурации $(b, 6, \perp E * E, 245ss145ss245s$.

В данной конфигурации нет возможности выполнить другую свертку, поэтому алгоритм переходит к шагу 5.2.

Шаг 5.2. Выполняется возврат к предыдущей свертке $(b, 6, \perp E * T, 45ss145ss245s)$. Шаг 5.2 повторяется, в результате чего алгоритм переходит

в конфигурацию (b, 6, $\perp E * P$, 5ss145ss245s), а затем в конфигурацию (b, 6, $\perp E * i$, ss145ss245s).

Теперь в вершине магазина находится символ, обозначающий перенос, и алгоритм выполняет два шага 5.4 алгоритма, сдвигая входной указатель влево:

(b, 6, $\perp E * i$, ss145ss245s)

$\vdash$ (b, 5, $\perp E *$, s145ss245s)

$\vdash$ (b, 4, $\perp E$, 145ss245s).

Шаг 5.3. В конфигурации (b, 4, $\perp E$, 145ss245s) невозможно выполнить никакую другую свертку, поэтому алгоритм возвращается к предыдущей свертке, а затем переходит к шагу 1:

(b, 4, $\perp E$, 145ss245s)

$\vdash$ (q, 5, $\perp E + T *$, s45ss245s),

Шаг 1. Алгоритм определяет, что свертка невозможна и делает перенос:

(q, 5, $\perp E + T *$, s45ss245s) — шаг 2;

$\vdash$ (q, 6, $\perp E + T * i$, ss45ss245s).

Шаг 1. После переноса алгоритм возвращается к шагу 1 и делает несколько сверток:

(q, 6, $\perp E + T * i$, ss45ss245s)

$\vdash$ (q, 6, $\perp E + T * P$, 5ss45ss245s)

$\vdash$ (q, 6, $\perp E + T$, 35ss45ss245s)

$\vdash$ (q, 6, $\perp E$, 135ss45ss245s).

Теперь свертки и переносы невозможны, алгоритм переходит к шагу 3.

Шаг 3.

(q, 6, $\perp E$, 135ss45ss245s)

$\vdash$ (t, 6, $\perp E$, 135ss45ss245s).

Алгоритм переходит в состояние допуска и при помощи гомоморфизма h выдает правый разбор входной цепочки 13545245. □

Можно доказать [3] две теоремы, аналогичные теоремам о нисходящем анализе:

1. Алгоритм 6.2 правильно находит правый разбор цепочки w, если он существует, и сигнализирует об ошибке в противном случае.
2. Пусть для каждого символа из магазинных цепочек, участвующих в конфигурациях алгоритма 6.2, требуется только одна ячейка, и пусть число элементарных операций, необходимых для вычисления одного

шага алгоритма 6.2, ограничено. Тогда для входной цепочки длины n алгоритму 6.2 требуются емкость памяти $c_1 n$ и время c_2^n, где c_1 и c_2 — некоторые константы.

Для ускорения работы алгоритма 6.2 можно его модифицировать.

- Можно определить множество Q всех терминальных цепочек x длиной k, которые могут следовать за нетерминалом A во всех правовыводимых сентенциальных формах. Если перед сверткой по A-правилу k символов входной цепочки, стоящих справа от входного указателя, не принадлежат множеству Q, то свертку делать не нужно.
- Можно упорядочить свертки так, чтобы *наиболее вероятные* из них делались первыми.
- Для ускорения возвратов можно в каждой точке, где возможна альтернативная свертка, запомнить информацию, позволяющую выполнить переход в нее без обратного прослеживания.

6.7. Алгоритм Эрли

Рассмотрим алгоритм синтаксического анализа, основный на понятии ситуации.

Основная идея алгоритма состоит в следующем. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика и $w = a_1 \ldots a_n$ ($w \in \Sigma^*$) — входная цепочка. Объект вида

$$[A \to X_1 X_2 \ldots X_k \bullet X_{k+1} \ldots X_m, i]$$

называется *ситуацией*, относящейся к цепочке w, если грамматика G содержит правило вида $A \to X_1 X_2 \ldots X_m$ и $0 \le i \le n$. Точка '•' между символами X_k и X_{k+1} является *метасимволом*, не принадлежащим ни множеству нетерминальных символов N, ни множеству терминалов Σ. Число k может быть любым целым числом от 0 (в этом случае точка — первый символ) до m (точка — последний символ).

Замечание

Если правило грамматики имеет вид $A \to \varepsilon$, то ситуация — это объект, имеющий вид $[A \to \bullet, i]$.

Если в грамматике для некоторых цепочек γ и δ существуют выводы

$$S \Rightarrow^* \gamma A \delta,\ \gamma \Rightarrow^* a_1 \ldots a_i \quad \text{и} \quad \alpha \Rightarrow^* a_{i+1} \ldots a_j,$$

то для каждого $0 \le j \le n$ можно построить список ситуаций I_j, такой, что $[A \to \alpha \bullet \beta, i] \in I_j$ и $0 \le i \le j$.

Способ построения ситуации позволяет утверждать, что между второй компонентой ситуации и номером списка, в котором она появляется, заключена часть входной цепочки, выводимая из α. Другие условия, налагаемые на

ситуацию, гарантируют возможность применения правила $A \to \alpha\beta$ в выводе некоторой входной цепочки, совпадающей с w до позиции j.

Список $I_0, I_1, \ldots, I_n$ назовем *списком разбора* для входной цепочки w.

Замечание

Очевидно, что $w \in L(G)$ тогда и только тогда, когда в списке разбора I_n есть ситуация вида $[S \to \alpha \bullet, 0]$.

Рассмотрим алгоритм, который по произвольной нелеворекурсивной КС-грамматике порождает список разбора любой входной цепочки.

Алгоритм 6.3. *Алгоритм Эрли*

Вход: Нелеворекурсивная КС-грамматика $G = (\Sigma, N, S, P)$ и входная цепочка $w = a_1 a_2 \ldots a_n$.

Выход: Список разбора $I_0, I_1, \ldots, I_n$ для входной цепочки w.

Описание алгоритма:

□ Сначала выполнить построение списка I_0:

1. Для каждого правила $S \to \alpha$ из P, в левой части которого находится начальный символ грамматики, включить в I_0 ситуацию $[S \to \bullet\, \alpha, 0]$.

Далее выполнять шаги 2 и 3 до тех пор, пока в список I_0 можно включать новые ситуации

2. Если $[B \to \gamma \bullet, 0] \in I_0$ (γ может быть и пустой цепочкой), включить в I_0 ситуацию $[A \to \alpha B \bullet \beta, 0]$ для всех ситуаций $[A \to \alpha \bullet B\beta, 0]$, уже входящих в I_0.
3. Пусть ситуация $[A \to \alpha \bullet B\beta, 0] \in I_0$. Для каждого правила из P вида $B \to \gamma$ включить в I_0 ситуацию $[B \to \bullet\, \gamma, 0]$, если ее там нет.

После построения списков $I_0, I_1, \ldots, I_{j-1}$ строится список I_j следующим образом:

4. Для каждой ситуации $[B \to \alpha \bullet a\beta, i]$ из I_{j-1}, для которой $a = a_j$, включить в I_j ситуацию $[B \to \alpha a \bullet \beta, i]$.

Далее выполнять шаги 5 и 6, пока в I_j можно включать новые ситуации.

5. Пусть $[A \to \alpha \bullet, i] \in I_j$. Найти в I_j ситуации вида $[B \to \alpha \bullet A\beta, k]$ и для каждой из них включить в I_j ситуацию $[B \to \alpha A \bullet \beta, k]$.
6. Пусть $[A \to \alpha \bullet B\beta, i] \in I_j$. Для каждого правила $B \to \gamma$ из P включить в I_j ситуацию $[B \to \bullet\gamma, j]$.

Замечание

Рассмотрение ситуации, в которой справа от точки стоит терминал, на шагах 2, 3, 5 и 6 не дает новых ситуаций.

■

Пример 6.10

🗁 Построим списки разбора для грамматики G_0 и входной цепочки $i + (i * i)$. Правила грамматики G_0:

(1) $E \to E + T$ (4) $T \to P$

(2) $E \to T$ (5) $P \to i$

(3) $T \to P * T$ (6) $P \to (E)$

Выполняем алгоритм 6.3 по шагам.

Шаг 1. Начальный символ грамматики E является левой частью двух первых правил. Поэтому в I_0 включаем ситуации $[E \to \bullet\ T + E, 0]$ и $[E \to \bullet\ T, 0]$.

Шаг 2. Переходим к выполнению шагов 3 и 4.

Шаг 3. Этот шаг не выполняется, т. к. в I_0 нет ситуации вида $[B \to \gamma \bullet, 0]$.

Шаг 4. Ситуация $[E \to \bullet\ T + E, 0] \in I_0$, поэтому в I_0 мы должны включить ситуацию $[T \to \bullet\ P * T, 0]$. По аналогичной причине включаем в I_0 ситуацию $[T \to \bullet\ P, 0]$.

Повторяем шаг 4 для только что включенных ситуаций и получаем окончательный вид списка разбора I_0:

$[E \to \bullet\ T + E, 0]$, $[E \to \bullet\ T, 0]$, $[T \to \bullet\ P * T, 0]$, $[T \to \bullet\ P, 0]$, $[P \to \bullet\ i, 0]$, $[P \to \bullet\ (E), 0]$.

Шаг 5. Первый символ входной цепочки ($j = 1$) — символ i, ситуация $[P \to \bullet\ i, 0] \in I_0$, поэтому в I_1 нужно включить ситуацию $[P \to i \bullet, 0]$.

Шаг 6. Выполняем шаги 7 и 8.

Шаг 7. Список I_1 содержит ситуацию $[P \to i \bullet, 0]$, для которой в списке I_0 необходимо найти ситуации вида $[B \to \alpha \bullet P\beta, k]$, где P — нетерминал рассматриваемой грамматики.

Это ситуации $[T \to \bullet\ P * T, 0]$ и $[T \to \bullet\ P, 0]$. Для первой из них в I_1 нужно включить ситуацию $[T \to P \bullet * T, 0]$, а для второй — $[T \to P \bullet, 0]$.

Для ситуации $[T \to P \bullet, 0]$ шаг 7 требуется повторить. В списке I_0 мы должны найти ситуации вида $[B \to \alpha \bullet T\beta, k]$, где T — нетерминал рассматриваемой грамматики. Это ситуации $[E \to \bullet\ T + E, 0]$ и $[E \to \bullet\ T, 0]$. Пополняем I_1 ситуациями $[E \to T \bullet + E, 0]$ и $[E \to T \bullet, 0]$.

Для ситуации $[E \to T \bullet, 0]$ шаг 7 также повторяется. В списке I_0 нужно найти ситуации вида $[B \to \alpha \bullet E\beta, k]$, где E — нетерминал грамматики. Таких ситуаций нет.

Список разбора I_1, построенный к данному моменту, содержит следующие ситуации:

$[P \to i \bullet, 0]$, $[T \to P \bullet * T, 0]$, $[T \to P \bullet, 0]$, $[E \to T \bullet + E, 0]$, $[E \to T \bullet, 0]$.

Переходим к шагу 8.

Шаг 8. Теперь для каждой ситуации из I_1, в которой перед нетерминалом стоит точка, например $[A \to \alpha \bullet B\beta, i]$, и для каждого $B \to \gamma$ — правила из P мы должны включить в список I_1 ситуацию $[B \to \bullet \gamma, j]$. Таких ситуаций в I_1 нет.

Окончательно список I_1 выглядит следующим образом:

$[P \to i \bullet, 0]$, $[T \to P \bullet * T, 0]$, $[T \to P \bullet, 0]$, $[E \to T \bullet + E, 0]$, $[E \to T \bullet, 0]$.

Переходим к построению списка ситуаций I_2.

Шаг 5. Следующий символ входной цепочки ($j = 2$) $a_2 = *$. В I_1 нужно найти ситуацию, содержащую "• *". Это $[T \to P \bullet * T, 0]$, поэтому в I_1 включаем ситуацию $[T \to P * \bullet T, 0]$.

Шаг 7. Не выполняется.

Шаг 8. Для ситуации $[T \to P * \bullet T, 0]$ мы должны включить в I_2 ситуации $[T \to \bullet P * T, 2]$ и $[T \to \bullet P, 2]$.

Повторяя шаг 8, получаем окончательный вид I_2:

$[T \to P * \bullet T, 0]$, $[T \to \bullet P * T, 2]$, $[T \to \bullet P, 2]$, $[P \to \bullet i, 2]$, $[P \to \bullet (E), 2]$.

Переходим к построению списка I_3.

Шаг 5. Очередной символ входной цепочки $a_3 = ($. Включаем в I_3 ситуацию $[P \to (\bullet E), 2]$.

Шаг 7. Не выполняется.

Шаг 8. После выполнения этого шага получаем I_3:

$[P \to (\bullet E), 2]$, $[E \to \bullet T + E, 3]$, $[E \to \bullet T, 3]$, $[T \to \bullet P * T, 3]$, $[T \to \bullet P, 3]$, $[P \to \bullet i, 3]$, $[P \to \bullet (E), 3]$.

Переходим к построению I_4.

Шаг 5. Очередной символ входной цепочки $a_4 = i$, ситуация $[P \to i \bullet, 3] \in I_3$, поэтому в I_1 нужно включить ситуацию $[P \to i \bullet, 3]$.

Теперь $I_4 = \{[P \to i \bullet, 3]\}$, и нужно перейти к шагу 7 алгоритма.

Шаг 7. Заметим, что поиск ситуации, содержащей нетерминал, перед которым стоит точка, мы должны выполнять в списке разбора, номер которого определяется вторым элементом $[P \to i \bullet, 3]$. В данном случае поиск ситуации выполняется в I_3.

В результате поиска имеем ситуации:

$[P \to (\bullet E), 2]$, $[E \to \bullet T + E, 3]$, $[E \to \bullet T, 3]$, $[T \to \bullet P * T, 3]$, $[T \to \bullet P, 3]$,

для которых мы должны включить в I_4 следующие:

$[P \to (\bullet E), 2]$, $[E \to T \bullet + E, 3]$, $[E \to T \bullet, 3]$, $[T \to P \bullet * T, 3]$, $[T \to P \bullet, 3]$.

Шаг 8. Не выполняется.

Окончательно список I_4 выглядит следующим образом:

$[P \to i \bullet, 3]$, $[P \to (\bullet E), 2]$, $[E \to T \bullet + E, 3]$, $[E \to T \bullet, 3]$, $[T \to P \bullet * T, 3]$, $[T \to P \bullet, 3]$.

Построение I_5 не составляет труда, поэтому приведем только его окончательный вид:

$[E \to T + \bullet E, 3]$, $[E \to \bullet T + E, 5]$, $[E \to \bullet T, 5]$, $[T \to \bullet P * T, 5]$, $[T \to \bullet P, 5]$, $[P \to \bullet i, 5]$, $[P \to \bullet (E), 5]$.

Шаг 5. Начнем построение списка I_6. Символ входной цепочки $a_6 = i$, ситуация $[P \to i \bullet, 5] \in I_5$, поэтому в I_6 нужно включить ситуацию $[P \to i \bullet, 5]$.

Шаг 7. Мы должны в I_5 найти ситуации, содержащие $\bullet P$.

В результате поиска получим ситуации $[T \to \bullet P * T, 5]$ и $[T \to \bullet P, 5]$, для которых нужно включить в I_6 $[T \to P \bullet * T, 5]$ и $[T \to P \bullet, 5]$.

Рассмотрение ситуации $[T \to P \bullet * T, 5]$ ничего не дает, но ситуация $[T \to P \bullet, 5]$ заставляет повторно обратиться к списку I_5 для поиска ситуаций, содержащих "$\bullet T$". Список I_5 содержит подходящие ситуации, и мы включаем в I_6 следующие:

$[E \to T \bullet + E, 5]$, $[E \to T \bullet, 5]$.

Выполняя аналогичные действия, мы пополняем список ситуацией $[E \to T + E \bullet, 3]$.

Теперь для выполнения шага 7 мы должны найти ситуацию, содержащую "$\bullet E$" в списке ситуаций I_3. Это ситуация $[P \to (E \bullet), 2]$. В I_6 включаем ситуацию $[P \to (E) \bullet, 2]$.

Окончательно список I_6 примет следующий вид:

$[P \to \bullet i, 5]$, $[T \to P \bullet * T, 5]$, $[T \to P \bullet, 5]$, $[E \to T \bullet + E, 5]$, $[E \to T \bullet, 5]$, $[E \to T + E \bullet, 3]$, $[P \to (E \bullet), 2]$.

Аналогично строится I_7, который имеет вид:

$[P \to (E) \bullet, 2]$, $[T \to P \bullet * T, 2]$, $[T \to P \bullet, 2]$, $[T \to P * T \bullet, 0]$, $[E \to T \bullet + E, 0]$, $[E \to T \bullet, 0]$.

Все списки разбора приведены на рис. 6.17.

Вторым этапом работы алгоритма Эрли является порождение по списку разбора правого разбора входной цепочки. Для простоты будем предполагать, что исходная грамматика не содержит циклов.

Таблица 6.1. *Списки разбора для примера 6.10*

I_0	I_1	I_2
[$E \to \bullet T + E$, 0]	[$E \to T\bullet + E$, 0]	[$T \to P * \bullet T$, 0]
[$E \to \bullet T$, 0]	[$E \to T\bullet$, 0]	[$T \to \bullet P * T$, 2]
[$T \to \bullet P * T$, 0]	[$T \to P\bullet * T$, 0]	[$T \to \bullet P$, 2]
[$T \to \bullet P$, 0]	[$T \to P\bullet$, 0]	[$P \to \bullet i$, 2]
[$P \to \bullet i$, 0]	[$P \to i\bullet$, 0]	[$P \to \bullet(E)$, 2]
[$P \to \bullet(E)$, 0]		
I_3	I_4	I_5
[$E \to \bullet T + E$, 3]	[$E \to T\bullet + E$, 3]	[$E \to T + \bullet E$, 3]
[$E \to \bullet T$, 3]	[$E \to T\bullet$, 3]	[$E \to \bullet T + E$, 5]
[$T \to \bullet P * T$, 3]	[$T \to P\bullet * T$, 3]	[$E \to \bullet T$, 5]
[$T \to \bullet P$, 3]	[$T \to P\bullet$, 3]	[$T \to \bullet P * T$, 5]
[$P \to \bullet i$, 3]	[$P \to i\bullet$, 3]	[$T \to \bullet P$, 5]
[$P \to (\bullet E)$, 2]	[$P \to (\bullet E)$, 2]	[$P \to \bullet i$, 5]
[$P \to \bullet(E)$, 3]		[$P \to \bullet (E)$, 5]
I_6	I_7	
[$E \to T + E\bullet$, 3]	[$E \to T\bullet + E$, 0]	
[$E \to T\bullet + E$, 5]	[$E \to T\bullet$, 0]	
[$E \to T\bullet$, 5]	[$T \to P * T\bullet$, 0]	
[$T \to P\bullet * T$, 5]	[$T \to P\bullet * T$, 2]	
[$T \to P\bullet$, 5]	[$T \to P\bullet$, 2]	
[$P \to \bullet i$, 5]	[$P \to (E) \bullet$, 2]	
[$P \to (E\bullet)$, 2]		

Алгоритм 6.4. *Построение правого разбора по списку разбора*

Вход: КС-грамматика $G = (\Sigma, N, S, P)$ без циклов, правила которой перенумерованы числами от 1 до p, входная цепочка $w = a_1 a_2 \ldots a_n$ и список разбора $I_0, I_1, \ldots I_n$ для цепочки w.

Выход: Правый разбор π цепочки w или сообщение об ошибке.

***Описание алгоритма*:**

□ Если в I_n нет ситуации вида $[S \to \alpha \bullet, 0]$, то $w \notin L(G)$ (алгоритм должен выдать сообщение об ошибке и остановиться), иначе присвоить $\pi = \varepsilon$ и выполнить рекурсивно описанную далее процедуру **R** с параметрами $([S \to \alpha \bullet, 0], n)$.

Процедура **R** $([A \to \beta \bullet, i], j)$ использует глобальную переменную π для записи правого разбора, причем запись номеров правил выполняется в *левый* конец переменной π. Действия, выполняемые процедурой, следующие:

1. Пусть $A \to \beta$ — правило грамматики с номером h. Записать h в переменную π.
2. Если $\beta = X_1 X_2 \ldots X_m$, то присвоить $k = m$ и $l = j$.
3. Повторять шаг 3 пока $k \neq 0$.
   3.1. Если $X_k \in \Sigma$, то $k = k - 1$, $l = l - 1$.
   3.2. Если $X_k \in N$, то в списке I_l найти ситуацию $[X_k \to \gamma \bullet, r]$, для которой в списке I_r имеется ситуация вида $[A \to X_1 X_2 \ldots X_{k-1} \bullet X_k \ldots X_m, i]$.
   3.3. Выполнить **R**$([X_k \to \gamma \bullet, r], l)$.
   3.4. $k = k - 1$, $l = r$.

■

Алгоритм 6.4 прослеживает правый вывод входной цепочки с помощью списка разбора, позволяющего определять правила, примененные в ходе вывода. Вызов процедуры **R** с параметрами $([A \to \beta \bullet, i], j)$ добавляет к левому концу текущего частичного разбора номер правила $A \to \beta$. Если $\beta = v_0 B_1 v_1 B_2 v_2 \ldots B_s v_s$, где $B_1 B_2 \ldots B_s$ — все вхождения нетерминалов в цепочку β, то процедура **R** определяет первое правило, использованное при развертке B_t $(1 \le t \le s)$, например $B_t \to \beta_t$, и позицию во входной цепочке w, непосредственно предшествующую первому терминальному символу, выводимому из B_t. Затем делаются рекурсивные вызовы процедуры **R** в следующем порядке:

R$([B_s \to \beta_s \bullet, i_s], j_s)$,

R$([B_{s-1} \to \beta_{s-1} \bullet, i_{s-1}], j_{s-1})$,

...

R$([B_1 \to \beta_1 \bullet, i_1], j_1)$, где $j_s = j - |v_s|$, $j_q = i_q + 1 - |v_q|$ для $1 \le q < s$.

Пример 6.11

🗁 Применим алгоритм 6.4 к списку разбора, приведенному в табл. 6.1, для получения правого разбора входной цепочки $i * (i + i)$. Для простоты будем нумеровать рекурсивные вызовы процедуры **R** (рис. 6.17).

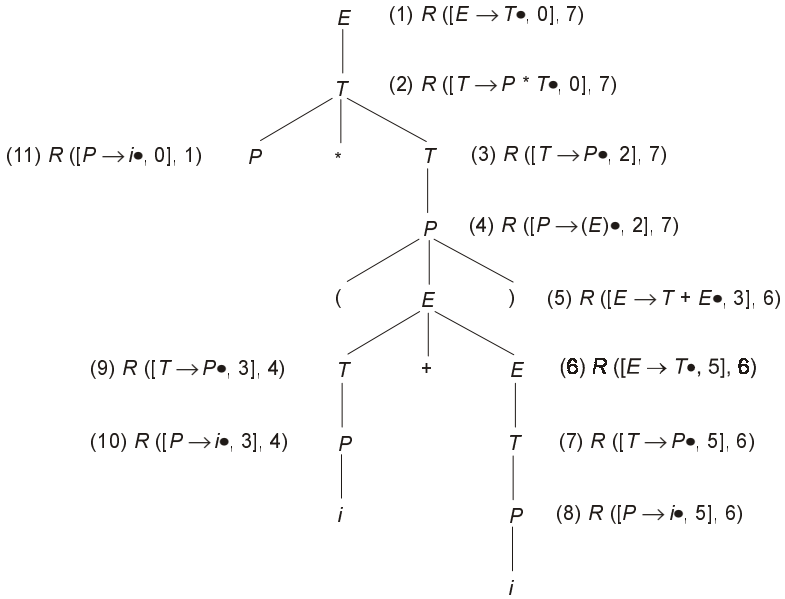


Рис. 6.17. Вызовы рекурсивной процедуры **R**

В списке I_0 существуют две ситуации $[E \to \bullet\ T + E, 0]$ и $[E \to \bullet\ T, 0]$, каждая из которых может быть использована при первом вызове **R**. Мы используем вторую из них, тогда первое обращение имеет вид:

(1) **R**($[E \to \bullet\ T, 0], 7$).

Процедура выполняет следующие шаги.

Шаг 1. Правило грамматики $E \to T$ имеет номер (2). Поместим его в π, т. е. $\pi = 2$.

Шаг 2. В правой части правила — один символ, поэтому $k = 1$. Значение переменной l равно значению второго параметра вызова, т. е. $l = 7$.

Рассматриваемый символ правой части — нетерминал, поэтому выполняется шаг 3.2 алгоритма.

Шаг 3.2. В I_7 нужно найти ситуацию $[T \to \gamma\ \bullet, r]$. Таких ситуаций две: $[T \to P * T\ \bullet, 0]$ и $[T \to P\ \bullet, 2]$. Проверим дополнительное условие выбора. Для этого мы должны в I_0 найти ситуацию, содержащую суффикс "$\bullet\ T$". Такая ситуация есть, это $[E \to \bullet\ T, 0]$. Для формирования обращения к процедуре выберем ситуацию $[T \to P * T\ \bullet, 0]$, тогда обращение к **R** будет иметь вид:

(2) **R**($[T \to P * T\ \bullet, 0], 7$).

Шаг 1. $\pi = 32$.

Шаг 2. $k = 3$, $l = 7$.

Первый рассматриваемый символ правой части правила — нетерминал T, поэтому выполняется шаг 3.2.

Шаг 3.2. В I_7 находим нужную ситуацию $[T \rightarrow P \bullet, 2]$ и выполняем очередное обращение к процедуре **R**:

(3) **R**($[T \rightarrow P \bullet, 2]$, 7).

Шаг 1. $\pi = 432$.

Шаг 2. $k = 1$, $l = 7$.

Шаг 3.2. В I_7 находим ситуацию $[P \rightarrow (E) \bullet, 2]$, удовлетворяющую условиям и выполняем обращение к **R**:

(4) **R**($[P \rightarrow (E) \bullet, 2]$, 7).

Шаг 1. $\pi = 6432$.

Шаг 2. $k = 3$, $l = 7$. Третий символ правой части правила $P \rightarrow (E)$ — это терминальный символ, поэтому далее нужно выполнить шаг 3.1.

Шаг 3.1. $k = k - 1 = 2$, $l = l - 1 = 6$.

Шаг 3.2. В I_6 находим ситуацию $[E \rightarrow \gamma \bullet, r]$. Таких ситуаций две: $[E \rightarrow T \bullet, 5]$ и $[E \rightarrow T + E \bullet, 3]$. Для проверки дополнительного условия в I_5 нужно найти ситуацию, имеющую суффикс "$\bullet E)$". Так как такой ситуации нет, то рассматриваемая ситуация не подходит. Для ситуации $[E \rightarrow T + E \bullet, 3]$ в I_3 будем искать ситуацию, имеющую суффикс "$\bullet E)$". Такая ситуация есть — это $[P \rightarrow (\bullet E), 2]$, значит, на этом шаге нужно выбрать ситуацию $[E \rightarrow T + E \bullet, 3]$ и выполнить следующее обращение к **R**:

(5) **R**($[E \rightarrow T + E \bullet, 3]$, 6).

Шаг 1. $\pi = 16432$.

Шаг 2. $k = 3$, $l = 6$.

Шаг 3.2. В I_6 находим ситуацию $[E \rightarrow \gamma \bullet, r]$, удовлетворяющую требуемому условию $[E \rightarrow T \bullet, 5]$, и выполняем обращение к **R**:

(6) **R**($[E \rightarrow T \bullet, 5]$, 6).

Шаг 1. $\pi = 216432$.

Шаг 2. $k = 1$, $l = 6$.

Шаг 3.2. В I_6 находим ситуацию $[T \rightarrow P \bullet, 5]$ и выполняем обращение к **R**:

(7) **R**($[T \rightarrow P \bullet, 5]$, 6).

Шаг 1. $\pi = 4216432$.

Шаг 2. $k = 1$, $l = 6$.

Шаг 3.2. В I_6 находим ситуацию $[P \rightarrow i \bullet, 5]$ и выполняем обращение к **R**:

(8) **R**($[P \rightarrow i \bullet, 5]$, 6).

Шаг 1. $\pi = 54216432$.

Шаг 2. $k = 1$, $l = 6$.

Шаг 3.1. $k = 0$, $l = 5$. Осуществляется выход из обращения (8) к процедуре **R**.

Продолжаем выполнять шаг 3.2 обращения (7) $k = 0$, $l = 5$.

Выполняется выход из обращения (7).

Шаг 3.2 обращения (6). $k = 0$, $l = 5$.

Ввполняется выход из обращения (6).

Шаг 3.2 обращения (6). $k = 2$, $l = 5$.

Повторяем шаг 3.2 для $k = 2$, $l = 5$. Находим нужную ситуацию $[T \rightarrow P \bullet, 3]$ и обращаемся к **R**:

(9) $\mathbf{R}([T \rightarrow P \bullet, 3], 4)$.

Шаг 1. $\pi = 454216432$.

Шаг 2. $k = 1$, $l = 4$.

Шаг 3.2 обращения (7). $k = 0$, $l = 4$. Находим нужную ситуацию $[P \rightarrow i \bullet, 3]$ и обращаемся к **R**:

(10) $\mathbf{R}([P \rightarrow i \bullet, 3], 4)$.

Шаг 1. $\pi = 5454216432$.

Шаг 2. $k = 1$, $l = 4$.

Шаг 3.1. $k = 0$, $l = 3$.

Далее выполняются выходы из процедур (10), (9), (5), (4) и (3).

Затем выполняется обращение:

(11) $\mathbf{R}([P \rightarrow i \bullet, 0], 1)$.

Шаг 1. $\pi = 55454216432$.

Шаг 2. $k = 1$, $l = 1$.

Шаг 3.1. $k = 0$, $l = 0$.

В конце своей работы алгоритм выполняет выходы из (11), (2) и (1)-го обращений и успешно завершает свою работу. Получен правый разбор $\pi = 55454216432$. □

В заключение можно заметить, что алгоритм Эрли позволяет для произвольной КС-грамматики разобрать входную цепочку за время $O(n^3)$, использовав при этом емкость памяти $O(n^2)$, где n — длина входной цепочки [3].

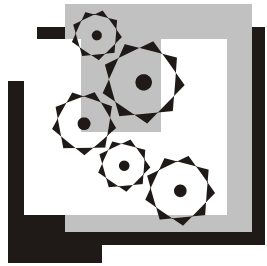
Контрольные вопросы

1. Дайте определение левого и правого разборов. Как левый и правый разборы связаны с левым и правым выводом?
2. Какова процедура построения дерева разбора при нисходящем анализе?
3. Какие основные действия выполняет левый анализатор?
4. Какова процедура построения дерева разбора при восходящем анализе?
5. Какие основные действия выполняет правый анализатор?
6. Дайте определение основы. Какие цепочки могут не иметь основы?
7. Какой МП-автомат называется незацикливающимся?
8. Для чего используются магазины L_1 и L_2 в алгоритме нисходящего разбора?
9. Как выполняется возврат в алгоритме восходящего разбора с возвратами?
10. Как используются списки разбора в алгоритме Эрли?

Упражнения

1. Постройте для заданной грамматики левый анализатор и приведите всевозможные такты его работы для входной цепочки:
   1.1. (1) $S \to aSbS$, (2) $S \to aS$, (3) $S \to c$.
   1.2. (1) $S \to (AS)$, (2) $S \to (b)$, (3) $A \to (SaA)$, (4) $A \to (a)$.
2. Постройте для заданной грамматики правый анализатор и приведите всевозможные такты его работы для входной цепочки:
   2.1. (1) $E \to E + T$, (2) $E \to T$, (3) $T \to (E)$, (4) $T \to i$.
   2.2. (1) $S \to aAb$, (2) $S \to c$, (3) $A \to bS$, (4) $A \to Bb$, (5) $B \to aA$, (6) $B \to c$.
3. Постройте списки разбора для заданных грамматик и входных цепочек:
   3.1. (1) $S \to bAb$, (2) $A \to cB$, (3) $A \to a$, (4) $B \to Aad$.
   3.2. (1) $S \to 0S11$, (2) $S \to 011$.

Выполните правый разбор, используя построенные в упражнении 2 списки разбора.



Глава 7

LL(*k*)-грамматики

При рассмотрении в *гл. 6* нисходящего анализа с возвратами были даны рекомендации по его ускорению, в одной из которых советуется заглядывать вперед на k входных символов, чтобы определить, надо ли использовать данную альтернативу. Выделим класс грамматик, которые позволяют реализовать эту рекомендацию.

7.1. Простые *LL*(1)-грамматики

Рассмотрим процесс нисходящего анализа цепочек языка на примере.

Пример 7.1

🗁 Пусть задана КС-грамматика:

(1) $S \to aA$ (3) $A \to cAd$ (5) $B \to cBf$

(2) $S \to bB$ (4) $A \to e$ (6) $B \to g$

Рассмотрим разбор входной цепочки *accedd*, которая принадлежит языку, определяемому заданной грамматикой.

Начнем с начального символа грамматики S и попытаемся породить рассматриваемую цепочку, используя левый вывод и проверяя на равенство начальные терминалы сентенциальной формы с символами цепочки.

На первом шаге порождения можно использовать первое или второе правило грамматики. Учитывая, что входная цепочка начинается с символа a, мы используем первое правило и получаем сентенциальную форму aA.

Для замены нетерминала A на следующем шаге порождения можно применить третье или четвертое правило грамматики. Для выбора правила, мы анализируем второй символ входной цепочки (первый символ был использован на предыдущем шаге). Так как рассматриваемый символ — это символ c, то для порождения выбираем третье правило грамматики, которое также начинается с символа c.

На рис. 7.1. проиллюстрирован процесс порождения цепочки *accedd*. Стрелкой помечен символ входной цепочки, анализируемый на данном шаге.

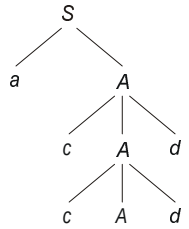


Рис. 7.1. Процесс порождения цепочки *accedd*

Особенностью рассматриваемой грамматики является то, что каждое ее правило начинается с терминального символа, причем альтернативы начинаются с различных символов, что позволяет достаточно просто построить *детерминированный* синтаксический анализатор.

Рассмотрим последовательность шагов работы такого анализатора для входной цепочки *accedd*, левый вывод которой равен 1334 (рис. 7.1).

Конфигурация, описывающая состояние анализатора (ax, $\alpha\perp$, π), включает в себя содержимое магазина $\alpha\perp$ (вершина слева), необработанную часть входной цепочки ax (a — текущий символ) и построенный к данному моменту времени разбор π.

Замечание

В [28] содержимое магазина анализатора интерпретируется как следующее утверждение о необработанной части цепочки входных символов: *вся входная цепочка допустима тогда и только тогда, когда цепочка оставшихся входных символов, включая текущий, выводима из цепочки символов, находящихся в магазине*. Например, если в процессе анализа в магазине оказалась цепочка $acAd\perp$, то это означает, что вся входная цепочка допустима тогда и только тогда, когда ее необработанная часть состоит из символов ac, за которыми идет цепочка, порождаемая нетерминалом A. Завершаться входная цепочка должна символом d.

Анализатор начинает свою работу в конфигурации (*accedd*, $S\perp$, ε), состояние магазина которой утверждает, что входная цепочка должна порождаться начальным символом грамматики.

Шаг 1. Выполняется порождение из символа S цепочки символов, начинающейся с символа a. Для этого анализатор может использовать правило грамматики (1) либо (2). Учитывая, что правило (2) порождает цепочку, начинающуюся символом b, выбираем правило (1) и переходим в конфигурацию (*accedd*, $aA\perp$, 1).

Шаг 2. В вершине магазина находится терминальный символ a. Так как этот символ *совпадает* с первым символом цепочки, мы можем вычеркнуть его из магазина и продолжить анализ входной цепочки.

Шаг 3. Текущий символ входной цепочки — символ *c* (второй символ входной цепочки), а в верхушке магазина находится нетерминал *A*. Для продолжения разбора анализатор может выбрать правило грамматики (3) или (4). Текущий символ требует, чтобы цепочка, порождаемая нетерминалом *A*, начиналась с символа *c*, поэтому выбираем правило (3). Анализатор переходит в конфигурацию (*ccedd*, *cAd*⊥, 13).

Продолжая аналогичные действия, получим последовательность шагов, приведенную в табл. 7.1 (входная цепочка разделена на две части: слева от вертикальной линии — обработанная часть, справа — необработанная, текущий символ выделен полужирным шрифтом).

Таблица 7.1. *Анализ цепочки accedd*

	Входная цепочка		Магазин	Действия анализатора
1		***a****ccedd*	*S*⊥	Применить правило 1
2		***a****ccedd*	*aA*⊥	Вычеркнуть
3	*a*	***c****cedd*	*Ad*⊥	Применить правило 3
4	*a*	***c****cedd*	*cAd*⊥	Вычеркнуть
5	*ac*	***c****edd*	*Ad*⊥	Применить правило 3
6	*ac*	***c****edd*	*cAdd*⊥	Вычеркнуть
7	*acc*	***e****dd*	*Add*⊥	Применить правило 4
8	*acc*	***e****dd*	*edd*⊥	Вычеркнуть
9	*acce*	***d****d*	*dd*⊥	Вычеркнуть
10	*acced*	***d***	*d*⊥	Вычеркнуть
11	*accedd*	ε	⊥	Допустить

Рассмотренная грамматика относится к классу простых (simple) $LL(1)$-грамматик.

КС-грамматика $G = (N, \Sigma, P, S)$ без ε-правил называется *простой LL(1)-грамматикой* (*s-грамматикой*, *разделенной грамматикой*), если для каждого $A \in N$ все его альтернативы начинаются различными терминальными символами.

Для *s*-грамматик можно сформулировать простые правила построения детерминированного магазинного автомата, выполняющего синтаксический разбор. Из рассмотренного примера видно, что для МП-автомата достаточно определить две операции: замену верхнего символа магазина цепочкой символов (правой частью соответствующего правила) и вычеркивание символа из магазина.

К сожалению, большинство конструкций языков программирования не могут быть описаны при помощи *s*-грамматик.

Пример 7.2

🗁 Рассмотрим еще одну грамматику, правила которой имеют следующий вид:

(1) $S \rightarrow aAS$ (3) $A \rightarrow cAS$

(2) $S \rightarrow b$ (4) $A \rightarrow \varepsilon$

Эта грамматика не относится к классу *s*-грамматик, т. к. имеет ε-правило. Несмотря на это, попробуем ее использовать для разбора цепочки *acbb*, действуя так же, как и в предыдущем случае.

Шаг 1. Анализатор выбирает правило (1) грамматики для порождения из символа *S* цепочки, начинающейся с символа *a*, и переходит из начальной конфигурации в конфигурацию (*acbb*, *aAS*⊥, 1).

Шаг 2. Далее анализатор вычеркивает символ из магазина и читает второй символ входной цепочки, переходя в конфигурацию (*cbb*, *AS*⊥, 1).

Шаг 3. Для порождения из верхнего символа магазина цепочки, начинающейся с символа *c*, анализатор выбирает правило (3) грамматики и после его использования переходит в конфигурацию (*cbb*, *cASS*⊥, 13).

Шаг 4. Повторяя шаг, аналогичный шагу 2, анализатор переходит в конфигурацию (*bb*, *ASS*⊥, 13).

Заметим, что пока действия анализатора такие же, как в предыдущем примере.

Шаг 4. Продолжая разбор, анализатор должен выбрать правило грамматики, которое позволяет из нетерминала *A* породить цепочку, начинающуюся с символа *b*. Применить правило (3), имеющее вид $A \rightarrow cAS$, нельзя, т. к. оно начинается с символа *c*, а текущий входной символ — *b*.

Остается сделать предположение, что требуемая цепочка порождается из *A* при помощи правила (4) $A \rightarrow \varepsilon$ и она пустая. В этом случае символ *b* — это начальный символ цепочки, выводимой из символа, находящегося в магазине ниже *A*, т. е. из нетерминала *S*.

Шаг 5. Применение правила (4) приводит к тому, что анализатор вычеркивает верхний символ (нетерминал *A*) из магазина, но не изменяет текущий

символ, т. е. из конфигурации (bb, $ASS\perp$, 13) он переходит в конфигурацию (bb, $SS\perp$, 134).

Оставшиеся действия анализатора не требуют пояснений:

$$(bb, SS\perp, 134) \vdash (bb, bS\perp, 1342)$$
$$\vdash (b, S\perp, 1342)$$
$$\vdash (b, b\perp, 13422)$$
$$\vdash (\varepsilon, \perp, 13422).$$

□

Анализ действий анализатора показывает, что успешное применение ε-правила для некоторого нетерминала возможно в том случае, если цепочка, порождаемая символом, лежащим ниже этого нетерминала, начинается с символа, совпадающего с текущим символом входной цепочки (шаг 5).

Перейдем к процессу анализа грамматики, определив два множества символов.

Множество FIRST (ПЕРВЫЙ) — это множество терминальных символов, которыми начинаются цепочки, выводимые из нетерминала A:

$$\text{FIRST}(A) = \{a \in \Sigma \mid A \Rightarrow_l^+ a\beta, \text{ где } \beta \in (N \cup \Sigma)^*\}.$$

Множество FOLLOW (СЛЕДУЮЩИЙ) — множество терминальных символов, которые могут встретиться непосредственно справа от нетерминала A в некоторой сентенциальной форме:

$$\text{FOLLOW}(A) = \{a \in \Sigma \mid S \Rightarrow_l^* \alpha A\gamma \text{ и } a \in \text{FIRST}(\gamma)\}.$$

Пример 7.3

Для рассмотренной ранее грамматики с правилами:

(1) $S \rightarrow aAS$ (3) $A \rightarrow cAS$

(2) $S \rightarrow b$ (4) $A \rightarrow \varepsilon$

множество $\text{FIRST}(A) = \{c\}$, а множество $\text{FOLLOW}(A) = \{a, b\}$, т. к. непосредственно справа от нетерминала A в сентенциальных формах может находиться символ S и $\text{FIRST}(S) = \{a, b\}$. □

При моделировании работы нисходящего синтаксического анализатора было установлено, что при разборе применяется некоторое правило грамматики, если анализатор находится в конфигурации (ax, $A\alpha\perp$, π) и выполняется одно из условий:

- правило имеет вид $A \rightarrow a\beta$;
- правило имеет вид $A \rightarrow \varepsilon$ и символ $a \in \text{FOLLOW}(A)$.

Для рассмотрения этих условий одновременно, введем понятие *множества выбора правила* SELECT, которое определим для данного правила следующим образом:

- если правило имеет вид $A \to a\beta$, то $SELECT(A \to a\alpha) = FIRST(a\alpha) = \{a\}$;
- если правило имеет вид $A \to \varepsilon$, то $SELECT(A \to a\alpha) = FOLLOW(A)$.

Замечание

Если правила грамматики перенумерованы, то в некоторых случаях удобнее писать SELECT(*i*), если заданное правило имеет номер *i*.

Теперь мы можем определить более широкий, чем *s*-грамматики, класс КС-грамматик:

КС-грамматика $G = (N, \Sigma, P, S)$ называется *q-грамматикой*, если:

1. Правая часть любого ее правила начинается с терминала или пуста.
2. Множества выбора правил для каждого $A \in N$ не пересекаются.

Очевидно, что построение МП-автомата по *q*-грамматике может быть выполнено так же, как и для *s*-грамматики.

7.2. Определение *LL*(1)-грамматики

Обобщим определение множества FIRST так, чтобы его можно было применить для правил произвольного вида.

$FIRST(\alpha) = \{a \in \Sigma \mid S \Rightarrow_l^* \alpha \Rightarrow_l^* a\beta$, где $\alpha \in (N \cup \Sigma)^+$, $\beta \in (N \cup \Sigma)^*\}$.

Данное определение означает, что множество FIRST(α) состоит из множества терминальных символов, которыми начинаются цепочки, выводимые из цепочки α.

КС-грамматика $G = (N, \Sigma, P, S)$ называется *LL*(1)-грамматикой, если из существования двух левых выводов:

$$S \Rightarrow_l^* wA\alpha \Rightarrow_l w\beta\alpha \Rightarrow^* wx$$
$$S \Rightarrow_l^* wA\alpha \Rightarrow_l w\gamma\alpha \Rightarrow^* wy,$$

для которых $FIRST(x) = FIRST(y)$, следует, что $\beta = \gamma$.

В определении *LL*(1)-грамматики утверждается, что выбор правила для замены нетерминала *A* в цепочке $wA\alpha$ однозначно определяется цепочкой *w* и следующим за ней входным символом. На самом деле это не так.

В [3] доказано следующее утверждение.

Утверждение

КС-грамматика $G = (N, \Sigma, P, S)$ является $LL(1)$-грамматикой тогда и только тогда, когда для двух различных правил $A \to \beta$ и $A \to \gamma$ из множества P пересечение $\mathrm{FIRST}(\beta\alpha) \cap \mathrm{FIRST}(\gamma\alpha)$ пустое при всех таких $wA\alpha$, что $S \Rightarrow_l^* wA\alpha$.

Из этого утверждения следует, что для $LL(1)$-грамматики правило для замены нетерминала однозначно определяется текущим входным символом. На рис. 7.2 приведено дерево вывода цепочки wxy. Если к настоящему времени было построено дерево с кроной $wA\alpha$, то для определения правила, которое должно быть использовано для замены нетерминала A, достаточно рассмотреть текущий символ — первый символ цепочки x.

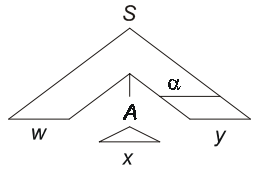


Рис. 7.2. Дерево вывода цепочки *wxy*

Можно доказать [3], что КС-грамматика $G = (N, \Sigma, P, S)$ является $LL(1)$-грамматикой тогда и только тогда, когда для двух различных правил $A \to \beta$ и $A \to \gamma$:

$$\mathrm{FIRST}(\beta\ \mathrm{FOLLOW}(A)) \cap \mathrm{FIRST}(\gamma\ \mathrm{FOLLOW}(A)) = \varnothing \text{ для всех } A \in N.$$

Замечание

Если αA — сентенциальная форма, то ε также принадлежит множеству FOLLOW(A).

На основании этого утверждения можно дать более конструктивное определение $LL(1)$-грамматики.

КС-грамматика $G = (N, \Sigma, P, S)$ является $LL(1)$-грамматикой тогда и только тогда, когда для каждого A-правила грамматики:

$$A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n,\ n \geq 1$$

выполняются следующие условия:

1. Множества $\mathrm{FIRST}(\alpha_1)$, $\mathrm{FIRST}(\alpha_2)$, ... , $\mathrm{FIRST}(\alpha_n)$ попарно не пересекаются.
2. Если $\alpha_i \Rightarrow^* \varepsilon$, то $\mathrm{FIRST}(\alpha_j) \cap \mathrm{FOLLOW}(A) = \varnothing$ для $1 \leq j \leq n$, $i \neq j$.

Важным следствием этого определения является то, что леворекурсивная грамматика не может быть $LL(1)$-грамматикой.

Другое важное свойство $LL(1)$-грамматики — ее однозначность.

7.3. Алгоритм разбора для *LL*(1)-грамматик

Разбор для $LL(1)$-грамматики можно осуществить с помощью так называемого "1-предсказывающего" (один-предсказывающего) алгоритма разбора, который использует входную ленту, магазин и выходную ленту (рис. 7.3). Единица в названии алгоритма означает, что при чтении анализируемой цепочки, находящейся на входной ленте, входная головка может заглядывать вперед на один символ.

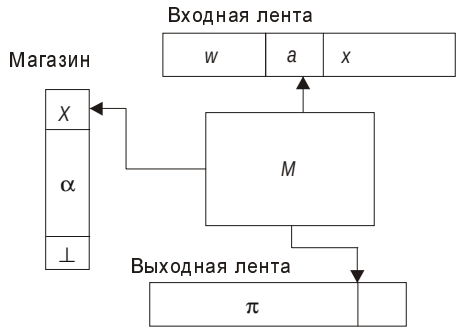


Рис. 7.3. "1-предсказывающий" алгоритм

Магазин содержит цепочку $X\alpha\perp$, где $X\alpha$ — цепочка магазинных символов (X — верхний символ магазина), а символ ($\perp$) — специальный символ, называемый *маркером дна* магазина. Если верхним символом магазина является маркер дна, то магазин пуст. Выходная лента содержит цепочку номеров правил π, представляющую собой текущее состояние левого разбора.

Конфигурацию "1-предсказывающего" алгоритма разбора будем представлять в виде $(x, X\alpha\perp, \pi)$, где x — неиспользованная часть входной цепочки, $X\alpha$ — цепочка в магазине, а π — цепочка на выходной ленте. Например, на рис. 7.3 изображена конфигурация $(ax, X\alpha\perp, \pi)$.

Обозначим алфавит магазинных символов (без символа ($\perp$)) как V_p.

Работой алгоритма управляет управляющая таблица M, задающая отображение множества $(V_p \cup \{\perp\}) \times (\Sigma \cup \{\varepsilon\})$ в множество, состоящее из следующих элементов:

1. (β, i), где $\beta \in V_p^*$ — правая часть правила вывода с номером i.
2. *ВЫБРОС*.

3. *ДОПУСК.*
4. *ОШИБКА.*

Алгоритм анализирует входную цепочку, выполняя последовательность тактов, похожих на такты ДМП-преобразователя. Такты алгоритма будем описывать в терминах отношения ($\vdash$), определенного на множестве конфигураций.

Пусть FIRST(x) = {a}. Тогда работа алгоритма в зависимости от элемента управляющей таблицы $M(X, a)$ определяется следующим образом:

1. $(x, X\alpha, \pi) \vdash (x, \beta\alpha, \pi i)$, если $M(X, a) = (\beta, i)$. В этом случае верхний символ магазина X заменяется на цепочку $\beta \in V_p^*$, и в выходную цепочку дописывается номер правила i. Входная головка при этом не сдвигается.
2. $(ax, a\alpha, \pi) \vdash (x, \alpha, \pi)$, если $M(a, a) =$ *ВЫБРОС*. Это означает, что, если верхний символ магазина совпадает с текущим входным символом, он выбрасывается из магазина, и входная головка сдвигается на один символ вправо.
3. Если алгоритм достигает конфигурации $(\varepsilon, \perp, \pi)$, что соответствует элементу управляющей таблицы $M(\perp, \varepsilon) =$ *ДОПУСК*, то его работа прекращается, и выходная цепочка π является левым разбором входной цепочки.
4. Если алгоритм достигает конфигурации $(x, X\alpha, \pi)$ и $M(X, a) =$ *ОШИБКА*, то разбор прекращается и выдается сообщение об ошибке.

Конфигурация $(w, S\perp, \varepsilon)$, где $S \in V_p$ — начальный символ магазина (начальный символ грамматики), называется *начальной конфигурацией.*

Если $(w, S\perp, \varepsilon) \vdash (\varepsilon, \perp, \pi)$, то π называется *выходом* алгоритма для входа w.

Пример 7.4

🗁 Рассмотрим работу "1-предсказывающего" алгоритма разбора для $LL(1)$-грамматики G_1, имеющей следующие правила вывода:

(1) $E \to TE'$ (5) $T' \to *PT'$

(2) $E' \to +TE'$ (6) $T' \to \varepsilon$

(3) $E' \to \varepsilon$ (7) $P \to i$

(4) $T \to PT'$ (8) $P \to (E)$

Замечание

Грамматика G_1 эквивалентна грамматике G_0, но не является леворекурсивной.

Управляющая таблица алгоритма M для грамматики G_1 представлена табл. 7.2, где пустые клетки таблицы имеют значение *ОШИБКА*.

Таблица 7.2. *Управляющая таблица M для грамматики* G_1

	i	**(**	**)**	**+**	*****	**ε**
E	*TE′*, 1	*TE′*, 1				
E′			ε, 3	+***T E′***, 2		ε, 3
T	*PT′*, 4	*PT′*, 4				
T′			ε, 6	ε, 6	**PT′*, 5	ε, 6
P	*i*, 8	(*E*), 7				
i	*ВЫБРОС*					
(		*ВЫБРОС*				
)			*ВЫБРОС*			
+				*ВЫБРОС*		
*****					*ВЫБРОС*	
⊥						*ДОПУСК*

Начальное содержимое магазина — $E\perp$

Рассмотрим работу алгоритма при разборе цепочки $i + i * i$. Для удобства будем отмечать полужирным шрифтом текущий символ входной цепочки и верхний символ магазина.

Шаг 1. Алгоритм находится в начальной конфигурации ($\boldsymbol{i} + i * i$, $\boldsymbol{E}\perp$, ε).

Определив значение управляющей таблицы для верхнего символа магазина E и текущего символа i, получим $M(E, i) = (TE', 1)$. Это значит, что нужно выполнить следующие действия:

- заменить верхний символ магазина E цепочкой TE';
- не сдвигать читающую головку;
- на выходную ленту поместить номер использованного правила 1.

Алгоритм переходит в конфигурацию ($\boldsymbol{i} + i * i$, $\boldsymbol{T}E'\perp$, 1).

Шаг 2. Выполняя действия, аналогичные описанным для шага 1, пройдем следующие конфигурации:

Текущая конфигурация		**Значение *M***
($\boldsymbol{i} + i * i$, $\boldsymbol{T}E'\perp$, 1)	⊢	$M(T, i) = (PT', 4)$
($\boldsymbol{i} + i * i$, $\boldsymbol{P}T'E'\perp$, 14)	⊢	$M(P, i) = (i, 8)$
($\boldsymbol{i} + i * i$, $\boldsymbol{i}T'E'\perp$, 148).		

Шаг 3. Алгоритм находится в конфигурации (***i*** + *i* * *i*, ***i****T′E′*⊥, 148). Значение управляющей таблицы *M*(*i*, *i*) = *ВЫБРОС*. Выполняем следующие действия:

- удаляем верхний символ магазина;
- сдвигаем читающую головку на один символ вправо.
- При этом выходная лента не изменяется.

Алгоритм переходит в конфигурацию (+ *i* * *i*, ***T′****E′*⊥, 148).

Шаг 4. Выполняя шаги типа 1 и 2, алгоритм произведет следующие действия:

Текущая конфигурация		Значение *M*
(+ *i* * *i*, ***T′****E′*⊥, 148)	⊢	*M*(*T′*, +) = (ε, 6)
(+ *i* * *i*, ***E′***⊥, 1486)	⊢	*M*(*E′*, +) = (+ *TE′*, 2)
(+ *i* * *i*, + ***TE′***⊥, 14862)	⊢	*M*(+, +) = *ВЫБРОС*
(***i*** * *i*, ***TE′***⊥, 14862)	⊢	*M*(*T*, *i*) = (*PT′*, 4)
(***i*** * *i*, ***PT′E′***⊥, 148624)	⊢	*M*(*P*, *i*) = (*i*, 8)
(***i*** * *i*, ***i****T′E′*⊥, 1486248)	⊢	*M*(*i*, *i*) = *ВЫБРОС*
(* *i*, ***T′****E′*⊥, 1486248)	⊢	*M*(*T′*, *) = (* *PT′*, 5)
(* *i*, * *PT′E′*⊥, 14862485)	⊢	*M*(*, *) = *ВЫБРОС*
(***i***, ***P****T′E′*⊥, 14862485)	⊢	*M*(*P*, *i*) = (*i*, 8)
(***i***, ***i****T′E′*⊥, 148624858)	⊢	*M*(*i*, *i*) = *ВЫБРОС*
(ε, *T′E′*⊥, 148624858)	⊢	*M*(*T′*, ε) = (ε, 6)
(ε, *E′*⊥, 1486248586)	⊢	*M*(*E′*, ε) = (ε, 3)
(ε, ⊥, 14862485863)		

Шаг 5. Алгоритм находится в конфигурации (ε, ⊥, 14862485863).

Так как значение *M*(⊥, ε) = *ДОПУСК*, то цепочка *i* + *i* * *i* принадлежит языку, и последовательность номеров правил 14862485863 на выходной ленте — это ее разбор. □

"1-предсказывающий" алгоритм разбора может быть использован для анализа любого входного языка, синтаксис которого описывается *LL*(1)-грамматикой. Этот алгоритм можно промоделировать с помощью ДМП-преобразователя с одним состоянием, входная лента которого имеет концевой маркер.

Рассмотрим, как можно построить управляющую таблицу для заданной *LL*(1)-грамматики.

Алгоритм 7.1. *Алгоритм построения управляющей таблицы M для LL(1)-грамматики*

Вход: *LL*(1)-грамматика $G = (N, \Sigma, P, S)$.

Выход: Корректная управляющая таблица M для грамматики G.

Описание алгоритма:

□ Таблица M определяется на множестве $(N \cup \Sigma \cup \{\perp\}) \times (\Sigma \cup \{\varepsilon\})$ по следующим правилам:

1. Если $A \to \beta$ — правило вывода грамматики с номером i, то $M(A, a) = (\beta, i)$ для всех $a \neq \varepsilon$, принадлежащих множеству FIRST(β).

   Если $\varepsilon \in$ FIRST(β), то $M(A, b) = (\beta, i)$ для всех $b \in$ FOLLOW(A).

2. $M(a, a) =$ *ВЫБРОС* для всех $a \in \Sigma$.
3. $M(\perp, \varepsilon) =$ *ДОПУСК*.
4. В остальных случаях $M(X, a) =$ *ОШИБКА* для $X \in (N \cup \Sigma \cup \{\perp\})$ и $a \in \Sigma \cup \{\varepsilon\}$.

■

Пример 7.5

🗁 Рассмотрим, как строится управляющая таблица M для грамматики G_1 с правилами:

(1)	$E \to TE'$	(5)	$T' \to * PT'$
(2)	$E' \to + TE'$	(6)	$T' \to \varepsilon$
(3)	$E' \to \varepsilon$	(7)	$P \to i$
(4)	$T \to PT'$	(8)	$P \to (E)$

Управляющая таблица должна содержать 11 строк, помеченных символами из множества $(N \cup \Sigma \cup \{\perp\})$, и 6 столбцов, помеченных символами из множества $(\Sigma \cup \{\varepsilon\})$.

Заполнение таблицы начнем с выполнения шага 1 алгоритма.

Шаг 1. Будем строить таблицу построчно, для чего последовательно рассмотрим все нетерминальные символы.

1. Нетерминалу E соответствует правило вывода грамматики (1) $E \to TE'$. Так как FIRST(TE') = { (, i}, то:

   $M(E, ( ) = M(E, i) = (TE', 1)$.

2. Для нетерминала E' в грамматике имеются два правила вывода:

   - для правила (2) $E' \to + TE'$ множество FIRST($+TE'$) = {+} и, следовательно, $M(E', +) = (+TE', 2)$;

- правило (3) $E' \to \varepsilon$ имеет пустую правую часть, поэтому необходимо вычислить множество символов, следующих за нетерминалом E' в сетенциальных формах. Построив левый вывод $E \Rightarrow TE' \Rightarrow PT'E' \Rightarrow (E)T'E' \Rightarrow (TE')TE' \ldots$, получим FOLLOW($E'$) = {), ε}.

Таким образом, $M(E',))$ = $M(E', \varepsilon)$ = (ε, 3).

Выполняя шаг 1 алгоритма для нетерминалов T, T' и P, получим:

Правило грамматики	Множество	Значение M
(4) $T \to PT'$	FIRST(PT') = { (, i)}	$M(T', ()$ = $M(T', i)$ = (PT', 4)
(5) $T' \to *PT'$	FIRST($*PT'$) = {*}	$M(T', *)$ = ($*PT'$, 5)
(6) $T' \to \varepsilon$	FOLLOW(T') = {+,), ε}	$M(T',+)$ = $M(T',))$ = $M(T', \varepsilon)$= = (ε, 6)
(7) $P \to i$	FIRST(i) = {i}	M(P, i) = (I, 7)
(8) $P \to (E)$	FIRST((E)) = {(}	$M(P, ()$ = ((E), 8)

Шаг 2. Далее всем элементам таблицы, находящимся на пересечении строки и столбца, отмеченных одним и тем же терминальным символом, присвоим значение *ВЫБРОС*.

Шаг 3. Элементу таблицы $M(\perp, \varepsilon)$ присвоим значение *ДОПУСК*.

Шаг 4. Остальным элементам таблицы присвоим значение *ОШИБКА*. □

В заключение, можно сравнить таблицу, построенную при помощи алгоритма 7.1, с управляющей таблицей, приведенной в табл. 7.2, и убедиться, что они совпадают. В [3] доказано, что алгоритм 7.1 строит корректную управляющую таблицу.

7.4. *LL*(*k*)-грамматики

LL(1)-грамматики часто используются для описания языков программирования, т. к. позволяют строить эффективные синтаксические анализаторы. Нужно отметить, что мощности *LL*(1)-грамматик недостаточно для описания синтаксиса любых конструкций языков программирования. Естественным обобщением *LL*(1)-грамматик являются *LL*(k)-грамматики [2—4, 28].

Для КС-грамматики G = (N, Σ, P, S) определим множество:

$$\text{FIRST}_k(\alpha) = \{x \mid \alpha \Rightarrow_l^* x\beta \text{ и } |x| = k \text{ или } \alpha \Rightarrow^* x \text{ и } |x| < k\}.$$

Множество $\text{FIRST}_k(\alpha)$ является обобщением множества FIRST(α) (*см. разд. 7.1*) и представляет собой множество терминальных префиксов длины k терминальных цепочек, выводимых из α (или меньшей длины, если из цепочки α выводится терминальная цепочка длины, меньшей k).

Таким образом определение $LL(k)$-грамматик можно сформулировать следующим образом.

КС-грамматика $G = (N, \Sigma, P, S)$ называется $LL(k)$-грамматикой, если из существования двух левых выводов:

$$S \Rightarrow_l^* wA\alpha \Rightarrow_l w\beta\alpha \Rightarrow^* wx$$
$$S \Rightarrow_l^* wA\alpha \Rightarrow_l w\gamma\alpha \Rightarrow^* wy,$$

для которых $FIRST_k(x) = FIRST_k(y)$, следует, что $\beta = \gamma$.

Разбор для $LL(k)$-грамматик выполняется значительно сложнее, чем для $LL(1)$-грамматик. Связано это с двумя проблемами:

- применение алгоритма 7.1 требует использования аванцепочек длиной до k символов, что существенно увеличивает размеры управляющей таблицы;
- для некоторых $LL(k)$-грамматик ($k > 1$) верхний символ магазина и аванцепочка длиной k (или меньше) символов не всегда однозначно определяют правило, которое должно быть применено при разборе.

Рассмотрим вторую проблему более подробно. Мы уже приводили утверждение, лежащее в основе алгоритма 7.1, что КС-грамматика $G = (N, \Sigma, P, S)$ является $LL(1)$-грамматикой тогда и только тогда, когда для двух ее различных правил $A \to \beta$ и $A \to \gamma$:

$$FIRST(\beta\ FOLLOW(A)) \cap FIRST(\gamma\ FOLLOW(A)) = \varnothing \text{ для всех } A \in N.$$

Обобщение этого утверждения на $LL(k)$-грамматики приводит к дополнительным ограничениям.

Если в КС-грамматике $G = (N, \Sigma, P, S)$ для двух различных A-правил $A \to \beta$ и $A \to \gamma$ выполняется:

$$FIRST_k(\beta\ FOLLOW_k(A)) \cap FIRST_k(\gamma\ FOLLOW_k(A)) = \varnothing,$$

то такая КС-грамматика называется *сильно* $LL(k)$-грамматикой.

Алгоритм 7.1 позволяет построить корректную управляющую таблицу только для сильно $LL(k)$-грамматики. В [3] приведен пример $LL(2)$-грамматики, не являющейся сильно $LL(2)$-грамматикой.

Пример 7.6

Для $LL(2)$-грамматики G с правилами $\{S \to aAaa \mid bAba,\ A \to b \mid \varepsilon\}$ множество $FOLLOW_2(A) = \{aa, ba\}$, а значит, для A-правила:

$$FIRST_2(b\ FOLLOW_2(A)) \cap FIRST_2(FOLLOW_2(A)) = \{ba\},$$

и эта грамматика не является сильно $LL(2)$-грамматикой.

Сложности построения синтаксических анализаторов и их неэффективность не позволяют использовать $LL(k)$-грамматики (для $k > 1$) в практике построения компиляторов.

7.5. Рекурсивный спуск

Метод рекурсивного спуска, используемый во многих современных трансляторах, является одним из первых детерминированных методов синтаксического анализа. Широкое распространение этого метода объясняется тем, что он крайне прост для реализации, если для написания компилятора используется язык программирования, допускающий рекурсивные процедуры.

Замечание

Процедура называется *рекурсивной*, если она может вызывать саму себя либо непосредственно, либо через цепочку вызовов других процедур.

Основная *идея метода* рекурсивного спуска состоит в том, что *каждому нетерминалу* грамматики соответствует процедура, которая распознает цепочку, порождаемую этим нетерминалом. Таким образом, необходимость моделирования МП-преобразователя заменяется механизмом вызова рекурсивных процедур. Поскольку языки высокого уровня используют более сложные стековые механизмы, чем требуются для реализации МП-автомата, то метод рекурсивного спуска по затратам времени и памяти будет уступать методам синтаксического анализа, использующим прямую реализацию МП-автомата.

Метод рекурсивного спуска применим для любой $LL(1)$-грамматики, которая должна быть записана в форме с альтернативными правыми частями.

Используем язык программирования Pascal для иллюстрации реализации метода рекурсивного спуска. Головной модуль `Recurs_Method` синтаксического анализатора, реализующего метод рекурсивного спуска, инициируя вызовы процедур, проверяет, порождается ли входная цепочка из начального символа грамматики. Если входная цепочка принадлежит языку, порождаемому грамматикой, то выполняется процедура `Access` (Допуск), в противном случае выполняется процедура `Error` (Ошибка). При этом входная цепочка должна заканчиваться специальным символом — концевым маркером, например ($\perp$). Введение концевого маркера соответствует включению дополнительного правила грамматики вида $S' \rightarrow S\perp$.

Для работы алгоритма необходимо иметь функцию `NextSymb`, которая читает очередной символ входной цепочки и присваивает значение этого символа переменной `Symb`.

При составлении процедур для распознавания цепочек, порождаемых нетерминальными символами грамматики, необходимо придерживаться следующих правил:

- если текущим символом правой части правила грамматики является нетерминал, ему соответствует вызов процедуры, которая распознает цепочку, порождаемую этим нетерминалом;
- если текущим символом правой части правила грамматики является терминал, ему соответствует условный оператор, осуществляющий проверку на совпадение терминала из правила грамматики и значения текущего символа из переменной `Symb`. Если символы совпадают, то выполняется вызов процедуры `NextSymb`, в противном случае осуществляется переход к следующей альтернативе. Если все альтернативы исчерпаны, выполняется вызов процедуры `Error`;
- пустой правой части правила соответствует переход на конец процедуры.

В качестве примера написания синтаксического анализатора методом рекурсивного спуска рассмотрим грамматику G_1, правила вывода которой записаны в виде:

$E \rightarrow TE'$

$E' \rightarrow + TE' \mid \varepsilon$

$T \rightarrow PT'$

$T' \rightarrow * PT' \mid \varepsilon$

$P \rightarrow (E) \mid i$

Дополним правила грамматики еще одним правилом $S \rightarrow E\perp$.

Процедура, реализующая рекурсивный спуск для грамматики G_1 на языке Pascal, приведена в листинге 7.1. Нетерминалу E соответствует процедура `Proc_E`, нетерминалу E' — `Proc_E1`, нетерминалу T — `Proc_T`, нетерминалу T' — `Proc_T1`, а нетерминалу P — `Proc_P`.

Листинг 7.1

```
Procedure Recurs_Method (List_Token: tList);
                        {List_Token – входная цепочка}
   Procedure Proc_E;
      begin
         Proc_T;
         Proc_E1
   end {Proc_E};
```

```
Procedure Proc_E1;
   begin
      if Symb = '+' then
      begin
         NextSymb;
         Proc_T;
         Proc_E1
      end
end {Proc_E1};
Procedure Proc_T;
   begin
      Proc_P;
      Proc_T1
end {Proc_T};
Procedure Proc_T1;
   begin
      if Symb = '*' then
      begin
         NextSymb;
         Proc_P;
         Proc_T1
      end
end {Proc_T1};
Procedure Proc_P;
   begin
      if Symb = '(' then
      begin
         NextSymb;
         Proc_E;
         if Symb = ')' then
            NextSymb;
         else
             Error
      end
      else
         if Symb = 'i' then
```

```
            NextSymb
         else
            Error
   end; {Proc_P}
begin
   { В начале анализа переменная Symb содержит первый символ цепочки }
   Proc_E;
      if Symb = '⊥' then
         Access
      else
         Error
end {Recurs_Method};
```

Список имен процедур в порядке их вызова при обработке входной цепочки $(i + i) * i\perp$ приведен на рис. 7.4. Справа в строке, соответствующей вызову процедуры `NextSymb`, изображена непрочитанная часть входной цепочки (текущий символ находится слева) непосредственно после вызова этой процедуры.

Контрольные вопросы

1. Какие КС-грамматики называются *s*-грамматиками?
2. Что такое множества выбора и как они используются при анализе *s*-грамматик?
3. Какие КС-грамматики называются LL(1)-грамматиками?
4. Дайте определения s-, q- и LL(1)-грамматик. Как соотносятся классы этих грамматик?
5. Как вычисляются множества выбора?
6. Как описывается такт работы LL(1)-анализатора?
7. Как можно представить управляющую таблицу анализатора?
8. Как определяются множества FIRSTk(β) и FOLLOWk(A)?
9. Какие грамматики называются LL(k)-грамматиками?
10. Что представляет собой головной модуль в синтаксическом анализаторе, реализующем метод рекурсивного спуска?
11. Какие требования предъявляются к языку программирования для реализации метода рекурсивного спуска?
12. Как программируются процедуры для распознавания нетерминалов в методе рекурсивного спуска?

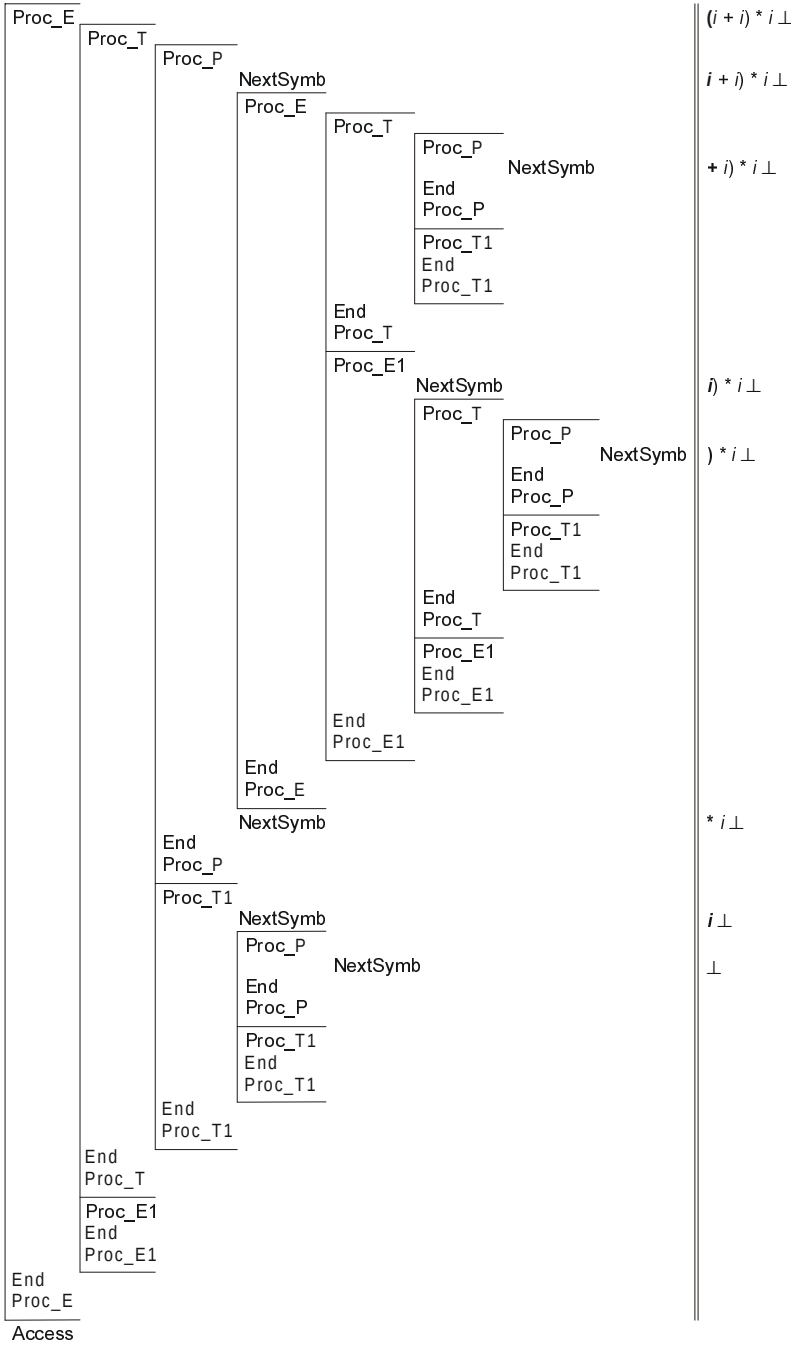
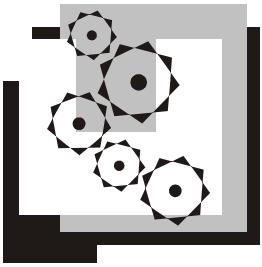


Рис. 7.4. Порядок вызова процедур

Упражнения

1. Проверить, является ли каждая из следующих грамматик $LL(1)$-грамматикой:
   1.1. $S \to BA,\ A \to BS \mid d,\ B \to aA \mid bS \mid c.$
   1.2. $S \to R \mid (S),\ R \to E = E,\ E \to i \mid (E + E).$
   1.3. $S \to aABbCD \mid \varepsilon,\quad A \to Asd \mid \varepsilon,\quad B \to Sac \mid eC \mid \varepsilon,\quad C \to Cf \mid Cg \mid \varepsilon,\ D \to aBD \mid \varepsilon.$
   1.4. $S \to begin\ D,\ L\ end,\ D \to d\ ,\ D,\ D \to \varepsilon,\ L \to sQ,\ L \to \varepsilon,\ Q \to ;\ Q,\ Q \to \varepsilon.$
   1.5. $S \to aAB,\ S \to bBS,\ S \to \varepsilon,\ A \to cBS,\ A \to \varepsilon,\ B \to dB,\ B \to \varepsilon.$
2. Построить управляющую таблицу для $LL(1)$-грамматики с правилами:
   2.1. $S \to Ab \mid Bd,\ A \to aA \mid \varepsilon,\ B \to cB \mid \varepsilon.$
   2.2. $S \to aAA \mid bSA \mid cA,\ A \to aAS \mid bSS \mid cS \mid d.$
   2.3. $S \to aR \mid (S)R,\ R \to \hat{}aR \mid \varepsilon.$
   2.4. $S \to aAbBbS,\quad S \to \varepsilon,\quad A \to aBC,\quad A \to bA,\quad B \to aB,\quad B \to \varepsilon,\quad C \to cC,\ C \to \varepsilon.$
   2.5. $S \to ZC,\ Z \to +,\ Z \to -,\ Z \to \varepsilon,\ C \to dAD,\ D \to \varepsilon,\ D \to .dA,\ A \to dA,\ A \to \varepsilon.$
3. Построить $LL(1)$-грамматику, порождающую логические выражения, составленные из идентификаторов, скобок и знаков логических операций: $\neg$, $\cap$, $\cup$. Знаки логических операций перечислены в соответствии с их приоритетом.
4. Составить процедуры для метода рекурсивного спуска для $LL(1)$-грамматик из упражнения 2.



Глава 8

LR(*k*)-грамматики

В *гл. 6* было показано, что восходящий синтаксический анализ выполняется алгоритмом типа "перенос-свертка", моделирующим работу недетерминированного МП-преобразователя. Синтаксический анализатор чередует переносы и свертки и использует два магазина, причем второй магазин необходим только для того, чтобы выполнить возврат. В этой главе мы рассмотрим широкий класс КС-грамматик, для которых возможен детерминированный восходящий синтаксический анализ. Этот класс грамматик — *LR*(*k*)-грамматики — был разработан Дональдом Кнутом [24] и нашел широкое применение при проектировании компиляторов.

8.1. Детерминированный разбор с помощью алгоритма "перенос-свертка"

Рассмотрим модифицированный алгоритм типа "перенос-свертка", который может выполнить *детерминированный восходящий синтаксический анализ.*

Анализ входной цепочки таким алгоритмом состоит в переносе входных символов в магазин до тех пор, пока в его верхней части не встретится основа. В этот момент производится свертка, в результате которой основа заменяется левой частью основывающего правила. Этот процесс продолжается до тех пор, пока не будет прочитана вся входная цепочка, а в магазине не останется начальный символ грамматики или алгоритм не выдаст сообщение об ошибке.

Проанализируем этот процесс подробнее. Пусть в КС-грамматике G имеется правовыводимая цепочка $S \Rightarrow^*_r \alpha x$, такая, что цепочка α заканчивается нетерминальным символом (или это пустая цепочка). Назовем цепочку α *открытой частью* цепочки αx, а x — *замкнутой частью* цепочки. Граница между открытой и замкнутой частями цепочки называется *рубежом*.

Алгоритм разбора типа "перенос-свертка" можно рассматривать как программу работы расширенного детерминированного МП-преобразователя. В *гл. 5* было показано, что такой преобразователь моделирует обращенный правый вывод цепочки, причем для каждой цепочки он хранит ее открытую часть в магазине, а замкнутая часть находится на входной ленте справа от читающей головки.

Рассмотрим правый вывод $S = \alpha_0 \Rightarrow_r \alpha_1 \Rightarrow_r ... \Rightarrow_r \alpha_m = w$. Допустим, что $\alpha_{i-1} = \gamma Bz$ и на i-ом шаге вывода $\alpha_{i-1} \Rightarrow_r \alpha_i$ было применено правило $B \rightarrow \beta y$ грамматики. Пусть $\gamma\beta = \alpha A$ и $yz = x$. Тогда цепочка $\alpha_i = \gamma\beta yz = \alpha Ax$. При моделировании этого шага вывода преобразователь моделирует обращение вывода, т. е. $\alpha_{i-1} \Leftarrow \alpha_i$.

В начале моделирования i-го шага состояние ДМП-преобразователя определяется цепочкой α_i: в магазине находится αA, а на входной ленте — x. Выполняя разбор, преобразователь будет переносить символы из цепочки x в магазин, пока не определит правый конец основы (возможно, ни одного переноса не потребуется). К этому моменту в магазин будет перенесена цепочка y и в нем будет находиться цепочка $\alpha Ay = \gamma\beta y$.

Преобразователь выполнит свертку, используя правило $B \rightarrow \beta y$, в магазине появится цепочка γB — открытая часть правовыводимой цепочки α_{i-1}, а на входной ленте останется цепочка z — замкнутая часть α_{i-1}. Моделирование шага вывода завершено.

Замечание

Основа правовыводимой цепочки αAx не может целиком лежать внутри цепочки α, но может полностью находиться внутри цепочки x.

Теперь можно перечислить те решения, которые алгоритм типа "перенос-свертка" должен принимать при выполнении восходящего детерминированного синтаксического анализа:

- определить тип очередного шага. Если правый конец основы находится в верхушке магазина, то выполнить свертку, иначе перенести очередной входной символ в магазин;
- локализовать в магазине левый конец основы;
- найти подходящий нетерминал для выполнения свертки.

Рассмотрим алгоритм типа "перенос-свертка", построенный в [29] для грамматики, имеющей следующие правила:

(1) $S \rightarrow (AS)$ (3) $A \rightarrow (SaA)$

(2) $S \rightarrow (b)$ (4) $A \rightarrow (a)$

Первой особенностью этого алгоритма является использование расширенного магазинного алфавита, что позволяет ему, анализируя верхний символ магазина, определить не только наличие в нем основы, но и *правило грамматики для свертки*.

Каждый магазинный символ алгоритма содержит сведения о символе грамматики и "кодируемой цепочке" (состоянии алгоритма). Пример кодирования магазинных символов для рассмотренной ранее грамматики приведен в табл. 8.1. Например, магазинный символ "$)_3$", находящийся на вершине магазина, означает, что:

- в верхушке магазина находится терминальный символ грамматики ")";

- в верхней части магазина находится кодируемая этим символом цепочка символов "(*SaA*)", т. е. *основа* (правая часть правила номер 3).

Таблица 8.1. *Кодирование магазинных символов*

Символ грамматики	Магазинный символ	Кодируемая цепочка	Символ грамматики	Магазинный символ	Кодируемая цепочка
S	S_1	$(AS$	b	b_1	$(b$
	S_2	$(S$	(	$(_1$	(
	S_3	$\perp S$	)	$)_1$	(AS)
A	A_1	$(A$		$)_2$	(b)
	A_2	$(SaA$		$)_3$	(SaA)
a	a_1	a		$)_4$	(a)
	a_2	$(a$	–	$\perp$	$\perp$

Магазинный алфавит построен таким образом, что для каждого магазинного символа (за исключением S_3 и $\perp$), кодируемая им цепочка является *префиксом правой части* некоторого правила грамматики. И наоборот, каждый непустой префикс правой части правила кодируется некоторым магазинным символом. Например, правая часть правила грамматики:

(1) $S \rightarrow (AS)$

имеет четыре непустых префикса: "(", "(A", "(AS" и "(AS)", которые кодируются четырьмя магазинными символами: "$(_1$", "A_1", "S_1" и "$)_1$" соответственно.

Второй особенностью алгоритма является то, что символ переносится в магазин только в том случае, если он кодирует цепочку, "совместимую" с цепочкой, которая будет находиться в магазине после переноса. Цепочка, кодируемая данным магазинным символом, *совместима* с цепочкой в магазине, если она является суффиксом магазинной цепочки после переноса данного символа. Например, если в магазине находится цепочка $\perp(_1(_1S_2a_1$, то алгоритм может втолкнуть в магазин только символ A_2, т. к. после этого в магазине будет находиться цепочка $\perp(_1(_1S_2a_1A_2$, суффиксом которой является цепочка, кодируемая символом A_2, а именно (SaA.

Управление алгоритмом осуществляется при помощи двух функций, приведенных в табл. 8.2, следующим образом:

- используя значения верхнего символа магазина и входного символа, алгоритм определяет значение функции действия: *ПЕРЕНОС* или *СВЕРТКА*;

- при выполнении переноса определяется значение функции переходов, равное магазинному символу, который нужно втолкнуть в магазин;
- значение функции действия, равное *СВЕРТКА*(i), однозначно определяет этот шаг.

Таблица 8.2. *Функции управления алгоритмом*

	Функция действий *f*					**Функция переходов *g***					
	(	a	b	)	$\perp$	S	A	a	b	(	)
S_1	П	П	П	П						$(_1$	$)_1$
S_2	П	П	П	П				a_1		$(_1$	
S_3	П	П	П	П	ДОП					$(_1$	
A_1	П	П	П	П		S_1				$(_1$	
A_2	П	П	П	П						$(_1$	$)_3$
a_1	П	П	П	П			A_2			$(_1$	
a_2	П	П	П	П						$(_1$	$)_4$
b_1	П	П	П	П						$(_1$	$)_2$
$(_1$	П	П	П	П		S_2	A_1	a_2	b_1	$(_1$	
$)_1$	С(1)	С(1)	С(1)	С(1)	С(1)						
$)_2$	С(2)	С(2)	С(2)	С(2)	С(2)						
$)_3$	С(3)	С(3)	С(3)	С(3)	С(3)						
$)_4$	С(4)	С(4)	С(4)	С(4)	С(4)						
$\perp$	П	П	П	П		S_3				$(_1$	

Пример 8.1

Рассмотрим работу алгоритма при разборе цепочки $(((b)a(a))((a)(b)))$. Эта цепочка принадлежит языку, определяемому рассматриваемой грамматикой, т. к. ее правый вывод:

$$
\begin{aligned}
S &\Rightarrow_{(1)} (AS) && \Rightarrow_{(1)} (A(AS))\\
&\Rightarrow_{(2)} (A(A(b))) && \Rightarrow_{(4)} (A((a)(b)))\\
&\Rightarrow_{(3)} ((SaA)((a)(b))) && \Rightarrow_{(4)} ((Sa(a))((a)(b)))\\
&\Rightarrow_{(2)} (((b)a(a))((a)(b))) \Rightarrow && \textit{ДОПУСК}.
\end{aligned}
$$

Дерево вывода цепочки $(((b)a(a))((a)(b)))$ приведено на рис. 8.1.

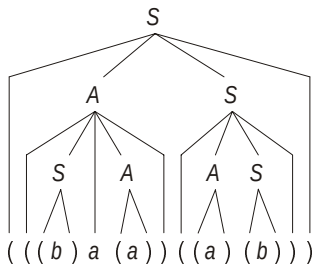


Рис. 8.1. Дерево вывода цепочки $(((b)a(a))((a)(b)))$

Шаг 1. Начальная конфигурация алгоритма $(\perp, (((b)a(a))((a)(b))), \varepsilon)$.

Функция действий определяется верхним символом магазина и входным символом. Для параметров начальной конфигурации (дно магазина "$\perp$" и первый входной символ "(") функция действий равна *ПЕРЕНОС*.

Определяем магазинный символ, который должен быть помещен в магазин. Так как значение функции переносов для тех же параметров "$\perp$" и "(" равно магазинному символу "$(_1$", то алгоритм переходит в конфигурацию:

$(\perp(_1, ((b)a(a))((a)(b))), \varepsilon)$.

Шаг 2. Выполняя аналогичные действия, алгоритм проходит следующие конфигурации:

$$
\begin{aligned}
(\perp, (((b)a(a))((a)(b))), \varepsilon) &\vdash_s (\perp(_1, ((b)a(a))((a)(b))), \varepsilon)\\
&\vdash_s (\perp(_1(_1, (b)a(a))((a)(b))), \varepsilon)\\
&\vdash_s (\perp(_1(_1(_1, b)a(a))((a)(b))), \varepsilon)\\
&\vdash_s (\perp(_1(_1(_1b_1,)a(a))((a)(b))), \varepsilon)\\
&\vdash_s (\perp(_1(_1(_1b_1)_2, a(a))((a)(b))), \varepsilon).
\end{aligned}
$$

Шаг 3. Алгоритм находится в конфигурации $(\perp(_1(_1(_1b_1)_2, a(a))((a)(b))), \varepsilon)$.

Значение функции действий для аргументов "$)_2$" и символа a равно *СВЕРТКА*(2). Для выполнения свертки по второму правилу грамматики мы должны вычеркнуть из магазина три верхних символа и втолкнуть в него некоторый магазинный символ. Для нахождения значения вталкиваемого символа нужно определить значение функции переходов для следующих аргументов: левой части правила, указанного в значении функции действий и входного символа. В данном случае это левая часть второго правила грамматики S и входной символ a, следовательно, получаем значение S_2, которое помещаем в верхушку магазина.

Шаг 4. Продолжая работу, алгоритм пройдет следующие конфигурации:

$$
\begin{aligned}
(\bot({}_1({}_1b_1)_2,\ a(a))((a)(b)))\ \bot,\ \varepsilon) &\vdash_{r(2)} (\bot({}_1({}_1S_2,\ a(a))((a)(b)))\ \bot,\ 2)\\
&\vdash_{s} (\bot({}_1({}_1S_2a_1,\ (a))((a)(b)))\ \bot,\ 2)\\
&\vdash_{s} (\bot({}_1({}_1S_2a_1({}_1,\ a))((a)(b)))\ \bot,\ 2)\\
&\vdash_{s} (\bot({}_1({}_1S_2a_1({}_1a_2,\ ))((a)(b)))\ \bot,\ 2)\\
&\vdash_{s} (\bot({}_1({}_1S_2a_1({}_1a_2)_4,\ )((a)(b)))\ \bot,\ 2)\\
&\vdash_{r(4)} (\bot({}_1({}_1S_2a_1A_2,\ )((a)(b)))\ \bot,\ 24)\\
&\vdash_{s} (\bot({}_1({}_1S_2a_1A_2)_3,\ ((a)(b)))\ \bot,\ 24)\\
&\vdash_{r(3)} (\bot({}_1A_1,\ ((a)(b)))\ \bot,\ 243)\\
&\vdash_{s} (\bot({}_1A_1({}_1,\ (a)(b)))\ \bot,\ 243)\\
&\vdash_{s} (\bot({}_1A_1({}_1({}_1,\ a)(b)))\ \bot,\ 243)\\
&\vdash_{s} (\bot({}_1A_1({}_1({}_1a_2,\ )(b)))\ \bot,\ 243)\\
&\vdash_{s} (\bot({}_1A_1({}_1({}_1a_2)_4,\ (b)))\ \bot,\ 243)\\
&\vdash_{r(4)} (\bot({}_1A_1({}_1A_1,\ (b)))\ \bot,\ 2434)\\
&\vdash_{s} (\bot({}_1A_1({}_1A_1({}_1,\ b)))\ \bot,\ 2434)\\
&\vdash_{s} (\bot({}_1A_1({}_1A_1({}_1b_1,\ )))\ \bot,\ 2434)\\
&\vdash_{s} (\bot({}_1A_1({}_1A_1({}_1b_1)_2,\ ))\ \bot,\ 2434)\\
&\vdash_{r(2)} (\bot({}_1A_1({}_1A_1S_1,\ ))\ \bot,\ 24342)\\
&\vdash_{s} (\bot({}_1A_1({}_1A_1S_1)_1,\ )\ \bot,\ 24342)\\
&\vdash_{r(1)} (\bot({}_1A_1S_1,\ )\ \bot,\ 243421)\\
&\vdash_{s} (\bot({}_1A_1S_1)_1,\ \bot,\ 243421)\\
&\vdash_{r(1)} (\bot S_3,\ \bot,\ 2434211)\\
&\vdash_{s} \textit{ДОПУСК}.
\end{aligned}
$$

Шаг 5. Алгоритм успешно завершил работу и выдал *правый разбор* входной цепочки $(((b)a(a))((a)(b)))$, равный 2434211. □

8.2. *LR*(*k*)-грамматика

Рассмотрим наиболее широкий класс КС-грамматик, для которых можно построить детерминированный восходящий анализатор. Такие грамматики получили название *LR*(*k*)-*грамматик* (входная цепочка читается слева (Left) направо, выходом анализатора является правый (Right) разбор, k — число символов входной цепочки, на которое можно "заглянуть" вперед для выделения основы).

Наиболее наглядно *LR*(*k*)-грамматику можно определить в терминах деревьев вывода. Грамматика $G = (N, \Sigma, P, S)$ является *LR*(*k*)-грамматикой, если,

просмотрев только часть кроны дерева вывода в этой грамматике, расположенную слева от данной внутренней вершины, часть кроны, выведенную из нее, и следующие k символов входной цепочки, можно установить правило вывода, которое было применено к этой вершине при порождении входной цепочки.

Например, рассмотрев цепочку uv и первые k символов цепочки w (рис. 8.2), можно определить, какое правило было применено к вершине A.

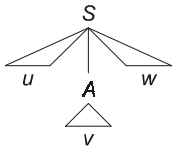


Рис. 8.2. Дерево вывода цепочки uvw

В определении $LR(k)$-грамматики используется множество $\mathrm{FIRST}_k(\gamma)$, состоящее из префиксов длины k терминальных цепочек, выводимых из γ. Если из γ выводятся терминальные цепочки, длина которых меньше k, то эти цепочки также включаются в множество $\mathrm{FIRST}_k(\gamma)$. Формально:

$$\mathrm{FIRST}_k(\gamma) = \{x \mid \gamma \Rightarrow^*_l xw \text{ и } |x| = k \text{ или } \gamma \Rightarrow^*_l x \text{ и } |x| < k\}.$$

Очевидно, что определение множества $\mathrm{FIRST}(\gamma)$, приведенное при определении $LL(k)$-грамматики в *гл. 7*, полностью согласуется с данным определением.

Введем еще одно понятие, которое понадобится для определения $LR(k)$-грамматики.

Пополненной грамматикой, полученной из КС-грамматики $G = (N, \Sigma, P, S)$, называется грамматика $G' = (N \cup \{S'\}, \Sigma, P \cup \{S' \to S\}, S')$.

Если правила грамматики G' пронумерованы числами 1, 2, ... , p, то будем считать, что $S' \to S$ — нулевое правило грамматики G', а нумерация остальных правил такая же, как и в грамматике G. Начальное правило $S' \to S$ вводится для того, чтобы свертку, в которой используется это правило, можно было интерпретировать как признак того, что входная цепочка допустима.

Дадим теперь определение $LR(k)$-грамматики.

Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика и $G' = (N', \Sigma, P', S')$ — полученная из нее пополненная грамматика. G называется *LR(k)-грамматикой* для $k \geq 0$, если из существования двух правых выводов:

$$S' \Rightarrow^*_r \alpha Aw \Rightarrow_r \alpha\beta w,$$
$$S' \Rightarrow^*_r \gamma Bx \Rightarrow_r \alpha\beta y,$$

для которых $\mathrm{FIRST}_k(w) = \mathrm{FIRST}_k(y)$ следует, что $\alpha Ay = \gamma Bx$.

Это определение говорит о том, что если $\alpha\beta w$ и $\alpha\beta y$ — правовыводимые цепочки пополненной грамматики G', у которых $\mathrm{FIRST}_k(w) = \mathrm{FIRST}_k(y)$ и $A \to \beta$ — последнее правило, использованное в правом выводе цепочки $\alpha\beta w$, то правило $A \to \beta$ должно использоваться также в правом разборе при свертке $\alpha\beta y$ к $\alpha A y$.

Поскольку правило грамматики $A \to \beta$ не зависит от w, то из определения $LR(k)$-грамматики следует, что во множестве $\mathrm{FIRST}_k(w)$ содержится информация, достаточная для определения основы.

8.3. Алгоритм разбора для *LR*(*k*)-грамматики

Для любой $LR(k)$-грамматики $G = (N, \Sigma, P, S)$ можно построить детерминированный анализатор, который выдает правый разбор входной цепочки.

Анализатор состоит из магазина, входной ленты, выходной ленты и управляющего устройства (пара функций f и g) (рис. 8.3).

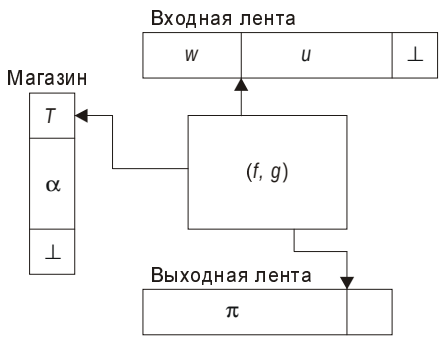


Рис. 8.3. *LR*(*k*)-анализатор

Магазинный алфавит V_p представляет собой множество специальных символов, соответствующих грамматическим вхождениям или их множествам. *Грамматическое вхождение* — это символы полного словаря грамматики, снабженные двумя индексами. Первый индекс задает номер i правила грамматики, в правую часть которого входит данный символ, а второй индекс j — номер позиции символа в этой правой части

Пример 8.2

🗁 Пусть грамматика содержит правила:

(1) $S \to aAb$ (4) $A \to Bb$

(2) $S \to c$ (5) $B \to aA$

(3) $A \to bS$ (6) $B \to c$

Тогда грамматическое вхождение $A_{5,2}$ представляет собой нетерминал A из второй позиции правой части пятого правила вывода. Если символ входит в правую часть i-го правила только один раз, то второй индекс будем опускать. Например, A_1 — это грамматическое вхождение нетерминала A в первое правило, а A_5 — грамматическое вхождение нетерминала A в пятое правило. □

Перед началом работы алгоритма магазин пуст (содержит маркер дна).

Управляющее устройство анализатора представляет собой таблицу $\mathscr{T}$, строки которой отмечены символами $T \in V_p \cup \{\perp\}$. Одна строка такой таблицы называется *LR(k)-таблицей*. Каждая $LR(k)$-таблица задает две функции: функцию действия f и функцию переходов g.

Функция действия f, аргументом которой служит цепочка $u \in (V_p \cup \{\perp\})^*$, принимает значения из множества {*ДОПУСК*, *ОШИБКА*, *ПЕРЕНОС*, (*СВЕРТКА*, i)}.

Аргументом функции переходов g является символ $a \in \Sigma \cup \{\perp\}$, а ее значениями — элементы множества $\{ОШИБКА\} \cup V_p$.

Алгоритм 8.1. *LR(k)-алгоритм разбора*

Вход: Анализируемая цепочка $z = t_1 t_2 \ldots t_j \ldots t_n \in \Sigma^*$, где j — номер текущего символа входной цепочки, находящегося под читающей головкой.

Управляющая таблица $\mathscr{T}$ (множество $LR(k)$-таблиц) для $LR(k)$-грамматики $G = (N, \Sigma, P, S)$.

Выход: Если $z \in L(G)$, то правый разбор цепочки z, в противном случае — сигнал об ошибке.

Описание алгоритма:

□

1. $j := 0$.
2. $j := j + 1$. Если $j > n$, то выдать сообщение об ошибке и перейти к шагу 5.
3. Определить цепочку u следующим образом:
   - если $k = 0$, то $u = t_j$;
   - если $k \geq 1$ и $j + k - 1 \leq n$, то $u = t_j t_{j+1} \ldots t_{j+k-1}$ — первые k символов цепочки $t_j t_{j+1} \ldots t_n$;
   - если $k \geq 1$ и $j + k - 1 > n$, то $u = t_j t_{j+1} \ldots t_n$ — остаток входной цепочки.
4. Применить функцию действия f из строки таблицы $\mathscr{T}$, отмеченной верхним символом магазина T, к цепочке u:
   - $f(u) = ПЕРЕНОС$. Определить функцию переходов $g(t_j)$ из строки таблицы $\mathscr{T}$, отмеченной символом T из верхушки магазина. Если

$g(t_j) = T'$ и $T' \in V_p \cup \{\perp\}$, то записать T' в магазин и перейти к шагу 2. Если $g(t_j) =$ *ОШИБКА*, то выдать сигнал об ошибке и перейти к шагу 5;

- $f(u) = (СВЕРТКА, i)$ и $A \to \alpha$ — правило вывода с номером i грамматики G. Удалить из верхней части магазина $|\alpha|$ символов, в результате чего в верхушке магазина окажется символ $T' \in V_p \cup \{\perp\}$, и выдать номер правила i на выходную ленту. Определить символ $T'' = g(A)$ из строки таблицы $\mathscr{T}$, отмеченной символом T', записать его в магазин и перейти к шагу 3.
- $f(u) =$ *ОШИБКА*. Выдать сообщение об ошибке и перейти к шагу 5.
- $f(u) =$ *ДОПУСК*. Объявить цепочку, записанную на выходной ленте, правым разбором входной цепочки z.

5. Останов.

■

Пример 8.3

▱ Рассмотрим алгоритм разбора для $LR(0)$-грамматики, имеющей следующие правила вывода:

(1) $S \to aAb$ (4) $A \to Bb$

(2) $S \to c$ (5) $B \to aA$

(3) $A \to bS$ (6) $B \to c$

Управляющая таблица $\mathscr{T}$ для этой грамматики приведена в табл. 8.3.

Таблица 8.3. *Управляющая таблица T*

T	$f(u)$				$g(X)$					
	a	b	c	$\perp$	a	b	c	S	A	B
S_0				Д						
a_1	П	П	П		a_5	b_3	c_6		A_1	B_4
A_1	П	П	П			b_1				
b_1	С,1	С,1	С,1	С,1						
c_2	С,2	С,2	С,2	С,2						
b_3	П	П	П		a_1		c_2	S_3		
S_3	С,3	С,3	С,3	С,3						

Таблица 8.3 *(окончание)*

T	$f(u)$				$g(X)$					
	a	*b*	*c*	⊥	*a*	*b*	*c*	*S*	*A*	*B*
B_4	П	П	П			b_4				
b_4	С,4	С,4	С,4	С,4						
a_5	П	П	П		a_5	b_3	c_6		A_5	B_4
A_5	С,5	С,5	С,5	С,5						
c_6	С,6	С,6	С,6	С,6						
⊥	П	П	П		a_1		c_2	S_0		

Замечание

Буквы С, П и Д в этой и во всех следующих таблицах служат условными обозначениями значений функции $f(u)$: *ПЕРЕНОС*, *СВЕРТКА* и *ДОПУСК* соответственно, а пустые элементы таблицы имеют значение *ОШИБКА*.

Рассмотрим алгоритм, управляющая таблица которого приведена в табл. 8.3. Работу алгоритма опишем в терминах конфигураций, представляющих собой тройки вида (αT, ax, π), где αT — цепочка магазинных символов (T — верхний символ магазина), ax — необработанная часть входной цепочки, начинающаяся символом a (для $k = 0$ длина цепочки u равна 1), π — выход, построенный к настоящему моменту времени.

Рассмотрим последовательность тактов, которую выполнит $LR(k)$-алгоритм при анализе входной цепочки *abcb*.

Шаг 1. Начальная конфигурация алгоритма — (⊥, *abcb*⊥, ε) (в вершине магазина находится маркер дна магазина, а текущим входным символом является символ a).

Для строки управляющей таблицы, отмеченной символом ⊥, $f(a)$ = *ПЕРЕНОС* и $g(a) = a_1$, поэтому:

- в магазин записывается символ a_1 (грамматическое вхождение символа a в правую часть первого правила);
- входная головка сдвигается на один символ вправо.

Алгоритм переходит в конфигурацию (⊥a_1, *bcb*⊥, ε).

Шаг 2. Для строки таблицы, отмеченной символом a_1, $f(b)$ = *ПЕРЕНОС*, а $g(b) = b_3$, следовательно, алгоритм перейдет в конфигурацию (⊥a_1b_3, *cb*⊥, ε).

Аналогично для магазинного символа b_3 и текущего символа входной цепочки магазин перейдет в конфигурацию (⊥$a_1b_3c_2$, *b*⊥, ε).

Шаг 3. Рассмотрим теперь строку управляющей таблицы $\mathscr{T}$, помеченную грамматическим вхождением c_2. В этом случае $f(b) = (c, 2)$, значит, необходимо выполнить свертку с использованием правила (2) $S \to c$.

Правая часть этого правила содержит только один символ, поэтому:

- удаляем из магазина символ c_2;
- определяем значение функции переходов для символа S из левой части правила (2) в строке управляющей таблицы $\mathscr{T}$, отмеченной символом b_3, который стал верхним символом магазина. Значение функции переходов $g(S) = S_3$.

Алгоритм переходит в конфигурацию $(\perp a_1 b_3 S_3,\ b\perp,\ 2)$, и в выходную цепочку записывается число 2 (номер использованного правила).

Поступая дальше подобным образом, получим следующую последовательность тактов работы анализатора:

$$\begin{aligned}(\perp a_1 b_3 S_3,\ b\perp,\ 2) &\vdash (\perp a_1 A_1,\ b\perp,\ 23)\\ &\vdash (\perp a_1 A_1 b_1,\ \perp,\ 23)\\ &\vdash (\perp S_0,\ \perp,\ 231).\end{aligned}$$

Заметим, что конфигурация $(\perp S_0,\ \perp,\ 231)$ является заключительной, а цепочка 231 — правым разбором цепочки *abcb*.

Приведем последовательность тактов, которую выполнит алгоритм при анализе входной цепочки *aabc*, содержащей синтаксическую ошибку.

$$\begin{aligned}(\perp,\ aabc\perp,\ \varepsilon) &\vdash (\perp,\ aabc\perp,\ \varepsilon)\\ &\vdash (\perp a_1,\ abc\perp,\ \varepsilon)\\ &\vdash (\perp a_1 a_5,\ bc\perp,\ \varepsilon)\\ &\vdash (\perp a_1 a_5 b_3,\ c\perp,\ \varepsilon)\\ &\vdash (\perp a_1 a_5 b_3 c_2,\ \perp,\ \varepsilon)\\ &\vdash (\perp a_1 a_5 b_3 S_3,\ \perp,\ 2)\\ &\vdash (\perp a_1 a_5 A_5,\ \perp,\ 23)\\ &\vdash (\perp a_1 B_4,\ \perp,\ 235)\\ &\vdash \textit{ОШИБКА}.\end{aligned}$$

$f(\varepsilon) = \textit{ОШИБКА}$. Алгоритм должен выдать сообщение об ошибке и закончить работу. □

Продемонстрировав работу $LR(k)$-анализатора для конкретных входных цепочек, рассмотрим построение управляющей таблицы $\mathscr{T}$ $LR(k)$-анализатора для $k = 0$ и некоторого подмножества $LR(1)$-грамматик. Это связано с тем, что алгоритм проверки принадлежности грамматики классу $LR(k)$-грамматик для

произвольного k предполагает построение большого числа вспомогательных множеств, поэтому его использование в случаях $k > 1$ для решения практических задач не оправдано.

8.4. Построение *LR*(*k*)-анализаторов

Пусть КС-грамматика такова, что *позволяет* распознать основу, анализируя только символы, помещенные в верхнюю часть магазина. В этом случае можно построить конечный автомат, который, считывая символы магазина сверху вниз, определит основу (если она есть в магазине). Диаграмма такого конечного автомата для грамматики с правилами вывода:

(1) $S \rightarrow aAb$ (4) $A \rightarrow Bb$

(2) $S \rightarrow c$ (5) $B \rightarrow aA$

(3) $A \rightarrow bS$ (6) $B \rightarrow c$

приведена на рис. 8.4.

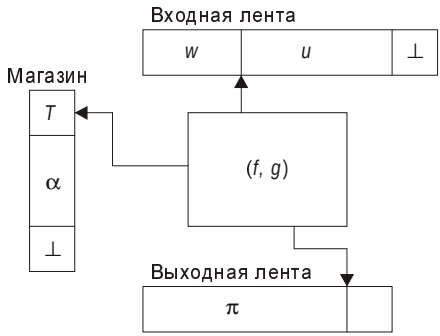


Рис. 8.4. Диаграмма конечного автомата

$LR(k)$-анализатор, приведенный на рис. 8.4, для распознавания основы не должен сканировать *весь* магазин. Это связано с тем, что символ в верхушке магазина содержит всю необходимую информацию об основе. Если верхнего символа магазина недостаточно для принятия решения, то $LR(k)$-анализатор может воспользоваться очередными k символами входной ленты. Как было замечено, практический интерес представляют случаи, для которых $k = 0$ или $k = 1$. Изучение $LR(k)$-разбора удобнее всего начинать с простейшего случая, для которого $k = 0$.

Пусть $S \Rightarrow^*_r \alpha Aw \Rightarrow \alpha\beta w$ — правый вывод в КС-грамматике G. Цепочка γ называется *активным префиксом* грамматики G, если γ — префикс цепочки $\alpha\beta$, т. е. γ — префикс некоторой правовыводимой цепочки, не выходящий за правый конец ее основы.

Мы уже видели, что основу $LR(k)$-анализатора составляют таблицы. Каждая $LR(k)$-таблица соответствует некоторому активному префиксу. Таблица, соответствующая активному префиксу γ, несет информацию о том, достигнут или нет правый конец основы и, если он достигнут, сообщает сведения об основе и правиле, которое нужно применить для ее свертки.

Замечание

В общем случае [3, 4] при построении $LR(k)$-таблиц возникает несколько проблем:

- $LR(k)$-условие говорит о том, что основу правовыводимой цепочки можно определить однозначно, если известен весь отрезок этой цепочки слева от основы и k очередных входных символов. Это означает, что цепочка γ может быть сколь угодно длинной. Возникает вопрос: можно ли в таком случае обойтись конечным множеством таблиц?
- Если существует вывод $S \Rightarrow^*_r \alpha Aw \Rightarrow_r \alpha\beta w$ и известно, что правый вывод цепочки $\alpha\beta w$ заканчивается правилом с номером p, то неясно, можно ли по таблицам, соответствующим цепочке $\alpha\beta$, простым способом определить таблицы, соответствующие αA. Поэтому $LR(k)$-таблицы должны содержать достаточно информации, чтобы по таблице, соответствующей цепочке $\alpha\beta$, можно было вычислить таблицу для αA, если $\alpha Aw \Rightarrow \alpha\beta w$.

Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. Назовем *LR(k)-ситуацией* конструкцию $[A \to \beta_1 \bullet \beta_2, u]$ в том случае, если $A \to \beta_1\beta_2$ — правило из P, а $u \in \Sigma^{*k}$. $LR(k)$-ситуация $[A \to \beta_1 \bullet \beta_2, u]$ *допустима* для активного префикса $\alpha\beta_1$, если существует такой вывод $S \Rightarrow^*_r \alpha Aw \Rightarrow \alpha\beta_1\beta_2 w$, что $u = \mathrm{FIRST}_k(w)$.

Система множеств допустимых $LR(k)$-ситуаций для пополненной грамматики G' называется *канонической системой множеств LR(k)-ситуаций* для грамматики G.

Цепочка β_1 может быть пустой, и для каждого активного префикса найдется хотя бы одна допустимая для него $LR(k)$-ситуация.

Пример 8.4

Так как $LR(0)$-ситуация — это просто правило грамматики с точкой в некоторой позиции правой части, то для правила $P \to (E)$ можно получить четыре ситуации:

$[P \to \bullet\ (E)]$,

$[P \to (\bullet\ E)]$,

$[P \to (E\ \bullet)]$,

$[P \to (E)\ \bullet]$.

Замечание

LR(0)-ситуация не содержит аванцепочку *u*, поэтому при ее записи можно опускать квадратные скобки.

Из определения *LR*(*k*)-ситуации следует, что для каждого активного префикса существует непустое множество ситуаций. Основная идея построения простого *LR*(0)-анализатора состоит в том, чтобы вначале построить на базе КС-грамматики детерминированный конечный автомат для распознавания активных префиксов. Для этого ситуации группируются в множества, которые приводят к состояниям анализатора.

Замечание

Ситуации могут рассматриваться как *состояния недетерминированного конечного автомата*, распознающего активные префиксы.

Существует несколько способов построения детерминированного конечного автомата, распознающего активные префиксы [2, 3, 28, 37]. Рассмотрим вначале достаточно простой способ построения канонической системы *LR*(0)-ситуаций, приведенный в [2], для чего определим две функции: CLOSURE и GOTO.

Пусть I — множество *LR*(0)-ситуаций КС-грамматики G. Тогда назовем замыканием множества I множество ситуаций CLOSURE(I), построенное по следующим правилам:

1. Включить в CLOSURE(I) все ситуации из I.
2. Если ситуация $A \to \alpha \bullet B\beta$ уже включена в CLOSURE(I) и $B \to \gamma$ — правило грамматики, то добавить в множество CLOSURE(I) ситуацию $B \to \bullet\, \gamma$ при условии, что там ее еще нет.
3. Повторять правило 2, до тех пор, пока в CLOSURE(I) нельзя будет включить новую ситуацию.

Второе правило построения можно пояснить следующим образом:

- наличие ситуации $A \to \alpha \bullet B\beta$ в множестве CLOSURE(I) говорит о том, что в некоторый момент разбора мы можем встретить во входном потоке анализатора подстроку, выводимую из $B\beta$;
- если в грамматике имеется правило $B \to \gamma$, то мы также можем встретить во входном потоке анализатора подстроку, выводимую из γ, следовательно, в CLOSURE(I) нужно включить ситуацию $B \to \bullet\, \gamma$.

Рассмотрим расширенную грамматику G'_0, имеющую следующие правила:

(0) $E' \to E$

(1) $E \to E + T$ (4) $T \to P$

(2) $E \to T$ (5) $P \to i$

(3) $T \to T * P$ (6) $P \to (E)$

Пусть вначале множество ситуаций CLOSURE(I) содержит одну ситуацию $E' \to \bullet\, E$, т. е. CLOSURE(I) = $\{E' \to \bullet\, E\}$.

Ситуация $E' \to \bullet\, E$ содержит точку перед символом E, поэтому по второму правилу в CLOSURE(I) необходимо включить E-правила с точкой слева:

$E \to \bullet\, E + T$ и $E \to \bullet\, T$.

Теперь в множестве CLOSURE(I) имеется ситуация, в которой за точкой следует T, а значит, в соответствии со вторым правилом построения в это множество нужно включить T-правила, а затем и P-правила с точкой слева:

$T \to \bullet\, T * P$, $T \to \bullet\, P$, $P \to \bullet\, i$, $P \to \bullet\, (E)$.

Окончательно получим:

CLOSURE(I) = $\{E' \to \bullet\, E,\ E \to \bullet\, E + T,\ E \to \bullet\, T,\ T \to \bullet\, T * P,\ T \to \bullet\, P,\ P \to \bullet\, i,\ P \to \bullet\, (E)\}$.

Алгоритм вычисления множества ситуаций CLOSURE(I) приведен в [2].

Неформально можно сказать, что если I — множество ситуаций, допустимых для некоторого активного префикса γ, то GOTO(I, X) — это множество ситуаций, допустимых для активного префикса γX.

Аргументами функции GOTO(I, X) являются множество ситуаций I и символ грамматики X. Функция GOTO(I, X) определяется как замыкание множества всех ситуаций $[A \to \alpha X \bullet \beta]$, таких, что $[A \to \alpha \bullet X\beta] \in I$.

Пусть для грамматики G'_0 множество $I = \{[E' \to E \bullet], [E \to E \bullet + T]\}$.

Для вычисления значения GOTO(I, +) необходимо рассмотреть ситуации, в которых сразу за точкой идет символ (+).Это ситуация $[E \to E \bullet + T]$. Перемещая точку за символ (+), построим множество ситуаций $\{[E \to E + \bullet\, T]\}$ и замыкание этого множества:

$\{T \to \bullet\, T * P,\ T \to \bullet\, P,\ P \to \bullet\, i,\ P \to \bullet\, (E)\}$.

Тогда GOTO(I, +) = $\{[E \to E + \bullet\, T],\ [T \to \bullet\, T * P],\ [T \to \bullet\, P],\ [P \to \bullet\, i],\ [P \to \bullet\, (E)]\}$

В [2] и [3] приведены алгоритмы, позволяющие построить каноническую систему множеств $LR(0)$-ситуаций $\mathscr{C}$. Процесс построения канонической системы множеств $LR(0)$-ситуаций можно описать с помощью следующих действий:

1. $\mathscr{C} = \varnothing$.
2. Включить в $\mathscr{C}$ множество A_0 = CLOSURE($\{[S' \to \bullet\, S]\}$), которое вначале "не отмечено".
3. Если множество ситуаций A, входящее в систему $\mathscr{C}$, "не отмечено", то:
   - отметить множество A;
   - вычислить для каждого символа $X \in (N \cup \Sigma)$ значение $A' = GOTO(A, X)$;
   - если множество $A' \neq \varnothing$ и еще не включено в $\mathscr{C}$, то включить его в систему множеств $\mathscr{C}$ как "неотмеченное" множество.

4. Повторять шаг 3, пока все множества ситуаций системы $\mathscr{C}$ не будут отмечены.

Пример 8.5

🗁 Определим каноническую систему множеств $LR(0)$-ситуаций для пополненной грамматики G_0 с правилами:

(0) $E' \to E$

(1) $E \to E + T$ (4) $T \to P$

(2) $E \to T$ (5) $P \to i$

(3) $T \to T * P$ (6) $P \to (E)$

В начале построения система множеств $\mathscr{C} = \varnothing$.

Определив множество $A_0 = \text{CLOSURE}(\{[S' \to \bullet S]\}) = \{E' \to \bullet E,\ E \to \bullet E + T,\ E \to \bullet T,\ T \to \bullet T * P,\ T \to \bullet P,\ P \to \bullet i,\ P \to \bullet (E)\}$, включим его в систему $\mathscr{C}$ в качестве "неотмеченного" множества:

A_0	$E' \to \bullet E$
	$E \to \bullet E + T$
	$E \to \bullet T$
	$T \to \bullet T * P$
	$T \to \bullet P$
	$P \to \bullet (E)$
	$P \to \bullet i$

Теперь мы должны отметить множество A_0 и определить множества $\text{GOTO}(A_0, X)$ для всех символов грамматики G_0:

$$A_1 = \text{GOTO}(A_0, E) = \text{GOTO}(\{E' \to \bullet E,\ E \to \bullet E + T\}, E) =$$
$$\text{CLOSURE}(\{E' \to E \bullet,\ E \to E \bullet + T\}) = \{E' \to E \bullet,\ E \to E \bullet + T\}.$$

Множество A_1 не пустое и отсутствует в системе $\mathscr{C}$. Включим A_1 в $\mathscr{C}$ как "неотмеченное" множество:

A_0	$E' \to \bullet E$
	$E \to \bullet E + T$
	$E \to \bullet T$
	$T \to \bullet T * P$
	$T \to \bullet P$
	$P \to \bullet (E)$
	$P \to \bullet i$

A_1	$E \to E \bullet$
	$E \to E \bullet + T$

Продолжая строить GOTO(A_0, X), получаем:

A_2 = GOTO(A_0, T) = GOTO({$E \to \bullet\ T$, $T \to \bullet\ T * P$}, T) =

CLOSURE({$E \to T \bullet$, $T \to T \bullet * P$}) = {$E' \to T \bullet$, $T \to T \bullet * P$}.

Включаем A_2 в $\mathscr{C}$.

A_3 = GOTO(A_0, P) = GOTO({$T \to \bullet\ P$}, P) = {$T \to P \bullet$}.

Включаем A_3 в $\mathscr{C}$.

A_4 = GOTO(A_0, ( ) = GOTO({$P \to \bullet\ (E)$}, ( ) = {$P \to (\bullet\ E)$, $E \to \bullet\ E + T$, $E \to \bullet\ T$, $T \to \bullet\ T * P$, $T \to \bullet\ P$, $P \to \bullet\ i$, $P \to \bullet\ (E)$}.

Включаем A_4 в $\mathscr{C}$.

A_5 = GOTO(A_0, i) = GOTO({$P \to \bullet\ i$}, i) = {$P \to i \bullet$}.

Включаем A_5 в $\mathscr{C}$.

Остальные множества GOTO(A_0, X), где $X \in$ {), +, *}, пусты, поэтому система $\mathscr{C}$ принимает вид:

A_0	$E' \to \bullet\ E$
	$E \to \bullet\ E + T$
	$E \to \bullet\ T$
	$T \to \bullet\ T * P$
	$T \to \bullet\ P$
	$P \to \bullet\ (E)$
	$P \to \bullet\ i$
A_1	$E' \to E \bullet$
	$E \to E \bullet + T$
A_2	$E \to T \bullet$
	$T \to T \bullet * P$

A_3	$T \to P \bullet$
A_4	$P \to (\bullet\ E)$
	$E \to \bullet\ E + T$
	$E \to \bullet\ T$
	$T \to \bullet\ T * P$
	$T \to \bullet\ P$
	$P \to \bullet\ (E)$
	$P \to \bullet\ i$
A_5	$P \to i \bullet$

Продолжая выполнять шаг 3 построения, "отмечаем" множество A_1 и строим множества A_3 = GOTO(A_1, X).

Кроме множества:

$$A_6 = \text{GOTO}(A_1, +) = \text{GOTO}(\{E \to E \bullet + T\}, +) = \text{CLOSURE}\{E \to E + \bullet\ T\} =$$
$$= \{E \to E + \bullet\ T,\ T \to \bullet\ T * P,\ T \to \bullet\ P,\ P \to \bullet\ (E),\ P \to \bullet\ i\},$$

остальные множества пусты. Включаем A_6 в $\mathscr{C}$ и получаем:

A_0	$E' \to \bullet E$ $E \to \bullet E + T$ $E \to \bullet T$ $T \to \bullet T * P$ $T \to \bullet P$ $P \to \bullet (E)$
A_1	$E' \to E \bullet$ $E \to E \bullet + T$
A_2	$E \to T \bullet$ $T \to T \bullet * P$
A_3	$T \to P \bullet$

A_4	$P \to (\bullet E)$ $E \to \bullet E + T$ $E \to \bullet T$ $T \to \bullet T * P$ $T \to \bullet P$ $P \to \bullet (E)$ $P \to \bullet I$
A_5	$P \to i \bullet$
A_6	$E \to E + \bullet T$ $T \to \bullet T * P$ $T \to \bullet P$ $P \to \bullet (E)$ $P \to \bullet i$

Продолжая выполнять шаг 3, получаем:

$$A_7 = \text{GOTO}(A_2, *) = \text{GOTO}(\{T \to T \bullet * P\}, *) = \text{CLOSURE}(\{T \to T * \bullet P\}) =$$
$$= \{T \to T * \bullet P,\ P \to \bullet i,\ P \to \bullet (E)\}.$$

Включаем A_7 в $\mathscr{C}$.

$$A_8 = \text{GOTO}(A_4,\ E) = \text{GOTO}(\{P \to (\bullet E),\ E \to \bullet E + T\},\ E) =$$
$$= \text{CLOSURE}(\{P \to (E \bullet),\ E \to E \bullet + T\}) = \{P \to (E \bullet),\ E \to E \bullet + T\}.$$

Включаем A_8 в $\mathscr{C}$.

Вычисляем множество GOTO(A_4, T).

$$\text{GOTO}\ (A_4,\ T) = \text{GOTO}(\{E \to \bullet T,\ T \to \bullet T * P\},\ T) =$$
$$= \text{CLOSURE}(\{E \to T \bullet,\ T \to T \bullet * P\}) = \{E' \to T \bullet,\ T \to T \bullet * P\}.$$

Такое множество (множество A_2) уже есть в $\mathscr{C}$.

Множества GOTO(A_4, P), GOTO(A_4, ( ) и GOTO(A_4, i) также уже помещены в систему $\mathscr{C}$.

Множество GOTO(A_5, X) = $\varnothing$ для всех $X \in (\text{N} \cup \Sigma)$.

Продолжая аналогичным образом, включаем в систему $\mathscr{C}$ множества:

$$A_9 = \text{GOTO}(A_6,\ T) = \text{GOTO}(\{E \to E + \bullet T,\ T \to \bullet T * P\},\ T) =$$
$$= \text{CLOSURE}(\{E \to E + T \bullet,\ T \to T \bullet * P\}) = \{E \to E + T \bullet,\ T \to T \bullet * P\},$$
$$A_{10} = \text{GOTO}(A_7,\ P) = \text{GOTO}(\{T \to T * \bullet P\},\ P) =$$
$$= \text{CLOSURE}(\{T \to T * P \bullet\}) = \{T \to T * P \bullet\},$$
$$A_{11} = \text{GOTO}(A_8,\ (\ ) = \text{GOTO}(\{E \to (E \bullet)\},\ (\ ) =$$
$$= \text{CLOSURE}(\{E \to (E) \bullet\}) = \{E \to (E) \bullet\}.$$

В результате получаем окончательную систему $LR(0)$-множеств $\mathscr{C}$:

A_0	E'	→	$\bullet E$
	E	→	$\bullet E + T$
	E	→	$\bullet T$
	T	→	$\bullet T * P$
	T	→	$\bullet P$
	P	→	$\bullet(E)$
	P	→	$\bullet i$
A_1	E'	→	$E\bullet$
	E	→	$E\bullet + T$
A_2	E	→	$T\bullet$
	T	→	$T\bullet * P$
A_3	T	→	$P\bullet$
A_4	P	→	$(\bullet E)$
	E	→	$\bullet E + T$
	E	→	$\bullet T$
	T	→	$\bullet T * P$
	T	→	$\bullet P$
	P	→	$\bullet(E)$
	P	→	$\bullet i$

A_5	P	→	$I\bullet$
A_6	E	→	$E + \bullet T$
	T	→	$\bullet T * P$
	T	→	$\bullet P$
	P	→	$\bullet(E)$
	P	→	$\bullet i$
A_7	T	→	$T * \bullet P$
	P	→	$\bullet(E)$
	P	→	$\bullet i$
A_8	P	→	$(E\bullet)$
	E	→	$E\bullet + T$
A_9	E	→	$E + T\bullet$
	T	→	$T\bullet * P$
A_{10}	T	→	$T * P\bullet$
A_{11}	P	→	$(E)\bullet$

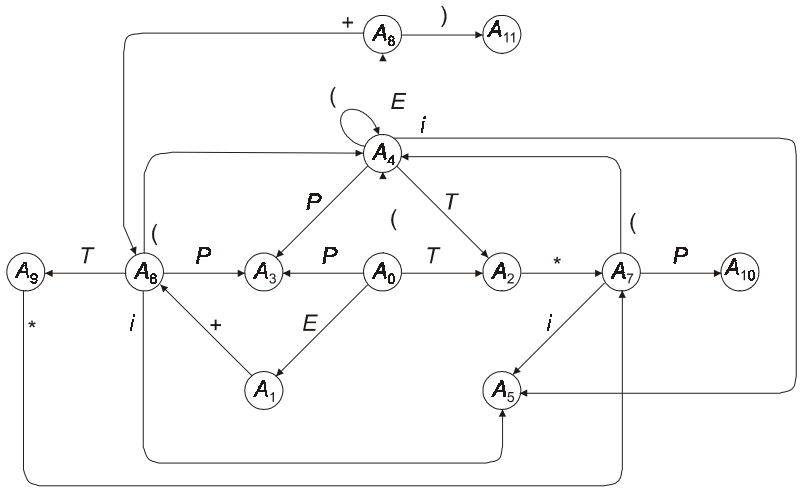


Рис. 8.5. Диаграмма переходов ДКА для активных префиксов грамматики G_0

Используя каноническую систему $LR(0)$-множеств, можно представить функцию GOTO в виде диаграммы детерминированного конечного автомата (рис. 8.5). □

8.5. Алгоритм построения анализатора для *LR*(0)-грамматики без ε-правил

Теперь мы можем более детально проанализировать работу $LR(0)$-анализатора. Рассмотрим анализатор, для грамматики с правилами:

(1) $S \rightarrow aAb$ (4) $A \rightarrow Bb$

(2) $S \rightarrow c$ (5) $B \rightarrow aA$

(3) $A \rightarrow bS$ (6) $B \rightarrow c$

управляющая таблица которого приведена в табл. 8.4.

Таблица 8.4. *LR(0)-анализатор*

T	f(u)				g(X)					
	a	b	c	⊥	a	b	c	S	A	B
S_0				Д						
a_1	П	П	П		a_5	b_3	c_6		A_1	B_4
A_1	П	П	П			b_1				
b_1	С,1	С,1	С,1	С,1						
c_2	С,2	С,2	С,2	С,2						
b_3	П	П	П		a_1		c_2	S_3		
S_3	С,3	С,3	С,3	С,3						
B_4	П	П	П			b_4				
b_4	С,4	С,4	С,4	С,4						
a_5	П	П	П		a_5	b_3	c_6		A_5	B_4
A_5	С,5	С,5	С,5	С,5						
c_6	С,6	С,6	С,6	С,6						
⊥	П	П	П		a_1		c_2	S_0		

Каждый символ магазинного алфавита V_p (грамматическое вхождение символа грамматики в правую часть правила вывода) можно интерпретировать как кодированное представление подцепочки из правой части правила, последним символом которой он является. При такой интерпретации магазинный символ вталкивается только тогда, когда символ из верхушки магазина является *кодированным представлением префикса подцепочки*, *представляемой вталкиваемым в магазин символом*, или когда вталкиваемый символ

представляет *новый префикс правой части какого-либо правила.* Перенос символов в магазин осуществляется до тех пор, пока в верхушке магазина не окажется символ, являющийся кодированным представлением основы. В этом случае цепочка магазинных символов, соответствующих основе, выталкивается из магазина, и в магазин вталкивается символ, который исполняет роль подцепочки, совместимой с подцепочкой, представляемой магазинным символом, расположенным под основой.

Рассмотрим, например, вторую строку управляющей таблицы, отмеченную грамматическим вхождением a_1, которое представляет собой префикс правой части правила (1). Пусть b — текущий входной символ. Согласно правилу (1), за a_1 может следовать грамматическое вхождение нетерминала A_1 или символ, представляющий подцепочку, порождаемую вхождением A_1. Цепочка, порождаемая A_1, может начинаться либо вхождением b_3 (правило (3)), либо вхождением B_4 (правило (4)), которое в свою очередь порождает цепочку, начинающуюся с a_5 или c_6. Следовательно, для входного символа b в магазин можно втолкнуть только грамматическое вхождение b_3, которое является кодированным представлением префикса правой части правила (3).

Строка таблицы, отмеченная маркером дна магазина ($\perp$), соответствует начальной конфигурации алгоритма. В первый момент времени в магазин можно записать только грамматическое вхождение a_1 или c_2, которые представляют собой префиксы цепочек, выводимых из начального вхождения S_0 (грамматического вхождения начального символа грамматики S, входящего в правую часть нулевого правила вывода пополненной грамматики).

Как следует из управляющей таблицы (табл. 8.4), функция действия $f(u)$ не зависит от текущего входного символа, а определяется только верхним символом магазина. Это означает, что для выбора нужного действия (переноса или свертки) алгоритм не должен заглядывать вперед на символы входной цепочки и, следовательно, $k = 0$. Алгоритм построения управляющей таблицы для $LR(0)$-грамматик основывается на рассмотрении пар грамматических вхождений, которые могут быть представлены соседними магазинными символами в процессе разбора допустимых цепочек.

Введем ряд понятий, которые понадобятся для построения управляющей таблицы.

Пусть X_i и Z_i — грамматические вхождения символов X и Z в правую часть i-го правила, а Y_j — грамматическое вхождение символа Y в правую часть j-го правила. Определим множество OFIRST(Y_i) (входит первым), в которое включим Y_j и все грамматические вхождения, с которых могут начинаться цепочки, выводимые из Y:

OFIRST(Y_i) = $\{Y_j\} \cup \{X_i \mid Y \Rightarrow^* A\beta \Rightarrow X\alpha\beta$ и X_i — самое левое грамматическое вхождение в правую часть правила $A \rightarrow X\}$.

Замечание

Если в грамматике есть правила с пустой правой частью, то на последнем шаге вывода они не применяются.

Используя множества OFIRST, определим отношение OBLOW (входит под) следующим образом:

- X_i OBLOW Y_j — это множество $\{(X_i, Y_j) \mid A \rightarrow \alpha X_i Z_i \beta \in P$ и $Y_j \in$ OFIRST $(Z_i)\}$.
- $\perp$ OBLOW Y_j — это множество $\{(\perp, Y_j) \mid Y_j$ OFIRST$(S_0)\}$, где S_0 — начальное вхождение.

Отношение X_i OBLOW Y_j определяет множество Q грамматических вхождений X_i, для которых представляющие их магазинные символы могут встретиться в магазине непосредственно под символом, представляющим Y_j.

Отношение OBLOW будем задавать с помощью матрицы, содержащей n столбцов и $(n + 1)$ строк, где n — число грамматических вхождений пополненной грамматики G'. Первые n строк матрицы отмечены грамматическими вхождениями, а последняя строка — маркером дна магазина. Если X_i OBLOW Y_j, то элемент матрицы, расположенный в строке X_i и столбце Y_j, равен 1.

Пример 8.6

🗁 Построим матрицу отношения OBLOW для грамматики с правилами:

(1) $S \rightarrow aAb$ (4) $A \rightarrow Bb$

(2) $S \rightarrow c$ (5) $B \rightarrow aA$

(3) $A \rightarrow bS$ (6) $B \rightarrow c$

Непосредственно из правил вывода грамматики (1) и (4) получим:

A_1 OBLOW b_1 и B_4 OBLOW b_4.

Из определения отношения OBLOW следует, что $\perp$ OBLOW Y_j тогда и только тогда, когда $Y_j \in$ OFIRST(S_0). Из S можно вывести цепочки $S \Rightarrow a_1 A_1 b_1$ и $S \Rightarrow c_2$. Следовательно, OFIRST$(S_0) = \{a_1, c_2, S_0\}$, и $\perp$ OBLOW a_1, $\perp$ OBLOW c_2 и $\perp$ OBLOW S_0.

Рассмотрим правило грамматики с номером (1). Из определения отношения OBLOW следует, что:

a_1 OBLOW Y_j для всех $Y_j \in$ OFIRST(A_1).

Из A можно вывести цепочки:

$A \Rightarrow b_3 S_3$, $A \Rightarrow B_4 b_4 \Rightarrow c_6 b_4$, $A \Rightarrow B_4 b_4 \Rightarrow a_5 A_5 b_4$.

Следовательно, OFIRST$(A_1) = \{a_5, b_3, c_6, A_1, B_4\}$ и a_1 OBLOW a_5, a_1 OBLOW b_3, a_1 OBLOW c_6, a_1 OBLOW A_1, a_1 OBLOW B_4.

Поступая подобным образом для правил (3) и (6), получим матрицу отношения OBLOW:

	S_0	a_1	A_1	b_1	c_2	b_3	S_3	B_4	b_4	a_5	A_5	c_6
S_0												
a_1			1			1		1		1		1
A_1				1								
b_1												
c_2												
b_3		1			1		1					
S_3												
B_4									1			
b_4												
a_5						1		1		1	1	1
A_5												
c_6												
$\perp$	1	1			1							

□

Опишем алгоритм построения управляющей таблицы $\mathscr{T}$ $LR(0)$-анализатора (множества $LR(0)$-таблиц) для $LR(0)$-грамматики, не содержащей правил с пустой правой частью. Этот алгоритм можно также использовать *для проверки принадлежности КС-грамматики классу LR(0).*

Алгоритм 8.2. *Построение множества LR(0)-таблиц*

Вход: $LR(0)$-грамматика $G = (N, \Sigma, P, S)$, не содержащая ε-правил.

Выход: Множество $\mathscr{T}$ $LR(0)$-таблиц для грамматики G или сообщение о том, что грамматика G не является $LR(0)$-грамматикой.

Описание алгоритма:

□

1. Построить пополненную грамматику G' для исходной грамматики G.
2. Вычислить отношения OBLOW для грамматических вхождений грамматики G'.

3. Определить функции переходов $g(X)$ следующим образом:
   - построить таблицу, содержащую по одному столбцу для каждого символа из $\Sigma \cup N \cup \{\varepsilon\}$ и одной строке для каждого грамматического вхождения грамматики G' и маркера дна. Элемент в строке, помеченной грамматическим вхождением X_i или маркером дна ($\perp$), и столбце, отмеченном символом грамматики Y, должен содержать все грамматические вхождения, для которых справедливо отношение X_i OBLOW Y_j. Заметам, что некоторые элементы построенной таким образом таблицы могут содержать более одного грамматического вхождения, т. е. таблица может быть *недетерминированной*;
   - интерпретируя построенную таблицу как *таблицу конечного автомата* (где состояния — это грамматические вхождения и маркер дна (начальное состояние), а входные символы — символы из $\Sigma \cup N \cup \{\varepsilon\}$), определить тип автомата: детерминированный или недетерминированный. Недетерминированный автомат преобразовать в эквивалентный ему детерминированный;
   - определить магазинный алфавит V_p так, чтобы каждому состоянию детерминированного конечного автомата соответствовал ровно один магазинный символ.

     В качестве символов алфавита V_p можно использовать любые символы, не являющиеся символами грамматики G' Для сохранения наглядности цепочек, представлямых магазином, будем считать, что символы из V_p совпадают по написанию с соответствующими грамматическими вхождениями. Если магазинный символ представляет множество грамматических вхождений, то индексы магазинных символов будем обозначать строчными латинскими буквами;
   - заменить совокупности грамматических вхождений, отмечающих состояния автомата, соответствующими символами из V_p.

   Полученная таблица представляет собой таблицу функций переходов $g(X)$ $LR(0)$-анализатора, причем элементы таблицы, соответствующие переходу в пустое множество состояний, имеют значение *ОШИБКА*.
4. Определить функции действия $f(a)$ для всех магазинных символов, каждому из которых соответствует одна строка таблицы. Количество столбцов таблицы $f(a)$ равняется количеству символов множества $\Sigma \cup \{\varepsilon\}$. Элементы таблицы $f(a)$ вычисляются следующим образом:
   - если магазинному символу T соответствует единственное вхождение S_0, то в строке, отмеченной символом T, $f(\varepsilon)$ = *ДОПУСК*, а все остальные элементы — *ОШИБКА*;
   - если магазинному символу T соответствует только одно грамматическое вхождение X_i, являющееся самым правым вхождением в i-е правило вывода грамматики G, то все элементы строки, помеченной T, имеют значение (*СВЕРТКА*, i);

- если магазинному символу T соответствует маркер дна магазина ($\perp$) или все грамматические вхождения, представляемые символом T, не являются самыми правыми в своих правилах, то в строке, отмеченной T, $f(\varepsilon)$ = *ОШИБКА*, а значения остальных элементов — *ПЕРЕНОС*;
- если множество вхождений, соответствующее магазинному символу T, содержит начальное вхождение S_0 и хотя бы еще одно вхождение, отличное от S_0, которое не является самым правым в своем правиле, то в строке, отмеченной T, $f(\varepsilon)$ = *ДОПУСК*, а значение всех остальных элементов — *ПЕРЕНОС*.

Если построение $f(a)$ закончено успешно, то грамматика является $LR(0)$-грамматикой, а таблица, полученная объединением таблиц, задающих функции $f(a)$ и $g(X)$, — управляющей таблицей $\mathcal{T}LR(0)$-анализатора.

■

Пример 8.7

🗁 Построим управляющую таблицу $LR(0)$-анализатора для грамматики, имеющей следующие правила:

(1) $E \to E + T$ (3) $T \to (E)$

(2) $E \to T$ (4) $T \to i$

Шаг 1. Определим пополненную грамматику для грамматики G:

$G' = (\mathrm{N} \cup \{S\}, \Sigma, P \cup \{S \to E\}, S)$.

Шаг 2. Вычислим отношение OBLOW для грамматических вхождений грамматики G'. Матрица отношения OBLOW имеет вид:

	E_0	E_1	$+_1$	T_1	T_2	$(_3$	E_3	$)_3$	i_4
E_0									
E_1			1						
$+_1$				1		1			1
T_1									
T_2									
$(_3$		1			1	1	1		1
E_3								1	
$)_3$									
i_4									
$\perp$	1	1			1	1			1

Шаг 3. Определим функции переходов $g(X)$:

- используя отношение OBLOW, построим таблицу переходов конечного (в данном случае недетерминированного) автомата:

	E	T	(	)	i	+
E_0						
E_1						$+_1$
$+_1$		T_1	$(_3$		i_4	
T_1						
T_2						
$(_3$	E_1, E_3	T_2	$(_3$		i_4	
E_3				$)_3$		
$)_3$						
i_4						
$\perp$	E_0, E_1	T_2	$(_3$		i_4	

- преобразуем недетерминированный автомат в эквивалентный ему детерминированный автомат, таблица переходов которого будет иметь вид:

	E	T	(	)	i	+
$\{\perp\}$	$(E_0, E_1\}$	$\{T_2\}$	$\{(_3\}$		$\{i_4\}$	
$\{E_0, E_1\}$						$\{+_1\}$
$\{T_2\}$						
$\{(_3\}$	$\{E_1, E_3\}$	$\{T_2\}$	$\{(_3\}$		$\{i_4\}$	
$\{i_4\}$						
$\{+_1\}$		$\{T_1\}$	$\{(_3\}$		$\{i_4\}$	
$\{E_1, E_3\}$				$\{)_3\}$		$\{+_1\}$
$\{T_1\}$						
$\{)_3\}$						

- определим множество магазинных символов:

	$\{E_0, E_1\}$	$\{E_1, E_3\}$	$\{\perp\}$	$\{T_2\}$	$\{(_3\}$	$\{i_4\}$	$\{+_1\}$	$\{T_1\}$	$\{)_3\}$
V_p	E_x	E_y	$\perp$	T_2	$(_3$	i_4	$+_1$	T_1	$)_3$

- тогда функции переходов $g(X)$ $LR(0)$-анализатора для грамматики G имеют вид:

$g(X)$	E	T	(	)	i	+
$\perp$	E_x	T_2	$(_3$		i_4	
E_x						$+_1$
T_2						
$(_3$	E_y	T_2	$(_3$		i_4	
i_4						
$+_1$		T_1	$(_3$		i_4	
E_y				$)_3$		$+_1$
T_1						
$)_3$						

Шаг 4. Определим функции действия $f(a)$. Построим таблицу, содержащую по одной строке для каждого магазинного символа и по одному столбцу для каждого символа $a \in (\Sigma \cup \{\varepsilon\})$.

Для строки таблицы, отмеченной символом $\perp$, соответствующее множество грамматических вхождений состоит из единственного символа $\perp$, поэтому в этой строке $f(\varepsilon)$ = *ОШИБКА*, а значения остальных элементов — *ПЕРЕНОС*.

Множество грамматических вхождений, соответствующих магазинному символу E_x, содержит два элемента: E_0 и E_1. Так как в это множество входит начальное вхождение E_0, а вхождение E_1 не является самым правым в правиле грамматики (1), то $f(\varepsilon)$ = *ДОПУСК*, а остальные элементы строки, отмеченные символом E_x, имеют значение *ПЕРЕНОС*.

Магазинному символу T_2 соответствует единственное грамматическое вхождение T_2, которое является самым правым в правиле вывода (2). Следовательно, значение всех элементов строки, помеченной T_2, будет равно (*СВЕРТКА*, 2).

Поступая подобным образом для остальных строк, получим таблицу функций переходов $f(a)$:

$f(a)$	(	)	i	+	ε
$\perp$	П	П	П	П	
E_x	П	П	П	П	Д
T_2	С,2	С,2	С,2	С,2	С,2
$(_3$	П	П	П	П	
i_4	С,4	С,4	С,4	С,4	С,4
$+_1$	П	П	П	П	
E_y	П	П	П	П	
T_1	С,1	С,1	С,1	С,1	С,1
$)_3$	С,3	С,3	С,3	С,3	С,3

Поскольку при построении функции переходов $f(a)$ не возникло конфликтных ситуаций, рассмотренная грамматика является $LR(0)$-грамматикой. Объединив функции $g(X)$ и $f(a)$ в одну таблицу, получим для управляющую таблицу $\mathcal{T}$ $LR(0)$-анализатора. □

8.6. Алгоритм построения анализатора для *SLR*(1)-грамматики без ε-правил

Усовершенствуем алгоритм 8.2 построения управляющей таблицы так, чтобы получить анализатор для некоторого класса грамматик, не принадлежащих классу $LR(0)$-грамматик. Этот класс является подклассом $LR(1)$-грамматик и получил название $SLR(1)$-грамматик без ε-правил. Буква S в названии грамматики является сокращением английского слова *Simple* (простой). Начнем с рассмотрения примера.

Пример 8.8

▱ Построим управляющую таблицу $LR(0)$-анализатора для грамматики G_0, правила вывода которой после пополнения имеют вид:

(0) $E' \to E$

(1) $E \to E + T$ (4) $T \to P$

(2) $E \to T$ (5) $P \to (E)$

(3) $T \to T * P$ (6) $P \to i$

В процессе построения анализатора получаем:

- матрицу отношения OBLOW, определенную на множестве грамматических вхождений пополненной грамматики:

	E_0	E_1	$+_1$	T_1	T_2	T_3	$*_3$	P_3	P_4	$(_5$	E_5	$)_5$	i_6
E_0													
E_1			1										
$+_1$				1		1			1	1			1
T_1													
T_2													
T_3							1						
$*_3$								1		1			1
P_3													
P_4													
$(_5$													
E_5		1			1	1			1	1	1		1
$)_5$												1	
i_6													
$\perp$	1	1			1	1			1	1			1

- таблицу переходов недетерминированного конечного автомата:

	+	*	(	)	*i*	*E*	*T*	*P*
⊥			$(_5$		i_6	E_0, E_1	T_2, T_3	P_4
E_0								
E_1	$+_1$							
$+_1$			$(_5$		i_6		T_1, T_3	P_4
T_1								
T_2								
T_3		$*_3$						
$*_3$			$(_5$		i_6			P_3
P_3								
P_4								
$(_5$			$(_5$		i_6	E_1, E_5	T_2, T_3	P_4
E_5								
$)_5$				$)_5$				
i_6								

- таблицу переходов детерминированного конечного автомата:

	+	*	(	)	i	*E*	*T*	*P*
{⊥}			$\{(_5\}$		$\{i_6\}$	{E_0, E_1}	{T_2, T_3}	{P_4}
$\{(_5\}$			$\{(_5\}$		$\{i_6\}$	{E_1, E_5}	{T_2, T_3}	{P_4}
$\{i_6\}$								
{E_0, E_1}	$\{+_1\}$							
{T_2, T_3}		$\{*_3\}$						
{P_4}								
{E_1, E_5}	$\{+_1\}$			$\{)_5\}$				
$\{+_1\}$			$\{(_5\}$		$\{i_6\}$		{T_1, T_3}	{P_4}
$\{*_3\}$			$\{(_5\}$		$\{i_6\}$			{P_3}
$\{)_5\}$								
{T_1, T_3}		$\{*_3\}$						
{P_3}								

- таблицу кодирования магазинных символов:

	$\{E_0, E_1\}$	$\{E_1, E_5\}$	$\{T_2, T_3\}$	$\{T_1, T_3\}$	$\{P_3\}$	$\{P_4\}$	$\{+_1\}$	$\{*_3\}$	$\{(_5\}$	$\{)_5\}$	$\{i_6\}$	$\{\perp\}$
V_p	E_x	E_y	T_x	T_y	P_3	P_4	$+_1$	$*_3$	$(_5$	$)_5$	i_6	⊥

- таблицу функций переходов $g(X)$:

$g(X)$	+	*	(	)	i	E	T	P
$\perp$			$(_5$		i_6	E_x	T_x	P_4
$(_5$			$(_5$		i_6	E_y	T_x	P_4
i_6								
E_x	$+_1$							
T_x		$*_3$						
P_4								
E_y	$+_1$			$)_5$				
$+_1$			$(_5$		i_6		T_y	P_4
$*_3$			$(_5$		i_6			P_3
$)_5$								
T_y		$*_3$						
P_3								

Далее, выполняя шаг 4 алгоритма, обнаруживаем, что G_0 не принадлежит классу $LR(0)$-грамматик, т. к. множества грамматических вхождений, соответствующие магазинным символам T_x и T_y, содержат самые правые вхождения символа T и не являются одноэлементными.

Покажем, как, сделав некоторый дополнительный анализ, можно определить функции действия $f(a)$ для данной грамматики. Анализ выполняется для каждого элемента строки таблицы отдельно и использует текущий входной символ ($k = 1$).

Рассмотрим, например, элемент управляющей таблицы для магазинного символа T_x, которому соответствует множество грамматических вхождений $\{T_2, T_3\}$. Наличие в T_x самого правого вхождения T_2 в правило вывода (2) говорит о том, что необходимо выполнить операцию (*СВЕРТКА*, 2), а тот факт, что в T_x содержится T_3, свидетельствует в пользу операции *ПЕРЕНОС*.

Допустим, что входным символом является символ (+). Для магазинного символа T_x функция $g(+)$ = *ОШИБКА*, поэтому втолкнуть его в магазин невозможно, и, следовательно, для допустимых входных цепочек в строке T_x функции действий должно быть значение $f(+)$ = (*СВЕРТКА*, 2).

Для входного символа '*' функция переходов $g(*) = *_3$, значит, при анализе допустимых входных цепочек перенос в магазин символа '*' будет правильным действием. Выполнение операции (*СВЕРТКА*, 2) в этом случае привело бы к тому, что в магазин был бы помещен магазинный символ, представляющий нетерминал E. Так как входным символом при этом остается '*', то на следующем шаге был бы выработан сигнал об ошибке, поскольку символ

'∗' не может непосредственно следовать за E (символ '∗' не принадлежит множеству FOLLOW(E)). Таким образом, операция (*СВЕРТКА*, 2) исключается, и в строке управляющей таблицы, помеченной T_x, $f(*)$ = *ПЕРЕНОС*. □

Принципы, продемонстрированные при выполнении данного примера, позволяют на основе алгоритма 8.2 сформулировать алгоритм построения управляющей таблицы для $SLR(1)$-грамматик.

Алгоритм 8.3. *Построение множества SLR(1)-таблиц*

Вход: $SLR(1)$-грамматика $G = (N, \Sigma, P, S)$, не содержащая ε-правил.

Выход: Множество $\mathscr{T}$ $SLR(1)$-таблиц для грамматики G или сообщение о том, что грамматика G не является $SLR(1)$-грамматикой.

Описание алгоритма:

□

1. Построить пополненную грамматику G' для исходной грамматики G.
2. Вычислить отношения OBLOW для грамматических вхождений грамматики G'.
3. Определить функции переходов $g(X)$ следующим образом:
   - построить таблицу, содержащую по одному столбцу для каждого символа из $\Sigma \cup N \cup \{\varepsilon\}$ и одной строке для каждого грамматического вхождения грамматики G' и маркера дна. Элемент в строке, помеченной грамматическим вхождением X_i или маркером дна ($\perp$), и столбце, отмеченном символом грамматики Y, должен содержать все грамматические вхождения, для которых справедливо отношение X_i OBLOW Y_j. Заметим, что некоторые элементы построенной таким образом таблицы могут содержать более одного грамматического вхождения, т. е. таблица может быть недетерминированной;
   - интерпретируя построенную таблицу как таблицу конечного автомата (где состояния — грамматические вхождения и маркер дна (начальное состояние), а входные символы — символы из $\Sigma \cup N \cup \{\varepsilon\}$), определить тип автомата: детерминированный или недетерминированный. Недетерминированный автомат преобразовать в эквивалентный ему детерминированный;
   - определить магазинный алфавит V_p так, чтобы каждому состоянию детерминированного конечного *автомата соответствовал ровно один магазинный* символ.

     В качестве символов алфавита V_p можно использовать любые символы, не являющиеся символами грамматики G'. Для сохранения наглядности

цепочек, представимых магазином, будем считать, что символы из V_p совпадают по написанию с соответствующими грамматическими вхождениями. Если магазинный символ представляет собой множество грамматических вхождений, то индексы магазинных символов будем обозначать строчными латинскими буквами;

- заменить совокупности грамматических вхождений, отмечающих состояния автомата, соответствующими символами из V_p.

Полученная таблица представляет собой таблицу функций переходов $g(X)$ $LR(0)$-анализатора, причем элементы таблицы, соответствующие переходу в пустое множество состояний, имеют значение *ОШИБКА*.

4. Для магазинного символа T, представляющего множество грамматических вхождений Q, и входного символа a значение функции действий $f(a)$ определяется следующим образом:
   - если начальное вхождение $S_0 \in Q$ и $a = \varepsilon$, то $f(\varepsilon) =$ *ДОПУСК*;
   - если $a \neq \varepsilon$ и $g(a) \neq$ *ОШИБКА*, то $f(a) =$ *ПЕРЕНОС*;
   - если $X_j \in Q$ — самое правое грамматическое вхождение i-го правила $A \rightarrow \alpha X_j$ и $a \in$ FOLLOW(A), то $f(a) =$ (*СВЕРТКА*, i). Значение остальных элементов таблицы для $f(a)$ — *ОШИБКА*.

Если имеется множество грамматических вхождений, не удовлетворяющих перечисленным в шаге 4 условиям, то грамматика не принадлежит классу $SLR(1)$.

Если построение $f(a)$ закончено успешно, то грамматика является $SLR(1)$-грамматикой, а таблица, полученная объединением таблиц, задающих функции $f(a)$ и $g(X)$, — управляющей таблицей $\mathcal{T}$ $SLR(1)$-анализатора.

■

Построим функцию действий $f(a)$ для грамматики G_0, используя шаг 4 алгоритма 8.3.

Начальное вхождение $E_0 \in E_x = \{E_0, E_1\}$, поэтому на основании первого правила шага 4 алгоритма $f(\varepsilon) =$ *ДОПУСК*.

Построим строку $f(a)$, помеченную состоянием T_y:

- $g(*) = *_3$, поэтому, используя второе правило построения из шага 4 алгоритма, $f(*) =$ *ПЕРЕНОС*;
- так как грамматическое вхождение $T_1 \in T_y = \{T_1, T_3\}$ является самым правым грамматическим вхождением в правило (1) $E \rightarrow E + T$ грамматики и FOLLOW(E) = {+,), ε}, получаем: $f(+) =$ (*СВЕРТКА*, 1), $f()) =$ (*СВЕРТКА*, 1), $f(\varepsilon) =$ (*СВЕРТКА*, 1), применяя третье правило из шага 4 алгоритма.

Поступая аналогичным образом, получим функцию действия $f(a)$:

$f(a)$	+	*	(	)	i	ε
⊥			П		П	
$(_5$			П		П	
i_6	С,6	С,6		С,6		С,6
E_x	П					Д
T_x	С,2	П		С,2		С,2
P_4	С,4	С,4		С,4		С,4
E_y	П			П		
$+_1$			П		П	
$*_3$			П		П	
$)_5$	С,5	С,5		С,5		С,5
T_y	С,1	П		С,1		С,1
P_3	С,3	С,3		С,3		С,3

Управляющая таблица (или совокупность $SLR(1)$-таблиц) $SLR(1)$-анализатора грамматики G_0 приведена в табл. 8.5.

Таблица 8.5. *SLR(1)-анализатор для грамматики G_0*

T	$f(a)$						$g(X)$							
	+	*	(	)	i	ε	+	*	(	)	i	E	T	P
⊥			П		П				$(_5$		i_6	E_x	T_x	P_4
$(_5$			П		П				$(_5$		i_6	E_y	T_x	P_4
i_6	С,6	С,6		С,6		С,6								
E_x	П					Д	$+_1$							
T_x	С,2	П		С,2		С,2		$*_3$						
P_4	С,4	С,4		С,4		С,4								
E_y	П			П			$+_1$			$)_5$				
$+_1$			П		П				$(_5$		i_6		T_y	P_4
$*_3$			П		П				$(_5$		i_6			P_3
$)_5$	С,5	С,5		С,5		С,5								
T_y	С,1	П		С,1		С,1		$*_3$						
P_3	С,3	С,3		С,3		С,3								

Пример 8.9

🗁 Рассмотрим работу построенного анализатора (табл. 8.5) при выполнении разбора цепочки $i + i * i$.

Шаг 1. Начальная конфигурация алгоритма равна $(\perp, i + i * i, \varepsilon)$.

Алгоритм выбирает строку управляющей таблицы, помеченную символом $(\perp)$ и, основываясь на значениях $f(i) = ПЕРЕНОС$ и $g(i) = i_6$, выполняет перенос и вталкивает в магазин грамматическое вхождение i_6, переходя в конфигурацию $(\perp i_6, + i * i, \varepsilon)$.

Шаг 2. Очередной шаг алгоритма определяется строкой управляющей таблицы, помеченной верхним символом магазина i_6.

Функция действий для очередного входного символа $(+)$ имеет значение $f(+) = (СВЕРТКА, 6)$, поэтому должна быть выполнена свертка цепочки в магазине, использующая правило (6) $P \rightarrow i$ грамматики. Символ, который должен быть помещен в магазин после свертки, определяется строкой таблицы, помеченной символом, находящимся в вершине магазина после вычеркивания основы (в данном случае — это $(\perp)$), и символом левой части основывающего правила (P).

В связи с тем, что $g(P) = P_4$, то алгоритм переходит в конфигурацию $(\perp P_4, + i * i, 6)$.

Продолжая работу, алгоритм проходит следующую последовательность конфигураций:

$$
\begin{aligned}
(\perp P_4, + i * i, 6) \; &\vdash_{r6} (\perp P_4, + i * i, 6) && \vdash_{r4} (\perp T_x, + i * i, 64)\\
&\vdash_{r2} (\perp E_x, + i * i, 642) && \vdash_{s} (\perp E_x +_1, i * i, 642)\\
&\vdash_{s} (\perp E_x +_1 i_6, * i, 642) && \vdash_{r6} (\perp E_x +_1 P_4, * i, 6426)\\
&\vdash_{r4} (\perp E_x +_1 T_y, * i, 64264) && \vdash_{s} (\perp E_x +_1 T_y *_3, i, 64264)\\
&\vdash_{s} (\perp E_x +_1 T_y *_3 i_6, \varepsilon, 64264) && \vdash_{r6} (\perp E_x +_1 T_y *_3 P_3, \varepsilon, 642646)\\
&\vdash_{r3} (\perp E_x +_1 T_y, \varepsilon, 6426463) && \vdash_{r1} (\perp E_x, \varepsilon, 64264631)\\
&\vdash_{s} ДОПУСК.
\end{aligned}
$$

Алгоритм достиг состояния *ДОПУСК*, поэтому входная цепочка $i + i * i$ принадлежит языку, определяемому грамматикой G_0, и имеет правый разбор 64264631. 🗀

8.7. Включение ε-правил в *LR*(0)- и *SLR*(1)-грамматики

Задача построения управляющей таблицы для $LR(0)$- и $SLR(1)$-грамматик с ε-правилами осложняется тем, что на каждом шаге работы анализатора можно считать, что верхушка магазина совпадает с правой частью любого ε-правила (например, с номером i), и, следовательно, имеются основания

полагать, что функция действия $f(a)$ для всех аргументов имеет значение (*СВЕРТКА*, i).

Доказано, что для данной пары магазинного T и входного a символов и правила вывода $A \to \varepsilon$ с номером i в строке T управляющей таблицы $\mathcal{T}$ $f(a) = (СВЕРТКА, i)$ тогда и только тогда, когда $a \in \text{FOLLOW}(A)$ и $g(A) \neq ОШИБКА$.

Первые три шага алгоритмов 8.2 и 8.3 построения управляющих таблиц для $LR(0)$- и $SLR(1)$-грамматик, описанные ранее, остаются без изменений, т. к. добавление к грамматике или удаление из нее ε-правил не изменяет множества грамматических вхождений и, следовательно, не влияет на отношение OBLOW.

Для учета ε-правил шаг 4 алгоритмов построения управляющей таблицы необходимо расширить следующим действием:

> если $A \to \varepsilon$ — правило вывода с номером i и множество:
>
> $\text{FOLLOW}(A) = \{a_j \mid 1 \le j \le m$, где m — количество символов в $\Sigma \cup \{\varepsilon\}\}$,
>
> то $f(a_j) = (СВЕРТКА, i)$ в тех строках управляющей таблицы, для которых $g(A) \neq ОШИБКА$.

Пример 8.10

🗁 Построим управляющую таблицу для приведенной $SLR(1)$-грамматики, правила вывода которой имеют вид:

(0) $S' \to S$ (3) $A \to \varepsilon$

(1) $S \to bABd$ (4) $B \to Bc$

(2) $A \to Aa$ (5) $B \to \varepsilon$

Используя алгоритм 8.2, построим:

- матрицу отношения OBLOW:

	S_0	b_1	A_1	B_1	d_1	A_2	a_2	B_4	c_4
S_0									
b_1			1			1			
A_1				1				1	
B_1					1				
d_1									
A_2							1		
a_2									
B_4									1
c_4									
$\perp$	1	1							

- таблицу переходов недетерминированного конечного автомата:

	S	A	B	a	b	c	d
S_0							
b_1		A_1, A_2					
A_1			B_1, B_2				
B_1							d_1
d_1							
A_2				a_2			
a_2							
B_4						c_4	
c_4							
$\perp$	S_0				b_1		

- таблицу переходов детерминированного конечного автомата:

	S	A	B	a	b	c	d
$\{S_0\}$							
$\{b_1\}$		$\{A_1, A_2\}$					
$\{A_1, A_2\}$			$\{B_1, B_4\}$	$\{a_2\}$			
$\{B_1, B_4\}$						$\{c_4\}$	$\{d_1\}$
$\{a_2\}$							
$\{c_4\}$							
$\{d_1\}$							
$\{\perp\}$	$\{S_0\}$				$\{b_1\}$		

- магазинный алфавит анализатора:

	$\{S_0\}$	$\{b_1\}$	$\{A_1, A_2\}$	$\{B_1, B_4\}$	$\{a_2\}$	$\{c_4\}$	$\{d_1\}$	$\{\perp\}$
V_p	S_0	b_1	A_x	B_x	a_2	c_4	d_1	$\perp$

После применения алгоритма построения управляющей таблицы $SLR(1)$-анализатора для грамматики G без ε-правил получим управляющую таблицу $\mathscr{T}$, приведенную в табл. 8.6 (в ячейках, выделенных серым цветом, в этот момент находятся значения *ОШИБКА*).

Теперь выполним расширенный шаг 4 алгоритма.

Для правила (3) имеем: FOLLOW(A) = {a, c, d} и $g(A) = A_x$ в строке, отмеченной символом b_1. Следовательно, в этой строке $f(a)$ = (*СВЕРТКА*, 3), $f(c)$ = (*СВЕРТКА*, 3) и $f(d)$ = (*СВЕРТКА*, 3).

Таблица 8.6. *SLR(1)-анализатор*

T	$f(u)$					$g(X)$						
	a	b	c	d	ε	S	A	B	a	b	c	d
S_0					Д							
b_1	С,3		С,3	С,3			A_x					
A_x	П		С,5	С,5				B_x	a_2			
B_x			П	П							c_4	d_1
a_2	С,2		С,2	С,2								
c_4			С,4	С,4								
d_1					С,1							
⊥		П				S_0				b_1		

Аналогично для правила (5): FOLLOW(B) = {c, d} и $g(B) = B_x$ в строке, отмеченной символом A_x. Таким образом, в строке A_x $f(c)$ = (*СВЕРТКА*, 5) и $f(d)$ = (*СВЕРТКА*, 5).

В заключение используем построенный алгоритм (см. табл. 8.6) для анализа входной цепочки *bacd*, правое порождение которой равно 14523.

Алгоритм выполнит следующую последовательность шагов:

$$
\begin{aligned}
(\perp, bacd, \varepsilon) &\vdash_s (\perp b_1, acd, \varepsilon) & &\vdash_{r3} (\perp b_1 A_x, acd, 3)\\
&\vdash_s (\perp b_1 A_x a_2, cd, 3) & &\vdash_{r2} (\perp b_1 A_x, cd, 32)\\
&\vdash_{r5} (\perp b_1 A_x B_x, cd, 325) & &\vdash_s (\perp b_1 A_x B_x c_4, d, 325)\\
&\vdash_{r4} (\perp b_1 A_x B_x, d, 3254) & &\vdash_s (\perp b_1 A_x B_x d_1, \varepsilon, 3254)\\
&\vdash_{r1} (\perp S_0, \varepsilon, 32541) & &\vdash_s \textit{ДОПУСК}.
\end{aligned}
$$

и выдаст правый разбор входной цепочки *bacd* равный 32541. □

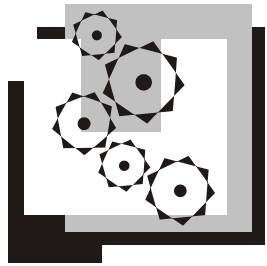
Контрольные вопросы

1. Как строится магазинный алфавит для алгоритма типа "перенос-свертка"?
2. Дайте определение $LR(k)$-грамматики. В чем особенности этого класса грамматик?
3. Что такое "грамматическое вхождение"?
4. Перечислите основные функции алгоритма $LR(k)$-алгоритма разбора. Как для него определяются функции действия и перехода?

5. Как строится конечный автомат для распознавания активных префиксов? Этот автомат детерминированный или нет?
6. Дайте определения множеств OFIRST и OBLOW. Каким образом они могут быть вычислены?
7. Какие изменения нужно внести в алгоритм построения SLR(1)-анализатора для учета правил с пустой правой частью?

Упражнения

1. Постройте управляющую таблицу и промоделируйте работу $LR(0)$-анализатора для КС-грамматики $G = (N, \Sigma, P, S)$ с правилами:
   1.1. $P = \{S \to aSb,\ S \to aSc,\ S \to ab\}$.
   1.2. $P = \{S \to aSSb,\ S \to aSSS,\ S \to c\}$.
   1.3. $P = \{S \to a,\ S \to (\ SR,\ R \to \hat{}\ SR,\ R \to)\ \}$.
   1.4. $P = \{S \to cA,\ S \to ccB,\ A \to cA,\ A \to a,\ B \to ccB,\ B \to b\}$.
   1.5. $P = \{S \to aAd,\ S \to bAc,\ A \to e,\ A \to \varepsilon\}$.
2. Постройте управляющую таблицу и промоделируйте работу $SLR(1)$-анализатора для КС-грамматики $G = (N, \Sigma, P, S)$ с правилами:
   2.1. $P = \{S \to bASB,\ S \to bA,\ A \to dSca,\ A \to \varepsilon,\ B \to cAa,\ B \to c\}$.
   2.2. $P = \{S \to Bv,\ S \to vC,\ A \to u,\ A \to vBS,\ B \to u,\ B \to yw,\ C \to Bv,\ C \to yAw\}$.
   2.3. $P = \{S \to B,\ S \to A,\ B \to C;D,\ C \to bd,\ C \to C;d,\ D \to ae,\ D \to a;D,\ A \to bD\}$.
   2.4. $P = \{S \to aA,\ S \to bB,\ A \to 1A0,\ A \to \varepsilon,\ B \to 1B00,\ B \to \varepsilon\}$.
   2.5. $P = \{S \to bABe,\ A \to Ad;,\ B \to B;C,\ B \to C,\ C \to a,\ C \to \varepsilon,\ A \to \varepsilon\}$.



Глава 9

Грамматики предшествования

Существует подкласс $LR(k)$-грамматик, который называется грамматиками предшествования. Основным достоинством грамматик предшествования является то, что для них легко построить алгоритм типа "перенос-свертка". Методы синтаксического анализа, основанные на таких грамматиках, были описаны в литературе одними из первых [6, 10, 28].

Как уже отмечалось, при восходящих методах синтаксического анализа в текущей сентенциальной форме выполняется поиск *основы* α, которая в соответствии с правилом грамматики $A \to \alpha$ сворачивается к нетерминальному символу A. Основная проблема восходящего синтаксического анализа заключается в поиске основы и нетерминального символа, к которому ее нужно приводить (сворачивать).

9.1. Понятие отношений предшествования

Попробуем, рассматривая только два соседних символа сентенциальной формы, найти *хвост* и *голову основы*. Пусть имеется сентенциальная форма $\alpha X_1 X_2 \beta$, где $X_1, X_2 \in N \cup \Sigma$. На некотором этапе разбора либо символ X_1, либо символ X_2, либо они вместе должны войти в основу. Рассмотрим три следующих случая:

X_1 — часть основы, символ X_2 не входит в основу (рис. 9.1, *а*). Символ X_1 будет свернут *раньше* X_2. В этом случае говорят, что символ X_1 *предшествует* X_2, и записывают это как $X_1 \cdot\!> X_2$. Очевидно, что символ X_1 — последний символ основы (*хвост основы*), т. е. последний символ правой части некоторого правила, а символ X_2 — терминальный символ.

Оба символа X_1 и X_2 входят в основу, т. е. сворачиваются *одновременно* (рис. 9.1, *б*). Обычно этот факт обозначают как $X_1 \doteq X_2$. В этом случае символы X_1 и X_2 должны входить в правую часть некоторого правила грамматики.

Символ X_2 — часть основы, а символ X_1 не входит в основу (рис. 9.1, *в*). Символ X_2 должен быть свернут *раньше* X_1. Это записывают как $X_1 <\!\cdot X_2$.

Очевидно, что в этом случае символ X_1 — это первый символ основы (*голова основы*), т. е. первый символ правой части некоторого правила.

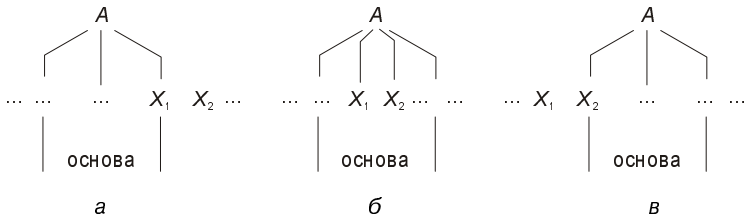


Рис. 9.1. Три случая вхождения символов X_1 и X_2 в основу сентенциальной формы $\alpha X_1 X_2 \beta$

Отношения, обозначаемые символами ($<\cdot$), ($\doteq$) и ($\cdot>$), называются *отношениями предшествования*.

Рассмотрим грамматику с правилами:

(1) $S \rightarrow bAb$ (3) $A \rightarrow a$

(2) $A \rightarrow cB$ (4) $B \rightarrow Aad$

Для того чтобы определить отношения между символами грамматики, необходимо рассмотреть некоторое множество сентенциальных форм. На рис. 9.2 приведены три сентенциальные формы, их синтаксические деревья с основами и отношения, которые можно определить при их анализе.

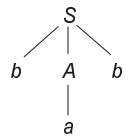
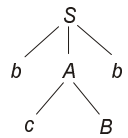
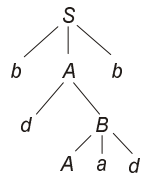
Сентенциальная форма:	*b a b*	*bcBb*	*bcAadb*
Синтаксическое дерево:			
Основа:	*a*	*cB*	*Aad*
Отношения, определяемые деревом:	$b <\cdot a$ $a \cdot> b$	$b <\cdot c$ $c \doteq B$ $B \cdot> b$	$c <\cdot A$ $A \doteq a$ $a \doteq d$ $d \cdot> b$

Рис. 9.2. Анализ сентенциальных форм

Замечание

Такой способ построения отношений не конструктивен, и в дальнейшем будут даны правила построения отношений, применимые к любой грамматике.

Все отношения предшествования, существующие между символами рассматриваемой грамматики, приведены в форме *матрицы отношений предшествования* в табл. 9.1. Строки и столбцы матрицы помечены нетерминальными и терминальными символами грамматики, а ее элементы содержат значения отношения предшествования.

Таблица 9.1. *Матрица отношений предшествования*

	S	*A*	*B*	*a*	*b*	*c*	*d*
S							
A				$\doteq$	$\doteq$		
B				$\cdot>$	$\cdot>$		
a				$\cdot>$	$\cdot>$		$\doteq$
b		$\doteq$		$<\cdot$		$<\cdot$	
c		$<\cdot$	$\doteq$	$<\cdot$		$<\cdot$	
d				$\cdot>$	$\cdot>$		

В данной грамматике между парами ее символов определено не более одного отношения предшествования. Это позволяет достаточно просто использовать матрицу отношений предшествования при анализе цепочек. Далее будет доказано, что основой *любой* сентенциальной формы $X_1 \ldots X_n$ является самая левая подцепочка $X_j \ldots X_i$, такая, что между символами выполняются следующие отношения предшествования:

$$X_{j-1} <\cdot X_j \doteq X_{j+1} \doteq X_{j+2} \doteq \ldots \doteq X_i \cdot> X_{i+1}.$$

Пример 9.1

Учитывая вышесказанное и используя матрицу отношений предшествования (табл. 9.1), рассмотрим разбор цепочки *bcaadb*.

Шаг 1. Начальная сентенциальная форма — это входная цепочка *bcaadb*. Используя матрицу отношений предшествования, получим следующие отношения между ее символами:

$$b <\cdot c <\cdot \boldsymbol{a} \cdot> a \doteq d \cdot> b.$$

Основой (выделенной полужирным шрифтом) является левый символ *a*, т. к. отношение его с предшествующим символом имеет значение "сворачивается позже" (<·), а с последующим символом — "сворачивается раньше" (·>). Для свертки нужно использовать правило грамматики (3) *A* → *a*, поэтому основа должна быть заменена символом *A* (левая часть правила). После замены получаем вторую сентенциальную форму *bcAadb*.

Шаг 2. Отношения между символами сентенциальной формы следующие:

$$b <\cdot c <\cdot \boldsymbol{A} \doteq \boldsymbol{a} \doteq \boldsymbol{d} \cdot> b.$$

Основа — цепочка *Aad*. Для свертки должно использоваться правило (4) *B* → *Aad*. После замены основы символом *B* получаем очередную сентенциальную форму *bcBb*.

Шаг 3. Анализируем отношения, существующие в сентенциальной форме:

$$b <\cdot \boldsymbol{c} \doteq \boldsymbol{B} \cdot> b.$$

Основа этой сентенциальной формы — цепочка *cB*. Для свертки должно использоваться правило (2) *A* → *cB*. Основа заменяется символом *A*. Очередная сентенциальная форма — *bAb*.

Шаг 4. Отношения между символами полученной сентенциальной формы выглядят так:

$$\boldsymbol{b} \doteq \boldsymbol{A} \doteq \boldsymbol{b}.$$

Основа сентенциальной формы — цепочка *bAb*. Для свертки должно использоваться правило (1) *S* → *bAb*. Основа заменяется символом *S*.

Анализ исходной цепочки закончен, а ее разбор равен 3421. □

На основании рассмотренного примера можно сделать предположение, что алгоритм типа "перенос-свертка" может быть легко реализован для грамматик предшествования.

9.2. Алгоритм типа "перенос-свертка"

Алгоритм типа "перенос-свертка" имеет входную ленту, читаемую слева направо, и магазин (см. рис. 8.3). Анализ входной цепочки с помощью этого алгоритма состоит в переносе входных символов в магазин до тех пор, пока в его верхней части не встретится основа. В этот момент производится свертка, в результате которой основа заменяется *левой частью* основывающего правила. Этот процесс продолжается до тех пор, пока вся входная цепочка не будет прочитана, а в магазине не останется начальный символ грамматики или алгоритм не выдаст сообщение об ошибке.

Определим формально алгоритм типа "перенос-свертка", применяемый для грамматик предшествования.

Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика, правила которой пронумерованы числами от 1 до p. *Алгоритмом типа "перенос-свертка"* для грамматики G назовем пару функций $\mathcal{A} = (f, g)$, где f называется *функцией переноса,* а g — *функцией свертки.* Функции f и g определяются следующим образом:

1. f отображает $(N \cup \Sigma \cup \{\perp\})^* \times (\Sigma \cup \{\varepsilon\})^*$ в множество {*ПЕРЕНОС*, *СВЕРТКА*, *ОШИБКА*, *ДОПУСК*}.
2. g отображает $(N \cup \Sigma \cup \{\perp\})^* \times (\Sigma \cup \{\varepsilon\})^*$ в множество $\{1, 2, \ldots, p,$ *ОШИБКА*$\}$. Если $g(\alpha, \omega) = i$, то правая часть i-го правила является суффиксом цепочки α.

Работу алгоритма типа "перенос-свертка" удобно описывать в терминах *конфигураций* вида $(\perp X_1 \ldots X_m, a_1 \ldots a_n, p_1 \ldots p_r)$, где:

- $\perp X_1 \ldots X_m$ — содержимое магазина, X_m — верхний символ магазина и $X_i \in N \cup \Sigma$, символ $(\perp)$ — символ дна магазина;
- $a_1 \ldots a_n$ — оставшаяся непрочитанной часть входной цепочки, a_1 — текущий входной символ;
- $p_1 \ldots p_r$ — разбор, полученный к данному моменту времени.

Один шаг алгоритма $\mathcal{A}$ можно описать с помощью двух отношений: $(\vdash^s)$ (перенос) и $(\vdash^r)$ (свертка), определенных на конфигурациях следующим образом:

1. Если $f(\alpha, aw) =$ *ПЕРЕНОС*, то входной символ переносится в верхушку магазина и читающая головка сдвигается на один символ вправо. В терминах конфигураций этот процесс описывается так:

   $(\alpha, aw, \pi) \vdash^s (\alpha a, w, \pi)$ для $\alpha \in (N \cup \Sigma \cup \{\perp\})^*$, $w \in (\Sigma \cup \{\varepsilon\})^*$ и $\pi \in \{1, \ldots, p\}^*$.

2. Если $f(\alpha\beta, w) =$ *СВЕРТКА*, $g(\alpha\beta, w) = i$ и $A \to \beta$ — правило грамматики с номером i, то цепочка β заменяется правой частью правила с номером i, а его номер помещается на выходную ленту, т. е. $(\alpha\beta, w, \pi) \vdash^r (\alpha A, w, \pi i)$.

3. Если $f(\alpha, w) =$ *ДОПУСК*, то $(\alpha, w, \pi) \vdash^s$ *ДОПУСК*.

В остальных случаях $(\alpha, w, \pi) \vdash^s$ *ОШИБКА*.

Замечание

Отношение $(\vdash)$ определяется как объединение отношений $(\vdash^s)$ и $(\vdash^r)$, а транзитивные замыкания отношений $(\vdash^+)$ и $(\vdash^*)$ определяются как обычно.

Для $w \in \Sigma^*$ будем записывать $\mathcal{A}(w) = \pi$, если $(\perp, w, \varepsilon) \vdash^* (\perp S, \varepsilon, \pi) \vdash^s$ *ДОПУСК*, и $\mathcal{A}(w) =$ *ОШИБКА*, в противном случае.

Рассмотрим работу алгоритма типа "перенос-свертка" для следующей грамматики:

(1) $S \to bAb$ (3) $A \to a$

(2) $A \to cB$ (4) $B \to Aad$

В табл. 9.2 при описании функции переноса f алгоритма $\mathscr{A}$ использованы следующие обозначения: П — *ПЕРЕНОС*, С — *СВЕРТКА*, Д — *ДОПУСК*. Если значение в таблице отсутствует, то значение соответствующей функции — *ОШИБКА*. В табл. 9.3 приведена функция свертки g алгоритма $\mathscr{A}$.

Таблица 9.2. *Функция переноса*

f	ax	bx	cx	dx	ε
αA	П	П			
αB	С	С			
αa	С	С		П	
αb	П		П		С
αc	П		П		
αd	С	С			
$\perp$		П			
$\perp S$					Д

Таблица 9.3. *Функция свертки*

g	ax	bx	x	ε
$\perp bAb$				1
αAad		4		
αcB	2	2		
αa			3	

Пример 9.2

🗁 Разберем с помощью построенного алгоритма $\mathscr{A}$ входную цепочку *bcaadb*.

Начальная конфигурация алгоритма равна ($\perp$, *bcaadb*, ε).

Шаг 1. Выполнение алгоритма определяется значением $f(\perp, bcaadb)$, которое, как видно из определения функции f, имеет значение *ПЕРЕНОС*. Алгоритм переходит в конфигурацию ($\perp b$, *caadb*, ε), выполняя шаг:

$$(\perp, bcaadb, \varepsilon) \vdash^{s} (\perp b, caadb, \varepsilon).$$

Аналогично выполняются несколько следующих шагов:

$(\perp b, caadb, \varepsilon) \vdash^s$

$(\perp bc, aadb, \varepsilon) \vdash^s$

$(\perp bca, adb, \varepsilon)$.

Шаг 2. Очередной шаг выполнения алгоритма определяется значением $f(\perp bca, adb)$, которое имеет значение *СВЕРТКА*. Правило, используемое при свертке, определяется значением функции $g(\perp bca, adb) = 3$. Алгоритм выполняет свертку, используя правило 3, и переходит в следующую конфигурацию:

$(\perp bca, adb, \varepsilon) \vdash^r (\perp bcA, adb, 3)$.

Продолжая свою работу, алгоритм $\mathscr{A}$ сделает такую последовательность шагов:

$(\perp bcA, adb, 3) \vdash^s$

$(\perp bcAa, db, 3) \vdash^s$

$(\perp bcAad, b, 3) \vdash^r$

$(\perp bcB, b, 34) \vdash^r$

$(\perp bA, b, 342) \vdash^s$

$(\perp bAb, \varepsilon, 342) \vdash^r$

$(\perp S, \varepsilon, 3421) \vdash^s$ *ДОПУСК.*

Таким образом, $\mathscr{A}(bcaadb) = 3421$, а значит, правый вывод цепочки *bcaadb* должен быть равен 1243.

Для проверки построим цепочку по заданному правому выводу:

$S \Rightarrow_{(1)} bAb \Rightarrow_{(2)} bcBb \Rightarrow_{(4)} bcAadb \Rightarrow_{(3)} bcaadb.$

Алгоритм корректно разобрал заданную входную цепочку. Синтаксическое дерево цепочки *bcaadb* приведено на рис. 9.3. □

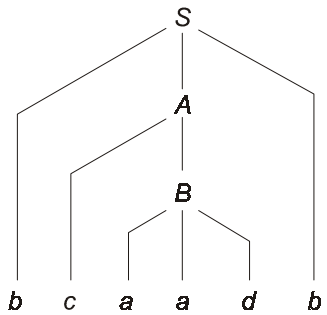


Рис. 9.3. Синтаксическое дерево цепочки *bcaadb*

Для практических целей рассмотренный алгоритм мало пригоден, т. к. для определения функции переноса он требует анализа всей цепочки в магазине и всей необработанной части входной цепочки. Аналогичный недостаток имеет алгоритм и при определении функции свертки.

Желательно, чтобы функция переноса зависела только от небольшого числа верхних символов магазина и от небольшого числа следующих входных символов, а функция свертки — от символов магазина, расположенных в нем не ниже, чем на один или два символа от сворачиваемой основы, и от одного или двух следующих входных символов.

Замечание

В алгоритме, приведенном в табл. 9.2 и 9.3, функция *f* зависит фактически только от верхнего символа магазина и следующего входного символа, а для функции *g* требуется анализ только одного символа ниже основы и следующего входного символа.

Простейший класс алгоритмов типа "перенос-свертка" основан на *отношениях предшествования*. Для грамматики предшествования границы основы *правовыводимой* цепочки можно определить с помощью отношений предшествования между символами, входящими в эту цепочку.

Разбор, основанный на отношениях предшествования, использовался при построении первых анализаторов для языков программирования. В литературе описано несколько вариантов грамматик предшествования. Мы рассмотрим основанный на отношениях предшествования детерминированный анализ, выполняющий правый разбор. При этом будут введены грамматики предшествования следующих типов:

- простого предшествования;
- слабого предшествования;
- операторного предшествования.

Для всех классов грамматик предшествования общим является способ поиска правого конца основы (*хвоста* основы). Если для разбора применяется алгоритм типа "перенос-свертка", то всякий раз, когда между верхним символом магазина и первым из необработанных входных символов выполняется отношение $(\cdot>)$, принимается решение о свертке, в противном случае делается перенос. Таким образом, с помощью отношения $(\cdot>)$ локализуется правый конец основы правовыводимой цепочки. Определение левого конца основы (*головы* основы) и нужной свертки осуществляется разными способами в зависимости от используемого типа отношений предшествования.

9.3. Грамматики простого предшествования

Синтаксический анализ, основанный на *простом предшествовании*, использует для выделения основы правовыводимой цепочки $\alpha\beta w$ три отношения предшествования $(<\cdot)$, $(\doteq)$ и $(\cdot>)$ следующим образом:

- если β — основа, то между всеми смежными символами цепочки α выполняется либо отношение ($<\cdot$), либо ($\doteq$);
- между последним символом цепочки α и первым символом цепочки β выполняется отношение ($<\cdot$);
- между смежными символами основы выполняется отношение ($\doteq$);
- между последним символом цепочки β и первым символом цепочки w выполняется отношение ($\cdot>$).

Очевидно, что правый конец основы правовыводимой цепочки грамматики простого предшествования можно выделить, просматривая эту цепочку слева направо до тех пор, пока впервые не встретится отношение ($\cdot>$). Для нахождения левого конца основы надо просмотреть ее назад, пока не встретится отношение ($<\cdot$). Цепочка, заключенная между отношениями ($<\cdot$) и ($\cdot>$), будет основой. Если грамматика является обратимой, т. е. не содержит правил с одинаковой правой частью, то основу можно однозначно свернуть. Этот процесс продолжается до тех пор, пока входная цепочка не свернется к начальному символу (либо пока дальнейшие свертки окажутся невозможными).

Отношения предшествования для КС-грамматики $G = (N, \Sigma, P, S)$ определяются на множестве $(N \cup \Sigma \cup \{\perp\}) \times (N \cup \Sigma \cup \{\varepsilon\})$ следующим образом:

- $X <\cdot Y$, если в множестве правил грамматики P есть правило $A \to \alpha X B \beta$ и существует вывод $B \Rightarrow^+ Y\gamma$;
- $X \doteq Y$, если в P содержится правило вида $A \to \alpha X Y \beta$;
- $X \cdot> a$, если в P есть правило вида $A \to \alpha B Y \beta$ и существуют выводы $B \Rightarrow^+ \gamma X$ и $Y \Rightarrow^* a\delta$ (если $Y \Rightarrow^0 a\delta$, то $Y = a$);
- $\perp <\cdot X$ для всех X, для которых $S \Rightarrow^+ X\alpha$;
- $Y \cdot> \varepsilon$ для всех Y, для которых $S \Rightarrow^+ \alpha Y$.

Замечание

Отношение ($\cdot>$) определяется на $(N \cup \Sigma \cup \{\perp\}) \times (\Sigma \cup \{\varepsilon\})$, т. к. справа от основы в правовыводимой цепочке может быть только терминальный символ.

Вычислять отношения предшествования для небольших грамматик достаточно просто, используя их определения. Практические алгоритмы определения отношений предшествования рассмотрены далее. Теперь можно перейти к определению грамматик предшествования.

Дадим ряд определений.

КС-грамматика G называется *приведенной*, если в ней нет выводов вида $A \Rightarrow^+ A$, бесполезных символов и ε-правил, за исключением правила $S \to \varepsilon$, причем в этом случае S не встречается в правых частях правил.

КС-грамматика $G = (N, \Sigma, P, S)$ называется *грамматикой предшествования*, если она приведенная, не содержит ε-правил и для любой пары символов из множества $N \cup \Sigma$ выполняется не более одного отношения предшествования.

Обратимая грамматика предшествования называется *грамматикой простого предшествования.*

Язык, порождаемый грамматикой простого предшествования, называется *языком простого предшествования.*

Обычно отношения предшествования представляют в виде квадратной матрицы, которую называют *матрицей отношений предшествования.*

В матрице простого предшествования строки отмечаются символами из $N \cup \Sigma \cup \{\perp\}$, а столбцы — символами из $N \cup \Sigma \cup \{\varepsilon\}$. На пересечении строки с номером i и столбца с номером j записывается отношение предшествования между символами X_i и Y_j. Элементами матрицы являются знаки отношений предшествования: $(\lessdot)$, $(\doteq)$, $(\gtrdot)$. Если между символами X_i и Y_j нет отношения предшествования, то элемент матрицы — пусто.

Пример 9.3

🗁 Построим матрицу простого предшествования для грамматики G_0 с правилами:

(1) $E \to E + T$ (4) $T \to P$

(2) $E \to T$ (5) $P \to i$

(3) $T \to T * P$ (6) $P \to (E)$

Для этих целей воспользуемся определениями отношений простого предшествования, стараясь применять систематический подход к их построению.

Шаг 1. Отношение $(\doteq)$ определяется непосредственно по правилам грамматики G_0.

Шаг 2. Определим отношение между терминальными символами '(' и '('.

Предположим, что между ними существует отношение $(\lessdot)$. Тогда, по определению, в грамматике должно быть правило вывода $A \to \alpha(B\beta$ и должен существовать вывод $B \Rightarrow^+ (\gamma$.

В грамматике есть правило $P \to (E)$ и существует вывод $E \Rightarrow T \Rightarrow P \Rightarrow (E)$. Следовательно, $(\lessdot ($.

Шаг 3. Теперь предположим, что между этими символами существует отношение $(\gtrdot)$. Тогда, по определению, в грамматике должно быть правило $A \to \alpha BY\beta$ и должны существовать два вывода: $B \Rightarrow^+ \gamma($ и $Y \Rightarrow^* (\delta$.

Поскольку в грамматике нельзя вывести цепочки, оканчивающиеся символом "(", предположение неверно, и между символами '(' и '(' имеет место единственное отношение (<·).

Поступая подобным образом, мы получим матрицу отношений простого предшествования, приведенную в табл. 9.4.

Таблица 9.4. *Матрица отношений предшествования грамматики* G_0

	E	*T*	*P*	*i*	(	)	+	*	ε
E						≐	≐		
T						·>	·>	≐	·>
P						·>	·>	·>	·>
i						·>	·>	·>	·>
(	<· ≐	<·	<·	<·	<·				
)						·>	·>	·>	·>
+		<· ≐	<·	<·	<·				
*			≐	<·	<·				
⊥	<·	<·	<·	<·	<·				

Из анализа матрицы отношений следует, что грамматика G_0 *не является грамматикой простого предшествования*, т. к. для пар символов ((, *E*) и (+, *T*) существует более одного отношения предшествования. □

Замечание

Отношения (≐), (<·) и (·>) не обладают свойствами, которые обычно приписываются отношениям (=), (<) и (>), заданным на вещественных или целых числах. Например, (≐) обычно не является отношением эквивалентности, отношения (<·) и (·>) не транзитивны, но могут быть симметричными или рефлексивными.

Для автоматизированного построения отношений простого предшествования можно использовать алгоритм, основанный на отношениях FIRSTN и LASTN, определенных на правилах грамматики [10].

Определим отношения FIRSTN(*A*) и LASTN(*A*) для нетерминального символа *A* грамматики следующим образом:

$$\text{FIRSTN}(A) = \{X \mid A \to X\alpha \in P\},\ \text{LASTN}(A) = \{X \mid A \to \alpha X \in P\},$$

т. е. множество FIRSTN — это множество нетерминальных символов, с которых могут начинаться цепочки, выводимые в грамматике из нетерминала A, а множество LASTN — множество нетерминальных символов, которыми могут заканчиваться такие цепочки.

Отношения FIRSTN и LASTN для КС-грамматики $G = (\text{N}, \Sigma, P, S)$ определим как:

$$\text{FIRSTN} = \bigcup_{i=1}^{\#\text{N}} \text{FIRSTN}(A_i),\ \text{LASTN} = \bigcup_{i=1}^{\#\text{N}} \text{LASTN}(A_i),$$

где #N — мощность множества нетерминальных символов N грамматики G.

Тогда отношение простого предшествования $(\lessdot\cdot)$ можно вычислить как произведение отношений $(\doteq)$ и FIRSTN^+, где FIRSTN^+ — транзитивное замыкание отношения FIRSTN, т. е.

$$(\lessdot\cdot) = (\doteq)\ (\text{FIRSTN}^+). \tag{9.1}$$

Отношение простого предшествования для пары символов $X \cdot\!\!> a$, где $a \in \Sigma$ и $\mathbf{E}$ — единичное отношение, определяется из соотношения:

$$X\ ((\text{LASTN}^+)^{-1})\ (\doteq)\ (\mathbf{E} + \text{FIRSTN}^+)\ a. \tag{9.2}$$

Если представлять отношения при помощи булевских матриц и использовать известные алгоритмы для определения транзитивного замыкания [10, 23], то вычисление отношений предшествования станет рутинной задачей.

Вычислим, например, отношения предшествования для грамматики с правилами:

(1) $S \to b\ A\ b$ (3) $A \to a$

(2) $A \to c\ B$ (4) $B \to A\ a\ d$

Отношения FIRSTN и LASTN определяются по грамматике тривиально:

FIRSTN	*S*	*A*	*B*	*a*	*b*	*c*	*d*
S					1		
A				1		1	
B		1					

LASTN	*S*	*A*	*B*	*a*	*b*	*c*	*D*
S					1		
A			1	1			
B							1

Транзитивные замыкания этих отношений будут определяться как:

FIRSTN$^+$	*S*	*A*	*B*	*a*	*b*	*c*	*d*
S					1		
A				1		1	
B		1				1	

LASTN$^+$	*S*	*A*	*B*	*a*	*b*	*c*	*d*
S					1		
A			1	1			1
B							1

Представим отношения при помощи булевых матриц, строки и столбцы которых обозначены символами грамматики в следующем порядке: *S*, *A*, *B*, *a*, *b*, *c*, *d*. Тогда после вычислений по формуле (9.1) получим:

$$\begin{bmatrix} 0&0&0&0&0&0&0\\ 0&0&0&1&1&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&1\\ 0&1&0&0&0&0&0\\ 0&0&1&0&0&0&0\\ 0&0&0&0&0&0&0 \end{bmatrix} \begin{bmatrix} 0&0&0&0&1&0&0\\ 0&0&0&1&0&1&0\\ 0&1&0&1&0&1&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0 \end{bmatrix} = \begin{bmatrix} 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&1&0&1&0\\ 0&1&0&1&0&1&0\\ 0&0&0&0&0&0&0 \end{bmatrix},$$

что соответствует следующим значениям отношения предшествования ($<\cdot$):

	S	*A*	*B*	*a*	*b*	*c*	*d*
S							
A							
B							
a							
b				$<\cdot$		$<\cdot$	
c		$<\cdot$		$<\cdot$		$<\cdot$	
d							

При вычислении отношений предшествования ($\cdot>$) по формуле (9.2) получаем:

$$\mathrm{LASTN}^{+} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad (\mathrm{LASTN}^{+})^{-1} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$((\mathrm{LASTN}^{+})^{-1})(\doteq) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix},$$

$$(\mathbf{E} + \mathrm{FIRSTN}^{+}) = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

$$((\mathrm{LASTN}^{+})^{-1})(\doteq)(\mathbf{E} + \mathrm{FIRSTN}^{+}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix},$$

что соответствует следующим значениям отношения предшествования ($\cdot>$):

	S	*A*	*B*	*a*	*b*	*c*	*d*
S							
A							
B				$\cdot>$	$\cdot>$		
a				$\cdot>$	$\cdot>$		
b							
c							
d				$\cdot>$	$\cdot>$		

Отношения предшествования обладают свойствами, которые используются при распознавании языков предшествования.

Лемма 9.1

Пусть задана приведенная КС-граммаитика $G = (N, \Sigma, P, S)$, не содержащая ε-правил.

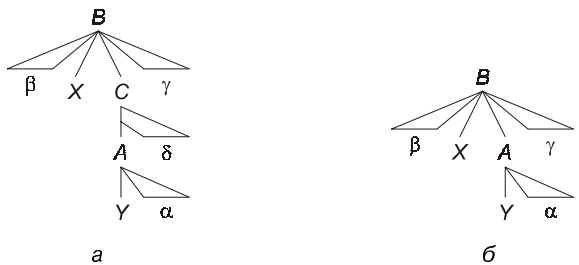
1. Если $X <\cdot A$ или $X \doteq A$ и $A \to Y\alpha$ содержится в P, то $X <\cdot Y$.
2. Если $A <\cdot a$, $A \doteq a$ или $A \cdot> a$ и $A \to Y\alpha$ содержится в P, то $Y \cdot> a$.

Доказательство

1. Докажем первое утверждение.

   □ Предположим, что $X <\cdot A$. По условию в грамматике должно быть правило $B \to \beta XC\gamma$ и вывод $C \Rightarrow^+ A\delta \Rightarrow Y\alpha\delta$, а значит $X <\cdot Y$ (рис. 9.4, *а*).

   Пусть $X \doteq A$. Тогда по условию в грамматике должно быть правило $B \to \beta XA\gamma$ и вывод $A \Rightarrow Y\alpha$, откуда получаем $X <\cdot Y$ (рис. 9.4, *б*). ■



а *б*

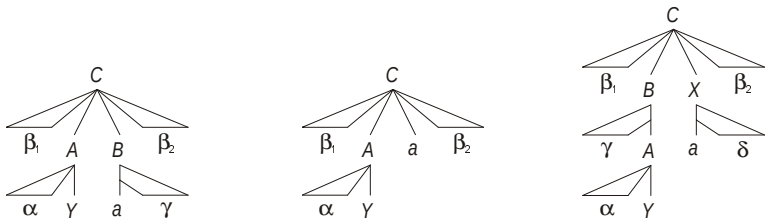
Рис. 9.4. Иллюстрации к доказательству первого утверждения леммы 9.1

2. Перейдем к доказательству второго утверждения.

   □ Предположим, что $A <\cdot a$. По условию леммы в грамматике должно быть правило $C \to \beta_1 AB\beta_2$ и вывод $B \Rightarrow^+ a\gamma$. Также по условию $A \Rightarrow \alpha Y$, т. е. $Y \cdot> a$ (рис. 9.5, *а*).

Пусть $A \doteq a$. По условию леммы в грамматике должно быть правило $C \to \beta_1 A a \beta_2$ и вывод $A \Rightarrow \alpha Y$, т. е. $Y \cdot\!> a$ (рис. 9.5, *б*).

Предположим теперь, что $A \cdot\!> a$. По условию леммы в грамматике должно быть правило $C \to \beta_1 B X \beta_2$ и выводы $B \Rightarrow^+ \gamma A$ и $X \Rightarrow^* a\delta$. Так как $A \Rightarrow \alpha Y$, то $Y \cdot\!> a$ (рис. 9.5, *в*). ■



а *б* *в*

Рис. 9.5. Иллюстрации к доказательству второго утверждения леммы 9.1

Для правовыводимой цепочки $\alpha\beta w$, в которой β — основа, между всеми смежными символами цепочки α выполняется либо отношение ($<\cdot$), либо ($\doteq$). Между последним символом цепочки α и первым символом цепочки β выполняется отношение ($<\cdot$), между смежными символами основы — отношение ($\doteq$), и между последним символом цепочки β и первым символом цепочки w — отношение ($\cdot>$). Приведем доказательство этого утверждения.

Теорема 9.1

Пусть $G = (N, \Sigma, P, S)$ — приведенная КС-грамматика без ε-правил. Если существует вывод:

$$\begin{aligned} \perp S &\Rightarrow^n_r X_p X_{p-1} \ldots X_{k+1} A\, a_1 \ldots a_q \\ &\Rightarrow_r X_p X_{p-1} \ldots X_{k+1} X_k \ldots X_1\, a_1 \ldots a_q, \end{aligned}$$

то:

1. $X_{i+1} <\cdot X_i$ или $X_{i+1} \doteq X_i$ для $p < i < k$.
2. $X_{k+1} <\cdot X_k$.
3. $X_{i+1} \doteq X_i$ для $k > i \geq 1$.
4. $X_1 \cdot\!> a_1$.

Доказательство

□ Доказательство теоремы проведем индукцией по длине вывода n.

Для $n = 0$ имеем $\perp S \Rightarrow_r \perp X_k X_{k-1} \ldots X_1$. По определению отношений предшествования $\perp <\cdot X_k$, $X_{i+1} \doteq X_i$ для $k > i \geq 1$ и $X_1 \cdot\!> \varepsilon$. По условию теоремы КС-грамматика не содержит ε-правил, поэтому $X_k X_{k-1} \ldots X_1 \neq \varepsilon$.

Предположим, что теорема верна для n. Рассмотрим вывод:

$$\begin{aligned} \perp S &\Rightarrow^n_r X_p \ldots X_{k+1} A\, a_1 \ldots a_q \\ &\Rightarrow_r X_p \ldots X_{k+1} X_k \ldots X_1\, a_1 \ldots a_q \\ &\Rightarrow_r X_p \ldots X_{j+1} Y_r \ldots Y_1 X_{j-1} \ldots X_1\, a_1 \ldots a_q, \end{aligned}$$

в котором на последнем шаге X_j заменяется цепочкой $Y_r \ldots Y_1$. Выполняется правый вывод, поэтому $X_{j-1} \ldots X_1$ — терминальные символы (возможно, $j = 1$).

По предположению индукции $X_{j+1} <\cdot X_j$ или $X_{j+1} \doteq X_j$. Поэтому по утверждению 1 леммы 9.1 $X_{j+1} <\cdot Y_r$. Кроме этого, X_j находится в одном из трех отношений с символом справа от него (который может быть равен a_1). Таким образом, $Y_1 \cdot> X_{j-1}$ либо $Y_1 \cdot> a_1$ при $j = 1$. Так как $Y_r \ldots Y_1$ — правая часть правила грамматики, то $Y_r \doteq Y_{r-1} \doteq \ldots \doteq Y_1$. По предположению индукции $X_{i+1} \cdot> X_i$ или $X_{i+1} \doteq X_i$ для $p < i < j$. Что и требовалось доказать. ■

Следствие 1

Если G — грамматика предшествования, то утверждение 1 теоремы 9.1 можно усилить, добавив слова "*точно одно из отношений* ($<\cdot$) *или* ($\doteq$)". Утверждения 1—4 можно усилить, добавив слова "и *не выполняются никакие другие отношения*".

Доказательство этого утверждения непосредственно следует из определения грамматики предшествования.

Следствие 2

Каждая грамматика простого предшествования однозначна.

Доказательство

□ Для любой правовыводимой цепочки β (отличной от S) предшествующая ей в выводе правовыводимая цепочка α ($\alpha \Rightarrow^r \beta$) определяется однозначно. Из следствия 1 вытекает, что основу цепочки β можно однозначно определить, просматривая эту цепочку, ограниченную концевыми маркерами, слева направо, пока впервые не обнаружится отношение ($\cdot>$), а затем вернуться назад до ближайшего ($<\cdot$). Основа лежит между ними. Так как грамматика простого предшествования обратима, то нетерминал, к которому надо свернуть основу, находится однозначно. Таким образом, α определяется по β однозначно. ■

Теперь опишем, как по грамматике простого предшествования можно построить *детерминированный* правый анализатор типа "перенос-свертка".

Алгоритм 9.1. *Построение анализатора типа "перенос-свертка" для грамматик простого предшествования*

Вход: Грамматика простого предшествования $G = (\Sigma, N, S, P)$, в которой правила вывода пронумерованы натуральными числами 1, 2, ... , p.

Выход: Алгоритм $\mathcal{A} = (f, g)$ типа "перенос-свертка" для грамматики G.

Описание алгоритма:

□ Функция переноса f зависит только от верхнего символа магазина и самого левого необработанного символа входной цепочки. Поэтому f определяется на множестве $(\Sigma \cup N \cup \{\perp\}) \times (\Sigma \cup \{\varepsilon\})$ (за исключением правила 3, которое имеет приоритет над правилами 1 и 2 в том случае, когда $X = S$ и $a = \varepsilon$):

1. $f(X, a) =$ *ПЕРЕНОС*, если $X \lessdot a$ или $X \doteq a$.
2. $f(X, a) =$ *СВЕРТКА*, если $X \cdot\!\!> a$.
3. $f(\perp S, \varepsilon) =$ *ДОПУСК*.
4. $f(X, a) =$ *ОШИБКА* в остальных случаях.

Функция свертки g зависит от верхней части магазинной цепочки, включающей основу и один символ ниже ее. Необработанная часть входной цепочки на g не влияет, поэтому g задается только на множестве $(\Sigma \cup N \cup \{\perp\})^*$:

1. $g(X_{k+1} X_k X_{k-1} \dots X_1, \varepsilon) = i$, если $X_{k+1} \lessdot X_k$, $X_{j+1} \doteq X_j$ для $k > j \geq 1$ и $A \to X_k X_{k-1} \dots X_1$ — правило вывода с номером i. Заметим, что функция g определяется только тогда, когда $X_1 \cdot\!\!> a$, где a — текущий входной символ.
2. $g(\alpha, \varepsilon) =$ *ОШИБКА* в остальных случаях.

■

Пример 9.4

🗁 Построим алгоритм разбора типа "перенос-свертка" для грамматики простого предшествования с правилами вывода:

(1) $S \to aSSb$

(2) $S \to c$

Для определения значений функций переноса и свертки необходимо предварительно построить отношения простого предшествования. Матрица отношений простого предшествования для этой грамматики имеет вид:

	S	a	b	c	ε
S	$\doteq$	$\lessdot$	$\doteq$	$\lessdot$	
a	$\doteq$	$\lessdot$		$\lessdot$	
b		$\cdot\!\!>$	$\cdot\!\!>$	$\cdot\!\!>$	$\cdot\!\!>$
$\perp$		$\lessdot$		$\lessdot$	

Функцию переносов f строим непосредственно по матрице предшествования: $f(X, a) = ПЕРЕНОС$, если $X \lessdot a$ или $X \doteq a$, и $f(X, a) = СВЕРТКА$, если $X \gtrdot a$. После добавления строки, соответствующей правилу 3 алгоритма 9.1, получим следующую функцию переносов f:

$f(X,a)$	a	b	c	ε
S	П	П	П	
a	П		П	
b	С	С	С	С
c	С	С	С	С
$\perp$	П		П	
$\perp S$				Д

Для построения функции свертки рассмотрим все возможные комбинации основ и символа, который может располагаться в магазине непосредственно под основой.

Грамматика содержит два правила и, следовательно, две основы: $aSSb$ и c. По матрице предшествования определяем множества символов, которые могут находиться под каждой из основ.

Для правила грамматики (1) — это символы (S, a, $\perp$), находящиеся в отношении ($\lessdot$) с символом a, а для правила (2) — символы (S, a, $\perp$), находящиеся в отношении ($\lessdot$) с символом c.

Таким образом, $g(\alpha) = 1$ для множества цепочек $\{SaSSb, aaSSb, \perp aSSb\}$ и $g(\alpha) = 2$ для множества цепочек $\{Sc, ac, \perp\}$. В остальных случаях $g(\alpha) = ОШИБКА$:

α		$g(\alpha)$
S	$aSSb$	1
a	$aSSb$	1
$\perp$	$aSSb$	1
S	c	2
a	c	2
$\perp$	c	2

Рассмотрим работу алгоритма для цепочки *acaccb*, *принадлежащей* языку, определяемому рассматриваемой грамматикой. Для этой цепочки алгоритм выполнит следующую последовательность шагов:

$(\perp, aaccbcb, \varepsilon) \vdash^s (\perp a, accbcb, \varepsilon)$

$\vdash^s (\perp aa, ccbcb, \varepsilon)$

$\vdash^s (\perp aac, cbcb, \varepsilon)$

$\vdash^s (\perp aaS, cbcb, 2)$

$\vdash^s (\perp aaSc, bcb, 2)$

$\vdash^r (\perp aaSS, bcb, 22)$

$\vdash^s (\perp aaSSb, cb, 22)$

$\vdash^r (\perp aS, cb, 221)$

$\vdash^s (\perp aSc, b, 221)$

$\vdash^r (\perp aSS, b, 2212)$

$\vdash^s (\perp aSSb, \varepsilon, 2212)$

$\vdash^r (\perp S, \varepsilon, 22121)$

$\vdash^s$ *ДОПУСК.*

В качестве результата алгоритм выдает разбор входной цепочки 22121.

Приведем поведение алгоритма для цепочки, *не принадлежащей* языку $L(G)$. Для цепочки *acab* алгоритм сделает следующую последовательность шагов:

$(\perp, acab, \varepsilon) \vdash^s (\perp a, cab, \varepsilon)$

$\vdash^s (\perp ac, ab, \varepsilon)$

$\vdash^r (\perp aS, ab, 2)$

$\vdash^s (\perp aSa, b, 2)$

$\vdash^s$ *ОШИБКА.*

Достигнув значения функции переносов, равного *ОШИБКА*, алгоритм прекращает свою работу. □

Синтаксический анализ языков простого предшествования можно выполнять, используя не только алгоритм типа "перенос-свертка", но и другие алгоритмы. Например, используя матрицу отношений предшествования, можно читать символы входной цепочки и записывать их в магазин до тех пор, пока верхний символ магазина не будет находиться в отношении $(\cdot>)$ со следующим входным символом. Это означает, что верхний символ магазина — хвост основы, а вся основа находится в магазине. Просмотрев верхнюю часть магазина, можно найти голову основы и правило грамматики для выполнения свертки, а затем выполнить свертку.

9.4. Грамматики слабого предшествования

Многие грамматики, описывающие языки программирования, не являются грамматиками простого предшествования. Попытки найти для данного языка грамматику простого предшествования могут привести к довольно громоздким грамматикам. Можно расширить класс грамматик, анализируемых методом предшествования, исключив ограничение, что отношения ($<\cdot$) и ($\doteq$) не могут пересекаться.

Для грамматик слабого предшествования отношение ($\cdot>$) по-прежнему будет использоваться для определения правого конца основы, а для локализации левого конца основы нужно будет найти правило грамматики, правая часть которого совпадает с символами, находящимися непосредственно слева от правого конца основы.

При использовании такого подхода могут возникнуть трудности, связанные с тем, что среди правил вывода грамматики могут быть такие, правые части которых являются суффиксами правых частей других правил. Например, если $\alpha\beta\gamma w$ — правовыводимая цепочка, в которой правый конец основы находится между символами цепочек γ и w, а $A \to \gamma$ и $B \to \beta\gamma$ — два разных правила грамматики, то не ясно, какое из них следует выбрать для свертки.

Допустим, что в случае конфликта правил для свертки мы выбираем самое длинное из применимых правил. Класс грамматик, для которых такое решение является правильным, образуют грамматики *слабого предшествования*.

Пусть $G = (N, \Sigma, S, P)$ — приведенная грамматика без ε-правил. Назовем G *грамматикой слабого предшествования*, если:

- отношение ($\cdot>$) не пересекается с объединением отношений ($<\cdot$) и ($\doteq$);
- для правил $A \to \alpha X \beta$ и $B \to \beta$ из P, где $X \in N \cup \Sigma$, не выполняется ни отношение $X <\cdot B$, ни отношение $X \doteq B$.

Вернемся к рассмотрению грамматики G_0. Матрица отношений простого предшествования для этой грамматики приведена в табл. 9.4.

Очевидно, что конфликты возникают только между отношениями ($<\cdot$) и ($\doteq$), и, следовательно, первое условие определения грамматики слабого предшествования выполняется.

Рассмотрим правила грамматики $E \to E + T$ и $E \to T$, в которых одна правая часть служит суффиксом другой. Между символами (+) и E нет никакого отношения предшествования, значит, для этих правил второе условие определения не нарушается. Аналогичное заключение можно сделать для правил (3) и (4) грамматики. Таким образом, G_0 — грамматика слабого предшествования.

Докажем, что в правовыводимой цепочке грамматики слабого предшествования основой всегда будет *самая длинная из правых частей правил*, применение которых возможно на данном шаге анализа [3].

Лемма 9.2

Пусть $G = (N, \Sigma, S, P)$ — грамматика слабого предшествования и в множестве правил P имеется правило $B \to \beta$. Пусть в грамматике существует правый вывод $\perp S \Rightarrow^*_r \gamma Cw \Rightarrow_r \delta X\beta w$. Если в грамматике есть правило $A \to \alpha X\beta$, то последним в этом выводе применялось не правило $B \to \beta$.

Доказательство

□ Допустим, что при выводе последним применялось правило $B \to \beta$. Это означает, что в выводе $\perp S \Rightarrow^*_r \gamma Cw \Rightarrow_r \delta X\beta w$ символ $C = B$ и цепочка $\gamma = \delta X$. Для вывода $S \Rightarrow^*_r \gamma Cw$ по теореме 9.1 получим, что $X <\cdot B$ или $X \doteq B$. Это следует из того, что основа цепочки γCw по предположению оканчивается правее символа C, и, значит, C — один из символов X_i этой теоремы. Полученное нарушение условия слабого предшествования доказывает лемму. ■

Лемма 9.3

Пусть $G = (N, \Sigma, S, P)$ — обратимая грамматика слабого предшествования, множество правил которой содержит правило $B \to \beta$, и существует правый вывод $\perp S \perp \Rightarrow^*_r \gamma Cw \Rightarrow_r \delta X\beta w$. Если правила вида $A \to \alpha X\beta$ в грамматике нет , то в выводе $\perp S \perp \Rightarrow^*_r \gamma Cw \Rightarrow_r \delta X\beta w$ должно быть $C = B$ и $= X$, т. е. последним применялось правило $B \to \beta$.

Доказательство

□ По условию леммы на последнем шаге вывода сворачивается символ C. Так как в грамматике нет правила $A \to \alpha X\beta$, то левый конец основы цепочки $\delta X\beta w$ не может быть левее символа X. Если основа оканчивается правее первого символа цепочки β, то нарушается лемма 9.2, в которой правило $B \to \beta$ играет роль правила $A \to \alpha X\beta$. Поэтому основа — это цепочка β. Если учесть, что по условию грамматика является обратимой, то лемма доказана. ■

Перейдем теперь к рассмотрению алгоритма разбора. Неформально алгоритм разбора для обратимых грамматик слабого предшествования работает следующим образом:

- алгоритм просматривает правовыводимую цепочку, ограниченную концевыми маркерами, слева направо до тех пор, пока впервые не встретит отношение $(\cdot>)$. Это отношение указывает на правый конец основы;
- алгоритм по одному просматривает символы слева от $(\cdot>)$.

Допустим, что в грамматике есть правило $B \to \beta$ и слева от $(\cdot>)$ находится цепочка $X\beta$. Если правило вида $A \to \alpha X\beta$ отсутствует в грамматике, то по лемме 9.3 цепочка β является основой. Если такое правило есть, то по лемме 9.2 правило $B \to \beta$ неприменимо. Следовательно, решение о том, свертывать ли β, можно принять, рассмотрев только один символ слева от β.

Заметим, что аналогичным образом работает алгоритм типа "перенос-свертка". Значит, для любой обратимой грамматики слабого предшествования можно построить такой алгоритм. Приведем алгоритм построения анализатора для грамматик слабого предшествования.

Алгоритм 9.2. *Построение анализатора типа "перенос-свертка" для грамматик слабого предшествования*

Вход: Грамматика слабого предшествования $G = (N, \Sigma, S, P)$, в которой правила вывода пронумерованы натуральными числами 1, 2, ... , p.

Выход: Алгоритм $\mathcal{A} = (f, g)$ типа "перенос-свертка" для грамматики G.

Описание алгоритма:

□ Функция переноса f зависит только от верхнего символа магазина и самого левого необработанного символа входной цепочки. Поэтому f определяется на множестве $(\Sigma \cup N \cup \{\perp\}) \times (\Sigma \cup \{\varepsilon\})$ (за исключением правила 3, которое имеет приоритет над правилами 1 и 2, когда $X = S$ и $a = \varepsilon$):

1. $f(X, a) =$ *ПЕРЕНОС*, если $X <\cdot a$ или $X \doteq a$.
2. $f(X, a) =$ *СВЕРТКА*, если $X \cdot> a$.
3. $f(\perp S, \varepsilon) =$ *ДОПУСК*.
4. $f(X, a) =$ *ОШИБКА* в остальных случаях.

Функция свертки g зависит от верхней части магазинной цепочки, включающей основу и еще один символ ниже нее. Необработанная часть входной цепочки на g не влияет, поэтому g задается только на множестве $(\Sigma \cup N \cup \{\perp\})^*$. Функция свертки g *определяется* так, чтобы при свертке *применялось самое длинное правило* из списка альтернатив.

1. $g(X, \beta) = i$, если $B \to \beta \in P$ и имеет номер i, и в P нет правила вида $A \to \alpha X\beta$.
2. $g(\alpha) =$ *ОШИБКА* в остальных случаях.

■

Построим алгоритм типа "перенос-свертка" для грамматики слабого предшествования G_0.

Функция переноса f (табл. 9.5) строится по матрице отношений предшествования (табл. 9.4) и не требует пояснений.

Таблица 9.5. *Функция переноса алгоритма "перенос-свертка" для грамматики G_0*

$f(X, a)$	i	(	)	+	*	ε
E			П	П		
T			С	С	П	С
P			С	С	С	С
i			С	С	С	С
(	П	П				
)			С	С	С	С
+	П	П				
*	П	П				
⊥	П	П				
⊥E						Д

Функция свертки g также строится с использованием матрицы отношений предшествования. Цепочка β соответствует правой части правила грамматики G_0, а X — первый символ слева от основы, т. е. между этим символом и первым символом основы β должно выполняться отношение (<·) (табл. 9.6).

Таблица 9.6. *Функция свертки алгоритма "перенос-свертка" для грамматики G_0*

α		$g(\alpha)$	α		$g(\alpha)$
X	β		X	β	
(	$E + T$	1	⊥	P	4
⊥	$E + T$	1	(	(E)	5
(	T	2	+	(E)	5
⊥	T	2	*	(E)	5
(	$T * P$	3	⊥	(E)	5
+	$T * P$	3	(	i	6
⊥	$T * P$	3	+	i	6
(	P	4	*	i	6
+	P	4	⊥	i	6

Последовательность тактов, которую выполнит анализатор при разборе входной цепочки $i + i * i$, имеет следующий вид:

$$
\begin{aligned}
(\perp, i + i * i, \varepsilon) &\vdash^{s} (\perp i, + i * i, \varepsilon)\\
&\vdash^{r} (\perp P, + i * i, 6)\\
&\vdash^{r} (\perp T, + i * i, 64)\\
&\vdash^{r} (\perp E, + i * i, 642)\\
&\vdash^{s} (\perp E +, i * i, 642)\\
&\vdash^{s} (\perp E + i, * i, 642)\\
&\vdash^{r} (\perp E + P, * i, 6426)\\
&\vdash^{r} (\perp E + T, * i, 64264)\\
&\vdash^{s} (\perp E + T *, i, 64264)\\
&\vdash^{s} (\perp E + T * i, \varepsilon, 64264)\\
&\vdash^{r} (\perp E + T * P, \varepsilon, 642646)\\
&\vdash^{r} (\perp E + T, \varepsilon, 6426463)\\
&\vdash^{r} (\perp E, \varepsilon, 64264631)\\
&\vdash^{s} \textit{ДОПУСК}.
\end{aligned}
$$

Таким образом, цепочка $i + i * i$ принадлежит языку, определяемому грамматикой G_0.

Замечание

Пусть алгоритм находится в конфигурации ($\perp E + T * P$, ε, 642646). На очередном шаге алгоритма при поиске номера правила для выполнения свертки необходимо проанализировать верхнюю часть магазина. При этом оказывается, что, согласно функции свертки g, возможно выполнение сверки по правилам:

(3) $T \to T * P$ или (4) $T \to P$.

Естественно, что верным решением является выполнение свертки по правилу (3). Значит функция g должна быть построена так, чтобы *при свертке применялось самое длинное из применимых правил грамматики*. В табл. 9.6 функция свертки упорядочена так, чтобы при последовательном поиске ее значения вначале проверялось более длинное правило.

Из лемм 9.2 и 9.3, из определения обратимой грамматики слабого предшествования и из конструкции самого алгоритма $\mathscr{A}$ следует, что алгоритм 9.2 строит корректный алгоритм разбора типа "перенос-свертка" для грамматики G.

9.5. Грамматики операторного предшествования

Существует эффективный алгоритм разбора для класса грамматик, называемых грамматиками *операторного предшествования*. Мы увидим, что этот метод прост для реализации и именно поэтому используется во многих компиляторах.

Рассмотрим анализ и вычисление значения арифметического выражения. В этом случае мы сначала выполняем операции (операторы) более высокого приоритета, затем более низкого и т. д. Неформально можно сказать, например, что оператор умножения (*) *предшествует* оператору сложения (+). В грамматиках простого и слабого предшествования отношения определялись между *всеми* символами — операциями и операндами. Анализ процесса вычисления арифметических выражений показывает, что порядок вычисления значения выражения *не зависит от операндов* и определяется только операторами. *Операторное предшествование формализует понятие предшествования между операторами, причем операторами могут быть только терминальные символы грамматики.*

Дадим формальное определение грамматик операторного предшествования и опишем алгоритм построения анализатора типа "перенос-свертка" для таких грамматик.

Операторной грамматикой называется приведенная КС-грамматика $G = (N, \Sigma, S, P)$ без ε-правил, правые части правил которой не содержат смежных нетерминалов.

Пусть $G = (N, \Sigma, S, P)$ — операторная грамматика. Отношения операторного предшествования задаются на множестве $(\Sigma \cup \{\perp\}) \times (\Sigma \cup \{\varepsilon\})$ следующим образом:

- $a \doteq b$, если $A \to \alpha aCb\beta \in P$ и $C \in N \cup \{\varepsilon\}$;
- $a <\cdot b$, если $A \to \alpha aB\beta \in P$ и $B \Rightarrow^{+} Cb\delta$, где $C \in N \cup \{\varepsilon\}$;
- $\perp <\cdot a$, если $S \Rightarrow^{+} Ca\alpha$ и $C \in N \cup \{\varepsilon\}$;
- $a \cdot> b$, если $A \to \alpha Bb\beta \in P$ и $B \Rightarrow^{+} \delta aC$, где $C \in N \cup \{\varepsilon\}$;
- $a \cdot> \varepsilon$, если $S \Rightarrow^{+} \alpha aC$ и $C \in N \cup \{\varepsilon\}$.

Отношения операторного предшествования можно задавать с помощью *матрицы операторного предшествования*. Строки и столбцы этой матрицы отмечаются символами из $\Sigma \cup \{\perp\}$. Если элемент матрицы, расположенный в строке, отмеченной терминалом a, и в столбце, отмеченном терминалом b, имеет значение пусто, то символ b ни в одной правильной входной цепочке не может следовать непосредственно за символом a.

Операторная грамматика $G = (N, \Sigma, S, P)$ называется *грамматикой операторного предшествования*, если между любыми двумя символами из множества $(\Sigma \cup \{\perp\}) \times (\Sigma \cup \{\varepsilon\})$ выполняется не более одного отношения операторного предшествования.

Пример 9.5

Используя определение, построим матрицу операторного предшествования для грамматики G_0:

(1) $E \to E + T$ (4) $T \to P$

(2) $E \to T$ (5) $P \to i$

(3) $T \to T * P$ (6) $P \to (E)$

Шаг 1. Отношение ($\doteq$) определяется непосредственно по правилам вывода грамматики. Для рассматриваемой грамматики из правила (6) следует, что символы '(' и ')' находятся в отношении ($\doteq$).

Отношения ($<\cdot$) и ($\cdot>$) строятся так же, как и для грамматики простого предшествования.

Шаг 2. Определим, например, отношение между символами '+' и '('.

Предположим, что эти символы находятся в отношении ($<\cdot$). Тогда, по определению, должно существовать правило $A \to \alpha + B\beta$ и вывод $B \Rightarrow^+ (\gamma$ или $B \Rightarrow^+ D(\gamma$, где $D \in N$. Из правила (1) грамматики и вывода $T \Rightarrow P \Rightarrow (E)$ следует, что $+ <\cdot$ (.

Шаг 3. Предположим, что $+ \cdot>$ (.Тогда должно существовать правило $A \to \alpha B(\beta$ и вывод $B \Rightarrow^+ \delta+$ или $B \Rightarrow^+ \delta + D$, где $D \in N$. Так как такого правила в G_0 нет, то второе предположение ошибочно, и, следовательно, $+ <\cdot$ (.

Рассуждая подобным образом для каждой пары символов, получим матрицу операторного предшествования, которая приведена в табл. 9.7.

Таблица 9.7. *Матрица операторного предшествования для грамматики G_0*

	(	*i*	*	+	)	ε
(	<·	<·	<·	<·	≐	
i			·>	·>	·>	·>
*	<·	<·	·>	·>	·>	·>
+	<·	<·	<·	·>	·>	·>
)			·>	·>	·>	·>
⊥	<·	<·	<·	<·		

Так как между терминальными символами грамматики существует не более одного отношения операторного предшествования, то G_0 — грамматика операторного предшествования.

Для грамматик операторного предшествования можно сформулировать теорему, аналогичную теореме 9.1.

Теорема 9.2

Пусть $G = (N, \Sigma, S, P)$ — грамматика операторного предшествования, в которой существует вывод $\perp S \Rightarrow^*_r \alpha A w \Rightarrow_r \alpha\beta w$. Тогда:

1. Между всеми соседними терминалами цепочки α (а также символом '$\perp$' и ε) выполняется отношение операторного предшествования ($<\cdot$) или ($\doteq$), и никакие другие отношения не выполняются.
2. Между самым правым терминалом цепочки α и самым левым терминалом цепочки β выполняется отношение операторного предшествования ($<\cdot$), и никакие другие отношения не выполняются.
3. Между всеми соседними терминалами цепочки β выполняется отношение операторного предшествования ($\doteq$), и никакие другие отношения не выполняются.
4. Между самым правым терминалом цепочки β и первым символом цепочки w (или ε) выполняется отношение операторного предшествования ($\cdot>$), и никакие другие отношения не выполняются.

Доказательство теоремы вытекает непосредственно из определения грамматики операторного предшествования.

Используя отношения операторного предшествования, не всегда можно локализовать основу. Это удается сделать только в тех случаях, когда *в основе имеются терминальные символы*, т. к. отношения операторного предшествования не определены на множестве нетерминальных символов. Поэтому при разборе анализатор для грамматик операторного предшествования ищет не основу, а так называемую *самую левую первичную фразу*.

Фразой называют часть цепочки, которую можно свернуть к нетерминалу. *Первичная фраза* — это фраза, содержащая, по крайней мере, один терминал и не содержащая никакой другой фразы, кроме самой себя.

Можно показать, что анализатор будет выполнять правильный разбор входной цепочки, если *все нетерминалы* исходной грамматики заменить одним нетерминалом. Более того, известны алгоритмы разбора, которые используют исходную грамматику только для построения матрицы операторного предшествования, а при разборе работают только с символами входной цепочки.

Рассмотрим анализатор для грамматик операторного предшествования, использующий *остовную грамматику*.

1. Пусть $G = (N, \Sigma, S, P)$ — операторная грамматика. *Остовной* грамматикой для грамматики G называется грамматика $G_S = G = (\{S\}, \Sigma, S, P')$, множество правил P' которой строится следующим образом:
2. Если правило $A \to Y_1Y_2 \ldots Y_m \in P$, то в P' включается правило $S \to X_1X_2 \ldots X_m$, где $X_i = Y_i$, если $Y_i \in \Sigma$, или $X_i = S$, если $Y_i \in N$.
3. Множество правил P' не должно содержать правил вида $S \to S$.

Замечание

Очевидно, что $L(G) \subseteq L(G_S)$, т. е. $L(G_S)$ может содержать цепочки, *не принадлежащие* $L(G)$.

Опишем алгоритм построения анализатора типа "перенос-свертка" для грамматик операторного предшествования.

Алгоритм 9.3. *Построение анализатора типа "перенос-свертка" для грамматик операторного предшествования*

Вход: Грамматика операторного предшествования $G = (\Sigma, N, S, P)$.

Выход: Алгоритм $\mathcal{A} = (f, g)$ типа "перенос-свертка" для грамматики G_S.

Описание алгоритма:

□ Пусть $C \in N \cup \{\varepsilon\}$.

Функция переноса f определяется по отношениям операторного предшествования следующим образом:

1. $f(a\beta, b) = ПЕРЕНОС$, если $a <\cdot b$ или $a \doteq b$;
2. $f(a\beta, b) = СВЕРТКА$, если $a \cdot> b$;
3. $f(\perp S, \varepsilon) = ДОПУСК$;
4. $f(\alpha, w) = ОШИБКА$ в остальных случаях.

Правила вычисления функции свертки следующие:

1. $g(a\beta b\gamma, w) = i$, если $a <\cdot b$, отношение ($\doteq$) выполняется для всех соседних терминалов цепочки γ (если они существуют) и $S \to \beta b\gamma$ — правило с номером i грамматики G_S.
2. $g(\alpha, w) = ОШИБКА$ в остальных случаях.

■

Пример 9.6

Построим алгоритм типа "перенос-свертка" для грамматики G_0, матрица операторного предшествования которой приведена в табл. 9.7.

Остовная грамматика G_{0S} для грамматики G_0 после преобразований (вычеркивание правил вида $E \to E$) содержит следующие правила (нумерация правил сохранена):

(1) $E \to E + E$ (5) $E \to i$

(3) $E \to E * E$ (6) $E \to (E)$

Функция переноса $f(aC, b)$, где $C \in N \cup \{\varepsilon\}$, строится по матрице операторного предшествования и имеет вид, приведенный в табл. 9.8.

Таблица 9.8. *Функция переноса для грамматики* G_0

	(	*i*	*	+	)	ε
(*C*	П	П	П	П	П	
i C			С	С	С	С
* *C*	П	П	С	С	С	С
+ *C*	П	П	П	С	С	С
) *C*			С	С	С	С
⊥	П	П	П	П		
⊥*E*	П	П	П	П		Д

Рассмотрим построение функции свертки $g(\alpha)$.

По определению, функция свертки принимает значение *j* для цепочки, состоящей из двух частей: правой части правила остовной грамматики G_{0S} и терминала, который может находиться непосредственно под этой правой частью в магазине.

Рассмотрим правило (5) $E \to i$ остовной грамматики G_{0S}. В магазине ниже правой части этого правила могут быть только такие символы *b*, для которых выполняется $b <\cdot i$. Это множество символов: { **(** , * , + , ⊥}. Поэтому $g((Ci) = 5$, $g(*Ci) = 5$, $g(+Ci) = 5$ и $g(\perp Ci) = 5$, где $C \in N \cup \{\varepsilon\}$.

Рассуждая аналогично для остальных правил грамматики G_{0S}, получим функцию свертки $g(\alpha)$ (табл. 9.9).

Таблица 9.9. *Функция свертки для грамматики* G_0

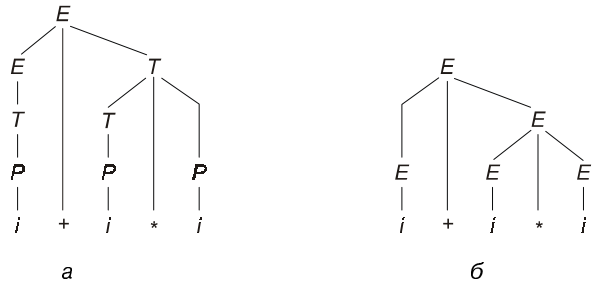
α	$g(\alpha)$	α	$g(\alpha)$
(*E* + *E*	1	+ (*E*)	5
⊥*E* + *E*	1	⊥(*E*)	5
(*E* * *E*	3	(*i*	6
+ *E* * *E*	3	* *i*	6
⊥*E* * *E*	3	+ *i*	6
((*E*)	5	⊥ *i*	6
* (*E*)	5		

Используя построенный алгоритм, рассмотрим анализ цепочки $i + i * i$.

Последовательность тактов, которую проделает алгоритм при разборе этой цепочки, имеет следующий вид:

$$
\begin{aligned}
(\perp, i + i * i, \varepsilon) &\vdash^s (\perp i, + i * i, \varepsilon)\\
&\vdash^r (\perp E, + i * i, 6)\\
&\vdash^s (\perp E +, i * i, 6)\\
&\vdash^s (\perp E + i, * i, 6)\\
&\vdash^r (\perp E + E, * i, 66)\\
&\vdash^s (\perp E + E *, i, 66)\\
&\vdash^s (\perp E + E * i, \varepsilon, 66)\\
&\vdash^r (\perp E + E * E, \varepsilon, 666)\\
&\vdash^r (\perp E + E, \varepsilon, 6663)\\
&\vdash^r (\perp E, \varepsilon, 66631)\\
&\vdash^s \textit{ДОПУСК}.
\end{aligned}
$$

Для сравнения на рис. 9.6, *а* и 9.6, *б* приведены деревья вывода входной цепочки $i + i * i$ в грамматике G_0 и G_{0S} соответственно. □



а *б*

Рис. 9.6. Синтаксические деревья цепочки $i + i * i$ в грамматиках G_0 и G_{0S}

На практике для построения перевода обычно нет необходимости дополнять синтаксическое дерево, построенное по остовной грамматике, до соответствующего дерева исходной грамматики.

Замечание

- Остовная грамматика G_{0S} *неоднозначна*, но отношения операторного предшествования *гарантируют однозначность разбора*.
- Если α — правовыводимая цепочка операторной грамматики, то она не содержит смежных нетерминалов.
- Если α — правовыводимая цепочка операторной грамматики, то символ, расположенный непосредственно слева от основы, не может быть нетерминалом.

9.6. Язык Флойда-Эванса

При выполнении детерминированного нисходящего или восходящего разбора соответствующий анализатор выполняет определенный этим алгоритмом набор стандартных операций над магазином, над входной цепочкой и выходной лентой. Анализ показывает, что этот набор невелик:

- сравнение верхнего символа (или цепочки, стоящей наверху) магазина с заданной цепочкой;
- замена верхнего символа (или цепочки, стоящей наверху) магазина некоторой определенной цепочкой (или символом);
- сравнение текущего входного символа с заданным символом;
- сдвиг входной головки на следующий символ (на один символ вправо);
- запись определенного символа (или цепочки) в конец входной ленты.

Это означает, что возможна разработка языка "программирования" детерминированных синтаксических анализаторов и средств интерпретации (или компиляции) программ на этом языке.

Язык, на котором можно описывать детерминированные алгоритмы синтаксического анализа, был предложен достаточно давно [10] и применялся при реализации нескольких компиляторов и систем построения трансляторов. Ранее этот язык называли языком продукций (продукционный язык), но это не совсем верно, т. к. операторы языка не имеют жесткой связи с правилами грамматики анализируемого языка. В соответствии с [3, 4] будем называть *язык программирования синтаксических анализаторов* языком Флойда-Эванса.

Язык Флойда-Эванса представляет собой *синтаксический метаязык*. Программа (синтаксический анализатор), написанная на этом языке, определяет алгоритм разбора, принимающий решения под воздействием управляющего устройства с конечной памятью.

Анализатор, записанный на языке Флойда-Эванса, представляет собой последовательность операторов определенного вида. Каждый оператор имеет уникальную метку, которую можно рассматривать как состояние управляющего устройства. Операторы работают над входной цепочкой и магазином и приводят к построению разбора входной цепочки.

Для определенности рассмотрим использование языка Флойда-Эванса для построения детерминированных восходящих анализаторов. Для таких анализаторов текущее состояние процесса анализа можно представить в виде конфигурации (q, $\perp X_m \ldots X_1$, $a_1 \ldots a_n$, π), где:

- q — метка текущего оператора;
- $X_m \ldots X_1$ — верхушка магазина, причем X_1 — верхний символ ($\perp$ — маркер дна магазина);

- $a_1 \ldots a_n$ — необработанная часть входной цепочки;
- π — готовая к данному моменту часть выхода анализатора (полный выход — правый разбор входной цепочки для некоторой КС-грамматики).

Программа на языке Флойда-Эванса автоматически использует магазин, входную и выходную ленты, т. е. является программой, управляющей работой синтаксического анализатора, аналогичного тому, что приведен на рис. 8.3.

Существует достаточно много модификаций языка Флойда-Эванса, обладающих различными возможностями [3, 10]. Рассмотрим наиболее простой для чтения и понимания вид оператора этого языка:

1	2	3	4	5	6	7	8
метка:	α	*a*	→	β	*действие*	*	*следующая метка*

Каждый оператор состоит из восьми полей (любое число полей может быть пустым):

- поле 1 — *метка* оператора;
- поле 2 — *образец для сопоставления* с магазином α;
- поле 3 — *образец для сопоставления* с входным символом *a*;
- поле 4 — метасинтаксический символ '→' — заменить;
- поле 5 — способ преобразования магазина (цепочка β, заменяющая верх магазина);
- поле 6 — выполнение дополнительных *действий*;
- поле 7 — действия с входной головкой;
- поле 8 — метка следующего выполняемого оператора программы.

Неформально семантику оператора можно описать следующим образом:

1. выполнение текущего оператора программы с меткой начинается с сопоставления цепочки-образца α (из поля 2) с верхушкой магазина и символа-образца *a* (из поля 3) с текущим символом входной цепочки;
2. если соответствие не *полное*, то текущим становится следующий (по порядку) оператор программы;
3. если сравнение произошло успешно, то цепочка α заменяется цепочкой β из поля 5 текущего оператора;
4. выполняются действия, перечисленные в поле 6 оператора;
5. если в поле 7 находится символ '*', то читающая головка сдвигается вправо на следующий символ входной цепочки;
6. текущим оператором становится оператор с меткой, записанной в поле 8.

Если синтаксический анализатор находится в конфигурации ($L1$, $\gamma\alpha$, ax, π) и оператор с меткой $L1$ имеет вид:

L1:	α	*a*	→	β	*ВЫДАЧА s*	*	*L2*

то анализатор переходит в конфигурацию ($L2$, $\gamma\beta$, x, πs), выполняя шаг анализа:

$$(L1, \gamma\alpha, ax, \pi) \vdash (L2, \gamma\beta, x, \pi s),$$

т. е. если при выполнении оператора с меткой $L1$ в верхушке магазина расположена цепочка α и a — текущий входной символ, то надо заменить α на β, выдать цепочку s, сдвинуть входную головку на один символ вправо (на это указывает наличие символа '*') и перейти к следующему оператору $L2$.

Как уже было сказано, любое из полей оператора может быть пустым:

- метка у каждого оператора удобна при обозначении операторов в конфигурациях, но в реальной программе помечаются только операторы, на которые есть переходы;
- если метасимвол '→' опущен, то изменение состояния магазина не предполагается (цепочку β в этом случае указывать бессмысленно);
- если опущено *действие*, то, очевидно, никакие дополнительные действия не выполняются;
- если опущен символ '*', то входная головка не сдвигается;
- если опущена *следующая метка*, то текущим после данного оператора становится следующий по порядку оператор программы.

Замечание

По своей сути, сдвиг входной головки, обозначаемый символом '*', и переход к оператору "*следующая метка*" являются действиями. Они выделены в отдельные поля только потому, что присутствуют практически в каждом операторе программы на языке Флойда-Эванса. Кроме уже указанного действия *ВЫДАЧА*, возможны действия *ДОПУСК*, *ОШИБКА* и другие.

Программа на языке Флойда-Эванса выполняется следующим образом:

1. Вначале анализатор находится в конфигурации (L, $\perp$, w, ε), где w — анализируемая входная цепочка, L — метка первого оператора программы.
2. Затем последовательно выполняются операторы программы, пока среди них не найдется оператор, у которого произошло успешное сопоставление образцов с магазином и входным символом. После выполнения всех действий, предписываемых этим оператором, управление передается оператору с меткой, указанной в его последнем поле.

3. Анализатор продолжает работать до тех пор, пока не выполнится действие ошибка или допуск. Выход анализатора (выходная лента) принимается во внимание только в случае действия допуск.

Будем описывать работу анализатора на языке Флойда-Эванса при помощи конфигураций, имеющих вид (L, $\perp\alpha$, x, π), где:

- L — метка текущего (выполняемого) оператора программы;
- $\perp\alpha$ — содержимое магазина (вершина слева);
- x — входная цепочка;
- π — содержимое выходной ленты, построенное к данному моменту времени.

Отношение перехода ($\vdash$) определяется аналогично тому, как оно было определено для алгоритма типа "перенос-свертка".

В программах на языке Флойда-Эванса удобно использовать метасинтаксический символ '#', который используется для обозначения *любого* символа грамматики. Если символ '#' встречается слева и справа от метасимвола '→', то он обозначает *один и тот же* символ.

Программа на языке Флойда-Эванса, реализующая анализатор для грамматики G_0 с правилами:

(1) $E \to E + T$ (4) $T \to P$

(2) $E \to T$ (5) $P \to i$

(3) $T \to T * P$ (6) $P \to (E)$

приведена в табл. 9.10.

Таблица 9.10. *Анализатор для грамматики G_0*

L0:		#	→	#				Читаем первый символ входной цепочки в магазин
L1:	(	#	→	(#		*	*L1*	
L2:	*i*	#	→	*P*#	*ВЫДАЧА* 5	*	*L4*	Наверху магазина символ *i*: свертка по правилу $P \to i$ и выдача на выход номера этого правила — 5. Переход к анализу *P* (множителя)
L3:	#				*ОШИБКА*			Ошибка — останов программы
L4:	*T*P*#		→	*T*#	*ВЫДАЧА* 3		*L6*	Свертка (3) $T \to T * P$

Таблица 9.10 *(окончание)*

L5:	*P#*		→	*T#*	*ВЫДАЧА* 4			Свертка (4) *T* → *P*
L6:	*T**	#	→	*T* #*		*	*L1*	Переход к выбору *P* (очередного множителя)
L7:	*E + T#*		→	*E#*	*ВЫДАЧА* 1		*L9*	Свертка (1) *E* → *E + T*
L8:	*T#*		→	*E#*	*ВЫДАЧА* 2			Свертка (2) *E* → *T*
L9:	*E+*	#	→	*E+ #*		*	*L1*	Переход к выбору *T* (очередного слагаемого)
L10:	(*E*)	#	→	*P*	*ВЫДАЧА* 6	*	*L4*	Свертка (6) *P* → (*E*)
L11:	⊥ *E* ⊥				*ДОПУСК*			Допуск — останов
L12:	#				*ОШИБКА*			Ошибка — останов

Пример 9.7

Работу анализатора проиллюстрируем разбором входной цепочки $i + i * i$. Цепочка принадлежит языку, определяемому исходной грамматикой, при ее анализе программа выполнит следующую последовательность конфигураций:

$$
\begin{aligned}
(L0, \perp, i + i * i, \varepsilon) &\vdash (L1, \perp i, + i * i, \varepsilon)\\
&\vdash (L2, \perp i, + i * i, \varepsilon)\\
&\vdash (L4, \perp P +, i * i, 5)\\
&\vdash (L5, \perp P +, i * i, 5)\\
&\vdash (L6, \perp T +, i * i, 54)\\
&\vdash (L7, \perp T +, i * i, 54)\\
&\vdash (L8, \perp T +, i * i, 54)\\
&\vdash (L9, \perp E +, i * i, 542)\\
&\vdash (L1, \perp E + i, * i, 542)\\
&\vdash (L2, \perp E + i, * i, 542)\\
&\vdash (L4, \perp E + P *, i, 5425)\\
&\vdash (L5, \perp E + P *, i, 5425)\\
&\vdash (L6, \perp E + T *, i, 54254)\\
&\vdash (L1, \perp E + T * i, \varepsilon, 54254)\\
&\vdash (L2, \perp E + T * i, \varepsilon, 54254)
\end{aligned}
$$

$\vdash (L4, \perp E + T * P, \varepsilon, 542545)$

$\vdash (L6, \perp E + T, \varepsilon, 5425453)$

$\vdash (L7, \perp E + T, \varepsilon, 5425453)$

$\vdash (L9, \perp E, \varepsilon, 54254531)$

$\vdash (L10, \perp E, \varepsilon, 54254531)$

$\vdash (L11, \perp E, \varepsilon, 54254531)$

$\vdash$ *ДОПУСК.*

На выходной ленте анализатора получен правый разбор входной цепочки 54254531. □

Можно заметить, что программа синтаксического анализатора, написанная на языке Флойда-Эванса, не так тесно связана с грамматикой, для которой она производит разбор, как уже рассмотренные в этой главе алгоритмы синтаксического анализа со своими грамматиками. Передача управления операторами может привести к тому, что не все свертки будут реализованы. Поэтому распознаваемый анализатором Флойда-Эванса язык может не совпадать с языком, определяемым грамматикой.

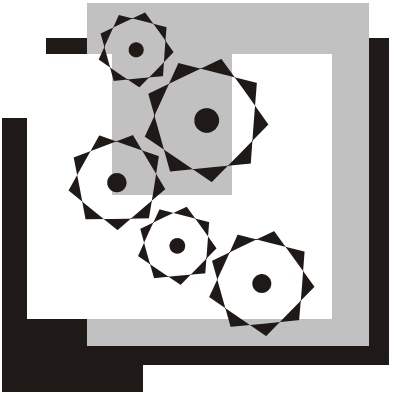
Язык Флойда-Эванса можно модифицировать так, чтобы под действиями понималось выполнение некоторых семантических функций. В этом случае анализатор будет выполнять синтаксически управляемую трансляцию, вычисляющую перевод нетерминала A в терминах переводов его составных частей. Кроме этого существуют модификации языка [10], позволяющие определять дополнительные структуры данных различной организации, описывать и использовать в программах подпрограммы и т. д.

Контрольные вопросы

1. Дайте определение алгоритма типа "перенос-свертка". Каким образом описывается его работа?
2. Дайте определение грамматик простого и слабого предшествования. Чем различаются эти грамматики?
3. Как определяются отношения предшествования? Какие средства существуют для их определения?
4. Как представляются отношения предшествования? Как они используются при выполнении синтаксического анализа?
5. Для чего может быть использован язык Флойда-Эванса?
6. Какие модификации языка Флойда-Эванса могут расширить его возможности?

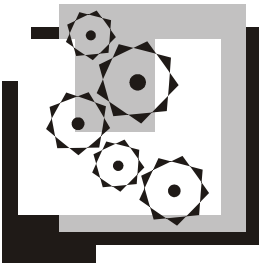
Упражнения

1. Постройте управляющую таблицу и промоделируйте работу анализатора типа "перенос-свертка" для КС-грамматики слабого предшествования $G = (\Sigma, N, P, S)$, правила которой имеют вид:
   1.1. $S \to N;Q$, $N \to n$, $Q \to N;n$, $Q \to q$, $Q \to Q;q$.
   1.2. $S \to SA$, $S \to A$, $A \to (S)$, $A \to ()$.
   1.3. $S \to SA$, $S \to A$, $A \to 0S$, $A \to 01S$.
   1.4. $E \to E\ or\ T$, $E \to T$, $T \to T\ and\ P$, $T \to P$, $P \to i$, $P \to (E)$, $P \to not\ P$.
   1.5. $S \to B$, $B \to S$, $B \to B1;E$, $B1 \to begin\ d$, $B1 \to B1;d$, $E \to s\ end$, $E \to s;E$, $S \to begin\ E$.
2. Постройте управляющую таблицу и промоделируйте работу анализатора типа "перенос-свертка" для КС-грамматики операторного предшествования $G = (\Sigma, N, P, S)$ с правилами:
   2.1. $S \to if\ E\ then\ S\ else\ S$, $S \to a$, $E \to E\ or\ b$, $E \to b$.
   2.2. $S \to aAc$, $S \to b$, $A \to aSc$, $A \to b$.
   2.3. $S \to Ac$, $S \to Bd$, $A \to a$, $B \to a$.
   2.4. $E \to E\ or\ T$, $E \to T$, $T \to T\ and\ P$, $T \to P$, $P \to i$, $P \to true$, $P \to false$, $P \to (E)$, $P \to not\ P$.
3. Постройте управляющую таблицу и промоделируйте работу анализатора типа "перенос-свертка" для КС-грамматики простого предшествования $G = (\Sigma, N, P, S)$ с правилами $S \to L = R$, $S \to R$, $L \to * R$, $L \to i$, $R \to (L)$.
4. Постройте анализатор на языке Флойда-Эванса для грамматик из упр.1.



Часть IV

Формальные методы описания и реализации синтаксически управляемого перевода



Глава 10

Промежуточные формы представления программ

Математически перевод определяется как множество пар, причем первый элемент принадлежит множеству объектов, которые требуется перевести, а второй элемент — множеству объектов, являющихся результатом перевода.

Компилятор в целом определяет перевод исходной программы, представленной в виде цепочки символов на языке программирования, в объектный код. Если компиляцию рассматривать как многоэтапный процесс, то можно говорить о переводе, присущем каждому этапу компиляции. Например, в процессе лексического анализа исходная программа преобразуется из цепочки символов в цепочку лексических единиц (лексем), а на этапе синтаксического анализа цепочка лексем преобразуется в некоторую *промежуточную форму*, являющуюся линейной формой записи синтаксического дерева программы.

Такой подход имеет определенные преимущества:

- перевод в промежуточную форму позволяет не учитывать особенности целевого языка, которые значительно усложняют перевод;
- упрощается перенос на другие целевые машины, для чего потребуется только реализовать преобразование промежуточной формы в язык новой машины;
- к программе в промежуточной форме можно применять алгоритмы машинно-независимой оптимизации.

Различные формы представления промежуточной программы отличаются друг от друга, в основном, способом соединения операторов и операндов. Общепринятыми промежуточными формами представления программы являются [2, 11, 28—30, 34, 45]:

- польская инверсная запись;
- тетрады;

- триады;
- связные списочные структуры.

В настоящее время ряд компиляторов в качестве промежуточной формы программы генерирует файлы, содержащие байт-код (Byte-code), представляющий собой инструкции для виртуальной машины Java (JVM — Java Virtual Machine). Поскольку ядро виртуальной машины Java реализовано практически для любого типа компьютеров, то файлы байт-кодов можно рассматривать как независимые от платформы приложения.

10.1. Польская запись

Одной из основных конструкций языков программирования является выражение, которое:

- может быть описано только рекурсивной самовложенной КС-грамматикой, т. к. оно включает в себя парные символы (скобки), т. е. по своей природе рекурсивно;
- содержит операции, имеющие различное число операндов и выполняющиеся в определенном порядке (приоритет операций);
- может содержать операнды различного типа, что требует дополнительного анализа при переводе.

Перечисленные особенности, с одной стороны, усложняют анализ и перевод выражений, требуют тщательного проектирования описания перевода, а с другой стороны, позволяют использовать выражения в качестве модели при рассмотрении проблем перевода.

10.1.1. Вычисление выражений

Порядок вычисления значения выражения определяется приоритетом операций и наличием в выражении скобок. Для простоты рассмотрим вычисление полноскобочного выражения, т. е. выражения, в котором при помощи дополнительных скобок явно определен порядок выполнения операций. Например, выражение

$$A + B - C + D * E$$

в полноскобочной форме может быть записано как

$$(((A + B) - C) + (D * E)).$$

При выполнении вычислений необходимо найти самую внутреннюю пару скобок, выполнить операцию, удалить использованную пару скобок, заменив ее вычисленным значением, и повторять этот процесс, пока есть скобки.

Замечание

Если каждой левой скобке придать вес +1, а каждой правой — вес –1, то процесс нахождения выполняемой на данном шаге операции упростится: нужно будет просмотреть выражение и найти максимум веса, который соответствует самой внутренней левой скобке.

Существуют более простые рекурсивные алгоритмы вычисления выражений [2, 11, 28].

Польский математик Ян Лукашевич обнаружил, что при использовании функциональной записи выражения, в которой операция предшествует своим операндам (а не записывается между ними), скобки становятся излишними.

Пример 10.1

🗁 Бинарная операция сложения обычно записывается в виде $A + B$. Ее функциональная запись — это $+(A, B)$. При рассмотрении выражений, содержащих только бинарные операции скобки и запятую в записи можно опустить, тогда сложение можно записать как $+AB$. 🗀

10.1.2. Префиксная форма

Префиксная форма (или польская запись) формально может быть определена следующим образом:

Всякая переменная или константа есть выражение.

1. Если θ_1 — знак унарной (одноместной) операции, а α — выражение, то $\theta_1\alpha$ является выражением.
2. Если θ_2 — знак бинарной (двухместной) операции, а α_1 и α_2 — выражения, то $\theta_2\alpha_1\alpha_2$ является выражением.
3. Если θ_n — знак n-местной операции, а α_1, α_2, ... , α_n являются выражениями, то $\theta_n\alpha_1\alpha_2 \ldots \alpha_n$ — выражение.
4. Других выражений не существует.

Пример 10.2

🗁 Выражение языка ALGOL-60 $(A + B) * C\uparrow(D / (E + F))$, где $\uparrow$ — обозначение операции возведения в степень, можно представить в префиксной форме как $*+ AB\uparrow C / D + EF$. 🗀

Можно заметить, что префиксная форма не нарушает порядка следования операндов в выражении:

$A + B$ записывается как $+AB$;

$B + A$ записывается как $+BA$;

т. е. эта форма может быть использована как для записи коммутативных, так и некоммутативных операций. Кроме того, в отличие от обычной алгебраической записи выражений префиксная форма однозначно определяет порядок выполнения операций. Например, выражение $A + B + C + D$ можно интерпретировать по-разному (см. первый столбец табл. 10.1), а префиксная запись выражения всегда однозначно определяет порядок выполнения операций (см. второй столбец табл. 10.1).

Таблица 10.1. *Полноскобочная и префиксная формы выражения*

Полноскобочная форма выражения	Префиксная форма выражения
$(((A + B) + C) + D)$	$+++ABCD$
$((A + B) + (C + D))$	$++AB+CD$
$((A + (B + C)) + D)$	$++A+BCD$
$(A + ((B + C) + D))$	$+A++BCD$
$(A + (B + (C + D)))$	$+A+B+CD$

Замечание

Обычно в языках программирования операция сложения выполняется слева направо.

Префиксная форма довольно редко используется при проектировании трансляторов. Это связано с тем, что преобразование выражения в префиксную запись требует его просмотра справа налево, а в большинстве трансляторов просмотр и интерпретация входной программы выполняется слева направо.

10.1.3. Постфиксная форма

В компиляторах чаще всего используют постфиксную форму, в которой операции записываются справа от операндов. Постфиксная форма (польская инверсная запись — ПОЛИЗ) определяется следующими правилами:

1. Всякая переменная или константа есть выражение.
2. Если θ_1 — знак унарной (одноместной) операции, а α — выражение, то $\alpha\theta_1$ является выражением.

   Если θ_2 — знак бинарной (двухместной) операции, а α_1 и α_2 — выражения, то $\alpha_1\alpha_2\theta_2$ является выражением.

   Если θ_n — знак n-местной операции, а α_1, α_2, ... , α_n являются выражениями, то $\alpha_1\alpha_2 \ldots \alpha_n\theta_n$ — выражение.
3. Других выражений не существует.

Замечание

Поскольку изображение операции "унарный минус" совпадает с изображением операции "бинарный минус", то унарный минус можно представлять двумя способами: либо записывать его как бинарный оператор, т. е. $-B$ представлять в виде $0 - B$, либо для унарного минуса ввести новый символ операции, например символ '!'.

Пример 10.3

🗁 Выражение языка ALGOL-60 $(A + B) * C\uparrow(D / (E + F))$, где '↑' — обозначение операции возведения в степень, можно представить в польской инверсной записи как $AB + CDEF + / \uparrow *$. 🗀

Так же, как и префиксную форму выражения, польскую инверсную запись можно рассматривать как программу работы стековой машины, содержащую команды двух типов:

1. Чтение операнда из выражения вызывает запись его в магазин.
2. Чтение операции влечет за собой выполнение следующих действий:
   - выполнение операции, операнды для которой читаются из верхушки магазина;
   - запись результата операции в магазин.

При вычислении выражения, представленного в постфиксной записи, его элементы читаются слева направо.

Например, выражение 7 5 2 * + соответствует последовательности действий, приведенной в табл. 10.2 (рассматриваемый элемент выражения выделен полужирным шрифтом).

Таблица 10.2. *Порядок вычисления выражения 7 5 2 * +*

Выражение	Тип элемента	Действие	Магазин
7 5 2 * +	операнд	Записать в магазин операнд (число 7)	7⊥
7 **5** 2 * +	операнд	Записать в магазин операнд (число 5)	5 7⊥
7 5 **2** * +	операнд	Записать в магазин операнд (число 2)	2 5 7⊥
7 5 2 ***** +	операция (*)	Прочитать из магазина операнд (число 2)	2 5 7⊥
		Прочитать из магазина операнд (число 5)	5 7⊥
		Выполнить операцию 2 * 5	7⊥
		Записать в магазин результат операции (число 10)	10 7⊥
7 5 2 * **+**	операция (+)	Прочитать из магазина операнд (число 10)	10 7⊥
		Прочитать из магазина операнд (число 7)	7⊥
		Выполнить операцию 10 + 7	⊥
		Записать в магазин результат операции (число 17)	17⊥

10.1.3.1. Свойства ПОЛИЗ

Польская инверсная запись обладает следующими свойствами:

- однозначно определяет порядок выполнения операций;
- имеет простую синтаксическую структуру.

Свяжем с каждым элементом e_i польской инверсной записи выражения вес p в соответствии со следующими правилами:

- если элемент e_i — переменная или константа, то $p(e_i) = 1$;
- если элемент e_i — знак унарной операции, то $p(e_i) = 0$;
- если элемент e_i — знак бинарной операции, то $p(e_i) = -1$;
- если элемент e_i — знак n-местной операции, то $p(e_i) = -(n - 1)$;

и определим функцию $P(k) = \Sigma\, p(i)$. Тогда для каждого выражения справедливы утверждения:

- $P(i) > 0$ для $1 < i < k$;
- $P(n) = 1$ для правильного выражения из n элементов.

10.1.3.2. Графическое представление выражений

Выражение можно изобразить в виде дерева. Пусть узлы дерева представляют собой операции, а ветви — операнды. Для бинарных операций левая ветвь соответствует левому операнду, а правая ветвь — правому операнду, причем в каждой ветви дерева узлы, которые лежат ниже, соответствуют операциям, которые выполняются раньше.

Например, выражению $a * (b + c)$ соответствует дерево, изображенное на рис. 10.1.

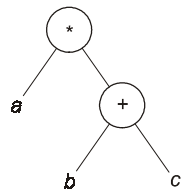


Рис. 10.1. Дерево выражения $a * (b + c)$

Префиксная форма этого выражения равна $*a + bc$, а постфиксная — $abc +*$.

Интересно, что все три формы представления выражения (инфиксная, префиксная и постфиксная) являются, по существу, различными формами представления или обхода построенного бинарного дерева. Можно убедиться, что префиксная польская запись выражения $*+abc$ — это левое скобочное представление дерева, из которого удалены скобки, или *прямой обход*

дерева. Аналогично, постфиксная запись *abc +** — это правое скобочное представление дерева, из которого удалены все скобки, или *обратный обход* дерева.

10.1.4. ПОЛИЗ как промежуточный язык

Польская инверсная запись выражений имеет два важных свойства, которые позволяют использовать ее не только для представления выражений, но и в качестве промежуточного языка для языковых процессоров:

1. Действия, описываемые в ПОЛИЗ, можно выполнять (или программировать) в процессе ее одностороннего безвозвратного просмотра слева направо, т. к. операнды в ПОЛИЗ всегда предшествуют знаку операции.
2. Операнды ПОЛИЗ расположены в том же порядке, что в исходном (инфиксном) выражении. Перевод выражения в ПОЛИЗ сводится только к изменению порядка следования знаков операций.

Расширение ПОЛИЗ для представления других конструкций языков программирования (переменных с индексами, указателей функций, условных выражений, основных операторов языка программирования) выполняется очень просто: нужно придерживаться одного правила — *знак операции должен следовать непосредственно за соответствующими ему операндами.*

Замечание

Термин "вычисление выражения" будем интерпретировать более широко. Под ним будем понимать как вычисление численного значения выражения в трансляторах интерпретирующего типа, так и *процесс анализа выражения и построения для него объектного кода* для компиляторов.

10.1.4.1. Элементы массивов

Пусть требуется вычислить выражение:

$$(a + b[i, j + 1]) * c + d,$$

в котором $b[i, j + 1]$ — элемент двухмерного массива.

Для представления в постфиксной записи элементов массива необходимо определить дополнительную операцию ПОЛИЗ, вычисляющую *адрес элемента массива* (код операции SUBS). Операндами операции SUBS являются имя массива и значения индексов. Число операндов этой операции переменно. Оно зависит от размерности массива и определяется по формуле $k = n + 1$, где n — размерность массива. Операнды операции SUBS должны располагаться в определенном порядке: имя массива, значения индексных выражений и константа k (целое без знака), определяющая число операндов.

С учетом представления элементов массива ПОЛИЗ рассматриваемого выражения имеет вид:

$$a\ b\ i\ j\ 1\ +\ 3\ \text{SUBS}\ +\ c\ *\ d\ +.$$

Графическое представление этого выражения приведено на рис. 10.2.

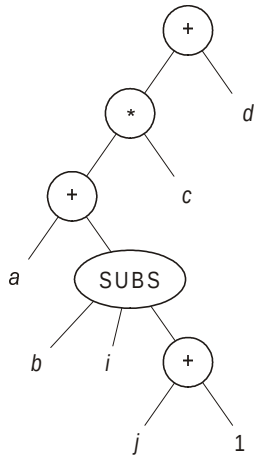


Рис. 10.2. Дерево выражения $(a + b[i, j + 1]) * c + d$

Замечание

При представлении в ПОЛИЗ элементов массивов можно явно не указывать число операндов операции SUBS. В этом случае имя массива должно располагаться непосредственно перед кодом операции, а информация о размерности массива, определяющая число операндов операции, должна храниться в другом месте, например, в таблице идентификаторов языкового процессора.

10.1.4.2. Указатели функций

С точки зрения представления в польской инверсной записи указатель функции (оператор обращения к функции) мало чем отличается от переменной с индексом. Для представления указателя функции введем в ПОЛИЗ дополнительную операцию CALL, имеющую переменное число операндов, зависящее от числа аргументов в вызове функции. Количество операндов операции CALL определяется выражением:

$$k = n + 1,$$

где n — число аргументов функции.

Например, ПОЛИЗ выражения $a + f(i, j + 1)$ имеет вид $a\ f\ i\ j\ 1\ +\ 3\ \text{CALL}+$ (рис. 10.3).

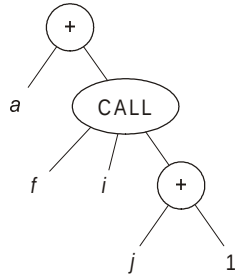


Рис. 10.3. Дерево выражения $a + f(i, j + 1)$

10.1.4.3. Условные выражения

В рассмотренных ранее выражениях порядок выполнения операций определялся приоритетом операций и скобками. В некоторых языках программирования имеются *условные выражения*, в которых порядок выполнения операций и значения операндов определяются *динамически* в зависимости от некоторых условий. Например, в языке C++ выражение `a > b?1:2` принимает значение `1`, если `a > b`, и значение `2` в противном случае. Условное арифметическое выражение `x + (if a > b then 1 else 2)` языка ALGOL-60 принимает значение `x + 1`, если `a > b`, и значение `x + 2` в противном случае.

В приведенных примерах значения операндов при вычислении выражения изменяются динамически в зависимости от значения условия `a > b`.

Для обеспечения возможности представления условных выражений добавим в польскую инверсную запись следующие объекты:

- метки, которые могут помечать некоторые элементы ПОЛИЗ и использоваться для динамического изменения хода вычислений;
- дополнительные операции:
  - BF — бинарная операция "Условный переход по значению ложь". Выражение *a m* BF означает, что при *a* равном значению "ложь" необходимо перейти к рассмотрению элемента ПОЛИЗ, помеченного меткой *m*, в противном случае — перейти к рассмотрению очередного элемента ПОЛИЗ;
  - BRL — унарная операция "Безусловный переход", операндом которой является метка элемента, который будет рассматриваться следующим;
  - DEFL — унарная операция "Определить метку", операндом которой является метка, которую необходимо поместить перед некоторым элементом ПОЛИЗ.

Используя введенные допущения, условное выражение $x + (\textbf{if}\ a > b\ \textbf{then}\ 1\ \textbf{else}\ 2)$ можно представить в виде:

$$x\ a\ b > m_1\ \text{BF}\ 1\ m_2\ \text{BRL}\ m_1\ \text{DEFL}\ 2\ m_2\ \text{DEFL}\ +$$

На рис. 10.4 приведен порядок использования элементов ПОЛИЗ при вычислении рассмотренного условного выражения: верхняя стрелка соответствует случаю $a > b$, а нижние стрелки — $a \le b$.

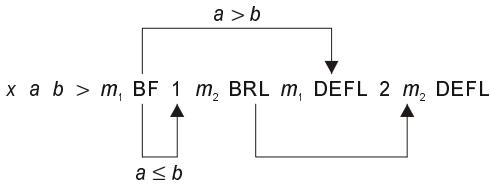


Рис. 10.4. ПОЛИЗ выражения $x + (\textbf{if}\ a > b\ \textbf{then}\ 1\ \textbf{else}\ 2)$

Для представления условного выражения с помощью дерева требуется ввести пустые узлы, метки и дополнительные операции (BF, BRL, DEFL). Пустые узлы соответствуют разветвлению вычислительного процесса и вводятся для того, чтобы число ветвей, исходящих из узла, было равно числу операндов операции. На рис. 10.5 приведено дерево условного выражения $x + (\textbf{if}\ a > b\ \textbf{then}\ 1\ \textbf{else}\ 2)$, при обратном обходе которого получается приведенное ранее представление выражения в ПОЛИЗ.

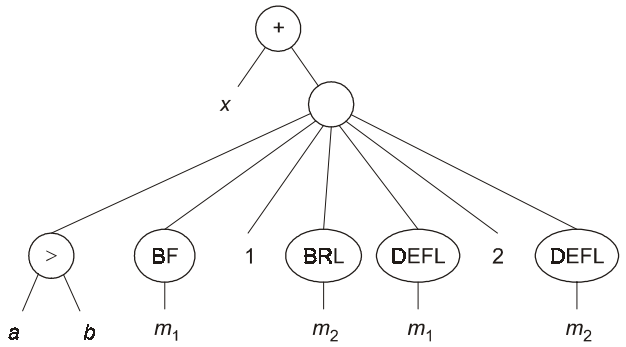


Рис. 10.5. Дерево условного выражения $x + (\textbf{if}\ a > b\ \textbf{then}\ 1\ \textbf{else}\ 2)$

Замечание

Включение дополнительных объектов, отражающих динамические свойства условного выражения, усложняет построение дерева и его интерпретацию.

10.1.4.4. Оператор присваивания

Для представления оператора присваивания в польской инверсной записи требуется ввести бинарную операцию "Присвоить значение" (:=). Левый

операнд операции определяет объект, которому нужно присвоить значение, а правый — вычисленное значение. Например, оператор присваивания языка Паскаль `a[j, k + 1] := sqr(m[j])/ 2` может быть записан в ПОЛИЗ в виде:

a j k 1 + 3 SUBS sqr *m j* 2 SUBS 2 CALL 2 / := .

10.1.4.5. Условный оператор

Используя результаты, полученные при рассмотрении условного выражения, условный оператор языка C++, имеющий формат:

if (<выражение>) <оператор1> **else** <оператор2>,

может быть представлен в польской инверсной записи следующим образом:

<выражение> m_1 BF <оператор1> m_2 BRL m_1 DEFL <оператор2> m_2 DEFL

В табл. 10.3 приведено представление основных операций, включенных в большинство языков программирования, в польской инверсной записи. Используя эти операции, можно разработать представление других, более сложных операторов языков программирования. При необходимости перечисленные операции могут быть модифицированы, а их список расширен.

Таблица 10.3. *Представление основных операций в ПОЛИЗ*

Оператор	Код операции	Операнды	Семантика оператора
Вычисление адреса элемента массива	SUBS	a, i_1, ... , i_n, k	Вычисление адреса элемента массива a; i_1, ... , i_n — индексные выражения; $k = n + 1$ — число операндов операции
Вызов функции	CALL	f, x_1, ... , x_n, k	Вызов функции f с аргументами x_1, ... , x_n; $k = n + 1$ — число операндов операции
Преобразование типа целый → вещественный вещественный → целый	 I2A A2I	 E E	Преобразование типа значения выражения E
Оператор присваивания	:=	a, b	Оператор присваивания $b := a$
Определение метки	DEFL	m	Определение метки m
Безусловный переход на метку	BRL	m	Переход на метку m (m — идентификатор метки)

Таблица 10.3 *(окончание)*

Оператор	Код операции	Операнды	Семантика оператора
Безусловный переход	BR	*n*	Безусловный переход на *n* (*n* — номер элемента ПОЛИЗ)
Условный переход по значению "ложь"	BF	*r*, *m*	Переход на метку *m*, если *r* = **false**
Переход по условию: равно нулю больше нуля меньше нуля не меньше нуля не больше нуля	 BZ BP BM BPZ BMZ	*m*, *r*	Переход на *m* (*m* — метка или номер элемента ПОЛИЗ) при выполнении соответствующего условия для выражения *r*
Начало блока	BLBEG	—	Отмечает начало блока
Конец блока	BLEND	—	Отмечает конец блока

Рассмотрим далее разработку представления в ПОЛИЗ конструкций языков программирования, отсутствующих в табл. 10.3, на примере оператора цикла.

10.1.4.6. Оператор цикла

Разработку представления оператора цикла в ПОЛИЗ можно выполнить за два шага. Рассмотрим, например, оператор цикла языка Pascal, имеющего формат:

```
for i := <Выражение_1> to <Выражение_2> do <Оператор>
```

Сначала опишем порядок выполнения оператора **for**, используя условный оператор и оператор перехода:

```
    i := <Выражение_1>;
m1: if (i ≤ <Выражение_2>)
    then
       <Операторы>
       i := i + 1;
       goto m1
    endif;
m4:
```

Теперь, зная представление в ПОЛИЗ условного оператора и оператора перехода, можно получить ПОЛИЗ оператора **for**:

i <Выражение_1> := m_1 DEFL *i* <Выражение_2> − m_4 BP <Оператор> *i* 1 + m_1 BRL m_4 DEFL

Недостатки этого представления оператора цикла становятся очевидными при разработке описания перевода. Это связано с тем, что часть операторов ПОЛИЗ, формируемая при просмотре заголовка цикла, должна следовать в ней *после* тела цикла, т. е. порядок следования операндов в ПОЛИЗе и в исходной программе не совпадает.

Опишем выполнение оператора цикла другим способом:

```
    i := <Выражение1>;
m1: if i ≤ <Выражение2>
    then goto m2
    else goto m4
    endif
m3: i := i + 1;
    goto m1;
m2: <Операторы>;
    goto m3;
m4:
```

В этом случае польская инверсная запись имеет вид:

i <Выражение_1> := m_1 DEFL *i* <Выражение_2> ≤ m_4 BF m_2 BRLm_3 DEFL *i i* 1 + := m_1 BRL m_2 DEFL <Оператор> m_3 BRL m_4 DEFL

При таком представлении следование операндов в исходном тексте и в ПОЛИЗ совпадает, что упрощает реализацию перевода.

Возможны и другие способы представления. Например, в [26] за счет введения другого набора операций, польская инверсная запись операторов цикла значительно упрощена.

Дополним ПОЛИЗ следующими операциями:

- m_1 m_2 *a* BRET — безусловный переход с возвратом. Эта операция имеет три операнда:
  - m_1 — метка, на которую должен быть выполнен безусловный переход;
  - m_2 — метка, на которую должен быть выполнен возврат по операции RET;
  - *a* — вспомогательная переменная, куда должна быть записана метка возврата;
- *a* RET — операция возврата, которая выполняет операцию перехода на метку, записанную в переменной *a*.

Пример 10.4

🗁 Рассмотрим выполнение польской инверсной записи:

m_1 DEFL <Операторы_1> m_2 m_1 a BRET <Операторы_2> m_2 DEFL <Операторы_3> a RET

при условии, что представление элементов <Операторы_1>, <Операторы_2>, <Операторы_3> в ПОЛИЗ известно:

- ❒ выполняются <Операторы_1>;
- ❒ выполняется оператор BRET: значение метки m_1 записывается во вспомогательную переменную a, выполняется переход к элементу ПОЛИЗ, помеченному меткой m_2;
- ❒ выполняются <Операторы_3>;
- ❒ операция возврата RET выполняет переход к элементу ПОЛИЗ, метка которого записана в переменной a, т, е. к элементу, помеченному меткой m_1 (рис. 10.6), и процесс повторяется.

Заметим, что <Операторы_2> никогда не выполняются. 🗀

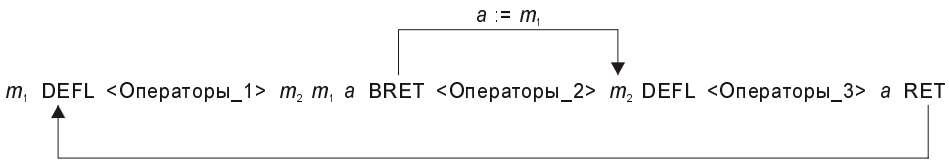


Рис. 10.6. Порядок выполнения ПОЛИЗ из примера 10.4

Используя введенные операции, рассматриваемый оператор цикла

for i := <Выражения_1> **to** <Выражения_2> **do** <Оператор>

можно представить в ПОЛИЗ следующим образом:

i <Выражения_1> := m_1 i <Выражения_2> ≤ m_4 BF m_2 m_1 a BRET

m_2 DEFL <Оператор> a RET m_4 DEFL.

Порядок выполнения ПОЛИЗ для этого оператора цикла приведен на рис. 10.7.

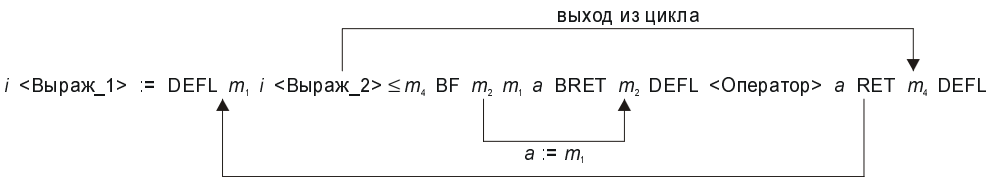


Рис. 10.7. Порядок выполнения ПОЛИЗ для оператора цикла for

Возможны и другие способы представления оператора цикла.

Замечание

Разработка представления конструкций языка программирования в ПОЛИЗ — это довольно сложная работа, при выполнении которой необходимо учитывать особенности языка программирования, выбранных методов анализа и целевой машины.

10.2. Тетрады

Для бинарных операций удобной формой представления программы после синтаксического анализа являются тетрады. Формат тетрад:

<код операции>, <операнд_1>, <операнд_2>, <результат>,

где <операнд_1> и <операнд_2> специфицируют аргументы, а <результат> — временное имя для хранения результата выполнения операции (переменная из рабочей области). В качестве операндов тетрад наряду с переменными и константами, определенными в исходной программе, могут выступать результаты ранее выполненных тетрад. Например, выражение $a * b + c * d$ представляется в виде последовательности следующих тетрад:

* , a, b, t_1

* , c, d, t_2

\+ , t_1, t_2, t_3.

Замечание

В тетрадах запрещены встроенные в операнды выражения.

Последовательность тетрад представляет собой программу, инструкции которой обрабатываются последовательно. Операнды одной тетрады должны быть одинакового типа. Для преобразования типа операнда можно использовать тетрады преобразования типа с кодами операций I2A (целый — в вещественный) и A2I (вещественный — в целый). Поскольку операция преобразования типа одноместная, она записывается с пустым вторым операндом, например,

A2I, a, , t

Для представления унарного минуса в тетрадах также можно не использовать второй операнд. Тетрада

\− , a, , t

интерпретируется как присвоение временной переменной t значения $-a$.

Множество очевидных (для представления выражений) операций может быть при необходимости расширено.

10.2.1. Переменные с индексами

Для представления переменных с индексами можно использовать тетраду

SUBS, a, i, t,

которая интерпретируется как операция доступа $a[i]$ и позволяет выбрать элемент одномерного массива.

Представление элементов многомерных массивов с помощью тетрад основано на приведении многомерного массива к одномерному (*см. разд. 1.7.5.2*). Так, если двухмерный массив a, имеющий n строк и m столбцов, отображается в памяти по строкам, нумерация индексов начинается с единицы и размер элемента массива — одно слово, то обращение к элементу массива $a[i, j]$ можно записать следующим образом:

– , i, 1, t_1

* , t_1, i, t_2

\+ , t_2, j, t_3

– , t_3, 1, t_4

SUBS, a, t_4, t_5

Первые четыре тетрады необходимы для вычисления значения смещения элемента массива $a[i, j]$ от начала массива по формуле: $(i - 1) * m + j - 1$.

10.2.2. Указатели функций

Обращение к функции записывается с помощью одной или нескольких тетрад в зависимости от числа аргументов функции. Так, обращение к функции одной переменной abs(x) записывается в виде одной тетрады

CALL, abs, x, t,

а обращение к функции f с аргументами t_1, t_2, t_3 может быть записано при помощи двух тетрад, где отсутствие кода операции означает продолжение описания операндов предшествующей тетрады:

CALL, f, t_1,

, t_2, t_3, t_4

Возможна и другая реализация обращения к функции с использованием тетрад. Так обращение к функции двух переменных $f(x, y)$ можно записать в виде:

PARAM, x, ,

PARAM, y, ,

CALL , f, 2.

Здесь тетрады с кодом PARAM используются для определения параметров функции, а в тетраде CALL указывается имя функции и число ее параметров.

10.2.3. Операторы

Формат тетрад для представления оператора присваивания и операторов безусловного и условного переходов приведен в табл. 10.4.

Таблица 10.4. *Формат тетрад для представления основных операторов*

Тетрада				Семантика тетрады
:=	*b*		*a*	*a* := *b*
DEFL	*L*			Определение метки *l*
BRL	*l*			Безусловный переход к тетраде с меткой *l*
BF	*l*	*E*		Переход к тетраде с меткой *l*, если значение выражения *E* равно "ложь"
BR	*m*			Безусловный переход на тетраду с номером *m*
BZL (BPL, BML)	*m*	*E*		Переход к тетраде с меткой *m*, если значение выражения *E* равно нулю (больше нуля, меньше нуля)
BZ (BP, BM)	*m*	*E*		Переход на тетраду с номером *m*, если значение выражения *E* равно нулю (больше нуля, меньше нуля)
BE (BP, BM)	*m*	*e*		Переход на тетраду с номером *m*, если значение выражения *E* равно нулю (больше нуля, меньше нуля)
BE (BL, BG)	*m*	e_1	e_2	Переход на тетраду с номером *m*, если значение выражения E_1 равно (меньше, больше) значения выражения E_2

В табл. 10.5 приведено представление оператора условного перехода и операторов цикла с помощью тетрад из табл. 10.4.

Замечание

В табл. 10.5 функция signum служит для проверки знака выражения (первый операнд задает значение проверяемого выражения, а второй операнд возвращает знак выражения).

Тетрады удобно использовать для выполнения машинно-независимой оптимизации, в частности, исключения лишних операций. Основным недостатком тетрад является большой объем памяти, необходимый для их хранения. Несмотря на то, что во многих тетрадах имеются свободные поля, результат выполнения операции всегда записывается в четвертое поле.

Внутреннее представление тетрады — запись, состоящая из четырех лексем. При этом коды операций тетрад, пустые поля, номера тетрад и временные переменные t_i являются лексемами специального типа.

Таблица 10.5. *Представление оператора условного перехода и операторов цикла помощью тетрад из табл. 10.4*

Оператор	Формат оператора			
	Код операции	**Операнд_1**	**Операнд_2**	**Результат**
if <выражение> **then** <операторы_1> **else** <операторы_2> **end**	<выражение> (результат в *t*)			
	BF	*Lelse*	*t*	
	<операторы_1>			
	BRL	*Lend*		
	DEFL	*Lelse*		
	<операторы_2>			
	DEFL	*Lend*		
for j := <выражение_1> **step** <выражение_2> **until** <выражение_3> **do** <оператор>	<выражение_1> (результат в t_1)			
	:=	t_1		*j*
	<выражение_2> (результат в t_2) <выражение_3> (результат в t_3)			
	DEFL – CALL * BPL	*Lbegin* *j* signum t_4 *Lend*	 t_3 t_2 t_5 t_6	 t_4 t_5 t_6
	<оператор>			
	+ BRL DEFL	*j* *Lbegin* *Lend*	t_2	*j*
while <выражение> **do** <оператор>	DEFL	*Lbegin*		
	<выражение> (результат в *t*)			
	BF	*t*	*Lend*	
	<оператор>			
	BRL DEFL	*Lbegin* *Lend*		

10.3. Триады

Триады так же, как и тетрады, являются удобной промежуточной формой представления бинарных операций. Формат триады имеет вид:

<код операции>, <операнд_1>, <операнд_2>.

В качестве кодов операций триад можно использовать коды операций тетрад, за исключением операций условного перехода BE (BL, BM), которые для триад не используются.

В триадах отсутствует поле для записи результата. Если в качестве операнда некоторой триады нужно использовать результат выполнения другой (ранее выполненной) триады, то в соответствующее поле триады записывается ссылка на триаду, в которой этот результат был получен. Результаты выполнения триад хранятся отдельно. Например, запись в виде последовательности триад выражения

$a * b - c / d$

будет иметь следующий вид (номера триад указаны в скобках):

(1) *, *a*, *b*

(2) /, *c*, *d*

(3) −, (1), (2)

Количество триад и тетрад, необходимое для представления одних и тех же конструкций языка, является одним и тем же, но каждая триада занимает в памяти меньше места. Экономия памяти при использовании триад в качестве промежуточной формы программы по сравнению с тетрадами достигается также за счет того, что после выполнения триады, в которой одним из операндов является результат выполнения *i*-ой триады, ее результат может быть уничтожен.

При выполнении машинно-независимой оптимизации некоторые операции могут удаляться из промежуточного представления программы или переставляться на другое место. При использовании тетрад такие преобразования не вызывают затруднений. В случае же использования триад могут возникнуть осложнения из-за того, что триады ссылаются друг на друга. Поэтому, если машинно-независимая оптимизация будет выполняться, необходимо либо хранить все результаты выполнения триад, либо использовать *косвенные триады*.

Косвенные триады представляются двумя списками. В первом списке хранятся сами триады, причем одинаковые триады в список заносятся только один раз. Второй список содержит номера триад в порядке их выполнения. При использовании косвенных триад перестановка и исключение идентичных операций во время машинно-независимой оптимизации производятся путем преобразования списка, содержащего номера триад. При этом вид

триад и их количество в списке не изменяются. Использование косвенных триад приводит к дополнительному сокращению объема памяти, необходимого для хранения промежуточной формы программы, если исходная программа содержит большое число идентичных операций (обычно это имеет место при наличии в программе большого числа индексированных переменных).

Пример представления последовательности операторов присваивания языка Паскаль с помощью косвенных триад приведены в табл. 10.7.

Таблица 10.7. *Представление последовательности операторов присваивания языка Паскаль с помощью косвенных триад*

Оператор	Список триад			Порядок выполнения триад
A := *B*[*j*] + *B*[*j* + 1];	SUBS	*B*	*j*	(1), (2), (3), (4), (5), (1), (2), (3), (6), (7), (1), (2), (3), (8), (9), (10)
C := *B*[*j*] – *B*[*j* + 1];	+	*j*	1	
D := *B*[*j*] + 2 * *B*[*j* + 1]	SUBS	*B*	(2)	
	+	(1)	(3)	
	:=	(4)	*A*	
	-	(1)	(3)	
	:=	(6)	*C*	
	*	2	(3)	
	+	(1)	(8)	
	:=	(9)	*D*	

10.4. Байт-коды JVM

Виртуальная машина Java базируется на понятии виртуального компьютера с логическим набором инструкций (*псевдокодов*), определяющих операции, которые может выполнить этот компьютер. *Интерпретатор* JVM — это программа, которая понимает семантику байт-кодов JVM и преобразует их в инструкции объектной машины. JVM базируется на концепции *стековой машины*.

Стековая машина не использует никакие физические регистры для передачи информации от одной инструкции к другой. Вместо этого используется стек, в котором производятся вычисления. В JVM имеется единственный

регистр, называемый *регистром PC*, который содержит *адрес* текущей выполняемой инструкции в потоке байт-кодов. Инструкции байт-кода выполняются последовательно. Для изменения порядка выполнения байт-кодов имеются специальные команды, изменяющие содержимое регистра PC.

Большинство семантических процедур, реализующих инструкции байт-кода, получают свои операнды из стека и помещают результаты обратно в стек. Инструкции могут иметь также аргументы, которые следуют в потоке байт-кодов непосредственно за самой инструкцией. В табл. 10.7 приводятся наиболее часто используемые инструкции байт-кода, упорядоченные по коду операции. Полный список всех инструкций JVM можно найти, например, в [8]. Табл. 10.7 содержит следующие столбцы:

- инструкция — символическое имя инструкции;
- код операции — фактическое значение байт-кода в шестнадцатиричной системе счисления;
- количество аргументов — количество однобайтовых операндов в потоке, непосредственно следующих за инструкцией;
- описание — описание семантики (правил выполнения) инструкции.

Таблица 10.7. *Основные инструкции байт-кода*

Инструкция	Код операции	Количество аргументов	Описание
`aconst_null`	01	0	Помещает в стек ссылку `null` на объект
`iconst_0`	03	0	Помещает в стек константу 0 типа `int`
`iconst_1`	04	0	Помещает в стек константу 1 типа `int`
`lconst_0`	09	0	Помещает в стек константу 0 типа `long`
`lconst_1`	10	0	Помещает в стек константу 1 типа `long`
`fconst_0`	0B	0	Помещает в стек константу 0 типа `float`
`fconst_1`	0C	0	Помещает в стек константу 1 типа `float`
`dconst_0`	0E	0	Помещает в стек константу 0 типа `double`
`dconst_1`	0F	0	Помещает в стек константу 1 типа `double`
`bipush`	10	1	Помещает в стек однобайтовое число со знаком как целое
`sipush`	11	2	Помещает в стек 16-битовое число со знаком как целое

Таблица 10.7 *(продолжение)*

Инструкция	Код операции	Количество аргументов	Описание
`iload`	15	1	Помещает в стек значение локальной переменной типа `int`, найденной во фрейме текущего метода по индексу. Индекс содержится в аргументе
`lload`	16	1	Помещает в стек значение локальной переменной типа `long`, найденной во фрейме текущего метода по индексу (и индексу + 1). Индекс содержится в аргументе
`fload`	17	1	Помещает в стек значение локальной переменной типа `float`, найденной во фрейме текущего метода по индексу. Индекс содержится в аргументе
`dload`	18	1	Помещает в стек значение локальной переменной типа `double`, найденной во фрейме текущего метода по индексу (и индексу + 1). Индекс содержится в аргументе
`aload`	19	1	Помещает в стек значение локальной переменной типа "ссылка на объект", найденной во фрейме текущего метода по индексу. Индекс содержится в аргументе
`iload_0`	1A	0	Помещает в стек значение локальной переменной типа `int`, найденной во фрейме текущего метода по индексу 0
`iload_1`	1B	0	Помещает в стек значение локальной переменной типа `int`, найденной во фрейме текущего метода по индексу 1
`iload_2`	1C	0	Помещает в стек значение локальной переменной типа `int`, найденной во фрейме текущего метода по индексу 2
`iload_3`	1D	0	Помещает в стек значение локальной переменной типа `int`, найденной во фрейме текущего метода по индексу 3
`lload_0`	1E	0	Помещает в стек значение локальной переменной типа `long`, найденной во фрейме текущего метода по индексам 0 и 1

Таблица 10.7 *(продолжение)*

Инструкция	Код операции	Количество аргументов	Описание
lload_1	1F	0	Помещает в стек значение локальной переменной типа long, найденной во фрейме текущего метода по индексам 1 и 2
lload_2	20	0	Помещает в стек значение локальной переменной типа long, найденной во фрейме текущего метода по индексам 2 и 3
lload_3	21	0	Помещает в стек значение локальной переменной типа long, найденной во фрейме текущего метода по индексам 3 и 4
fload_0	22	0	Помещает в стек значение локальной переменной типа float, найденной во фрейме текущего метода по индексам 0 и 1
fload_1	23	0	Помещает в стек значение локальной переменной типа float, найденной во фрейме текущего метода по индексам 1 и 2
fload_2	24	0	Помещает в стек значение локальной переменной типа float, найденной во фрейме текущего метода по индексам 2 и 3
fload_3	25	0	Помещает в стек значение локальной переменной типа float, найденной во фрейме текущего метода по индексам 3 и 4
dload_0	26	0	Помещает в стек значение локальной переменной типа double, найденной во фрейме текущего метода по индексам 0 и 1
dload_1	27	0	Помещает в стек значение локальной переменной типа double, найденной во фрейме текущего метода по индексам 1 и 2
dload_2	28	0	Помещает в стек значение локальной переменной типа double, найденной во фрейме текущего метода по индексам 2 и 3
dload_3	29	0	Помещает в стек значение локальной переменной типа double, найденной во фрейме текущего метода по индексам 3 и 4
aload_0	2A	0	Помещает в стек значение локальной переменной типа "ссылка на объект", найденной во фрейме текущего метода по индексам 0 и 1

Таблица 10.7 *(продолжение)*

Инструкция	Код операции	Количество аргументов	Описание
aload_1	2B	0	Помещает в стек значение локальной переменной типа "ссылка на объект", найденной во фрейме текущего метода по индексам 1 и 2
aload_2	2C	0	Помещает в стек значение локальной переменной типа "ссылка на объект", найденной во фрейме текущего метода по индексам 2 и 3
aload_3	2D	0	Помещает в стек значение локальной переменной типа "ссылка на объект", найденной во фрейме текущего метода по индексам 3 и 4
iaload	2E	0	Читает из стека индекс массива и ссылку на массив типа int и помещает в стек значение элемента с этим индексом
laload	2F	0	Читает из стека индекс массива и ссылку на массив типа long и помещает в стек значение элемента с этим индексом
faload	30	0	Читает из стека индекс массива и ссылку на массив типа float и помещает в стек значение элемента с этим индексом
daload	31	0	Читает из стека индекс массива и ссылку на массив типа double и помещает в стек значение элемента с этим индексом
aaload	32	0	Читает из стека индекс массива и ссылку на массив ссылок и помещает в стек значение элемента с этим номером
caload	34	0	Читает из стека индекс массива и ссылку на массив типа char и помещает в стек значение элемента с этим индексом
istore	36	1	Читает из стека значение типа int и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексу. Индекс содержится в аргументе

Таблица 10.7 *(продолжение)*

Инструкция	Код операции	Количество аргументов	Описание
`lstore`	37	1	Читает из стека значение типа `long` и записывает его в локальную переменную по указателю (и индексу + 1) на фрейм текущего метода. Индекс содержится в аргументе
`fstore`	38	1	Читает из стека значение типа `float` и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексу. Индекс содержится в аргументе
`dstore`	39	1	Читает из стека значение типа `double` и записывает его в локальную переменную по указателю (и индексу + 1) на фрейм текущего метода. Индекс содержится в аргументе
`astore`	3A	1	Читает из стека ссылку на объект и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексу. Индекс содержится в аргументе
`istore_0`	3B	0	Читает из стека значение типа `int` и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексу 0
`istore_1`	3C	0	Читает из стека значение типа `int` и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексу 1
`istore_2`	3D	0	Читает из стека значение типа `int` и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексу 2
`istore_3`	3E	0	Читает из стека значение типа `int` и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексу 3
`lstore_0`	3F	0	Читает из стека значение типа `long` и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексам 0 и 1

Таблица 10.7 *(продолжение)*

Инструкция	Код операции	Количество аргументов	Описание
`lstore_1`	40	0	Читает из стека значение типа `long` и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексам 1 и 2
`lstore_2`	41	0	Читает из стека значение типа `long` и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексам 2 и 3
`lstore_3`	42	0	Читает из стека значение типа `long` и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексам 3 и 4
`fstore_0`	43	0	Читает из стека значение типа `float` и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексу 0
`fstore_1`	44	0	Читает из стека значение типа `float` и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексу 1
`fstore_2`	45	0	Читает из стека значение типа `float` и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексу 2
`fstore_3`	46	0	Читает из стека значение типа `float` и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексу 3
`dstore_0`	47	0	Читает из стека значение типа `double` и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексам 0 и 1
`dstore_1`	48	0	Читает из стека значение типа `double` и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексам 1 и 2

Таблица 10.7 *(продолжение)*

Инструкция	Код операции	Количество аргументов	Описание
dstore_2	49	0	Читает из стека значение типа double и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексам 2 и 3
dstore_3	4A	0	Читает из стека значение типа double и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексам 3 и 4
astore_0	4B	0	Читает из стека ссылку на объект и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексу 0
astore_1	4C	0	Читает из стека ссылку на объект и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексу 1
astore_2	4D	0	Читает из стека ссылку на объект и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексу 2
astore_3	4E	0	Читает из стека ссылку на объект и записывает его в локальную переменную, расположенную во фрейме текущего метода по индексу 3
iastore	4F	0	Читает из стека значение типа int, индекс массива и ссылку на массив типа int и записывает это значение в элемент массива с этим индексом
lastore	50	0	Читает из стека значение типа long, индекс массива и ссылку на массив типа long и записывает это значение в элемент массива с этим индексом
fastore	51	0	Читает из стека значение типа float, индекс массива и ссылку на массив типа float и записывает это значение в элемент массива с этим индексом
dastore	52	0	Читает из стека значение типа double, индекс массива и ссылку на массив типа double и записывает это значение в элемент массива с этим индексом

Таблица 10.7 *(продолжение)*

Инструкция	Код операции	Количество аргументов	Описание
aastore	53	0	Читает из стека ссылку на объект, индекс массива и ссылку на массив ссылок и записывает ссылку на объект в элемент массива с этим индексом
castore	55	0	Читает из стека значение типа char, индекс массива и ссылку на массив типа char и записывает это значение в элемент массива с этим индексом
pop	57	0	Читает из стека слово
pop2	58	0	Читает из стека два слова
dup	59	0	Дублирует слово из верхушки стека
dup_x1	5A	0	Дублирует слово из верхушки стека и помещает это значение двумя словами ниже
dup2	5C	0	Дублирует два слова из верхушки стека
dup2_x1	5D	0	Дублирует два слова из верхушки стека и помещает эти значения двумя словами ниже
swap	5F	0	Меняет местами два слова в верхушке стека
iadd	60	0	Читает из стека два значения типа int, складывает их и помещает результат в стек
ladd	61	0	Читает из стека два значения типа long, складывает их и помещает результат в стек
fadd	62	0	Читает из стека два значения типа float, складывает их и помещает результат в стек
dadd	63	0	Читает из стека два значения типа double, складывает их и помещает результат в стек
isub	64	0	Читает из стека два значения типа int, вычитает их и помещает результат в стек

Таблица 10.7 *(продолжение)*

Инструкция	Код операции	Количество аргументов	Описание
lsub	65	0	Читает из стека два значения типа long, вычитает их и помещает результат в стек
fsub	66	0	Читает из стека два значения типа float, вычитает их и помещает результат в стек
dsub	67	0	Читает из стека два значения типа double, вычитает их и помещает результат в стек
imul	68	0	Читает из стека два значения типа int, перемножает их и помещает результат в стек
lmul	69	0	Читает из стека два значения типа long, перемножает их и помещает результат в стек
fmul	6A	0	Читает из стека два значения типа float, перемножает их и помещает результат в стек
dmul	6B	0	Читает из стека два значения типа double, перемножает их и помещает результат в стек
idiv	6C	0	Читает из стека два значения типа int, делит их и помещает результат в стек
ldiv	6D	0	Читает из стека два значения типа long, делит их и помещает результат в стек
fdiv	6E	0	Читает из стека два значения типа float, делит их и помещает результат в стек
ddiv	6F	0	Читает из стека два значения типа double, делит их и помещает результат в стек
irem	70	0	Читает из стека два значения типа int, вычитает их и остаток помещает в стек
lrem	71	0	Читает из стека два значения типа long, вычитает их и остаток помещает в стек
frem	72	0	Читает из стека два значения типа float, вычитает их и остаток помещает в стек
drem	73	0	Читает из стека два значения типа double, вычитает их и остаток помещает в стек

Таблица 10.7 *(продолжение)*

Инструкция	Код операции	Количество аргументов	Описание
ineg	74	0	Читает из стека значение типа int, вычисляет его арифметическое отрицание и помещает результат в стек
lneg	75	0	Читает из стека значение типа long, вычисляет его арифметическое отрицание и помещает результат в стек
fneg	76	0	Читает из стека значение типа float, вычисляет его арифметическое отрицание и помещает результат в стек
dneg	77	0	Читает из стека значение типа double, вычисляет его арифметическое отрицание и помещает результат в стек
iand	7E	0	Читает из стека два значения типа int, выполняет поразрядное И и помещает результат в стек
ior	80	0	Читает из стека два значения типа int, выполняет поразрядное ИЛИ и помещает результат в стек
i2l	85	0	Читает из стека значение типа int, преобразует его в значение типа long и помещает в стек
i2f	86	0	Читает из стека значение типа int, преобразует его в значение типа float и помещает в стек
i2d	87	0	Читает из стека значение типа int, преобразует его в значение типа double и помещает в стек
l2i	88	0	Читает из стека значение типа long, преобразует его в значение типа int и помещает в стек
l2f	89	0	Читает из стека значение типа long, преобразует его в значение типа float и помещает в стек
l2d	8A	0	Читает из стека значение типа long, преобразует его в значение типа double и помещает в стек

Таблица 10.7 *(продолжение)*

Инструкция	Код операции	Количество аргументов	Описание
f2i	8B	0	Читает из стека значение типа float, преобразует его в значение типа int и помещает в стек
f2l	8C	0	Читает из стека значение типа float, преобразует его в значение типа long и помещает в стек
f2d	8D	0	Читает из стека значение типа float, преобразует его в значение типа double и помещает в стек
d2i	8E	0	Читает из стека значение типа double, преобразует его в значение типа int и помещает в стек
d2l	8F	0	Читает из стека значение типа double, преобразует его в значение типа long и помещает в стек
d2f	90	0	Читает из стека значение типа double, преобразует его в значение типа float и помещает в стек
int2char	92	0	Читает из стека значение типа int, преобразует его в значение типа char и помещает в стек
lcmp	94	0	Читает из стека значение_2 и значение_1, оба типа long. Если значение_1 больше значения_2, в стек помещается 1. Если значение_1 равно значению_2, в стек помещается 0. Если значение_1 меньше значения_2, в стек помещается –1
fcmpl	95	0	Читает из стека значение_2 и значение_1, оба типа float. Если значение_1 больше значения_2, в стек помещается 1. Если значение_1 равно значению_2, в стек помещается 0. Если значение_1 меньше значения_2 или одно из значений не является числом, в стек помещается –1

Таблица 10.7 *(продолжение)*

Инструкция	Код операции	Количество аргументов	Описание
dcmpl	97	0	Читает из стека значение_2 и значение_1, оба типа double. Если значение_1 больше значения_2, в стек помещается 1. Если значение_1 равно значению_2, в стек помещается 0. Если значение_1 меньше значения_2 или одно из значений не является числом, в стек помещается –1
ifeq	99	2	Читает из стека значение типа int. Если оно равно 0, два аргумента складываются и результат прибавляется к текущему значению PC. В противном случае выполняется следующая инструкция
ifne	9A	2	Читает из стека значение типа int. Если оно не равно 0, два аргумента складываются, и результат прибавляется к текущему значению PC. В противном случае выполняется следующая инструкция
iflt	9B	2	Читает из стека значение типа int. Если оно меньше 0, два аргумента складываются и результат прибавляется к текущему значению PC. В противном случае выполняется следующая инструкция
ifge	9C	2	Читает из стека значение типа int. Если оно больше или равно 0, два аргумента складываются и результат прибавляется к текущему значению PC. В противном случае выполняется следующая инструкция
ifgt	9D	2	Читает из стека значение типа int. Если оно больше 0, два аргумента складываются и результат прибавляется к текущему значению PC. В противном случае выполняется следующая инструкция
ifle	9E	2	Читает из стека значение типа int. Если оно меньше или равно 0, два аргумента складываются и результат прибавляется к текущему значению PC. В противном случае выполняется следующая инструкция

Таблица 10.7 *(продолжение)*

Инструкция	Код операции	Количество аргументов	Описание
if_icmpeq	9F	2	Читает из стека значение_2 и значение_1, оба типа int. Если значение_1 равно значению_2, два аргумента складываются и результат прибавляется к текущему значению PC. В противном случае выполняется следующая инструкция
if_icmpne	A0	2	Читает из стека значение_2 и значение_1, оба типа int. Если значение_1 не равно значению_2, два аргумента складываются и результат прибавляется к текущему значению PC. В противном случае выполняется следующая инструкция
if_icmplt	A1	2	Читает из стека значение_2 и значение_1, оба типа int. Если значение_1 меньше значения_2, два аргумента складываются, и результат прибавляется к текущему значению PC. В противном случае выполняется следующая инструкция
if_icmpge	A2	2	Читает из стека значение_2 и значение_1, оба типа int. Если значение_1 больше или равно значению_2, два аргумента складываются и результат прибавляется к текущему значению PC. В противном случае выполняется следующая инструкция
if_icmpgt	A3	2	Читает из стека значение_2 и значение_1, оба типа int. Если значение_1 больше значения_2, два аргумента складываются и результат прибавляется к текущему значению PC. В противном случае выполняется следующая инструкция
if_icmple	A4	2	Читает из стека значение_2 и значение_1, оба типа int. Если значение_1 меньше или равно значению_2, два аргумента складываются и результат прибавляется к текущему значению PC. В противном случае выполняется следующая инструкция
if_acmpeq	A5	2	Читает из стека ссылку_2 на объект и ссылку_1 на объект. Если они ссылаются на один и тот же объект, два аргумента складываются и результат прибавляется к текущему значению PC. В противном случае выполняется следующая инструкция

Таблица 10.7 *(продолжение)*

Инструкция	Код операции	Количество аргументов	Описание
`if_acmpne`	A6	2	Читает из стека `ссылку_2` на объект и `ссылку_1` на объект. Если они не ссылаются на один и тот же объект, два аргумента складываются и результат прибавляется к текущему значению PC. В противном случае выполняется следующая инструкция
`goto`	A7	2	Два аргумента складываются, образуя 16-разрядное значение, и результат прибавляется к текущему значению PC
`jsr`	A8	2	Два аргумента складываются, образуя 16-разрядное значение. В стек записывается значение PC, указывающее на инструкцию, непосредственно следующую за данной. 16-разрядное значение прибавляется к PC, чтобы поток выполнения перешел к подпрограмме. При входе в подпрограмму адрес возврата читается из стека и сохраняется в локальной переменной для последующего использования инструкциями `ret` и `ret_w`
`ret`	A9	1	Аргумент используется как указатель на локальную переменную во фрейме метода, которая содержит адрес возврата в вызвавшую программу. Этот адрес записывается в регистр PC, чтобы поток управления вернулся в вызвавшую программу
`ireturn`	AC	0	Читает из стека текущего метода значение типа `int`. Затем оно помещается в стек фрейма вызвавшего метода. Управление возвращается вызвавшему методу
`lreturn`	AD	0	Читает из стека текущего метода значение типа `long`. Затем оно помещается в стек фрейма вызвавшего метода. Управление возвращается вызвавшему методу
`freturn`	AE	0	Читает из стека текущего метода значение типа `float`. Затем оно помещается в стек фрейма вызвавшего метода. Управление возвращается вызвавшему методу

Таблица 10.7 *(окончание)*

Инструкция	Код операции	Количество аргументов	Описание
dreturn	AF	0	Читает из стека текущего метода значение типа double. Затем оно помещается в стек фрейма вызвавшего метода. Управление возвращается вызвавшему методу
areturn	B0	0	Читает из стека текущего метода ссылку на объект. Затем она помещается в стек фрейма вызвавшего метода. Управление возвращается вызвавшему методу
return	B1	0	Управление возвращается вызвавшему методу без помещения какого-либо значения в стек
new	BB	2	Создает новый объект, основываясь на типе класса, определяемого аргументами. Два аргумента складываются, образуя 16-разрядное смещение в пуле констант к элементу "ссылка на класс". Определяется класс и создается новая ссылка на объект для этого класса. Затем эта ссылка помещается в стек
newarray	BC	1	Размещает в памяти новый массив элементов одного из встроенных типов. Тип задается аргументом: 5 — char, 6 — float, 10 — int. Количество элементов массива уже находится в стеке к моменту входа в данную инструкцию
ifnull	C6	2	Читает из стека ссылку на объект. Если это null, два аргумента складываются и результат прибавляется к текущему значению PC. В противном случае выполняется следующая инструкция
ifnonnull	C7	2	Читает из стека ссылку на объект. Если это не null, два аргумента складываются и результат прибавляется к текущему значению PC. В противном случае выполняется следующая инструкция
ret_w	D1	2	Два аргумента складываются, образуя 16-разрядный указатель на локальную переменную во фрейме метода, которая содержит адрес возврата в вызвавшую программу. Этот адрес записывается в регистр PC, чтобы поток управления вернулся в вызвавшую программу

Пример 10.5

С использованием операций из табл. 10.7 байт-код фрагмента программы на Java

```
x = 0;  a = 1;  b = 2;  c = 5;
x = a + b * c;
```

имеет следующий вид:

03		Записать в стек 0
3C		Прочитать содержимое стека в `x`
04		Записать в стек 1
3D		Прочитать содержимое стека в `a`
10	02	Записать в стек 2
3E		Прочитать содержимое стека в `b`
10	05	Записать в стек 5
36	04	Прочитать содержимое стека в `c`
1C		Поместить в стек `a`
1B		Поместить в стек `b`
15	04	Поместить в стек `c`
68		Поместить в стек `bc*`
60		Поместить в стек `a+`
3C		Прочитать содержимое стека в `x`

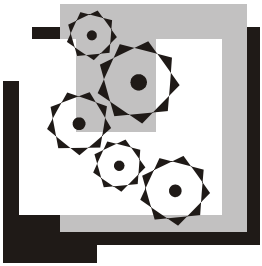
Контрольные вопросы

1. Почему исходная программа переводится сначала в промежуточное представление, а затем в объектный код?
2. Как арифметическое выражение записывается в ПОЛИЗ? Как в ПОЛИЗ записывается унарный минус?
3. Почему в компиляторах используется для представления выражений польская инверсная запись?
4. Как производится вычисление выражения, представленного в польской инверсной записи?
5. Как графически представляется польская инверсная запись?
6. Как можно представить в тетрадах переменную с индексами, указатель функции?

7. Чем различаются триады и косвенные триады?
8. Что представляет собой виртуальная машина Java?
9. Как обрабатываются инструкции байт-кода?

Упражнения

1. Представьте графически и в польской инверсной записи выражения:
   1.1. $\left|\cos x + \sin y - z\right|$.
   1.2. $\sqrt{b + \left|a\right|}$.
   1.3. $1 + a_{i,j} - \operatorname{ctg}\frac{x}{\sqrt{x^2 + 1}}$.
2. Представьте в ПОЛИЗ, тетрадах и триадах операторы:
   2.1. `a := b² + 4ac`.
   2.2. `if (b && c < d) a++; else a += 2`.
   2.3. `for j := n downto 1 do a[j + 1] := a[j]`.



Глава 11

Формальные методы описания перевода

Формальные грамматики позволяют описывать только синтаксические аспекты формальных языков. При построении трансляторов необходимо помнить, что каждой входной цепочке (программе на входном языке) необходимо поставить в соответствие другую цепочку (программу на выходном языке). Отношение между входными и выходными цепочками, которые определяют значение или смысл входных цепочек, называют *переводом*.

При разработке формальных методов описания перевода так же, как при описании синтаксиса языка, возникает проблема задания бесконечного перевода конечными средствами.

11.1. Перевод и семантика

Существуют два подхода к описанию перевода:

- использование системы, аналогичной формальной грамматике или являющейся ее расширением, которая порождает перевод (точнее пару цепочек, принадлежащих переводу);
- применение автомата, распознающего входную цепочку и выдающего на выходе перевод этой цепочки.

Естественными требованиями к устройству, выполняющему трансляцию, являются:

- простота анализа и синтеза: мы должны, с одной стороны, по описанию перевода достаточно легко установить, какие пары цепочек принадлежат переводу, и, с другой стороны, построение описания перевода не должно быть непосильной задачей для разработчика;
- возможность автоматического построения транслятора по описанию перевода;

- небольшой объем и эффективность трансляции;
- корректность и т. п.

Перейдем к рассмотрению простейших способов описания перевода.

Пусть Σ — входной алфавит и Δ — выходной алфавит. *Переводом* с языка $L_1 \subseteq \Sigma^*$ на язык $L_2 \subseteq \Delta^*$ называется отношение τ из Σ^* в Δ^*, для которого L_1 — область определения, а L_2 — множество значений.

Если $(x, y) \in \tau$, то цепочка y называется *выходом* для цепочки x. В общем случае в переводе τ для одной входной цепочки может быть задано более одной выходной цепочки.

Замечание

Для языков программирования перевод должен быть *функцией*, т. е. для каждой входной программы должно быть не более одной выходной программы.

Простейшие переводы можно задать при помощи *гомоморфизма*.

Пример 11.1

Пусть требуется перевести символы, образующие целые числа со знаком, в их названия. В этом случае гомоморфизм h можно определить следующим образом:

- $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, –, +\}$;
- $h(a) = a$, если $a \in \Sigma$;
- для $a \in \Sigma$ гомоморфизм $h(a)$ определяется в соответствии с табл. 11.1.

Таблица 11.1. *Определение гомоморфизма h(a)*

a	*h(a)*	*a*	*h(a)*	*a*	*h(a)*
1	*один*	5	*пять*	9	*девять*
2	*два*	6	*шесть*	0	*нуль*
3	*три*	7	*семь*	+	*плюс*
4	*четыре*	8	*восемь*	–	*минус*

Используя данное определение, перевод входной цепочки "–15", содержащей запись числа –15, будет иметь вид: "минус один пять".

Гомоморфизм позволяет описывать только простейшие переводы. Даже такие простые переводы, как $\tau_1 = \{(x, y) \mid x$ — инфиксное выражение и y — префиксная запись выражения $x\}$ и $\tau_2 = \{(x, y) \mid x$ — инфиксное выражение и y — постфиксная запись выражения $x\}$ нельзя задать при помощи гомоморфизма, нужны более мощные формализмы.

11.2. СУ-схемы

Схема синтаксически управляемого (СУ-схема) перевода представляет собой систему, порождающую пары цепочек, принадлежащих переводу. Неформально схема синтаксически управляемого перевода представляет собой грамматику, в которой каждому правилу приписан элемент перевода. При порождении цепочки каждый раз, когда правило участвует в этом процессе, элемент перевода генерирует часть выходной цепочки, соответствующей части входной цепочки, порожденной этим правилом.

Рассмотрим схему синтаксически управляемого перевода, описывающего перевод скобочного арифметического выражения, порождаемого грамматикой G_0, в соответствующую польскую инверсную запись. Схема перевода представлена в табл. 11.2.

Таблица 11.2. *СУ-схема перевода*

	Правило грамматики	**Элемент перевода**
(1)	$E \rightarrow E + T$	$E = ET +$
(2)	$E \rightarrow T$	$E = T$
(3)	$T \rightarrow T * P$	$T = TP *$
(4)	$T \rightarrow P$	$T = P$
(5)	$P \rightarrow (E)$	$P = E$
(6)	$P \rightarrow i$	$P = i$

В этой схеме правилу грамматики $E \rightarrow E + T$ соответствует элемент перевода $E = E\ T +$, который интерпретируется следующим образом: "Перевод нетерминала E, стоящего в *левой части* элемента перевода, представляет собой перевод, порожденный символом E, стоящим в *правой части* этого элемента, за которым следуют перевод, порождаемый символом T, и символ '+' ".

Используя СУ-схему, приведенную в табл. 11.2, определим перевод цепочки $a * (b + c)$.

Замечание

Для указания местоположения *символов* выходной цепочки, являющихся переводом символов входной цепочки, будем считать символы *a*, *b*, *c* значениями терминального символа *i* грамматики.

Вначале определим левый вывод входной цепочки:

$$E \Rightarrow_{(2)} T \Rightarrow_{(3)} T* P \Rightarrow_{(4)} P* P \Rightarrow_{(6)} a* P \Rightarrow_{(5)} a* (E) \Rightarrow_{(1)} a* (E + T) \Rightarrow_{(2)}$$
$$a* (T + T) \Rightarrow_{(4)} \quad a* (P + T) \Rightarrow_{(6)} \quad a* (b + T) \Rightarrow_{(4)} \quad a* (b + P) \Rightarrow_{(6)}$$
$$a* (b + c).$$

Полученный вывод 23465124646 определяет последовательность использования элементов перевода при порождении выходной цепочки, поэтому последовательность выводимых *пар цепочек* имеет следующий вид:

$$(E, E) \Rightarrow_{(2)} (T, T)$$
$$\Rightarrow_{(3)} (T* P, TP*)$$
$$\Rightarrow_{(4)} (P* P, PP*)$$
$$\Rightarrow_{(6)} (a* P, aP*)$$
$$\Rightarrow_{(5)} (a* (E), a* E)$$
$$\Rightarrow_{(1)} (a* (E + T), a* ET+)$$
$$\Rightarrow_{(2)} (a* (T + T), a* TT+)$$
$$\Rightarrow_{(4)} (a* (P + T), a* PT+)$$
$$\Rightarrow_{(6)} (a* (b + T), a* bT+)$$
$$\Rightarrow_{(4)} (a* (b + P), a* bP+)$$
$$\Rightarrow_{(6)} (a* (b + c), a* bc+).$$

Каждая выходная цепочка при порождении части входной цепочки получается путем замены (подходящего) нетерминала выходной цепочки значением соответствующего ему элемента перевода.

Рассмотренная схема перевода относится к классу схем, называемых *схемами синтаксически управляемого перевода.*

Схемой синтаксически управляемого перевода называется пятерка $T = (N, \Sigma, \Delta, R, S)$, где:

- N — конечное множество нетерминальных символов;
- Σ — конечный входной алфавит;
- Δ — конечный выходной алфавит;
- R — конечное множество правил вида $A \rightarrow \alpha, \beta$, где $\alpha \in (N \cup \Sigma)^*$, $\beta \in (N \cup \Delta)^*$ и вхождения нетерминалов в цепочку β образуют *перестановку* вхождений нетерминалов в цепочку α;
- S — начальный символ ($S \in N$).

По определению, каждому вхождению определенного нетерминала в цепочку α соответствует некоторое вхождение этого нетерминала в цепочку β. Если некоторый нетерминал B входит в цепочку α более одного раза, то может возникнуть неоднозначность. Для исключения этого будем пользоваться верхними целочисленными индексами, например в правиле $A \to B^{(1)}CB^{(2)}$, $B^{(2)}B^{(1)}C$ первой, второй и третьей позиции цепочки $B^{(1)}CB^{(2)}$ соответствуют вторая, третья и первая позиции цепочки $B^{(2)}B^{(1)}C$.

Выводимая пара цепочек СУ-схемы T определяется рекурсивно следующим образом:

1. (S, S) — выводимая пара, в которой символы S соответствуют друг другу.
2. Если $(\alpha A\beta, \alpha' A\beta')$ — выводимая пара, в которой два выделенных вхождения нетерминала A соответствуют друг другу, и $A \to \alpha$, β — правило из R, то $(\alpha\gamma\beta, \alpha'\gamma'\beta')$ — выводимая пара.

Вхождения нетерминалов в цепочки γ и γ' соответствуют друг другу точно так же, как они соответствовали в правиле СУ-схемы. Вхождения нетерминалов в цепочки α и β соответствуют вхождениям нетерминалов в цепочки α' и β' в новой выводимой паре точно так же, как они соответствовали в старой выводимой паре. При необходимости это соответствие будет указываться верхними индексами.

Если между парами цепочек $(\alpha A\beta, \alpha' A\beta')$ и $(\alpha\gamma\beta, \alpha'\gamma'\beta')$ установлена описанная ранее связь, то будем писать $(\alpha A\beta, \alpha' A\beta') \Rightarrow_T (\alpha\gamma\beta, \alpha'\gamma'\beta')$.

Транзитивное замыкание, рефлексивно-транзитивное замыкание и k-ю степень отношения $\Rightarrow_T$ будем обозначать как $\Rightarrow^+_T$, $\Rightarrow^*_T$ и $\Rightarrow^k_T$ соответственно. В очевидных случаях можно опускать индекс 'T'.

Переводом, определяемым схемой T, или $\tau(T)$ называют множество пар цепочек $\{(x, y) \mid (S, S) \Rightarrow^* (x, y), x \in \Sigma^*$ и $y \in \Delta^*\}$.

Пример 11.2

Пусть СУ-схема задана следующим образом [3]:

$$T = (\{S\}, \{i, +\}, \{i, +\}, R, S),$$

где множество правил R содержит правила

(1) $S \to + S^{(1)} S^{(2)}, S^{(1)} + S^{(2)}$

(2) $S \to i, i$

Рассмотрим вывод в данной СУ-схеме:

$$
\begin{aligned}
(S,\ S) &\Rightarrow_{(1)} (+\ S^{(1)}\ S^{(2)} &,\quad & S^{(1)} + S^{(2)})\\
&\Rightarrow_{(1)} (+\ +\ S^{(3)}\ S^{(4)}\ S^{(2)} &,\quad & S^{(3)} + S^{(4)} + S^{(2)})\\
&\Rightarrow_{(1)} (+\ +\ +\ S^{(5)}\ S^{(6)}\ S^{(4)}\ S^{(2)} &,\quad & S^{(5)} + S^{(6)} + S^{(4)} + S^{(2)})\\
&\Rightarrow_{(2)} (+\ +\ +\ a\ S^{(6)}\ S^{(4)}\ S^{(2)} &,\quad & a + S^{(6)}\ S^{(4)} + S^{(2)})\\
&\Rightarrow_{(2)} (+\ +\ +\ a\ b\ S^{(4)}\ S^{(2)} &,\quad & a + b + S^{(4)} + S^{(2)})\\
&\Rightarrow_{(2)} (+\ +\ +\ a\ b\ c\ S^{(2)} &,\quad & a + b + c + S^{(2)})\\
&\Rightarrow_{(2)} (+\ +\ +\ a\ b\ c\ d &,\quad & a + b + c + d).
\end{aligned}
$$

Входной цепочке $+\ +\ +\ i\ i\ i\ i$ соответствует выходная цепочка $i + i + i + i$. □

Очевидно, что перевод, определяемый СУ-схемой, имеет вид: $\tau(T) = \{(x,\ i(+\ i)^k) \mid k \geq 0$ и x — префиксная запись выражения $i(+i)^k\}$.

Если $T = (\mathrm{T}, \Sigma, \Delta, R, S)$ — СУ-схема, то $\tau(T)$ называется *синтаксически управляемым переводом*.

Грамматика $G_i = (\mathrm{N}, \Sigma, P, S)$, где $P = \{A \to \alpha \mid A \to \alpha, \beta, \in R\}$, называется *входной грамматикой СУ-схемы T*.

Грамматика $G_0 = (\mathrm{N}, \Delta, P', S)$, где $P' = \{A \to \beta \mid A \to \alpha, \beta, \in R\}$, называется *выходной грамматикой СУ-схемы T*.

К достоинствам СУ-схемы относится не только ее простота, но и наглядность ее интерпретации. Синтаксически управляемый перевод можно трактовать как метод преобразования деревьев выводов входной грамматики G_i в деревья выводов выходной грамматики G_0. Перевод данной входной цепочки x можно получить, построив ее дерево вывода, затем преобразовав это дерево в дерево вывода в выходной грамматике и, наконец, взяв крону выходного дерева в качестве перевода цепочки x [3].

Алгоритм 11.1. *Преобразование деревьев при помощи СУ-схемы*

Вход: СУ-схема $T = (\mathrm{N}, \Sigma, \Delta, R, S)$ с входной грамматикой $G_i = (\mathrm{N}, \Sigma, P, S)$ и выходной грамматикой $G_0 = (\mathrm{N}, \Delta, P', S)$ и дерево вывода D в G_i, с кроной, принадлежащей Σ^*.

Выход: Некоторое дерево вывода D' в G_0, такое, что если x и y — кроны деревьев D и D' соответственно, то $(x, y) \in \tau(T)$.

***Описание алгоритма*:**

□

1. Пусть данный шаг применяется к внутренней вершине n дерева D, имеющей k прямых потомков $n_1, \ldots, n_k$.

   1.1. Устранить из множества вершин $n_1, \ldots, n_k$ листья, помеченные терминальными символами или ε.

   1.2. Пусть $A \rightarrow \alpha$ — правило входной грамматики G_i, соответствующее вершине n и ее прямым потомкам, т. е. A — метка вершины n, и α образуется конкатенацией меток вершин $n_1, \ldots, n_k$. Выбрать из R некоторое правило вида $A \rightarrow \alpha, \beta$ (если таких правил несколько, то выбор произволен).

   Переставить оставшиеся прямые потомки вершины n (если они есть) согласно соответствию между вхождениями нетерминалов в α и β. (Поддеревья, корнями которых служат эти потомки, переставляются вместе с ними).

   1.3. Добавить в качестве прямых потомков вершины n листья с метками так, чтобы метки всех ее прямых потомков образовали цепочку β.

   1.4. Применить шаг 1 к прямым потомкам вершины n, не являющимся листьями, в порядке слева направо.

2. Повторять шаг 1 рекурсивно, начиная с корня дерева D.

Результирующим деревом будет D'.

■

Пример 11.3

🗁 Рассмотрим СУ-схему $T = (\{S, A\}, \{0, 1\}, \{a, b\}, R, S)$, где R состоит из правил:

(1) $S \rightarrow 0AS, SAa$

(2) $A \rightarrow 0SA, ASa$

(3) $S \rightarrow 1, b$

(4) $A \rightarrow 1, b$

Во входной грамматике этой схемы цепочка 0010111 имеет вывод 1232343 (дерево вывода приведено на рис. 11.1).

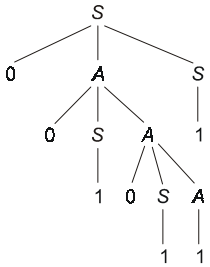


Рис. 11.1. Дерево вывода цепочки 0010111

Преобразуем это дерево при помощи алгоритма 11.1.

Шаг 1. Применим шаг 1 алгоритма к корню дерева *S*:

Устраняем левый лист дерева, помеченный 0. Находим правило входной грамматики (1) $S \rightarrow 0AS$, соответствующее корню дерева *S*. У этого правила только один элемент перевода *SAa*, который определяет, что нужно *поменять местами* прямые потомки корня *A* и *S*. *Добавляем* прямой потомок к корню дерева и *помечаем* его *a*.

Результат этих действий приведен на рис. 11.2, *а*.

Шаг 1. Рассматриваем прямой потомок корня дерева — вершину, помеченную символом *A*.

Устраняем лист 0. Используя правило (2) СУ-схемы, меняем местами прямые потомки рассматриваемой вершины. Добавляем прямой потомок *a* к рассматриваемой вершине и получаем дерево, приведенное на рис. 11.2, *б*.

Шаг 1. Рассматриваем прямой потомок корня дерева — вершину, помеченную символом *S*.

Устраняем лист 1. Находим соответствующее правило СУ-схемы — правило номер (3), и так как потомков у рассматриваемой вершины нет, то создаем прямой потомок *b* для рассматриваемой вершины и получаем дерево, приведенное на рис. 11.2, *в*.

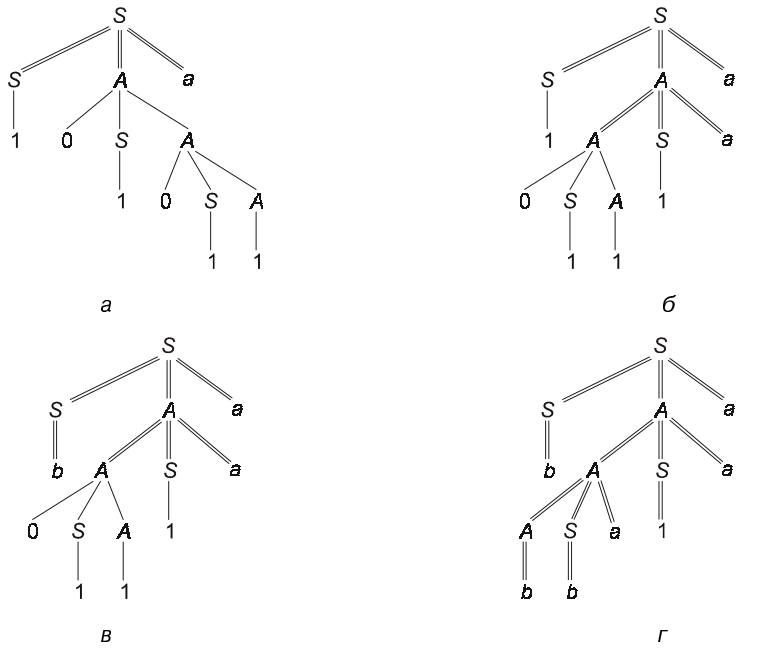


Рис. 11.2. Преобразование деревьев

Продолжая рекурсивно выполнение шага 1 алгоритма, получаем результирующее дерево, приведенное на рис. 11.2, *г*. □

В [3] доказаны следующие утверждения:

1. Если x и y — кроны деревьев D и D' из алгоритма 11.1, то $(x, y) \in \tau(T)$.
2. Если $(x, y) \in \tau(T)$, то существуют дерево вывода D с кроной x и такая последовательность выборов вершин при обращении к шагу 2.2 алгоритма, что в результате получается дерево D' с кроной y.

Замечание

Порядок применений шага 2 алгоритма 11.1 к вершинам дерева не важен. Можно выбрать любой порядок, при котором каждая внутренняя вершина рассматривается точно один раз.

СУ-схема $T = (N, \Sigma, \Delta, R, S)$ называется *простой*, если для каждого правила $A \to \alpha, \beta \in R$ соответствующие друг другу вхождения нетерминалов встречаются в α и β в одном и том же порядке.

Перевод, определяемый простой СУ-схемой, называется *простым синтаксически управляемым переводом*.

Соответствие нетерминалов в выводимой паре простой СУ-схемы определяется порядком, в котором эти нетерминалы появляются в цепочках, поэтому для нее легко построить транслятор, представляющий собой преобразователь с магазинной памятью.

По этой причине простые СУ-переводы образуют важный класс переводов.

Пример 11.4

Рассмотрим простую СУ-схему, имеющую следующие правила:

(1) $E \to (E)$, E (5) $T \to A * A$, $A * A$

(2) $E \to E + E$, $E + E$ (6) $T \to i$, i

(3) $E \to T$, T (7) $A \to (E + E)$, $(E + E)$

(4) $T \to (T)$, T (8) $A \to T$, T

и вывод входной цепочки $((i * (i * i) + i) * i)$, равный 3457358684586863686.

Соответствующий вывод *пар цепочек* имеет вид:

$$
\begin{aligned}
(E, E) &\Rightarrow_{(3)} (T & , T)\\
&\Rightarrow_{(4)} ((T) & , T)\\
&\Rightarrow_{(5)} ((A * A) & , A * A)\\
&\Rightarrow_{(7)} (((E + E) * A & , (E + E) * A)\\
&\Rightarrow_{(3)} (((T + E) * A & , (T + E) * A)\\
&\Rightarrow_{(5)} (((A * A + E) * A & , (A * A + E) * A)\\
&\Rightarrow_{(8)} (((T * A + E) * A & , (T * A + E) * A)\\
&\Rightarrow_{(6)} (((i * A + E) * A & , (i * A + E) * A)\\
&\Rightarrow_{(8)} (((i * T + E) * A & , (i * T + E) * A)\\
&\Rightarrow_{(4)} (((i * (T) + E) * A & , (i * T + E) * A)\\
&\Rightarrow_{(5)} (((i * (A * A) + E) * A & , (i * A * A + E) * A)\\
&\Rightarrow_{(8)} (((i * (T * A) + E) * A & , (i * T * A + E) * A)\\
&\Rightarrow_{(6)} (((i * (i * A) + E) * A & , (i * i * A + E) * A)\\
&\Rightarrow_{(8)} (((i * (i * T) + E) * A & , (i * i * T + E) * A)\\
&\Rightarrow_{(6)} (((i * (i * i) + E) * A & , (i * i * i + E) * A)\\
&\Rightarrow_{(3)} (((i * (i * i) + T) * A & , (i * i * i + T) * A)\\
&\Rightarrow_{(6)} (((i * (i * i) + i) * A & , (i * i * i + i) * A)\\
&\Rightarrow_{(8)} (((i * (i * i) + i) * T & , (i * i * i + i) * T)\\
&\Rightarrow_{(6)} (((i * (i * i) + i) * i & , (i * i * i + i) * i).
\end{aligned}
$$

Для входной цепочки $((i * (i * i) + i) * i$ СУ-схема порождает выходную цепочку $(i * i * i + i) * i$. Анализ позволяет сделать вывод, что СУ-схема отображает арифметические выражения, порождаемые грамматикой G_0, в арифметические выражения, не содержащие избыточных скобок. □

Рассмотрение СУ-схем позволяет понять идеи, лежащие в основе методов описания синтаксически управляемого перевода: включение в грамматику, описывающую порождение цепочек языка, элементов перевода, позволяющих описывать порождение выходных цепочек.

Для получения дополнительных сведений о СУ-схемах можно обратиться к [3, 4].

11.3. Транслирующие грамматики

Рассматривая процесс перевода инфиксных арифметических выражений в польскую инверсную запись, попытаемся построить некоторый процессор, выполняющий этот перевод.

Если на входной ленте процессора находится цепочка $a + b * c$, то процессор должен работать следующим образом:

- прочитать входной символ a;
- выдать символ a на выходную ленту;
- прочитать входной символ '+';
- прочитать входной символ b;
- выдать символ b на выходную ленту;
- прочитать входной символ '*';
- прочитать входной символ c;
- выдать символ c на выходную ленту;
- выдать символ '*' на выходную ленту;
- выдать символ '+' на выходную ленту.

Этот план работы определяется тем, что порядок следования операндов в инфиксной записи и ПОЛИЗ совпадает, а операции в ПОЛИЗ должны следовать сразу за своими операндами. Если операцию чтения символа из входной ленты мы будем обозначать читаемыми символами, а операцию записи — символами, помещаемыми на выходную ленту, заключенными в фигурные скобки (*операционными* символами), то работа по переводу может быть записана следующим образом:

$a\ \{a\} + b\ \{b\} * c\ \{c\}\ \{*\}\ \{+\}$.

Транслирующая грамматика — это КС-грамматика, множество терминалов которой разбито на два множества: множество входных и множество операционных символов.

Дадим формальное определение транслирующей грамматики.

Транслирующей грамматикой называется пятерка объектов $G^T = (N, \Sigma_i, \Sigma_a, P, S)$, где Σ_i — словарь входных символов, Σ_a — словарь операционных символов, N — нетерминальный словарь, $S \in N$ — начальный символ транслирующей грамматики, P — конечное множество правил вывода вида $A \rightarrow \alpha$, в которых $A \in N$, а $\alpha \in (\Sigma_i \cup \Sigma_a \cup N)^*$.

Транслирующая грамматика $G_0{}^T$, описывающая перевод инфиксных арифметических выражений в ПОЛИЗ, содержит следующие правила:

(1) $E \rightarrow E + T\ \{+\}$ (4) $T \rightarrow P$

(2) $E \rightarrow T$ (5) $P \rightarrow i\ \{i\}$

(3) $T \rightarrow T * P\ \{*\}$ (6) $P \rightarrow (E)$

Грамматика, полученная из транслирующей грамматики путем вычеркивания всех операционных символов, называется *входной грамматикой* для этой транслирующей грамматики. Язык, порождаемый входной грамматикой, называется *входным языком*.

Например, входной грамматикой G_0 для транслирующей грамматики $G_0{}^T$ является КС-грамматика для инфиксных арифметических выражений, содержащая правила:

(1) $E \to E + T$ (4) $T \to P$

(2) $E \to T$ (5) $P \to i$

(3) $T \to T * P$ (6) $P \to (E)$

Рассмотрим левый вывод цепочки $i + i * i$ во входной грамматике G_0:

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow P + T \Rightarrow i + T \Rightarrow i + T * P \Rightarrow i + P * P \Rightarrow i + i * P \Rightarrow i + i * i.$$

Применив эту же последовательность правил к соответствующим нетерминалам транслирующей грамматики $G_0{}^T$, получим следующий левый вывод:

$$E \Rightarrow E + T\,\{+\} \Rightarrow T + T\,\{+\} \Rightarrow P + T\,\{+\} \Rightarrow i\,\{i\} + T\,\{+\} \Rightarrow i\,\{i\} + T * P\,\{*\}\,\{+\} \Rightarrow i\,\{i\} + P * P\,\{*\}\,\{+\} \Rightarrow i\,\{i\} + i\,\{i\} * P\,\{*\}\,\{+\} \Rightarrow i\,\{i\} + i\,\{i\} * i\,\{i\}\,\{*\}\,\{+\}.$$

Цепочки языка, порождаемого транслирующей грамматикой, называют *активными цепочками*. *Входной частью* активной цепочки называется последовательность входных символов, полученная из активной цепочки после удаления всех операционных символов. *Операционной частью* активной цепочки называется последовательность операционных символов, полученная путем вычеркивания из активной цепочки всех входных символов.

Например, последовательность $i + i * i$ является входной частью активной цепочки $i\,\{i\} + i\,\{i\} * i\,\{i\}\,\{*\}\,\{+\}$, а последовательность $\{i\}\{i\}\{i\}\{*\}\{+\}$ — ее операционной частью.

Множество всех пар, первыми элементами которых являются входные части активной цепочки, а вторыми элементами — операционные части активной цепочки, называется *синтаксически управляемым переводом* $\tau(G^T)$, определяемым транслирующей грамматикой G^T.

В восходящих методах обработки языков широко применяются постфиксные транслирующие грамматики.

Транслирующая грамматика называется *постфиксной* транслирующей грамматикой тогда и только тогда, когда все операционные символы в правых частях правил вывода расположены правее всех входных и нетерминальных символов.

Примером постфиксной транслирующей грамматики является грамматика $G_0{}^T$.

Для получения транслирующей грамматики входной язык описывается входной КС-грамматикой. Затем в правила вывода этой грамматики вставляются операционные символы для описания семантических действий, связанных с соответствующими правилами.

Рассмотрим получение транслирующей грамматики $G_0{}^T$ по входной грамматике G_0.

В ПОЛИЗ операнды располагаются в той же последовательности, что и в инфиксной записи, а знаки операций следуют непосредственно после своих операндов в том порядке, в котором они выполняются. Для того чтобы идентификатор i выдавался сразу после его прочтения, правило (6) грамматики G_0 преобразуется к виду $P \rightarrow i\ \{i\}$. Чтобы знак операции сложения выдавался непосредственно после своих операндов, правило (1) заменяется на $E \rightarrow E + T\ \{+\}$. Это правило интерпретируется следующим образом: обработка арифметического выражения E состоит из обработки первого операнда E, чтения символа '+', обработки второго операнда T и выдачи символа '+'. Аналогичные рассуждения позволяют следующим образом преобразовать правило (3) грамматики:

$$T \rightarrow T * P\ \{*\}.$$

Один и тот же перевод можно определить разными транслирующими грамматиками.

Пример 11.5

🗁 Пусть две транслирующие грамматики $G_1{}^T$ и $G_2{}^T$, определяющие перевод в ПОЛИЗ инфиксных бесскобочных арифметических выражений, выполняемых в порядке написания операций, выглядят следующим образом:

$G_1{}^T$

(1) $E \rightarrow E + T\ \{+\}$

(2) $E \rightarrow E * T\ \{*\}$

(3) $E \rightarrow T$

(4) $T \rightarrow i\ \{i\}$

$G_2{}^T$

(1) $E \rightarrow i\ \{i\}\ R$

(2) $R \rightarrow + i\ \{i\}\ \{+\}\ R$

(3) $R \rightarrow * i\ \{i\}\ \{*\}\ R$

(4) $R \rightarrow \varepsilon$

Рассмотрим выполнение перевода цепочки $i + i * i$ с использованием транслирующих грамматик ${G_1}^T$ и ${G_2}^T$.

При использовании грамматики ${G_1}^T$ активная цепочка порождается следующим образом:

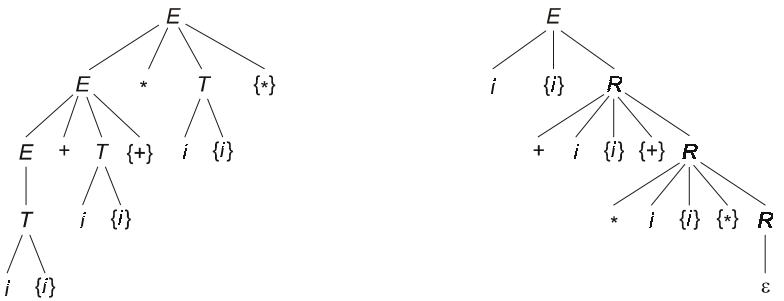
$$
\begin{aligned}
E &\Rightarrow_{(2)} E * T\{*\} \\
&\Rightarrow_{(1)} E + T\{+\} * T\{*\} \\
&\Rightarrow_{(3)} T + T\{+\} * T\{*\} \\
&\Rightarrow_{(4)} i\{i\} + T\{+\} * T\{*\} \\
&\Rightarrow_{(4)} i\{i\} + i\{i\}\{+\} * T\{*\} \\
&\Rightarrow_{(4)} i\{i\} + i\{i\}\{+\} * i\{i\}\{*\}
\end{aligned}
$$

Полученная операционная часть цепочки $\{i\}\{i\}\{+\}\{i\}\{*\}$ является переводом входной цепочки $i + i * i$.

Использование второй грамматики ${G_2}^T$ позволяет получить следующее порождение активной цепочки:

$$
\begin{aligned}
E &\Rightarrow_{(1)} i\{i\} R \\
&\Rightarrow_{(2)} i\{i\} + i\{i\}\{+\} R \\
&\Rightarrow_{(3)} i\{i\} + i\{i\}\{+\} * i\{i\}\{*\} R \\
&\Rightarrow_{(4)} i\{i\} + i\{i\}\{+\} * i\{i\}\{*\}
\end{aligned}
$$

Полученная операционная часть цепочки $\{i\}\{i\}\{+\}\{i\}\{*\}$ также является переводом входной цепочки. На рис. 11.3 приведены деревья вывода цепочки $i\{i\} + i\{i\}\{+\} * i\{i\}\{*\}$ с использованием грамматик ${G_1}^T$ и ${G_2}^T$ соответственно. □



а *б*

Рис. 11.3. Деревья вывода цепочки $i\{i\} + i\{i\}\{+\} * i\{i\}\{*\}$ в грамматиках ${G_1}^T$ (*а*) и ${G_2}^T$ (*б*)

Во многих практических приложениях появление входного символа в активной цепочке можно интерпретировать как обозначение операции чтения

этого символа некоторым устройством, а вхождению операционного символа в правило вывода можно сопоставить операцию выдачи символа, заключенного в фигурные скобки. При таком подходе входную часть активной цепочки называют входной цепочкой, операционную часть — выходной цепочкой, а транслирующую грамматику, определяющую перевод, — *грамматикой цепочечного перевода.*

Примером грамматики цепочечного перевода является грамматика $G_0{}^T$:

(1) $E \rightarrow E + T \{+\}$ (4) $T \rightarrow P$

(2) $E \rightarrow T$ (5) $P \rightarrow i \{i\}$

(3) $T \rightarrow T * P \{*\}$ (6) $P \rightarrow (E)$

Другая возможная интерпретация операционных символов — имена семантических процедур, вызываемых при обработке входной цепочки. В этом случае активная цепочка определяет последовательность вызовов семантических процедур и моменты времени, в которые выполняются эти вызовы по отношению к моментам чтения входных символов.

11.4. Атрибутные транслирующие грамматики

При определении транслирующей грамматики в понятие входного символа не включалось представление о том, что он является лексемой и состоит из двух компонентов: типа и значения. На самом деле транслирующая грамматика способна описывать перевод только той части символа, которая задает его тип. Рассмотрим, как можно расширить понятие грамматики цепочечного перевода, чтобы использовать при переводе оба компонента входного символа. Расширенная транслирующая грамматика получила название атрибутной транслирующей грамматики (АТ-грамматики). АТ-грамматики были предложены Дональдом Кнутом [24] и в дальнейшем изучались рядом ученых. Материал данного раздела в основном базируется на результатах, приведенных в [2, 27, 28].

В АТ-грамматике символы снабжаются атрибутами, которые могут принимать значения из некоторого множества допустимых значений и интерпретируются как семантическая информация, связанная с конкретным вхождением символа в правило вывода грамматики. При таком подходе значения атрибутов связываются с вершинами дерева вывода в АТ-грамматике, а правила вычисления значений атрибутов сопоставляются правилам вывода грамматики.

Все атрибуты нетерминальных и операционных символов делятся на *синтезированные* и *унаследованные* атрибуты.

11.4.1. Синтезированные атрибуты

Рассмотрим получение АТ-грамматики из исходной транслирующей грамматики.

Пример 11.6

Пусть задана транслирующая грамматика $G_3{}^T$, допускающая в качестве входа арифметические выражения, построенные из символов множества $\{c, +, *, (,)\}$, где c — лексема, являющаяся целочисленной константой и порождающая на выходе значение этого выражения:

(0) $S \rightarrow E$ {ОТВЕТ}

(1) $E \rightarrow E + T$ (4) $T \rightarrow P$

(2) $E \rightarrow T$ (5) $P \rightarrow c$

(3) $T \rightarrow T * P$ (6) $P \rightarrow (E)$

На рис. 11.4, *а* изображено дерево вывода для входной цепочки $c_5 + c_2 * c_3$ (индексы обозначают значения лексем). Этой входной цепочке должна соответствовать выходная цепочка $\{ОТВЕТ\}_{11}$.

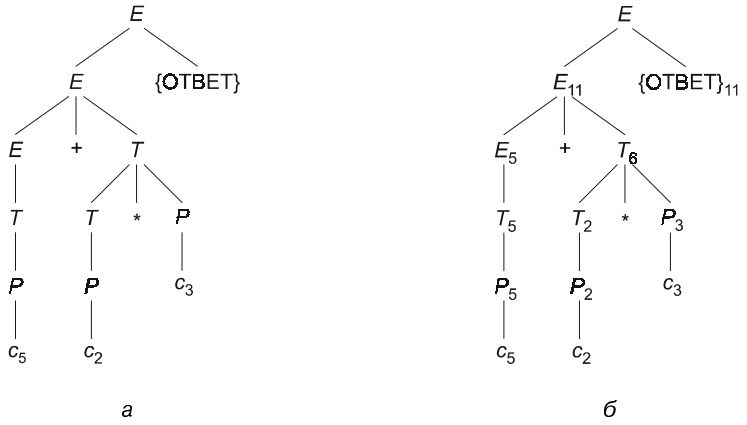


Рис. 11.4. Дерево вывода цепочки $c_5 + c_2 * c_3$ во входной (*а*) и атрибутной транслирующей (*б*) грамматиках

Поскольку любое вхождение нетерминалов E, T и P в дерево вывода представляет собой некоторое подвыражение входного выражения, введем для каждого из символов E, T и P по одному атрибуту, значением которого будет значение подвыражения, порождаемого этим нетерминалом.

Входной символ c и операционный символ {ОТВЕТ} также имеют по одному атрибуту. Значение атрибута символа c равно значению константы, которую он представляет, а значением символа {ОТВЕТ} является результат выражения.

Все атрибуты символов могут принимать любые целочисленные значения из области допустимых для выражений, описываемых соответствующей входной грамматикой G_3. На рис. 11.4, *б* изображено дерево вывода той же цепочки, что и на рис. 11.4, *а*, но на этом дереве нетерминалы и операционный символ {ОТВЕТ} помечены значениями своих атрибутов.

Выберем для каждого атрибута каждого вхождения символа в правило вывода грамматики уникальное имя и включим атрибуты в правила вывода в виде подстрочных индексов соответствующих символов.

Замечание

Одни и те же имена атрибутов можно использовать в нескольких правилах, т. к. они локальны в правиле грамматики.

Сопоставим каждому правилу вывода грамматики правило вычисления значения атрибута нетерминала из левой части правила, которому соответствует нетерминальная вершина дерева вывода.

Рассмотрим, например, вершину T дерева, изображенного на рис. 11.4, *а*. Правило вывода, примененное к этой вершине, имеет вид: $T \rightarrow T * P$. Оно определяет, что значение подвыражения, порожденного нетерминалом из левой части правила, равно значению подвыражений, порожденных символами P и T из правой части правила, которые являются прямыми потомками вершины T. Если p — атрибут символа T из левой части правила, а q и r — атрибуты символов T и P из правой части правила, то правило вычисления значения атрибута p будет иметь вид: $p \leftarrow q * r$, где символ '←' — знак операции присваивания.

Таким образом, начиная с атрибутов входных символов и поднимаясь по дереву от кроны к корню, можно определить все значения атрибутов нетерминалов из левых частей правил вывода. Правила вычисления значений атрибутов для АТ-грамматики $G_3{}^A$, полученной по транслирующей грамматике $G_3{}^T$, будут выглядеть следующим образом:

(0) $S \rightarrow E_q \{\text{ОТВЕТ}\}_p$ $\quad q \leftarrow p$

(1) $E_p \rightarrow E_q + T_r$ $\quad p \leftarrow q + r$

(2) $E_p \rightarrow T_q$ $\quad p \leftarrow q$

(3) $T_p \rightarrow T_q * P_r$ $\quad p \leftarrow q * r$

(4) $T_p \rightarrow P_q$ $\quad p \leftarrow q$

(5) $P_p \rightarrow (E_q)$ $\quad p \leftarrow q$

(6) $P_p \rightarrow c_q$ $\quad p \leftarrow q$

Атрибуты, значения которых вычисляются при движении по дереву вывода снизу вверх, традиционно называют *синтезированными*. Термин "синтезированный" подчеркивает, что значение атрибута синтезируется из значений атрибутов потомков.

Рассмотренные ранее правила вычисления значений атрибутов не распространяются на атрибут операционного символа {ОТВЕТ}, который относится к классу унаследованных атрибутов.

11.4.2. Унаследованные атрибуты

Термин "унаследованный" означает то, что значение атрибута зависит от значений атрибутов предка символа или атрибутов его соседей в дереве вывода. Например, в грамматике $G_3{}^T$, значение атрибута *r* символа {ОТВЕТ} равно значению атрибута нетерминала *E* (соседа слева), порождающего все выражение. Оно может быть вычислено только после того, как будут определены значения всех синтезированных атрибутов нетерминалов.

Рассмотрим еще один пример АТ-грамматики.

Пример 11.7

🗁 Пусть входная КС-грамматика *G*, порождающая описания переменных в некотором языке программирования, выглядит следующим образом:

(1) $D \rightarrow t\ i\ L$

(2) $L \rightarrow ,\ i\ L$

(3) $L \rightarrow \varepsilon$

Множество входных символов этой грамматики состоит из лексем: запятая, идентификатор *i* и имя типа *t*.

Семантика обработки описания заключается в занесении типа переменной в определенное поле элемента таблицы идентификаторов. Эту операцию можно выполнить с помощью семантической процедуры УСТАНОВИТЬ_ТИП с двумя параметрами: указатель на элемент таблицы идентификаторов, соответствующий описываемой переменной, и тип переменной. Процедуру УСТАНОВИТЬ_ТИП лучше всего вызывать сразу после распознавания идентификатора. Указанная последовательность действий может быть описана транслирующей грамматикой G^T:

(1) $D \rightarrow t\ i\ \{\text{ТИП}\}\ L$

(2) $L \rightarrow ,\ i\ \{\text{ТИП}\}\ L$

(3) $L \rightarrow \varepsilon$

В этой грамматике вызову процедуры УСТАНОВИТЬ_ТИП соответствует операционный символ {ТИП}.

Введем в транслирующую грамматику G^T атрибуты и правила их вычисления. Входные символы t и i имеют по одному атрибуту. Значением атрибута символа i является указатель на элемент таблицы идентификаторов, а атрибут символа t может принимать значения из множества {ЦЕЛЫЙ, ВЕЩЕСТВЕННЫЙ, ЛОГИЧЕСКИЙ}.

Операционный символ {ТИП} должен иметь два унаследованных атрибута, значения которых совпадают со значениями соответствующих фактических параметров процедуры УСТАНОВИТЬ_ТИП. Значения унаследованных атрибутов символа {ТИП} для первого правила вывода приравниваются значениям соответствующих атрибутов входных символов i и t, входящих в правую часть того же правила левее символа {ТИП}.

Во второе правило тип описываемых переменных можно передать через унаследованный атрибут нетерминала L. Значение этого унаследованного атрибута будет передаваться по дереву сверху вниз, начиная с вершины, где он получает начальное значение, равное значению атрибута входного символа t.

Атрибутная транслирующая грамматика G^A выглядит следующим образом:

(1) $D \rightarrow t_r\ i_a\ \{\text{ТИП}\}_{a_1,\ r_1}\ L_{r_2}$

$a_1 \leftarrow a$

$r_1,\ r_2 \leftarrow r$

(2) $L_r \rightarrow ,\ i_a\ \{\text{ТИП}\}_{a_1,\ r_1}\ L_{r_2}$

$a_1 \leftarrow a$

$r_1,\ r_2 \leftarrow r$

(3) $L_r \rightarrow \varepsilon$

Замечание

Запись атрибутного правила в виде $r_1, r_2 \leftarrow r$ означает, что значение r присваивается одновременно атрибутам r_1 и r_2.

Входной цепочке $t_{\text{ЦЕЛЫЙ}}\ i_5,\ i_9,\ i_2$ соответствует дерево вывода в грамматике G^A, изображенное на рис. 11.5. □

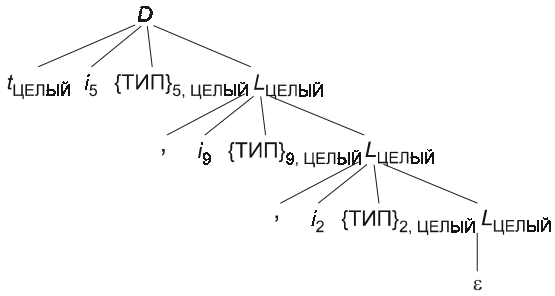


Рис. 11.5. Дерево вывода

В рассмотренных примерах все операционные символы имели унаследованные атрибуты. При определении атрибутной транслирующей грамматики можно обойтись без синтезированных атрибутов операционных символов. Необходимость в синтезированных атрибутах операционных символов может возникнуть в некоторых практических реализациях переводов, поэтому в определение АТ-грамматики включены оба типа атрибутов.

11.4.3. Определение атрибутной трансл ирующей грамматики

Атрибутная транслирующая грамматика — это транслирующая грамматика, обладающая следующими дополнительными свойствами:

1. Каждый символ грамматики (входной, нетерминальный и операционный) имеет конечное множество атрибутов, и каждый атрибут имеет (возможно, бесконечное) множество допустимых значений.
2. Все атрибуты нетерминальных и операционных символов делятся на синтезированные и унаследованные.
3. Унаследованные атрибуты подчиняются следующим правилам:
   - каждому вхождению унаследованного атрибута в правую часть правила вывода сопоставляется правило, позволяющее вычислить значение этого атрибута как функцию некоторых других атрибутов символов, входящих в левую или правую часть данного правила;
   - для каждого унаследованного атрибута начального символа грамматики задается начальное значение.
4. Правила вычисления значений синтезированных атрибутов определяются следующим образом:
   - каждому вхождению синтезированного атрибута нетерминального символа в левую часть правила вывода сопоставляется правило, позволяющее вычислить значение этого атрибута как функцию некоторых других атрибутов символов, входящих в левую или правую часть данного правила;
   - каждому синтезированному атрибуту операционного символа сопоставляется правило, позволяющее вычислить значение этого атрибута как функцию некоторых других атрибутов данного символа.

Будем записывать атрибуты в виде индексов соответствующих символов АТ-грамматики, при этом для каждого атрибута будем указывать, к какому классу этот атрибут относится. Например:

$X_{a,\,b}$ синтезированный a унаследованный b

Правила вычисления атрибутов будем записывать в виде операторов присваивания, левая часть которых — атрибут или список атрибутов, а правая часть — функция.

Рассмотрим АТ-грамматику G^A, описывающую перевод оператора присваивания некоторого языка программирования в цепочку тетрад с кодами операций: СЛОЖИТЬ, УМНОЖИТЬ, ПРИСВОИТЬ. Левой частью оператора присваивания является идентификатор, а правой частью — бесскобочное арифметическое выражение, выполняемое слева направо в порядке написания операций.

Входными символами грамматики являются символы множества $\{i, +, *, =\}$, где i — лексема, представляющая идентификатор. Каждый идентификатор имеет один атрибут, значением которого является указатель на соответствующий элемент таблицы идентификаторов.

Операционные символы {+}, {*}, {:=} соответствуют кодам операций тетрад, получаемых на выходе. Символы {+} и {*} имеют по три унаследованных атрибута, значениями которых являются указатели на элементы таблицы, в которой хранятся значения левого операнда, правого операнда и результата соответственно. Операционный символ {:=} имеет два унаследованных атрибута. Значение первого атрибута — указатель на элемент таблицы идентификаторов, соответствующий идентификатору из левой части оператора присваивания, а значением второго атрибута является указатель на элемент таблицы, в котором хранится значение выражения из правой части оператора присваивания.

Словарь нетерминальных символов состоит из символов S, E и R. Начальный символ S не имеет атрибутов, символ E имеет один синтезированный атрибут, значением которого является указатель на элемент таблицы, содержащий значение выражения, порождаемого E, а символ R имеет два атрибута: унаследованный (первый) и синтезированный. Значением унаследованного атрибута является указатель на элемент таблицы, соответствующий промежуточному результату, предшествующему R. Этот промежуточный результат является также левым операндом первой операции, порождаемой нетерминальным символом R, если такая операция имеет место. Значение синтезированного атрибута символа R — это указатель на элемент таблицы, соответствующий значению подвыражения, которое получается после присоединения цепочки, порожденной символом R, к цепочке, представляющей левый операнд.

Правила вывода АТ-грамматики G^A с соответствующими правилами вычисления следующих атрибутов:

E_t синтезированный t

$R_{p,t}$ унаследованный p синтезированный t

Атрибуты операционных символов унаследованные.

(1) $S \rightarrow i_{p_1} := E_{q_1} \{:=\}_{p_2, q_2}$

$p_2 \leftarrow p_1$

$q_2 \leftarrow q_1$

(2) $E_{t_2} \rightarrow i_{p_1} R_{p_2, t_1}$

$p_2 \leftarrow p_1$

$t_2 \leftarrow t_1$

(3) $R_{p_1, t_2} \rightarrow + i_{q_1} \{+\}_{p_2, q_2, r_1} R_{r_2, t_1}$

$r_1, r_2 \leftarrow$ GETNEW

$p_2 \leftarrow p_1$

$q_2 \leftarrow q_1$

$t_2 \leftarrow t_1$

(4) $R_{p_1, t_2} \rightarrow * i_{q_1} \{*\}_{p_2, q_2, r_1} R_{r_2, t_1}$

$r_1, r_2 \leftarrow$ GETNEW

$p_2 \leftarrow p_1$

$q_2 \leftarrow q_1$

$t_2 \leftarrow t_1$

(5) $R_{p_1, t_2} \rightarrow \varepsilon$

$p_2 \leftarrow p_1$

В атрибутных правилах, связанных с правилами вывода (3) и (4), используется процедура-функция без параметров GETNEW, которая выдает значение указателя на свободную позицию таблицы, где будут запоминаться промежуточные результаты. Строго говоря, эта процедура не является функцией, т. к. при разных обращениях к ней получаются разные результаты. Таким образом, пользуясь процедурой GETNEW, мы несколько отступаем от определения АТ-грамматики. Вместо того чтобы использовать процедуру GETNEW, можно ввести дополнительные атрибуты для запоминания адресов занятых позиций таблицы. Однако при использовании процедуры GETNEW атрибутные правила получаются более простыми, и такой подход можно порекомендовать при практическом конструировании компиляторов.

АТ-грамматики используются для получения *атрибутных деревьев вывода*, *атрибутных активных цепочек* и *атрибутных переводов*.

Процедура построения атрибутного дерева вывода включает следующие действия:

1. По соответствующей транслирующей грамматике построить дерево вывода активной цепочки, состоящей из входных и операционных символов без атрибутов.
2. Присвоить начальные значения унаследованным атрибутам начального символа грамматики.
3. Присвоить значения атрибутам входных символов, входящих в дерево вывода.
4. Найти атрибут, отсутствующий в дереве вывода, аргументы правила вычисления которого уже известны. Вычислить значение этого атрибута и добавить его в дерево.
5. Повторять шаг 4 до тех пор, пока значения всех атрибутов символов, входящих в дерево вывода, не будут вычислены или пока шаг 4 может выполняться.

На рис. 11.6 приведено атрибутное дерево вывода в АТ-грамматике, соответствующее входной цепочке $i_5 = i_7 + i_2 * i_3$ при условии, что процедура GETNEW выдает последовательные адреса ячеек, начиная с адреса 100. Заметим, что при применении правила (5) синтезированному атрибуту нетерминала R присваивается значение его унаследованного атрибута. Затем это значение передается вверх по дереву вывода как синтезированный атрибут нетерминала из левой части правил (4), (3) и (2) и используется в качестве второго атрибута операционного символа {ПРИСВОИТЬ}.

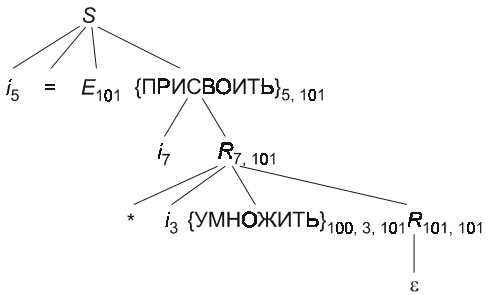


Рис. 11.6. Атрибутное дерево вывода

Последовательность атрибутных входных и операционных символов, полученная по атрибутному дереву вывода в АТ-грамматике, называется *атрибутной активной цепочкой.*

Множество пар, первым элементом которых является атрибутная входная цепочка, а вторым элементом — атрибутная последовательность операционных символов атрибутной активной цепочки, называется *атрибутным переводом* $\tau(G^A)$, определяемым АТ-грамматикой.

Пример 11.8

🗁 Грамматика G^A для входной цепочки $i_5 = i_7 + i_2 * i_3$ определяет следующую атрибутную последовательность операционных символов:

$$\{+\}_{7,\ 2,\ 100}\ \{*\}_{100,\ 3,\ 101}\ \{:=\}_{5,\ 101}.$$

🗀

Дерево вывода в АТ-грамматике называется *завершенным*, если в результате применения процедуры построения дерева значения всех атрибутов всех символов окажутся вычисленными.

Хотя для каждого атрибута, входящего в дерево вывода, существует правило его вычисления, может оказаться, что между атрибутами возникла *круговая зависимость*, в результате чего нельзя получить завершенное дерево.

АТ-грамматика называется *корректной* тогда и только тогда, когда в результате применения описанной процедуры построения атрибутного дерева вывода получается завершенное дерево.

Далее будут рассмотрены два подкласса корректных АТ-грамматик, которые часто используются при проектировании компиляторов: L-атрибутные транслирующие грамматики и S-атрибутные транслирующие грамматики.

11.4.4. Вычисление значений атрибутов

Вернемся еще раз к проблеме вычисления значений атрибутов. После построения дерева вывода входной цепочки возникает вопрос о *порядке* вычисления значений атрибутов. По определению, атрибут можно вычислить, если известны значения всех атрибутов, от которых зависит его значение. Число деревьев вывода так же, как и входных цепочек, бесконечно, поэтому важно по самой грамматике уметь определять, является ли множество ее правил вычисления атрибутов *корректным*.

Алгоритмы проверки корректности АТ-грамматики приведены в [2, 25]. Они основаны на построении *графа зависимостей* и его анализе.

Узлами графа зависимостей служат атрибуты, которые нужно вычислить, а дугам ставятся в соответствие зависимости, определяющие, какие атрибуты вычисляются раньше, а какие позже.

Граф зависимостей $R(D)$ представляет собой ориентированный граф, который строится для некоторого дерева вывода D следующим образом:

- ❒ узлами $R(D)$ являются пары (X, a), где X — узел дерева D, а a — атрибут символа, служащего меткой узла X;

- дуга из узла (X_1, a_1) в узел (X_2, a_2) проводится, если семантическое правило, вычисляющее значение атрибута a_1, непосредственно использует значение атрибута a_2.

Вычисление значения двоичного числа по его символьному представлению можно задать при помощи следующей атрибутной грамматики [24]:

$N_{v_4} \rightarrow L_{v_2, l_2, s_2} . L_{v_3, l_3, s_3}$
$v_4 \leftarrow v_2 + v_3$
$s_2 \leftarrow 0$
$s_3 \leftarrow -l_3$

$L_{v_2, l_2, s_2} \rightarrow B_{v_1, s_1}$
$v_2 \leftarrow v_1$
$s_1 \leftarrow s_2$
$l_2 \leftarrow 1$

$N_{v_4} \rightarrow L_{v_2, l_2, s_2}$
$v_4 \leftarrow v_2$
$s_2 \leftarrow 0$

$B_{v_1, s_1} \rightarrow 0$
$v_1 \leftarrow 0$

$L_{v_2, l_2, s_2} \rightarrow L_{v_3, l_3, s_3} B_{v_1, s_1}$
$v_2 \leftarrow v_3 + v_1$
$s_1 \leftarrow s_2$
$s_3 \leftarrow s_2 + 1$
$l_2 \leftarrow l_3 + 1$

$B_{v_1, s_1} \rightarrow 1$
$v_1 \leftarrow 1$

В этой грамматике атрибуты имеют следующую семантику: v — значение (рациональное число), s — масштаб (целое число) и l — длина числа (целое число).

Используя грамматику, построим дерево D вывода цепочки 101.01 (рис. 11.7).

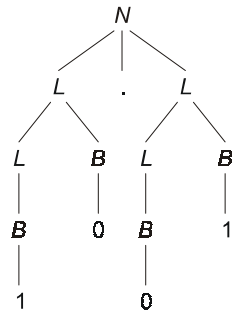


Рис. 11.7. Дерево вывода цепочки 101.01

Для построения узлов графа зависимостей необходимо последовательно рассмотреть вершины дерева D и для каждой из них построить столько узлов, сколько атрибутов имеет символ, которым помечена эта вершина.

Например, для корня дерева, обозначенного символом N с одним атрибутом, нужно в граф включить один узел, пометив его парой (N, v_4).

Для определения дуг графа необходимо использовать семантические правила. Например, рассмотрим узел дерева зависимостей (N, v_4). При построении дерева D непосредственные потомки корня N определялись правилом грамматики $N_{v_4} \rightarrow L_{v_2, l_2, s_2} \cdot L_{v_3, l_3, s_3}$. Атрибут v_4 нетерминала N зависит, в соответствии с семантическим правилом $v_4 \leftarrow v_2 + v_3$, от атрибута v_2 левого сына, помеченного символом L, и атрибута v_3 правого сына. Поэтому в граф зависимостей необходимо включить дуги $((L, v_2), (N, v_4))$ и $((L, v_3), (N, v_4))$. Поступая аналогичным образом, мы можем построить граф зависимостей (рис. 11.8).

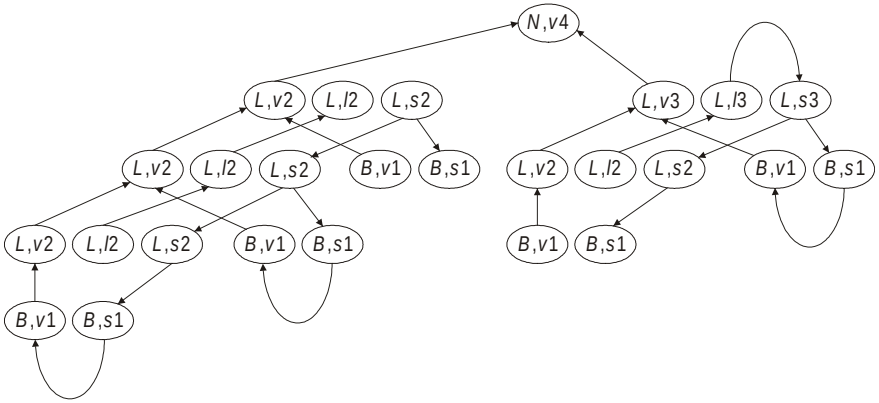


Рис. 11.8. Граф зависимостей

В [24] доказано, что атрибутная грамматика корректна, если граф зависимостей не имеет циклов.

Общие алгоритмы вычисления атрибутов весьма неэффективны, поэтому на практике используют методы, в которых для определения порядка вычисления не рассматриваются семантические правила. Например, если порядок вычисления атрибутов задается методом синтаксического анализа, то он не требует анализа семантических правил и построения графа зависимости.

11.5. Методика разработки описания перевода

При проектировании перевода следует руководствоваться следующими общими соображениями:

1. Описание перевода строится последовательно для конструкций входного языка в порядке их усложнения, начиная с простейших конструкций, например: выражение, оператор присваивания, оператор цикла.

2. Выходная цепочка (результат перевода) включает в себя объекты (сущности) трех видов:
   - объекты, передаваемые в выходную цепочку из входной цепочки;
   - объекты, *не изменяющиеся в процессе перевода* (терминалы выходного языка);
   - объекты, генерируемые во время перевода.
3. Транслирующая грамматика определяет *порядок применения операционных символов* (элементов перевода), строящих выходную цепочку, атрибуты же используются только для передачи значений символов из одних правил в другие.
4. Операционный символ, включаемый в правило вывода грамматики, может генерировать выходную цепочку до тех пор, пока в нее не должен быть включен *объект*, *перемещаемый из входной цепочки и недоступный в данном правиле грамматики.*

Проектирование описания перевода некоторой (одной!) конструкции входного языка выполняется в несколько этапов.

Неформальное описание перевода. Неформальное описание перевода представляет собой пару цепочек, первая из которых является конструкцией входного языка, а вторая — ее представлением в выходном языке. При разработке описания перевода рекомендуется:

- ❒ строить описание перевода на основе описания синтаксиса и семантики входного языка, учитывая все возможные форматы представления входной конструкции (например, **if-then** и **if-then-else**);
- ❒ конструкции, перевод которых уже описан и которые являются частью данной конструкции, считать *терминальными*;
- ❒ после построения неформального описания перевода выделить в нем символы, передаваемые из входной цепочки, и символы, генерируемые при его построении.

Описание синтаксиса. Описание синтаксиса выполняется в БНФ или расширенной БНФ. После стандартных преобразований БНФ преобразуется в КС-грамматику, описывающую рассматриваемую конструкцию входного языка.

Определение транслирующей грамматики. При определении транслирующей грамматики требуется анализировать последовательность используемых при построении перевода правил грамматики, а также моделировать действия, выполняемые операционными символами грамматики. В связи с этим на данном этапе должны быть уточнены:

- ❒ метод синтаксического анализа, используемый при анализе и переводе данной конструкции, и способ его реализации;

- интерпретация операционных символов (цепочка символов, помещаемых на выходную ленту, или имя подпрограммы, которую необходимо выполнить).

Исходными данными при построении транслирующей грамматики являются:

- КС-грамматика, описывающая синтаксис конструкции;
- последовательность правил грамматики (разбор), используемых процессором при построении перевода;
- объекты, доступные процессору, выполняющему перевод (для процессора с магазинной памятью это текущий символ входной цепочки и, в зависимости от метода синтаксического анализа, весь магазин или его часть).

Для построения транслирующей грамматики необходимо выполнить следующие действия:

1. Выбрать простейшую входную цепочку, соответствующую данной конструкции, и определить ее разбор.
2. В неформальном описании перевода выделить очередной (вначале первый) символ выходной цепочки, который должен быть передан в нее из входной цепочки.
3. На основании доступности (видимости) данных процессору и с учетом разбора, определить правило грамматики, в которое нужно включить элемент перевода (операционный символ), переносящий нужный символ в выходную цепочку. Если действия успешны, то перейти к шагу 5, иначе выполнить следующий шаг 4.
4. Если нельзя выполнить шаг 3, то соответствующий операционный символ включить в наиболее подходящее правило грамматики (в дальнейшем этот символ будет передан в него при помощи атрибутов), определив, если нужно, дополнительную выходную ленту процессора.
5. Возложить на новый (включенный) операционный символ действия по записи в выходную цепочку всех символов, вплоть до следующего символа, который должен быть передан из входной цепочки.
6. Повторить шаги 2—5 для всех символов, передаваемых из входной цепочки.
7. Тестировать транслирующую грамматику на более сложных, например, вложенных входных цепочках.

В простых случаях (простая конструкция входного языка, простое ее представление в выходном языке, удачные метод синтаксического анализа и его реализация, удачное расположение операционных символов) может оказаться, что для описания перевода достаточно только транслирующей грамматики.

Определение атрибутной транслирующей грамматики. При необходимости передачи данных между узлами синтаксического дерева транслирующую грамматику следует расширить до атрибутной транслирующей грамматики.

По определению, передача значений атрибутов в атрибутных транслирующих грамматиках возможна только в следующих направлениях:

- от символа левой части символам правой части правила вывода;
- от символов правой части правила символу левой части этого же правила вывода;
- между символами правой части правила вывода.

Для реализации передачи значения символа необходимо выполнить следующие действия:

1. Определить источник передаваемого значения (символ грамматики) и приписать ему атрибут.
2. Определить приемник значения (символ грамматики) и приписать ему атрибут.
3. Учитывая ограничения в передаче атрибутов, определить маршрут передачи значения, для чего удобно пользоваться синтаксическим деревом грамматики.
4. Описать передачу значения атрибута правилами АТ-грамматики, стараясь, чтобы функции вычисления атрибутов были простейшими (копирующие правила).

Тестирование АТ-грамматики. Тестирование АТ-грамматики заключается в моделировании работы построенного по ней процессора, выполняющего перевод. При большом числе тестов и/или большой их длине эта работа может быть выполнена только при использовании соответствующих средств автоматизации. Для простых языковых конструкций, когда длина теста и их число невелики, а тестирование выполняется отдельно для каждой конструкции, начиная с простейших, работа эта не только необходима, но и реально выполнима.

Тестирование АТ-грамматики позволяет определять два вида ошибок:

- ошибки в структуре выходной цепочки (неверный порядок следования символов в выходной цепочке);
- ошибки в значениях символов выходной цепочки, что связано с ошибками в передаче значений атрибутов.

При тестировании реального описания перевода желательно использовать комплексные тесты, определяющие оба вида ошибок.

11.6. Пример разработки АТ-грамматики

Выполним разработку описания перевода оператора цикла, имеющего следующий формат:

for <параметр> = <начальное значение>

to <конечное значение> **step** <шаг>

<тело цикла>

next <параметр>

Для упрощения введем следующие ограничения:

- <параметр> — идентификатор целой переменной;
- <начальное значение>, <конечное значение>, <шаг> — целые положительные константы;
- <тело цикла> — оператор цикла или терминал (не подлежащий переводу оператор);
- кодирование символов входной и выходной программы не рассматривается.

Оператор цикла, который требуется перевести, имеет вид:

for i = c1 **to** c2 **step** c3 <тело цикла> **next** i

Допустим, что выходной язык не содержит оператора цикла. Тогда результат перевода (рис. 11.9) можно представить следующей последовательностью операторов:

i := c1; m1: **if** i > c2 **then goto** m2; <тело цикла>; i := i + c3; **goto** m1; m2:

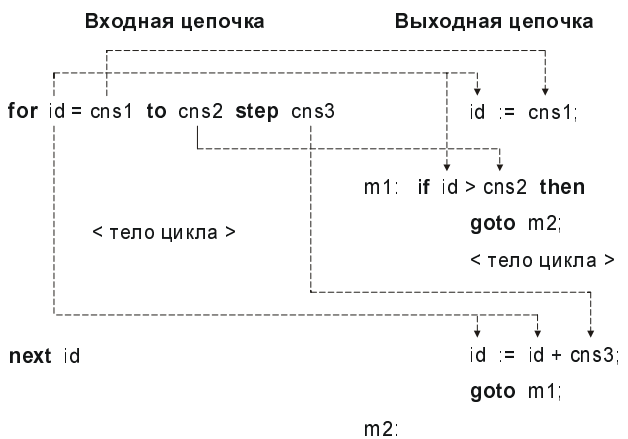


Рис. 11.9. Перевод оператора цикла

Анализ выполняемого перевода позволяет сделать вывод о том, что выходная цепочка включает в себя объекты (сущности) трех видов:

1. Объекты, *передаваемые в выходную цепочку из входной цепочки* (переменная цикла i, начальное значение c1, конечное значение c2, шаг c3).
2. Объекты, не изменяющиеся в процессе перевода (множество терминальных символов выходного языка { ;, :, :=, +, >, **if**, **then**, **goto** }.
3. Объекты, генерируемые во время перевода (метки m1 и m2).

Составим БНФ, которая описывает синтаксис заданного оператора цикла, выбирая такую структуру описания, которая позволяет проиллюстрировать практически все проблемы синтеза АТ-грамматик:

(1) <оператор цикла> ::= <заголовок цикла><тело цикла>< конец цикла>

(2) <заголовок цикла> ::= **for** <начальное значение><конечное значение><шаг>

(3) <начальное значение> ::= <идентификатор> = <константа>

(4) <конечное значение> ::= **to** <константа>

(5) <шаг> ::= **step** <константа>

(6) <тело цикла> ::= <оператор цикла>

(7) <тело цикла> ::= <терминал>

(8) <конец цикла> ::= **next** <идентификатор >

Введем обозначения: *S* — <оператор цикла>, *A* — <заголовок цикла>, *B* — <тело цикла>, *C* — < конец цикла >, *D* — <начальное значение>, *E* — <конечное значение>, *F* — <шаг>. Тогда правила вывода КС-грамматики, описывающей синтаксис входного языка, имеют вид:

(1) $S \rightarrow A\ B\ C$ (5) $F \rightarrow$ **step** cns

(2) $A \rightarrow$ **for** $D\ E\ F$ (6) $B \rightarrow S$

(3) $D \rightarrow$ id = cns (7) $B \rightarrow$ term

(4) $E \rightarrow$ **to** cns (8) $C \rightarrow$ **next** id

При построении транслирующей грамматики требуется сделать некоторые предположения о ходе выполнения перевода и порядке анализа входной цепочки и построения выходной цепочки.

Для определенности будем считать, что:

- используется восходящий метод синтаксического анализа, выполняемый процессором с магазинной памятью;
- выполняется цепочечный перевод, в котором операционный символ вида {OUT := "str"} означает запись в выходную ленту OUT цепочки символов "str".

Определим действия, выполняемые первым по порядку элементом перевода (операционным символом), и правило, в которое он должен быть помещен. Для этого еще раз рассмотрим неформальное описание перевода (см. рис. 11.9). Очевидно, что начать запись в выходную ленту процессор сможет только тогда, когда ему будет доступен параметр цикла id. Когда процессор выполняет свертку, используя правило грамматики с номером (3), то id входит в основу и доступен процессору. Следовательно, в это правило мы можем включить операционный символ, который записывает на выходную ленту начальный участок перевода, вплоть до символа cns. Правило транслирующей грамматики с номером (3) будет выглядеть следующим образом:

(3) *D* → id = cns { OUT:= "id := cns; m1 : **if** id >" }.

При выполнении свертки по этому правилу операционный символ записывает на выходную ленту параметр цикла id. Запись этой переменной в OUT может быть выполнена функцией "Читать третий сверху символ магазина процессора и записать его на выходную ленту". Аналогичные действия ("Читать верхний символ магазина и записать его на выходную ленту") выполняются для начального значения параметра цикла cns. Остальные символы не зависят от входной цепочки (терминалы выходного языка) и просто помещаются на выходную ленту.

Рассуждая подобным образом, мы получим следующие правила транслирующей грамматики, содержащие элементы перевода (в порядке их использования при анализе входной цепочки и построении выходной цепочки):

(4) *E* → **to** cns { OUT:= "cns **then goto** m2;" }

(7) *B* → term { OUT:= "term;" }

(5) *F* → **step** cns { OUT:= "id := id + cns; **goto** m1; m2 : " }

Добавив остальные правила грамматики, получим следующую транслирующую грамматику:

(1) *S* → *A B C*

(2) *A* → **for** D E F

(3) *D* → id = cns { OUT:= "id := cns; m1 : **if** id >" }

(4) *E* → **to** cns { OUT:= "cns **then goto** m2;" }

(5) *F* → **step** cns { OUT:= "id := id + cns; **goto** m1; m2: " }

(6) *B* → S

(7) *B* → term { OUT:= "term;" }

(8) *C* → **next** id

Для тестирования грамматики построим перевод входной цепочки:

```
for id = cns1 to cns2 step cns3 term next id
```

Синтаксическое дерево разбора этой цепочки приведено на рис. 11.10 (слева указаны номера правил грамматики, соответствующие вершинам дерева).

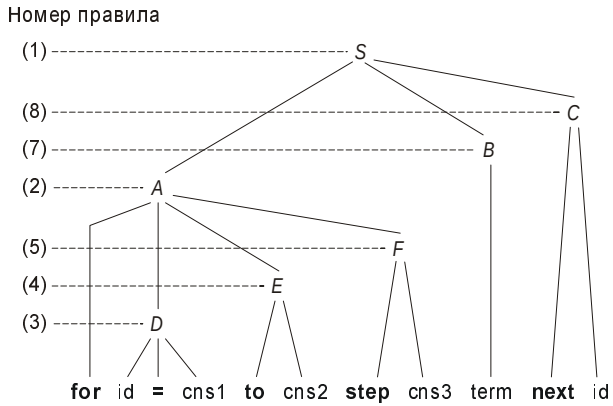


Рис. 11.10. Синтаксическое дерево разбора цепочки

При восходящих методах синтаксического анализа (*см. гл. 8, 9*) строится обращенный правый вывод (разбор) цепочки — последовательность номеров правил, используемых при свертках, — который для рассматриваемого примера равен: ((3), (4), (5), (2), (7), (8), (1)). При этом на выходной ленте формируется цепочка:

id := cns1; m1: **if** id > cns2 **then goto** m2; id := id + cns3; **goto** m1; m2: term;

Анализ полученной цепочки позволяет сделать следующие выводы:

1. Тело цикла (терминальный оператор term) включается неправильно (не перед увеличением параметра цикла на значение шага, а в конец выходной цепочки). Это связано с тем, что правило (5) применяется раньше, чем правило (7).
2. При использовании правила (5) значение параметра цикла id недоступно (id вытолкнут из магазина при свертке по третьему правилу грамматики). Значение id должно быть передано в элемент перевода правила (5) для правильной генерации выходной цепочки.
3. При переводе вложенных циклов возникнет конфликт меток m1 и m2, т. к. m1 и m2 — фиксированные значения. Значения меток должны генерироваться при выполнении перевода и передаваться от места возникновения метки к месту ее использования.

Ошибки, обнаруженные в выходной цепочке, можно легко устранить, расширив транслирующую грамматику до атрибутной транслирующей грамматики. Первую ошибку можно исправить несколькими способами:

- включить в правило (4) атрибут, в который записать координаты выходной цепочки, куда должно быть вставлено тело цикла, передать значение

этого атрибута в правило (5) и использовать его для выполнения включения тела цикла в выходную цепочку;

- связать с нетерминалом F атрибут, значением которого является строка символов. При использовании правила (5) записать в эту строку операторы, реализующие изменение параметра цикла, затем передать значение этого атрибута в правило (1), где и переписать его значение в выходную цепочку;
- пополнить атрибутный процессор, выполняющий перевод, дополнительной лентой, на которую записывать операторы, реализующие изменение переменной цикла в правиле (5). В конце перевода при свертке по правилу (1) сцепить вспомогательную ленту с выходной лентой.

Применение первых двух способов не вызывает принципиальных затруднений, но неэффективно при реализации процессора, поэтому используем последний способ устранения ошибки: на выходную ленту OUT1 будем записывать операторы установки начального значения параметра цикла, проверки завершения цикла и тело цикла, а на дополнительную ленту OUT2 — операторы изменения переменной цикла с заданным шагом.

Вторая и третья ошибки устраняются стандартным способом: включением в правила грамматики атрибутов и функций их вычисления.

Преобразуем транслирующую грамматику в транслирующую атрибутную грамматику.

Рассмотрим правило транслирующей грамматики:

$D \rightarrow$ id = cns { OUT1:= "id := cns; m1 : **if** id >" }

При выполнении свертки по этому правилу значения терминальных символов id и cns требуется передать из входной цепочки в выходную. Это реализуется передачей соответствующих атрибутов в операционный символ и использованием их при построении выходной цепочки:

$D \rightarrow \mathrm{id}_{a1}$ = cns_{a2} { OUT1:= "id_{n1} := cns_{n2}; m1: **if** id >" }$_{n1,\ n2}$

$n1 \leftarrow a1$; $n2 \leftarrow a2$

Запись "id_{n1}" в операционном символе означает, что в данное место выходной цепочки нужно поместить идентификатор, значение которого определяется унаследованным атрибутом $n1$ (значения id и cns — это *не числовые значения* этих объектов, а *ссылки на таблицы*, которые содержат необходимую информацию).

Так как переменная цикла id используется и в других правилах грамматики (правила (5) и (8)), то ее значение нужно передать в эти правила, связав с символом D атрибут и определив функцию для его вычисления. Окончательно правило грамматики примет вид:

(3) $D_{s1} \rightarrow \mathrm{id}_{a1}$ = cns_{a2} { OUT1:= "id_{n1} := cns_{n2}; m1: **if** id >" }$_{n1,\ n2}$

$s1, n1 \leftarrow a1$; $n2 \leftarrow a2$

Для определения способа передачи значения переменной цикла id (атрибут $a1$) в правила (5) и (8) грамматики, рассмотрим фрагмент дерева грамматики, изображенный на рис. 11.11.

Процесс передачи значения лексемы id (атрибут $a1$) из правила (3) грамматики отмечен на рис. 11.11 сплошными линиями:

1. Из правой части правила (3) значение атрибута входного символа $a1$ присваивается синтезированному атрибуту $s1$ символа D из левой части этого же правила.
2. Из правой части правила (2) значение синтезированного атрибута $s1$ нетерминального символа D присваивается унаследованному атрибуту $n1$ нетерминала F из правой части этого же правила.
3. Из левой части правила (5) значение унаследованного атрибута $n1$ нетерминала F передается в правую часть унаследованному атрибуту $n2$ операционного символа.

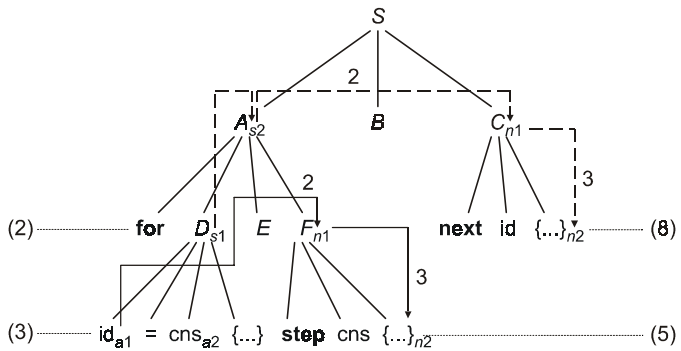


Рис. 11.11. Передача атрибутов символам грамматики

Соответствующие описанному процессу правила АТ-грамматики имеют вид:

(3) $D_{s1} \rightarrow \mathrm{id}_{a1} = \mathrm{cns}_{a2}$ { OUT1 := "id_{n1} := cns_{n2}; m1: **if** id >" }$_{n1,\ n2}$
$s1, n1 \leftarrow a1; \quad n2 \leftarrow a2$

(2) $A \rightarrow$ **for** $D_{s1}\ E\ F_{n1}$
$n1 \leftarrow s1$

(5) $F_{n1} \rightarrow$ **step** cns { OUT2 := " id := id +cns; **goto** m1; m2 : "}$_{n2}$
$n2 \leftarrow n1$

Передача значения параметра цикла в правило (8) грамматики изображено на рис. 11.11 штриховыми линиями и описывается следующими правилами АТ-грамматики:

(2) $A_{s2} \rightarrow$ **for** $D_{s1}\ E\ F_{n1}$
$s2, n1 \leftarrow s1$

(1) $S \rightarrow A_{s2}\ B\ C_{n1}$
$n1 \leftarrow s2$

(8) $C_{n1} \rightarrow$ **next** id { **if** $n2 \neq n3$ **then** Error $\}_{n2}$
$n2 \leftarrow n1$

Особенностью правила (8) построенной АТ-грамматики является то, что его операционный символ должен интерпретироваться не как элемент цепочечного перевода, а как выполнение действий, напрямую не связанных с генерацией выходной цепочки. В этот операционный символ передаются значения параметра цикла из заголовка цикла и оператора его завершения. Если эти значения не равны, то должно возникать состояние ошибки процессора (Error).

Процесс генерации и передачи меток иллюстрируется на рис. 11.12. Метки m1 и m2 генерируются в операционных символах правил (3) и (4) грамматики соответственно с помощью процедуры-функции NewLabel, возвращающей при обращении к ней уникальное значение метки. Правила АТ-грамматики, описывающие процесс генерации меток, имеют вид:

(3) $D_{s1,\,s2} \rightarrow \mathrm{id}_{a1} = \mathrm{cns}_{a2}$
{ OUT1:= "id_{n1} := cns_{n2}; Label_{s3}: **if** id >" $\}_{n1,\,n2,\,n3}$
$s1, n1 \leftarrow a1$; $n2 \leftarrow a2$; $s3 \leftarrow$ NewLabel; $n3 \leftarrow s3$; $s2 \leftarrow n3$;

(4) $E_{s3} \rightarrow$ **to** cns { OUT1 := "cns *then* **goto** Label_{s1};" $\}_{n2}$
$s1 \leftarrow$ NewLabel; $n2 \leftarrow s1$; $s3 \leftarrow n2$;

(2) $A \rightarrow$ **for** $D_{s1,\,s2}\ E_{s3}\ F_{n1,\,n2,\,n3}$
$n1 \leftarrow s1$; $n3 \leftarrow s3$

(5) $F_{n1,\,n2,\,n3} \rightarrow$ **step** cns { OUT2 := "id := id + *cns*; **goto** Label_{n5}; Label_{n4}:" $\}_{n4,\,n5}$
$n4 \leftarrow n2$; $n5 \leftarrow n3$

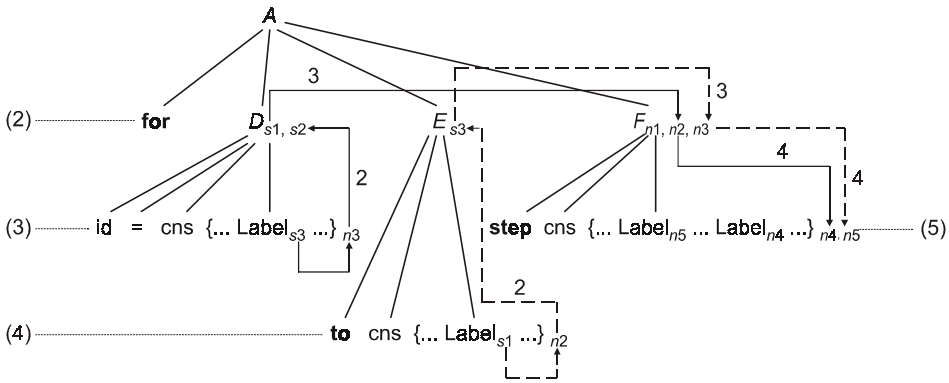


Рис. 11.12. Процесс генерации и передачи меток

Сцепление выходных лент OUT1 и OUT2 должно быть выполнено один раз сразу после анализа и перевода всей входной цепочки. Для реализации сцепления выходных лент грамматику необходимо пополнить новым (нулевым) правилом, включив в него операционный символ, выполняющий действия по конкатенации выходных лент. Результирующая АТ-грамматика будет иметь вид:

(0) $S0 \rightarrow S$ { OUT1 := OUT1 || OUT2 }

(1) $S \rightarrow A_{s1}\ B\ C_{n1}$
$n1 \leftarrow s1$

(2) $A_{s4} \rightarrow$ **for** $D_{s1,\,s2}\ E_{s3}\ F_{n1,\,n2,\,n3}$
$s4,\ n1 \leftarrow s1;\quad n2 \leftarrow s2;\quad n3 \leftarrow s3;$

(3) $D_{s1,\,s2} \rightarrow \mathrm{id}_{a1} = \mathrm{cns}_{a2}$
{ OUT1 := "id_{n1} := cns_{n2}; Label_{s3}: **if** id >" }$_{n1,\,n2,\,n3}$
$s1,\ n1 \leftarrow a1;\quad n2 \leftarrow a2;\quad s3 \leftarrow$ NewLabel; $\quad n3 \leftarrow s3;\quad s2 \leftarrow n3;$

(4) $E_{s3} \rightarrow$ **to** cns_{a1} { OUT1 := "cns_{n1} **then goto** Label_{s1};" }$_{n1,\,n2}$
$n1 \leftarrow a1;\quad s1 \leftarrow$ NewLabel; $\quad n2 \leftarrow s1;\quad s3 \leftarrow n2;$

(5) $F_{n1,\,n2,\,n3} \rightarrow$ **step** cns_{a1}
{ OUT2 := "id_{n6} := id_{n6} + cns_{n7}; **goto** Label_{n5}; Label_{n4}:" }$_{n4,\,n5,\,n6}$
$n4 \leftarrow n2;\quad n5 \leftarrow n3;\quad n6 \leftarrow n1;\quad n7 \leftarrow a1;$

(6) $B \rightarrow S$

(7) $B \rightarrow$ term; { OUT1:= "term"; }

(8) $C_{n1} \rightarrow$ **next** id_{a1} { **if** $n2 \neq n3$ **then** Error }$_{n2,\,n3}$
$n2 \leftarrow n1;\quad n3 \leftarrow a1$

При переводе вложенных операторов цикла использование двух выходных лент в процессоре и их сцепление при выполнении перевода может привести к ошибкам в структуре выходной цепочки. Рассмотрим перевод следующей входной цепочки:

```
for i1 = c11 to c12 step c13
    for i2 = c21 to c22 step c23
        top
    next i2
next i1
```

разбор которой в виде последовательности номеров правил и синтаксического дерева приведен на рис. 11.13.

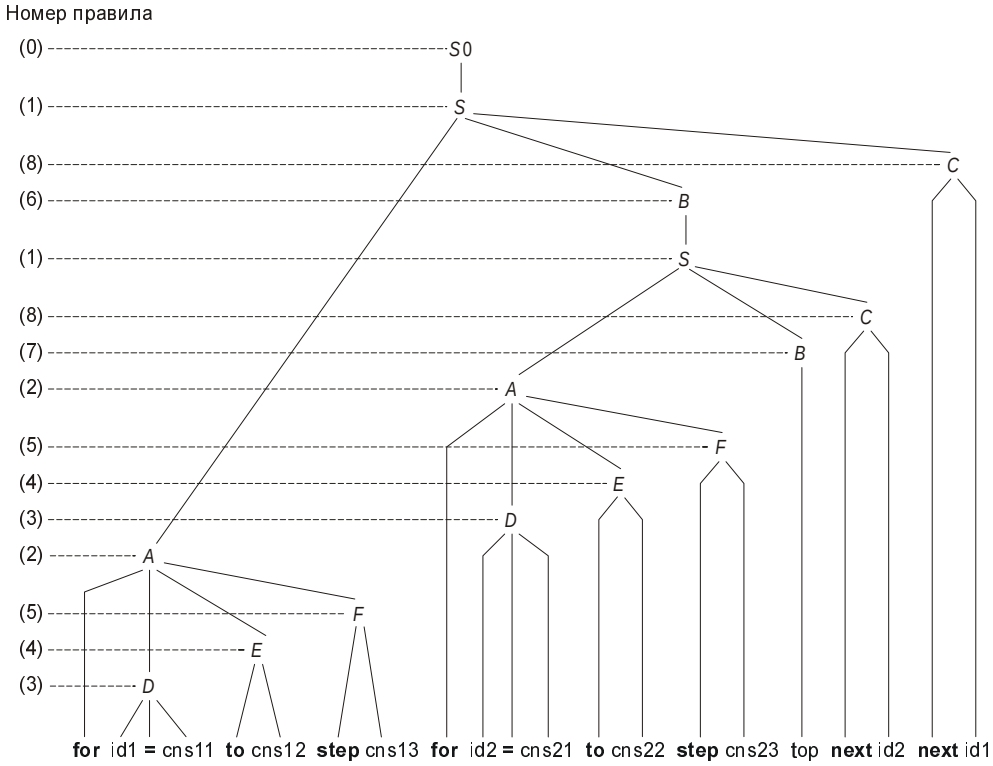


Рис. 11.13. Разбор входной цепочки с вложенными циклами

К моменту использования правила (0) на выходных лентах OUT1 и OUT2 будут находиться следующие цепочки (на данном этапе проверки считаем, что вычисление атрибутов выполняется правильно):

OUT1:

```
        i1 := c11;
Label11: if i1 > c12 then goto Label12;
            i2 := c21;
        Label21: if i1 > c22 then goto Label22;
            top;
```

OUT2:

```
        i1 := i1 + c13; goto Label11; Label12:
        i2 := i2 + c23; goto Label21; Label22:
```

После выполнения сцепления выходная (основная) лента будет содержать перевод:

```
OUT1:
        i1 := c11;
Label11: if i1 > c12 then goto Label12;
              i2 := c21;
        Label21: if i1 > c22 then goto Label22;
              top;
OUT2:
        i1 := i1 + c13; goto Label11; Label12:
        i2 := i2 + c23; goto Label21; Label22:
        i2 := c21;
Label21: if i1 > c22 then goto Label22;
        top;
        i1 := i1 + c13;  goto Label11;
Label12:
        i2 := i2 + c23;
        goto Label21;
Label22:
```

Анализ выходной цепочки показывает, что перевод построен неверно: операторы завершения внешнего и вложенного циклов следуют в обратном порядке. Связано это с тем, что вначале на выходную ленту OUT2 записываются операторы завершения внешнего цикла, и только после этого — вложенного. При формировании же перевода на выходную ленту вначале должны быть записаны операторы завершения вложенного цикла, а затем — внешнего. Для того чтобы исправить эту ошибку, необходимо запись на ленту OUT2 и чтение из нее выполнять с одного конца, т. е. реализовать ее как стек с операциями PushOUT2 ("Втолкнуть на ленту OUT2") и PopOUT2 ("Вытолкнуть с ленты OUT2").

После включения операций PushOUT2 и PopOUT2 в правила (0) и (5) окончательно получим следующий вид атрибутной транслирующей грамматики:

(0) $S0 \rightarrow S$ { **while** (OUT2 not empty) OUT1 := OUT1 ‖ PopOUT2 }

(1) $S \rightarrow A_{s1}\ B\ C_{n1}$
$n1 \leftarrow s1$

(2) $A_{s4} \rightarrow$ **for** $D_{s1,\ s2}\ E_{s3}\ F_{n1,\ n2,\ n3}$
$s4,\ n1 \leftarrow s1;\quad n2 \leftarrow s2;\quad n3 \leftarrow s3;$

(3) $D_{s1,\ s2} \rightarrow \mathrm{id}_{a1} = \mathrm{cns}_{a2}$
{ OUT1 := "id_{n1} := cns_{n2}; Label_{s3}: **if** id >" $\}_{n1,\ n2,\ n3}$
$s1,\ n1 \leftarrow a1;\quad n2 \leftarrow a2;\quad s3 \leftarrow$ NewLabel; $\quad n3 \leftarrow s3;\quad s2 \leftarrow n3;$

(4) $E_{s3} \rightarrow$ **to** cns_{a1} { OUT1 := "cns_{n1} **then goto** Label_{s1};" $\}_{n1,\ n2}$
$n1 \leftarrow a1;\quad s1 \leftarrow$ NewLabel; $\quad n2 \leftarrow s1;\quad s3 \leftarrow n2;$

(5) $F_{n1,\ n2,\ n3} \rightarrow$ **step** cns_{a1}
{ PushOUT2("id := "id_{n6} := id_{n6} + cns_{n7}; **goto** Label_{n5}; Label_{n4}:")$\}_{n4,\ n5,\ n6}$

(6) $B \rightarrow S$

(7) $B \rightarrow$ term; { OUT1 := "term; }

(8) $C_{n1} \rightarrow$ **next** id_{a1} { **if** $n2 \neq n3$ **then** Error $\}_{n2,\ n3}$
$n2 \leftarrow n1;\quad n3 \leftarrow a1$

Читателям рекомендуется самостоятельно проверить правильность окончательного варианта построения АТ-грамматики для различных тестовых входных цепочек.

Контрольные вопросы

1. Как определяется понятие семантики языков программирования?
2. Как можно описать простейшие переводы?
3. Какие переводы называются синтаксически управляемыми?
4. Что такое элемент перевода?
5. Дайте определение СУ-схемы. Какие цепочки называются входными и выходными?
6. Как определить входную и выходную грамматики СУ-схемы?
7. Как происходит порождение пар цепочек под управлением СУ-схемы?
8. Как происходит преобразование деревьев под управлением СУ-схем?
9. Какие СУ-схемы называются простыми?
10. Какие переводы позволяют описать простые СУ-схемы?
11. Какая грамматика называется транслирующей грамматикой?
12. Что такое входная и выходная грамматики транслирующей грамматики?
13. Как определяется активная цепочка, ее входная и операционная части?
14. Как связаны вывод цепочки и дерево вывода активной цепочки?
15. Какая грамматика называется атрибутной транслирующей грамматикой?
16. Какие атрибуты называются унаследованными и синтезированными?

17. В чем основные различия между унаследованными и синтезированными атрибутами?
18. Что представляет собой правило вычисления атрибутов?
19. Как строится граф зависимостей?
20. Как граф зависимостей используется при вычислении атрибутов?
21. Какие виды объектов включаются в выходную цепочку при переводе?
22. Из каких этапов состоит процесс построения транслирующей грамматики?
23. Из каких этапов состоит процесс построения АТ-грамматики?
24. Какие типы ошибок позволяет найти тестирование АТ-грамматики?

Упражнения

1. Разработайте простую СУ-схему, описывающую перевод арифметических скобочных выражений, содержащих операции '+' и '*' в:
   1.1. постфиксную запись.
   1.2. префиксную запись.
2. В языке ALGOL-60 выражения строятся с помощью операций, имеющих следующие приоритеты (в порядке убывания):

   (1) $\uparrow$ (4) $\leq < = \neq > \geq$ (7) $\vee$

   (2) $*/ \div$ (5) $\neg$ (8) $\rightarrow$

   (3) $+ -$ (6) $\wedge$ (9) $\equiv$

   Разработайте простую СУ-схему, описывающую перевод инфиксных выражений, содержащих перечисленные операции, в постфиксную запись.
3. Постройте МП-преобразователи, реализующие СУ-схемы из упр. 1.
4. Для простой СУ-схемы:

   $E \rightarrow + EE,\ EE+$

   $E \rightarrow * EE,\ EE*$

   $E \rightarrow a,\ a$

   постройте эквивалентный МП-преобразователь.
5. Для заданной СУ-схемы:

   $S \rightarrow a\ S^{(1)}\ b\ A\ c\ S^{(2)}\ d\ S^{(3)},\ e\ A\ c\ S^{(2)}\ d\ S^{(3)}\ f\ S^{(1)}\ g$

   $S \rightarrow i,\ i$

   $A \rightarrow i,\ i$

   постройте переводы для входных цепочек:
   5.1. *aibicdi.*
   5.2. *aaibicidibicidi.*

6. Постройте вывод в транслирующей грамматике $G_0{}^T$ цепочек:
   6.1. $i * (i + i)$.
   6.2. $(i + i) * (i + i)$.
7. Определите транслирующую грамматику, допускающую в качестве входа произвольную цепочку из нулей и единиц и порождающую на выходе:
   7.1. обращение входной цепочки.
   7.2. цепочку 0^n1^m, где n — число нулей, а m — число единиц во входной цепочке.
8. Постройте транслирующую грамматику, определяющую перевод логических выражений, составленных из логических переменных, скобок и знаков операций дизъюнкции, конъюнкции и отрицания:
   8.1. из *инфиксной* записи в ПОЛИЗ.
   8.2. из ПОЛИЗ в инфиксную запись.
   8.3. из инфиксной записи в функциональную запись такую, например, что выражение $a \cap (b \cup c)$ будет представлено в виде $f_\cap (a, f_\cup (b, c))$, где $f_\cap$ и $f_\cup$ — отдельные символы.
9. Постройте АТ-грамматику, описывающую перевод оператора присваивания некоторого гипотетического языка программирования в цепочку тетрад с кодами операций: ПРИСВОИТЬ, СЛОЖИТЬ, ВЫЧЕСТЬ, УМНОЖИТЬ, ДЕЛИТЬ. Левой частью оператора присваивания является идентификатор, а правой частью — бесскобочное арифметическое выражение, выполняемое справа налево в порядке написания операций. В арифметическом выражении можно использовать идентификаторы и знаки арифметических операций: +, −, *, /.
10. Постройте АТ-грамматику, описывающую перевод оператора присваивания некоторого гипотетического языка программирования в цепочку тетрад с кодами операций: ПРИСВОИТЬ, И, ИЛИ. НЕ. Левой частью оператора присваивания является идентификатор, а правая часть представляет собой логическое выражение, составленное из идентификаторов, скобок и знаков логических операций: $\cap$, $\cup$, $\neg$. Порядок выполнения логического выражения определяется обычным приоритетом логических операций.
11. Постройте АТ-грамматику, описывающую перевод в тетраду с кодом операции ПЕРЕХОД_ПО_РАВНО условного оператора, которому соответствует следующее правило входной грамматики:

    условный оператор → **if** $i = i$ **goto** c.

    Лексема c представляет собой номер оператора, к которому осуществляется переход в случае равенства, а i — идентификатор.

12. Для входной грамматики, описывающей арифметические выражения, с правилами вывода:

$S \rightarrow E$ $\qquad$ $E \rightarrow (E)$

$E \rightarrow E := E$ $\qquad$ $E \rightarrow i$

$E \rightarrow E + E$

где i — лексема, значением которой является указатель на элемент таблицы идентификаторов, а семантика выражений такая же, как в языке С, постройте АТ-грамматику, которая проверяет, представляет ли левая часть выражения l-значение, и, в случае успеха, стройте ПОЛИЗ выражения.

13. Для входной грамматики, описывающей объявление переменных, с правилами вывода:

(1) $D \rightarrow i\ L$ $\qquad$ (4) $T \rightarrow$ int

(2) $L \rightarrow , i\ L$ $\qquad$ (5) $T \rightarrow$ real

(3) $L \rightarrow : T$

постройте АТ-грамматику, которая заносит значение типа каждого идентификатора в таблицу идентификаторов.

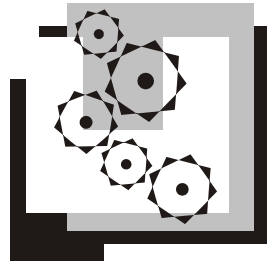
14. Для входной грамматики, описывающей арифметические выражения, с правилами вывода:

$E \rightarrow (E + E)$

$E \rightarrow (E * E)$

$E \rightarrow i$

где i — лексема, значением которой является указатель на элемент таблицы идентификаторов, определить постфиксную S-атрибутную грамматику, описывающую перевод этих выражений в цепочку тетрад с кодами операций: СЛОЖИТЬ и УМНОЖИТЬ, и построить S-атрибутный ДМП-процессор, выполняющий заданный перевод.



Глава 12

Разработка и реализация синтаксически управляемого перевода

Рассмотрим два подкласса корректных АТ-грамматик, которые часто используются при проектировании языковых процессоров: *L*-атрибутные транслирующие грамматики и *S*-атрибутные транслирующие грамматики.

12.1. *L*-атрибутные и *S*-атрибутные транслирующие грамматики

АТ-грамматика называется *L-атрибутной* тогда и только тогда, когда выполняются следующие условия:

1. Аргументами правила вычисления значения унаследованного атрибута символа из правой части правила вывода могут быть только унаследованные атрибуты символа из левой части и произвольные атрибуты символов из правой части, расположенные левее рассматриваемого символа.
2. Аргументами правила вычисления значения синтезированного атрибута символа из левой части правила вывода являются унаследованные атрибуты этого символа или произвольные атрибуты символов из правой части.
3. Аргументами правила вычисления значения синтезированного атрибута операционного символа могут быть только унаследованные атрибуты этого символа.

Является ли произвольная АТ-грамматика *L*-атрибутной, можно проверить, независимо исследуя каждое правило вывода и каждое правило вычисления значения атрибута. Примером *L*-атрибутной транслирующей грамматики является грамматика G^A, построенная в *разд. 11.4.3*.

При выполнении условия 1 определения унаследованные атрибуты каждой вершины дерева вывода зависят (непосредственно или косвенно) только от атрибутов входных символов, расположенных в дереве левее данной вершины,

что позволяет использовать L-атрибутные грамматики в нисходящих синтаксических анализаторах. Условия 2 и 3 введены с целью сделать АТ-грамматику корректной.

В L-атрибутной транслирующей грамматике атрибуты символов A, B и C из правила вывода $A \rightarrow BC$ можно вычислять в следующем порядке:

- унаследованные атрибуты символа A;
- унаследованные атрибуты символа B;
- синтезированные атрибуты символа B;
- унаследованные атрибуты символа C;
- синтезированные атрибуты символа C;
- синтезированные атрибуты символа A.

АТ-грамматика называется *S-атрибутной* тогда и только тогда, когда она является L-атрибутной и все атрибуты нетерминалов синтезированные.

Ограничения, накладываемые на L-атрибутную транслирующую грамматику, позволяют вычислять значения атрибутов в процессе нисходящего анализа входной цепочки. Нисходящий детерминированный анализатор для $LL(1)$-грамматик требует, чтобы L-атрибутная транслирующая грамматика, описывающая перевод, имела *форму простого присваивания*.

12.2. Форма простого присваивания

При определении АТ-грамматики в *разд. 11.4* правила вычисления атрибутов записывались в виде операторов присваивания, левые части которых представляют собой атрибут или список атрибутов, а правые части — функцию, использующую в качестве аргументов значения некоторых атрибутов. Простейший случай функции из правой части оператора присваивания — *тождественная или константная функция*, например $a \leftarrow b$ или $x, y \leftarrow 1$.

Правило вычисления атрибутов называется *копирующим правилом* тогда и только тогда, когда левая часть правила — это атрибут или список атрибутов, а правая часть — константа или атрибут. Правая часть называется *источником копирующего правила*, а каждый атрибут из левой части — *приемником копирующего правила*.

Если источники нескольких копирующих правил совпадают, то их приемники можно объединить в одну левую часть. Например, правила $z, w \leftarrow a$ и $x, y \leftarrow z$ можно записать в виде: $x, y, z, w \leftarrow a$, т. к. источнику второго

правила z, согласно первому правилу, присваивается значение a. Аналогично $x \leftarrow y$ и $a, b \leftarrow y$ можно записать как $a, b, x \leftarrow y$.

Множество копирующих правил называется *независимым*, если источник каждого правила из этого множества не входит ни в одно из других правил множества.

Если два копирующих правила независимы, их нельзя объединять.

Атрибутная транслирующая грамматика имеет форму простого присваивания тогда и только тогда, когда:

1. Некопирующими являются только правила вычисления синтезированных атрибутов операционных символов.
2. Для каждого правила вывода грамматики соответствующее множество копирующих правил независимо.

Примером АТ-грамматики в форме простого присваивания является грамматика G^A, приведенная в *разд. 11.4.3*.

Рассмотрим процедуру преобразования произвольной L-атрибутной транслирующей грамматики в эквивалентную L-атрибутную грамматику в форме простого присваивания. Смысл используемого здесь понятия эквивалентности объясняется далее.

1. Для каждой функции $f(x_1, x_2, \ldots, x_n)$, входящей в правило вычисления атрибутов, связанное с некоторым правилом вывода грамматики, создать соответствующий ей операционный символ $\{f\}$, который определяется следующим образом:

   $\{f\}_{x_1, x_2, \ldots, x_n, p}$ УНАСЛЕДОВАННЫЕ $x_1, x_2, \ldots, x_n$

   СИНТЕЗИРОВАННЫЙ p

   $p \leftarrow f(x_1, x_2, \ldots, x_n)$

2. Для каждого некопирующего правила $z_1, z_2, \ldots, z_m \leftarrow f(y_1, y_2, \ldots, y_n)$, связанного с некоторым правилом вывода грамматики, найти символы $a_1, \ldots, a_n$, которые не содержатся в этом правиле вывода, и вставить в его правую часть символ $\{f\}_{a_1, a_2, \ldots, a_n, r}$. Заменить некопирующее правило на следующие $(n + 1)$ копирующих правил:

   $a_i \leftarrow y_i$ для каждого аргумента y_i;

   $z_1, z_2, \ldots, z_m \leftarrow r$.

   При включении в правило вывода операционного символа необходимо соблюдать следующие условия:

   - операционный символ должен располагаться *правее* всех символов правой части правила вывода, атрибутами которых являются аргументы $y_1, y_2, \ldots, y_n$;

- операционный символ должен располагаться *левее* всех символов правой части правила вывода, атрибутами которых служат $z_1, z_2, \ldots, z_m$;

- с учетом предыдущих ограничений операционный символ может быть вставлен в любое место правой части правила вывода, но предпочтение следует отдать *самой левой из возможных позиций*, т. к. это позволяет упростить реализацию синтаксического анализатора.

3. Два копирующих правила, соответствующие одному и тому же правилу вывода, необходимо объединить, если источник одного из них входит в другое. Это достигается удалением правила с лишним источником и объединением его приемников с приемниками оставшегося правила.

Если в качестве источника копирующего правила используется константная функция, являющаяся процедурой-функцией без параметров (например, функция GETNEW из AT-грамматики, приведенной в *разд. 11.4.3*), то такие атрибутные правила объединять нельзя, т. к. два разных вызова функции без параметров могут давать разные значения.

Рассмотрим пример преобразования L-атрибутной транслирующей грамматики, порождающей префиксные арифметические выражения над константами, в форму простого присваивания. Атрибутные правила вывода этой грамматики имеют следующий вид:

E_p синтезированный p

$\{\text{ОТВЕТ}\}_r$ унаследованный r

(1) $S \rightarrow E_p \{\text{ОТВЕТ}\}_r$

$r \leftarrow p$

(2) $E_p \rightarrow + E_q E_r$

$p \leftarrow q + r$

(3) $E_p \rightarrow * E_q E_r$

$p \leftarrow q * r$

(4) $E_p \rightarrow c_r$

$p \leftarrow r$

Входными символами грамматики являются: лексема c, представляющая собой целочисленную константу, и знаки арифметических операций: '+' и '*'. Входной символ c имеет один атрибут, значением которого является значение константы. Нетерминальный символ E и операционный символ {ОТВЕТ} также имеют по одному атрибуту. Значением синтезированного атрибута символа E является значение подвыражения, порождаемого этим символом, а значением унаследованного атрибута символа {ОТВЕТ} — значение всего выражения, порождаемого грамматикой.

Исходная грамматика содержит два некопирующих правила: $p \leftarrow q + r$ и $p \leftarrow q * r$, правые части которых представляют собой функции сложения и умножения соответственно. Для преобразования заданной грамматики в форму простого присваивания введем операционные символы $\{\text{СЛОЖИТЬ}\}_{A,\ B,\ R}$ и $\{\text{УМНОЖИТЬ}\}_{A,\ B,\ R}$, каждый из которых имеет по два унаследованных атрибута A и B и один синтезированный атрибут R. Для операционного символа $\{\text{СЛОЖИТЬ}\}_{A,\ B,\ R}$ атрибутное правило имеет вид: $R \leftarrow A + B$, а для операционного символа $\{\text{УМНОЖИТЬ}\}_{A,\ B,\ R}$ — $R \leftarrow A \times B$.

Для того чтобы преобразованная грамматика также была L-атрибутной, символ {СЛОЖИТЬ} необходимо поместить правее всех символов правой части правила вывода (2), т. к. одним из аргументов сложения является атрибут самого правого символа E. Атрибут, получающий в качестве своего значения результат сложения, в определении места расположения символа {СЛОЖИТЬ} не участвует, т. к. он не приписан ни к одному из символов правой части. Аналогичные рассуждения относительно операционного символа {УМНОЖИТЬ} определяют крайнюю правую позицию правой части правила (3), как единственно возможное место расположения этого символа. Полученная в результате преобразования L-атрибутная грамматика в форме простого присваивания имеет вид:

E_p синтезированный p

$\{\text{ОТВЕТ}\}_r$ унаследованный r

$\{\text{СЛОЖИТЬ}\}_{A,\ B,\ R}$ унаследованный A, B

$R \leftarrow A + B$ синтезированный R

$\{\text{УМНОЖИТЬ}\}_{A,\ B,\ R}$ унаследованный A, B

$R \leftarrow A * B$ синтезированный R

(1) $S \rightarrow E_p\ \{\text{ОТВЕТ}\}_r$

$r \leftarrow p$

(2) $E_p \rightarrow +\ E_q\ E_r\ \{\text{СЛОЖИТЬ}\}_{A,\ B,\ R}$

$A \leftarrow q$

$B \leftarrow r$

$p \leftarrow R$

(3) $E_p \rightarrow *\ E_q\ E_r\ \{\text{УМНОЖИТЬ}\}_{A,\ B,\ R}$

$A \leftarrow q$

$B \leftarrow r$

$p \leftarrow R$

(4) $E_p \rightarrow c_r$

$p \leftarrow r$

Эта грамматика порождает те же входные цепочки и значение синтезированного атрибута нетерминала E, что и исходная грамматика. Однако формально она не определяет того же самого перевода, т. к. преобразованные правила (2) и (3) *удлиняют активную цепочку*, включив в нее действия, обеспечивающие вычисление функций сложения и умножения соответственно. Для того чтобы преобразованная грамматика определяла тот же перевод, что и исходная, введенные в процессе преобразования операционные символы не следует выдавать в выходную цепочку.

12.3. Атрибутный перевод для *LL*(1)-грамматик

Расширим "1-предсказывающий" алгоритм разбора так, чтобы он мог выполнять атрибутный перевод, определяемый L-атрибутной транслирующей грамматикой, входной грамматикой которой является $LL(1)$-грамматика. Моделирование такого алгоритма можно осуществить с помощью атрибутного ДМП-преобразователя с концевым маркером.

Сначала рассмотрим проблему выполнения синтаксически управляемого перевода, определяемого транслирующей грамматикой цепочечного перевода, входной грамматикой которого является $LL(1)$-грамматика.

12.3.1. Реализация синтаксически управляемого перевода для транслирующей грамматики

Преобразуем "1-предсказывающий" алгоритм разбора для $LL(1)$-грамматик, включив в него действия, обеспечивающие перевод входной цепочки, порождаемой входной грамматикой, в цепочку операционных символов и запись этой цепочки на выходную ленту. Преобразованный таким образом алгоритм в дальнейшем будем называть *нисходящим детерминированным процессором с магазинной памятью* (*нисходящий ДМП-процессор*). При этом, если из контекста ясно, что речь идет о нисходящем методе анализа, слово "нисходящий" будем опускать.

В транслирующей грамматике множество терминальных символов разбито на множество входных символов Σ_i и множество операционных символов Σ_a.

Расширим алфавит магазинных символов, добавив в него операционные символы. Тогда $V_p = \Sigma_i \cup \Sigma_a \cup N$. Затем доопределим управляющую таблицу M, которая для транслирующей грамматики цепочечного перевода задает отображение множества $(V_p \cup \{\perp\}) \times (\Sigma_i \cup \{\varepsilon\})$ в множество, состоящее из следующих элементов:

1. (β, y), где $\beta \in V_p^*$ — цепочка из правой части правила транслирующей грамматики $A \to y\beta$, а $y \in \Sigma_a^*$;
2. *ВЫДАЧА*(X), где $X \in \Sigma a$;
3. *ВЫБРОС*;

4. *ДОПУСК*;
5. *ОШИБКА*.

Пусть FIRST(x) = a, где x — неиспользованная часть входной цепочки. Тогда работу ДМП-процессора в зависимости от элемента управляющей таблицы $M(X, a)$ можно определить следующим образом:

1. $(x, X\alpha, \pi) \vdash (x, \beta\alpha, \pi y)$, если $M(X, a) = (\beta, y)$. Верхний символ магазина X заменяется цепочкой $\beta \in V_p^*$, и в выходную цепочку дописывается цепочка операционных символов y. Входная головка при этом не сдвигается.
2. $(x, X\alpha, \pi) \vdash (x, \alpha, \pi X)$, если $M(X, a) =$ *ВЫДАЧА*(X). Это означает, что если верхний символ магазина — операционный символ, то он выталкивается из магазина и записывается на выходную ленту. Входная головка не сдвигается.

Действия, соответствующие элементам управляющей таблицы: *ВЫБРОС*, *ДОПУСК* и *ОШИБКА*, остаются теми же, что и в "1-предсказывающем" алгоритме для $LL(1)$-грамматик.

Опишем алгоритм построения управляющей таблицы M.

Алгоритм 12.1. *Алгоритм построения управляющей таблицы для транслирующей грамматики цепочечного перевода, входной грамматикой которой является LL(1)-грамматика*

Вход: Транслирующая грамматика цепочечного перевода $G^T = (N, \Sigma_i, \Sigma_a, P, S)$, входная грамматика которой является $LL(1)$-грамматикой.

Выход : Корректная управляющая таблица M для грамматики G^T.

Описание алгоритма:

□ Управляющая таблица M определяется на множестве $(N \cup \Sigma_i \cup \Sigma_a \cup \{\perp\}) \times (\Sigma_i \cup \{\varepsilon\})$ по следующим правилам:

1. Если $A \rightarrow y\beta$ — правило вывода грамматики G^T, где $y \in \Sigma_a^*$, а y — либо ε, либо цепочка, начинающаяся с терминала или нетерминала, то $M(A, a) = (\beta, y)$ для всех $a \neq \varepsilon$, принадлежащих множеству FIRST(β).

   Если $\varepsilon \in$ FIRST(β), то $M(A, b) = (\beta, y)$ для всех $b \in$ FOLLOW(A).

   Заметим, что при вычислении FIRST(β) операционные символы, входящие в цепочку β, вычеркиваются.
2. $M(X, a) =$ *ВЫДАЧА*(X) для всех $X \in \Sigma_a$ и $a \in \Sigma_i \cup \{\varepsilon\}$.
3. $M(a, a) =$ *ВЫБРОС* для всех $a \in \Sigma_i$.
4. $M(\perp, \varepsilon) =$ *ДОПУСК*.
5. В остальных случаях $M(X, a) =$ *ОШИБКА* для $X \in (N \cup \Sigma_i \cup \Sigma_a \cup \{\perp\})$ и $a \in \Sigma_i \cup \{\varepsilon\}$.

■

Построим управляющую таблицу для транслирующей грамматики, описывающей перевод инфиксных арифметических выражений в ПОЛИЗ. Эта транслирующая грамматика получена из входной $LL(1)$-грамматики G_1 путем включения в нее операционных символов $\{i\}$, $\{+\}$ и $\{*\}$:

(1) $E \rightarrow TE'$ (5) $T' \rightarrow * P \{*\} T'$

(2) $E' \rightarrow + T \{+\} E'$ (6) $T' \rightarrow \varepsilon$

(3) $E' \rightarrow \varepsilon$ (7) $P \rightarrow i \{i\}$

(4) $T \rightarrow PT'$ (8) $P \rightarrow (E)$

Управляющая таблица должна содержать 14 строк, помеченных символами из множества $N \cup \Sigma_i \cup \Sigma_a \cup \{\perp\}$, и 6 столбцов, помеченных символами из множества $\Sigma_i \cup \{\varepsilon\}$.

Построение управляющей таблицы для строк, отмеченных символами из множества $N \cup \Sigma_i \cup \{\perp\}$, ничем не отличается от построения таблицы для соответствующей входной $LL(1)$-грамматики (табл. 12.1), а строки управляющей таблицы, отмеченные операционными символами, содержат значения *ВЫДАЧА*($\{X\}$), где $X \in \{\{i\}, \{+\}, \{*\}\}$. Заметим, что элементы таблицы, соответствующие строкам, помеченным операционными символами, для всех столбцов одинаковые, т. к. действия, выполняемые ДМП-процессором в случае, когда верхним символом магазина является операционный символ, не зависят от входного символа.

Таблица 12.1. *Управляющая таблица M для транслирующей грамматики, описывающей перевод инфиксных арифметических выражений в ПОЛИЗ*

	i	**(**	**)**	**+**	*****	**ε**
E	*TE′*, ε	*TE′*, ε				
E′			ε, ε	*+ T{+} E′*, ε		ε, ε
T	*PT′*, ε	*PT′*, ε				
T′			ε, ε	ε, ε	** P{*} T′*, ε	ε, ε
P	*i{i}*, ε	*(E)*, ε				
i	*ВЫБРОС*					
(		*ВЫБРОС*				
)			*ВЫБРОС*			
+				*ВЫБРОС*		
*****					*ВЫБРОС*	
⊥						*ДОПУСК*
{*i*}	*ВЫДАЧА*({*i*})					
{+}	*ВЫДАЧА*({+})					
{*}	*ВЫДАЧА*({*})					
Начальное содержимое магазина — *E*⊥						

Для входной цепочки $i + i * i$ ДМП-процессор проделает следующую последовательность тактов:

$$
\begin{aligned}
(i + i * i,\ E\bot,\ \varepsilon) &\vdash (i + i * i,\ TE'\bot,\ \varepsilon)\\
&\vdash (i + i * i,\ PT'E'\bot,\ \varepsilon)\\
&\vdash (i + i * i,\ i\{i\}\ T'E'\bot,\ \varepsilon)\\
&\vdash (+ i * i,\ \{i\}\ T'E'\bot,\ \varepsilon)\\
&\vdash (+ i * i,\ T'E'\bot,\ \{i\})\\
&\vdash (+ i * i,\ E'\bot,\ \{i\})\\
&\vdash (+ i * i,\ + T\{+\}\ E'\bot,\ \{i\})\\
&\vdash (\ i * i,\ T\{+\}\ E'\bot,\ \{i\})\\
&\vdash (\ i * i,\ PT'\ \{+\}\ E'\bot,\ \{i\})\\
&\vdash (\ i * i,\ i\{i\}\ T'\{+\}E'\bot,\ \{i\})\\
&\vdash (\ * i,\ \{i\}\ T'\ \{+\}\ E'\bot,\ \{i\})\\
&\vdash (\ * i,\ T'\ \{+\}\ E'\bot,\ \{i\}\{i\})\\
&\vdash (\ * i,\ * P\{*\}\ T'\ \{+\}\ E'\bot,\ \{i\}\{i\})\\
&\vdash (\ i,\ P\{*\}\ T'\ \{+\}\ E'\bot,\ \{i\}\{i\})\\
&\vdash (\ i,\ i\{i\}\{*\}\ T'\ \{+\}\ E'\bot,\{i\}\{i\})\\
&\vdash (\varepsilon,\ \{i\}\{*\}\ T'\ \{+\}\ E'\bot,\ \{i\}\{i\}\})\\
&\vdash (\varepsilon,\ \{i\}\{*\}\ T'\ \{+\}\ E'\bot,\ \{i\}\{i\})\\
&\vdash (\varepsilon,\ T'\ \{+\}\ E'\bot,\ \{i\}\{i\}\{i\}\{*\})\\
&\vdash (\varepsilon,\ \{+\}\ E'\bot,\ \{i\}\{i\}\{i\}\{*\})\\
&\vdash (\varepsilon,\ E'\bot,\ \{i\}\{i\}\{i\}\{*\}\{+\})\\
&\vdash (\varepsilon,\ \bot,\ \{i\}\{i\}\{i\}\{*\}\{+\}).
\end{aligned}
$$

ДМП-процессор можно использовать в качестве базового процессора для других видов синтаксически управляемого перевода, если операцию выдачи операционного символа в выходную ленту заменить операциями вызова соответствующих семантических процедур.

12.3.2. *L*-атрибутный ДМП-процессор

Процедура преобразования ДМП-процессора в атрибутный ДМП-процессор, которая описывается в данном пособии, требует, чтобы АТ-грамматика, определяющая перевод, имела *форму простого присваивания*.

Рассмотрим построение L-атрибутного ДМП-процессора, реализующего перевод, определяемый L-атрибутной транслирующей грамматикой в форме простого присваивания, входной грамматикой которого является $LL(1)$-грамматика.

Сначала построим ДМП-процессор, реализующий цепочечный перевод, описываемый транслирующей грамматикой цепочечного перевода, которая получается из заданной L-атрибутной транслирующей грамматики после удаления из нее всех атрибутов. Затем расширим полученный таким образом ДМП-процессор, включив для каждого магазинного символа множество полей для представления атрибутов символа и дополнив управляющую таблицу действиями по вычислению атрибутов и записи их в соответствующие поля.

Для удобства изложения будем считать, что магазинный символ с n атрибутами представляется в магазине $(n + 1)$-ой ячейками, верхняя из которых содержит имя символа, а остальные — поля для атрибутов. Поля магазинного символа доступны для записи и извлечения атрибутов от момента вталкивания символа в магазин до момента выталкивания его из магазина.

В момент вталкивания символа в магазин в поле для каждого синтезированного атрибута и атрибута входного символа заносится указатель на связанный список полей, соответствующих унаследованным атрибутам, где этот атрибут должен запоминаться, а поле для каждого унаследованного атрибута остается пустым. Содержимое полей синтезированных атрибутов и атрибутов входных символов остается неизменным в течение всего времени нахождения символа в магазине, а поля, соответствующие унаследованным атрибутам, приобретают значения атрибутов к моменту времени, когда магазинный символ окажется в верхушке магазина.

Пусть x — остаток входной цепочки, и $\mathrm{FIRST}(x) = a$. Опишем действия, которые должен выполнять L-атрибутный ДМП-процессор в зависимости от элемента управляющей таблицы $M(X, a)$, где X — символ в верхушке магазина.

- *Начальная конфигурация.* В магазине находится маркер дна и начальный символ грамматики. Поля начального символа грамматики, соответствующие унаследованным атрибутам, заполняются начальными значениями атрибутов, которые задаются L-атрибутной транслирующей грамматикой, а в поля, соответствующие синтезированным атрибутам, заносятся пустые указатели, которые служат маркерами конца списков.
- $M(X, a) = ВЫБРОС$ (символ в верхушке магазина совпадает с текущим входным символом). В этом случае каждое поле верхнего магазинного символа содержит указатель на список полей магазина, в которых требуется поместить значение атрибута текущего входного символа. Операция *ВЫБРОС* расширяется таким образом, что каждый атрибут текущего входного символа копируется во все поля списка на который указывает соответствующее поле верхнего магазинного символа.
- $M(X, a) = ВЫДАЧА(X)$ (в верхушке магазина находится операционный символ). Операция *ВЫДАЧА*(X) ДМП-процессора расширяется следующим образом:
  - значения унаследованных атрибутов извлекаются из соответствующих полей верхнего магазинного символа и используются затем при выдаче символа в выходную цепочку. При этом следует

помнить, что если операционный символ появился в результате преобразования исходной атрибутной транслирующей грамматики в форму простого присваивания, то такой символ в выходную цепочку не выдается;

- значения синтезированных атрибутов вычисляются по правилам вычисления атрибутов, связанным с данным операционным символом, после чего значение каждого синтезированного атрибута помещается во все поля списка, на который указывает соответствующее поле символа из верхушки магазина.

- $M(X, a) = (\beta, y)$ (в верхушке магазина находится нетерминал). В этом случае L-атрибутный ДМП-процессор вталкивает в магазин цепочку символов β из правой части распознаваемого правила и выдает цепочку операционных символов y. При этом вычисляются атрибуты операционных символов, которые не вталкиваются в магазин, и заполняются поля атрибутов магазинных символов и символов цепочки β, вталкиваемых в магазин.

Источниками атрибутных правил, связанных с правилами вывода L-атрибутной транслирующей грамматики в форме простого присваивания, могут быть только константы, унаследованные атрибуты нетерминалов из левой части правил вывода, атрибуты входных и операционных символов, синтезированные атрибуты нетерминалов из правой части правил вывода. В табл. 12.2 приведены значения источников копирующих правил в момент времени, когда верхним символом магазина является нетерминал.

Таблица 12.2. *Значения источников копирующих правил*

Источник	Доступ к источнику
Константа	Значение всегда доступно
Унаследованный атрибут нетерминала из левой части правила	Значение источника находится в соответствующем поле верхнего символа магазина
Синтезированный атрибут операционного символа, который не вталкивается в магазин	Значение источника вычисляется в соответствии с атрибутным правилом
Синтезированный атрибут вталкиваемых в магазин нетерминала и операционного символа	Недоступен
Атрибут входного символа	Недоступен

Приемниками атрибутных правил L-атрибутной транслирующей грамматики могут быть только синтезированные атрибуты нетерминалов из левой

части правил вывода и унаследованные атрибуты символов из правой части правил вывода. В табл. 12.3 приведены поля магазина, соответствующие приемникам атрибутных правил, которые необходимо заполнить во время перехода L-атрибутного ДМП-процессора при $M(X, a) = (\beta, y)$.

Таблица 12.3. *Значения приемников атрибутных правил*

Приемник	Поле
Унаследованный атрибут нетерминала и операционного символа, вталкиваемых в магазин	Соответствующее поле вталкиваемого символа
Синтезированный атрибут нетерминала из левой части правила	Все поля в списке, на который указывает соответствующее поле магазинного символа

При выполнении перехода L-атрибутный ДМП-процессор выполняет следующие атрибутные действия:

1. Вычисляет значения синтезированных атрибутов операционных символов, которые не вталкиваются в магазин, и выдает цепочку операционных символов с их атрибутами в выходную цепочку, если эти символы не появились в результате преобразования грамматики в форму простого присваивания.
2. Если источник копирующего правила доступен (первые две строки табл. 12.2), то значение источника помещается в соответствующее поле магазинного символа.
3. Если источник копирующего правила недоступен, то после вталкивания символа в магазин в соответствующие поля символа заносятся указатели на список полей, где будут храниться значения унаследованных атрибутов.

Действия L-атрибутного ДМП-процессора для элементов управляющей таблицы, имеющих значения *ДОПУСК* и *ОШИБКА*, остаются теми же самыми, что у ДМП-процессора.

Построим L-атрибутный ДМП-процессор для L-атрибутной транслирующей грамматики в форме простого присваивания, правила вывода которой вместе с атрибутными правилами приведены в *разд. 11.4.3*.

На рис. 12.1 показано представление полей магазинных символов, а в табл. 12.4 приведена управляющая таблица для транслирующей грамматики, полученной из исходной L-атрибутной транслирующей грамматики путем вычеркивания из нее атрибутов.

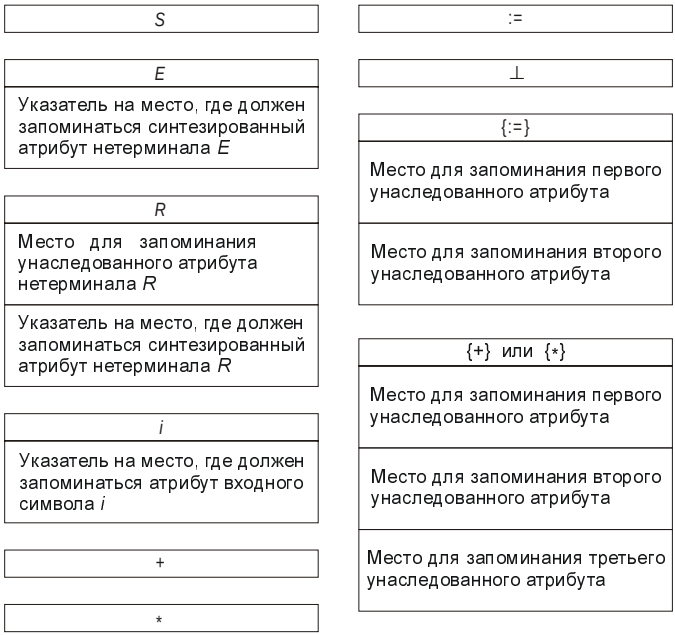


Рис. 12.1. Представление полей магазинных символов

Таблица 12.4. *Управляющая таблица для транслирующей грамматики, полученной из исходной грамматики*

`	***i***	**:=**	**+**	*****	**ε**
S	*i* := *E* {:=}, ε				
E	*i R*, ε				
R			+ *i* {+} *R*, ε	* *i* {*} *R*, ε	ε, ε
i	*ВЫБРОС*				
:=		*ВЫБРОС*			
+			*ВЫБРОС*		
*****				*ВЫБРОС*	
⊥					*ДОПУСК*
{:=}	*ВЫДАЧА*($\{:=\}_{p,\ q}$)				
{+}	*ВЫДАЧА*($\{+\}_{p,\ q,\ r}$)				
{*}	*ВЫДАЧА*($\{*\}_{p,\ q,\ r}$)				
Начальное содержимое магазина — *S*⊥					

Действия, выполняемые над атрибутами L-атрибутным ДМП-процессором при переходе из одной конфигурации в другую в соответствии с атрибутными правилами L-атрибутной транслирующей грамматики, приведены на рис. 12.2.

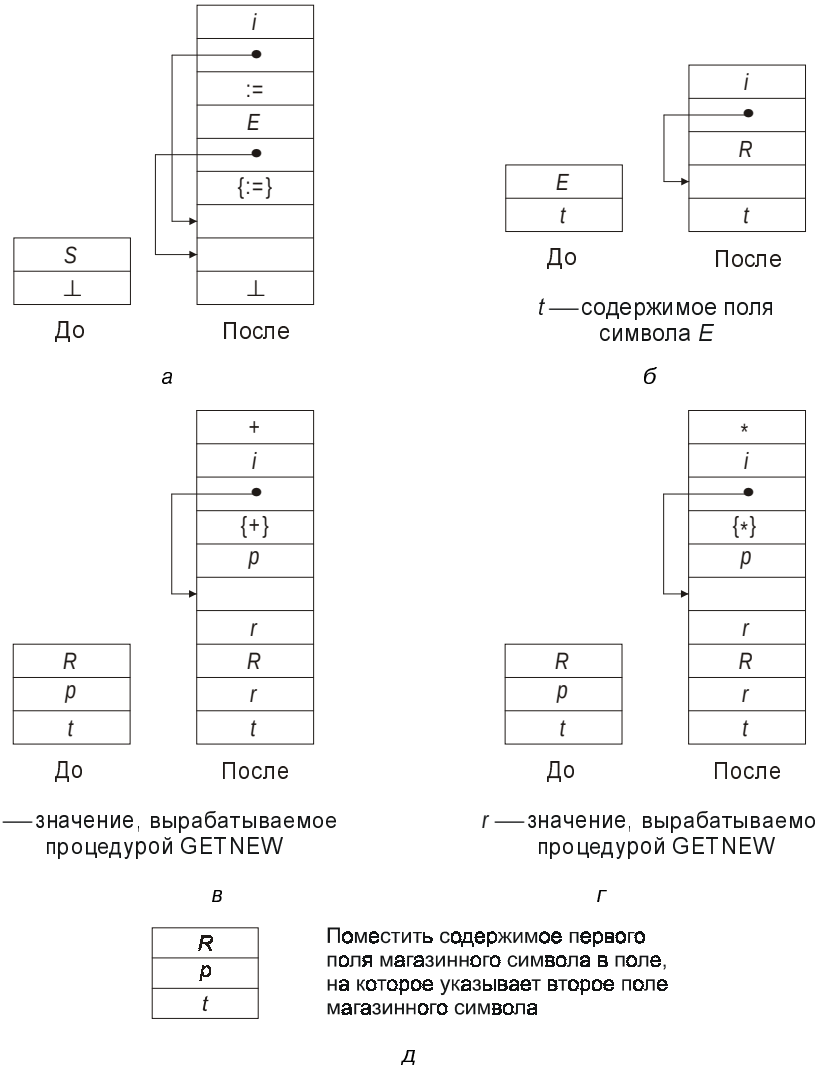


Рис. 12.2. Действия над атрибутами в магазине

На рис. 12.2 , *а*—*р* изображена последовательность состояний магазина построенного L-атрибутного ДМП-процессора, которая соответствует обработке входной цепочки $i_5 := i_1 + i_2 * i_3$.

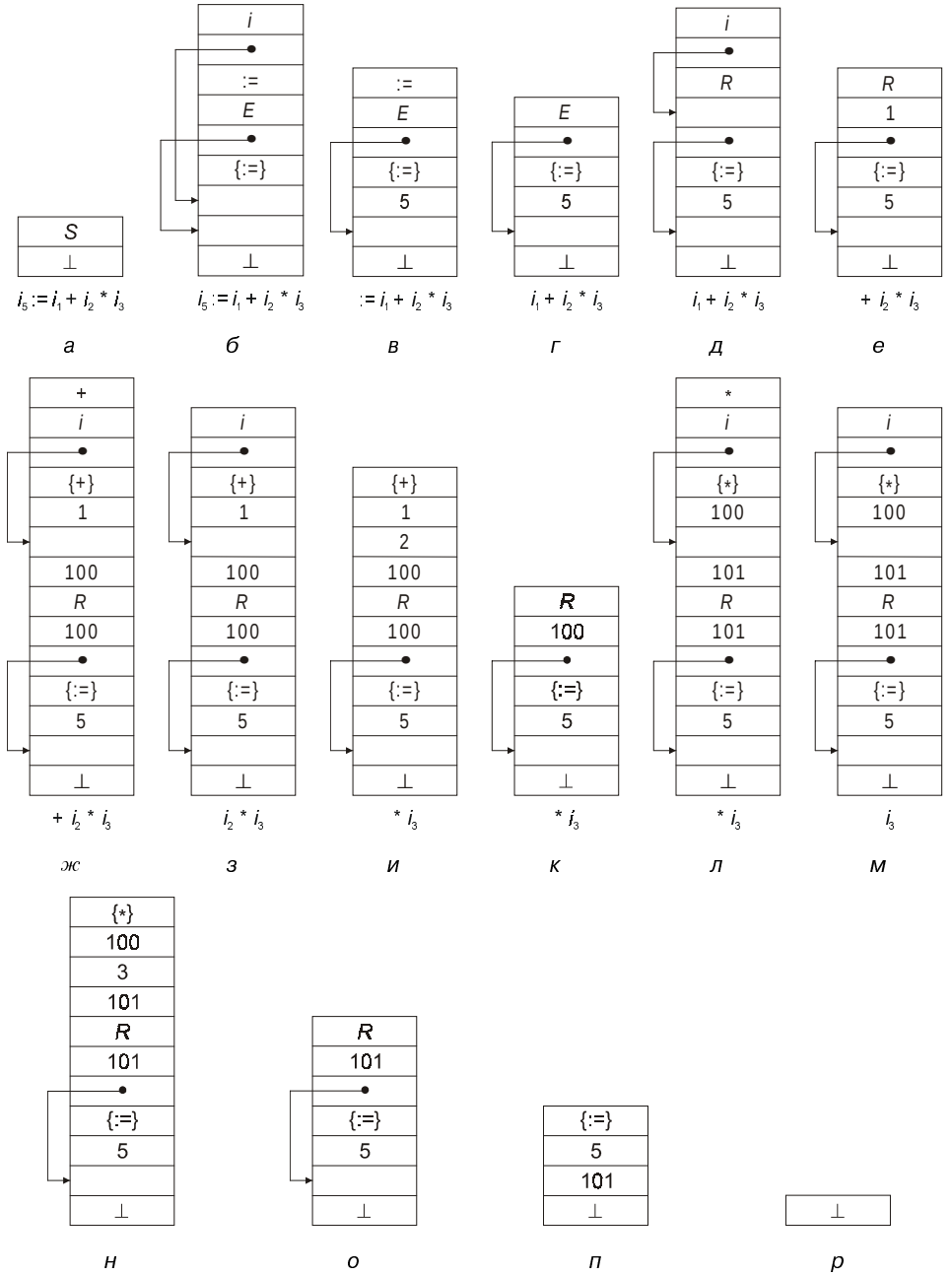


Рис. 12.3. Последовательность состояний магазина при обработке входной цепочки $i_5 := i_1 + i_2 * i_3$

Переход от рис. 12.3, *а* к рис. 12.3, *б* предопределяется элементом управляющей таблицы $M(S, i)$. В соответствии с копирующими правилами, связанными с правилом вывода (1) АТ-грамматики, в поле атрибута входного символа *i* и в поле синтезированного атрибута нетерминала *E* засылаются указатели на поля унаследованных атрибутов операционного символа {:=}. Так как символы *i* и *E* обрабатываются раньше, чем символ {:=} окажется в верхушке магазина, пустые поля символа {:=} будут заполнены значениями атрибутов до того, как символ {:=} станет верхним символом магазина.

Переход от рис. 12.3, *б* к рис. 12.3, *в* демонстрирует выталкивание символа *i* из магазина. При этом значение атрибута текущего входного символа (значение 5) копируется в поле, соответствующее первому унаследованному атрибуту символа {:=}, на которое указывает поле магазинного символа *i*.

Переход от рис. 12.3, *в* к рис. 12.3, *г* соответствует выталкиванию из магазина входного символа {:=}, у которого нет атрибутов.

На рис. 12.3, *д* изображено содержимое магазина после выполнения перехода, соответствующего элементу управляющей таблицы $M(E, i)$. Нетерминал *E* заменяется цепочкой *iR* в соответствии с рис. 12.2, *б*. При этом указатель из поля синтезированного атрибута нетерминала *E* переписывается в поле синтезированного атрибута нетерминала *R*, в результате чего выполняется копирующее правило $t2 \leftarrow t1$.

Переход от рис. 12.3, *д* к рис. 12.3, *е* соответствует выталкиванию из магазина символа *i* и копированию атрибута текущего символа из входной цепочки (значение 1) в поле унаследованного атрибута нетерминала *R*.

На рис. 12.3, *ж* изображено содержимое магазина после применения правила вывода (3), которому соответствует элемент управляющей таблицы $M(R, +)$. В этом случае значение унаследованного атрибута нетерминала *R* копируется в поле первого унаследованного атрибута операционного символа {+}, а в поле атрибута входного символа *i* помещается указатель на поле второго унаследованного атрибута операционного символа {+}, в котором в дальнейшем (рис. 12.3, *и*) будет находиться значение атрибута второго операнда операции сложения. Заполнение поля третьего унаследованного атрибута операционного символа {+} и поля унаследованного атрибута нетерминала *R* осуществляется путем вызова процедуры-функции GETNEW, которая вырабатывает значение указателя на свободную позицию таблицы, используемой для записи промежуточных результатов (например, значение 100). Копирование указателя из поля синтезированного атрибута нетерминала *R*, находящегося в верхушке магазина, в поле синтезированного атрибута нетерминала *R*, вталкиваемого в магазин, выполняется в соответствии с копирующим правилом $t2 \leftarrow t1$.

Переход от рис. 12.3, *ж* к рис. 12.3, *з* соответствует выталкиванию из магазина безатрибутного символа '+', а состояние магазина после выталкивания

из него входного символа *i*, сопровождающееся копированием атрибута текущего входного символа (значение 2) во второе поле унаследованного атрибута операционного символа {+}, приведено на рис. 12.3, *и*.

Переход от рис. 12.3, *и* к рис. 12.3, *к* соответствует выталкиванию из магазина операционного символа {+}. При этом в выходную цепочку выдается тетрада:

+	1	2	100

Переход от рис. 12.3, *к* к рис. 12.3, *л* осуществляется в результате применения правила вывода (4) АТ-грамматики подобно переходу от рис. 12.3, *е* к рис. 12.3, *ж*, а последовательность переходов от рис. 12.3, *л* к рис. 12.3, *о* выполняется аналогично последовательности переходов от рис. 12.3, *ж* к рис. 12.3, *к* и заканчивается выдачей тетрады:

*	100	3	101

Действия *L*-атрибутного ДМП-процессора при применении правила вывода (5) АТ-грамматики (переход от рис. 12.3, *о* к рис. 12.3, *п*) сводятся к выталкиванию из магазина нетерминала *R* и копированию значения унаследованного атрибута нетерминала *R* в поле, на которое ссылается указатель из поля синтезированного атрибута этого нетерминала.

При выталкивании из магазина операционного символа {:=} (рис. 12.3, *п*) в выходную цепочку выдается тетрада присваивания:

:=	101		5

и магазин опустошается (рис. 12.3, *р*).

12.3.3. Атрибутный перевод методом рекурсивного спуска

Сначала рассмотрим, каким образом можно изменить процедуры для распознавания цепочек, порождаемых нетерминальными символами грамматики, для реализации перевода, описываемого транслирующей грамматикой цепочечного перевода.

В этом случае правила составления процедур дополняются следующим правилом: если текущим символом правой части правила вывода грамматики является операционный символ `Y`, ему соответствует вызов процедуры записи операционного символа в выходную цепочку `Output(Y)`.

В качестве примера реализации перевода с использованием метода рекурсивного спуска рассмотрим транслирующую грамматику, описывающую перевод инфиксных арифметических выражений в польскую инверсную

запись. Эта грамматика построена на основе входной грамматики G_1 и имеет следующие правила:

$S \rightarrow E\perp$

$E \rightarrow TE'$

$E' \rightarrow + T \{+\}\ E' \mid \varepsilon$

$T \rightarrow PT'$

$T' \rightarrow * P \{*\}\ T' \mid \varepsilon$

$P \rightarrow (E) \mid i \{i\}$

Головной модуль `Recurs_Method` и процедуры для распознавания нетерминальных символов E (`Proc_E`) и T (`Proc_T`) не изменятся. Процедура на языке Pascal, реализующая перевод для транслирующих грамматик методом рекурсивного спуска приведена в листинге 12.1.

Листинг 12.1

```
Procedure Recurs_Method_TG (List_Token: tList, List_Oper: tListOper);
                           {List_Token – входная цепочка,
                            List_Oper – выходная цепочка}
   Procedure Proc_E;
      begin
         Proc_T;
         Proc_E1
   end {Proc_E};
   Procedure Proc_E1;
      begin
         if Symb = '+' then
         begin
            NextSymb;
            Proc_T;
            Output({+});
            Proc_E1
         end
   end {Proc_E1}
Procedure Proc_T;
      begin
         Proc_P;
         Proc_T1
   end {Proc_T};
```

```
Procedure Proc_T1;
      begin
         if Symb = '*' then
         begin
            NextSymb;
            Proc_P;
            Output({*});
            Proc_T1
         end
   end {Proc_T1};
   Procedure Proc_P;
      begin
         if Symb = '(' then
         begin
            NextSymb;
            Proc_E;
            if Symb = ')' then
               NextSymb;
            else
                Error
         end
         else
            if Symb = 'i' then
            begin
               NextSymb;
               Output({i})
            end
            else
               Error
   end; {Proc_P}
begin
   { В начале анализа переменная Symb содержит первый символ цепочки }
   Proc_E;
       if Symb = '⊥' then
          Access
       else
          Error
end {Recurs_Method_TG};
```

На рис. 12.4 приведен список имен процедур в порядке их вызова при переводе входной цепочки $(i + i) * i \perp$. В предпоследнем столбце в строке, соответствующей вызову процедуры `NextSymb`, изображена непрочитанная часть входной цепочки (текущий символ находится слева) непосредственно после

вызова этой процедуры, а в последнем столбце в строке, соответствующей вызову процедуры `Output(Y)`, изображена часть выходной цепочки, построенной непосредственно после вызова процедуры `Output(Y)`.

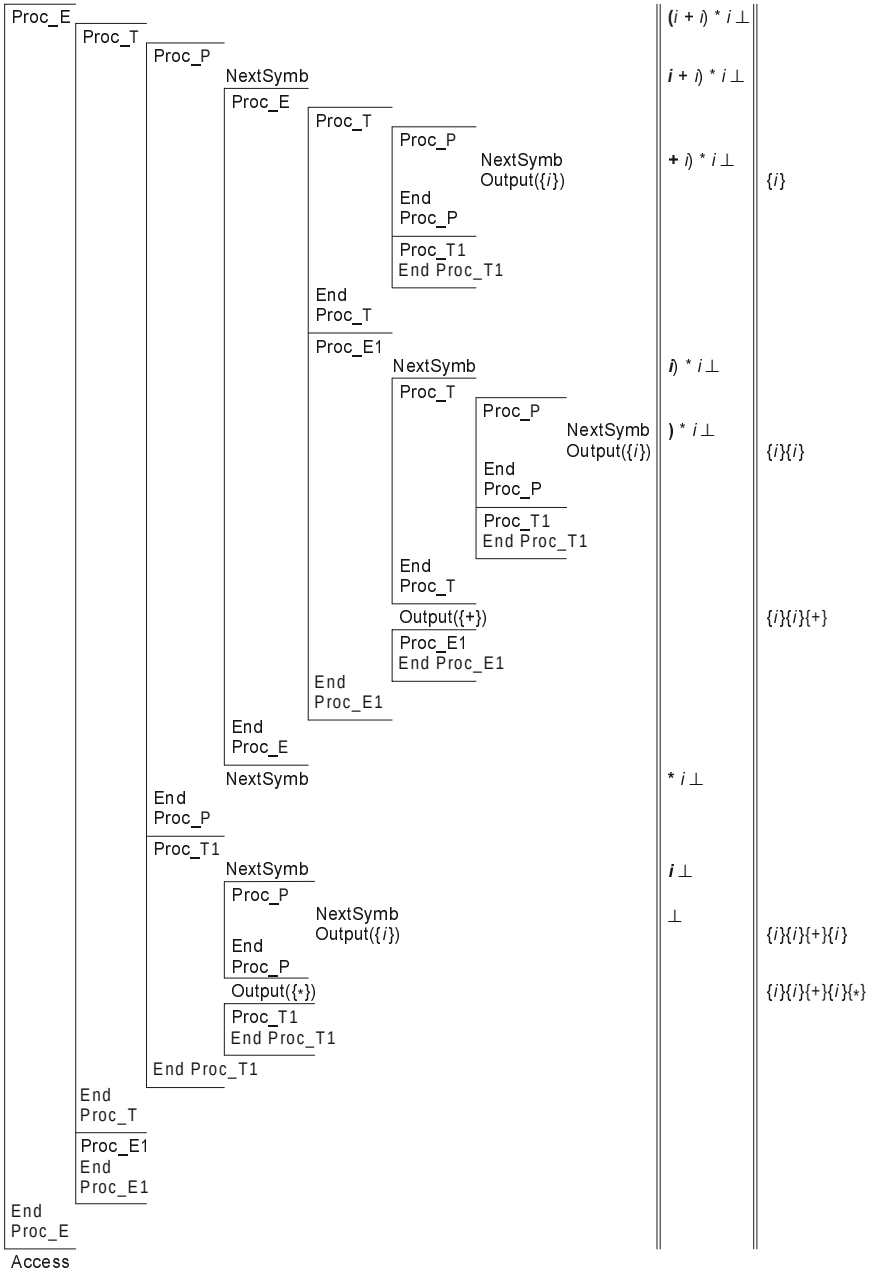


Рис. 12.4. Порядок вызова процедур при переводе цепочки $(i + i) * i \perp$

Для того, чтобы методом рекурсивного спуска можно было выполнять L-атрибутный перевод, введем в процедуры распознавания нетерминальных символов *параметр* для каждого атрибута этого символа. При этом если параметру процедуры соответствует унаследованный атрибут нетерминального символа, то вызов этой процедуры производится с фактическим параметром — *значением унаследованного атрибута*, а в случае синтезированного атрибута в качестве фактического параметра используется *переменная*, которой должно быть присвоено *значение синтезированного атрибута в момент выхода из процедуры*.

Для того чтобы процедуры обеспечивали правильную передачу параметров для унаследованных и синтезированных атрибутов, они должны быть написаны на языке программирования, который обеспечивает способы передачи параметров "вызов по значению" (для унаследованных атрибутов) и "вызов по ссылке" (для синтезированных атрибутов).

Замечание

Язык программирования Pascal поддерживает оба метода передачи параметров, а в языке C имеется единственный механизм передачи параметров — "вызов по значению" [36]. Эффект вызова по ссылке в языке C может быть получен, если использовать в качестве параметров указатели.

Поскольку имена атрибутов нетерминальных символов используются в качестве параметров процедур для распознавания этих нетерминалов, при именовании атрибутов необходимо выполнять дополнительное требование, заключающееся в том, что все вхождения некоторого нетерминала в левые части правил вывода AT-грамматики должны иметь один и тот же список имен атрибутов. Например, в грамматике не может быть таких правил вывода:

$$R_{p_1,\, t_2} \rightarrow + \; i_{q_1} \; \{*\}_{p_2,\, q_2,\, r_1} \; R_{r_2,\, t_1}$$

$$R_{p_1,\, t_2} \rightarrow * \; i_{q_1} \; \{*\}_{p_2,\, q_2,\, r_1} \; R_{r_2,\, t_1}$$

$$R_{p_1,\, p_2} \rightarrow \varepsilon$$

т. к. первые два вхождения нетерминала R имеют атрибуты p_1, t_2, а последнее вхождение — p_1, p_2. В этом случае необходимо выбрать какой-то один список имен атрибутов нетерминала R (например, p и t), использовать эти имена при описании типа атрибутов этого нетерминала (например, УНАСЛЕДОВАННЫЙ p, СИНТЕЗИРОВАННЫЙ t) и переименовать его атрибуты соответствующим образом.

Замечание

Указанные ограничения на имена атрибутов не распространяются на атрибуты символов из правых частей правил вывода грамматики.

После того как атрибуты в левых частях правил и в описании их типов переименованы, для упрощения или исключения некоторых правил вычисления атрибутов можно использовать новое соглашение об обозначениях атрибутов, которое формулируется следующим образом: "Если два атрибута получают одно и то же значение, то им можно дать *одно и то же имя* при условии, что для этого не нужно изменять имена атрибутов нетерминала в левой части правила вывода".

Рассмотрим несколько примеров.

1. $A \to B_x\, C_y\, D_z$

   $y, z \leftarrow x$

   Атрибутам x, y и z присваивается одно и то же значение, поэтому им можно дать общее имя. Используя новое имя a, получим правило вывода грамматики $A \to B_x\, C_x\, D_z$, которое не требует правила вычисления атрибута x.

2. $A_x \to B_y\, \{f\}_z$

   $y, z \leftarrow x$

   Атрибутам можно дать одно имя, но оно должно быть x, для того чтобы не изменилось имя атрибута в левой части правила. Выполнив замену имен, получим правило вывода грамматики $A_x \to B_x\, \{f\}_x$, при этом отпадает необходимость в атрибутном правиле.

3. $A_{x,\,y} \to a\, B_z\, C_t$

   $y, z, t \leftarrow x$

   Атрибуты y, z, t, x имеют одно и то же значение, но им нельзя дать одно имя, поскольку нетерминал в левой части правила вывода имеет два атрибута: x и y. В данном случае можно получить лишь частичное упрощение:

   $A_{x,\,y} \to a\, B_y\, C_y$

   $y \leftarrow x$.

Новый способ записи имен атрибутов позволяет непосредственно превращать списки атрибутов нетерминальных символов грамматики в списки параметров процедур для распознавания нетерминальных символов. Такой способ записи имеет недостаток, заключающийся в том, что из него не видно, каким образом между атрибутами передается информация. Например, из правила

$$A \to B\, C_x\, D_x$$

не ясно, присваивается ли атрибут нетерминала C атрибуту нетерминала D или наоборот. Если атрибут нетерминала C присваивается атрибуту нетерминала D, то атрибутное правило является L-атрибутным и может использоваться при реализации перевода методом рекурсивного спуска. В противном случае метод рекурсивного спуска неприменим.

Обратившись к описанию типов атрибутов, можно определить порядок передачи информации между атрибутами (значение синтезированного атрибута должно присваиваться унаследованному атрибуту). Однако на практике более удобно использовать обычный способ именования атрибутов и переходить к новому способу записи только после того, как будет доказано, что исходная АТ-грамматика является L-атрибутной.

Замечание

Для метода рекурсивного спуска не требуется, чтобы АТ-грамматика, описывающая перевод, имела форму простого присваивания.

Опишем детально, как необходимо расширить метод рекурсивного спуска, чтобы он выполнял атрибутный перевод.

Во-первых, изменим процедуру `NextSymbol` таким образом, чтобы она читала очередной символ входной цепочки (лексему) и присваивала класс текущего входного символа переменной `ClassSymb`, а значение лексемы (если оно есть) — переменной `ValSymb`.

Правила составления процедур, приведенные в *разд. 7.5*, при условии, что:

- АТ-грамматика, описывающая перевод, является L-атрибутной;
- левые части правил и описания нетерминалов используют одни и те же имена атрибутов;
- для атрибутов, имеющих одно и то же значение, можно использовать одинаковые имена;

дополняются следующими правилами:

- *формальные параметры*. Список имен атрибутов, соответствующий вхождениям нетерминала в левые части правил вывода, становится списком формальных параметров соответствующей процедуры;
- *спецификации параметров*. Спецификации атрибутов (УНАСЛЕДОВАННЫЙ или СИНТЕЗИРОВАННЫЙ) переводятся в спецификации формальных параметров по следующим правилам:
  - тип УНАСЛЕДОВАННЫЙ соответствует способу передачи параметров "вызов по значению";
  - тип СИНТЕЗИРОВАННЫЙ соответствует способу передачи параметров "вызов по ссылке";
- *локальные переменные*. Все имена атрибутов символов данного правила грамматики, кроме тех, что связаны с символом из левой части, становятся локальными переменными соответствующей процедуры;
- *обработка нетерминала из правой части правила*. Для каждого вызова процедуры, соответствующего вхождению нетерминального символа в правую часть правила вывода, список атрибутов этого вхождения используется в качестве списка фактических параметров;

- *обработка входного символа*. Для каждого вхождения входного символа в правую часть правила вывода грамматики перед вызовом процедуры `NextSymb` в процедуру включается фрагмент кода, который каждой переменной из списка атрибутов входного символа присваивает значение входного атрибута из переменной `ValSymb`;
- *обработка операционного символа*. Для каждого вхождения операционного символа в правую часть правила вывода грамматики в процедуру включается фрагмент кода, который по соответствующим атрибутным правилам вычисляет значения синтезированных атрибутов операционного символа и присваивает вычисленные значения переменным, соответствующим синтезированным атрибутам. Затем вызывается процедура выдачи операционного символа вместе с атрибутами в выходную строку;
- *обработка правил вычисления атрибутов*. Для каждого правила вычисления атрибутов, сопоставленного правилу вывода грамматики, в процедуру включается фрагмент кода, который вычисляет значение атрибута и присваивает это значение каждой переменной из левой части атрибутного правила (если соглашение об одинаковых именах выполнено, то в левой части атрибутного правила будет только один атрибут). Фрагмент кода можно поместить в любом месте процедуры, которое находится:
  - после точки, где используемые в правиле атрибуты уже вычислены;
  - перед точкой, где впервые используется вычисленное значение атрибута;
- *головной модуль*. Все имена синтезированных атрибутов начального символа грамматики становятся локальными переменными головного модуля. Список фактических параметров вызова процедуры распознавания начального символа грамматики содержит *начальные значения унаследованных атрибутов* и *имена синтезированных атрибутов*.

Пример 12.1

🗁 В качестве примера реализации атрибутного перевода с использованием метода рекурсивного спуска рассмотрим L-атрибутную транслирующую грамматику, приведенную в *разд. 11.4.3*.

Для повышения наглядности переименуем атрибуты символов грамматики таким образом, чтобы всем вхождениям символов в правые части *разных* правил вывода соответствовали *разные имена атрибутов*, а также выберем одинаковые имена для атрибутов нетерминала R из левой части правил вывода с номерами (3), (4) и (5). Пусть p — унаследованный атрибут нетерминала R, а t — его синтезированный атрибут. После преобразования получим следующую грамматику:

E_t синтезированный t

$R_{p,\,t}$ унаследованный p синтезированный t

Атрибуты операционных символов унаследованные.

(0) $S0 \rightarrow S\perp$

(1) $S \rightarrow I_a := E_b \{:=\}_{a,\,b}$

(2) $E_c \rightarrow I_d\, R_{d,\,c}$

(3) $R_{p,\,t} \rightarrow + i_{q_1} \{+\}_{p,\,q_1,\,r_1} R_{r_1,\,t}$

$r_1 \leftarrow$ GETNEW

(4) $R_{p,\,t} \rightarrow \times i_{q_2} \{*\}_{p,\,q_2,\,r_2} R_{r_2,\,t}$

$r_2 \leftarrow$ GETNEW

(5) $R_{p,\,t} \rightarrow \varepsilon$

$t \leftarrow p$

С учетом выполненных преобразований процедура на языке Pascal, реализующая атрибутный перевод операторов присваивания некоторого языка программирования в цепочку тетрад с кодами операций: СЛОЖИТЬ, УМНОЖИТЬ, ПРИСВОИТЬ методом рекурсивного спуска, приведена в листинге 12.2.

Листинг 12.2

```
Procedure Recurs_Method_ATG (List_Token: tList, List_Tetr: tListTetr);
                            {List_Token – входная цепочка,
                             List_Tetr – цепочка тетрад}
   Procedure Proc_S;
   begin
         if ClassSymb = ClassId {текущий символ – идентификатор} then
         begin
            a := ValSymb;
            NextSymb;
            if ClassSymb = ':=' then
            begin
               NextSymb;
               Proc_E(b);
               Output({:=}, a, b)
            end
            else
               Error
         end
         else
            Error
```

```
   end; {Proc_S}
   Procedure Proc_E(var c: integer);
      var d: integer; {локальная переменная Proc_E}
      begin
      if ClassSymb = ClassId {текущий символ – идентификатор} then
      begin
         d := ValSymb;
         NextSymb;
         Proc_R(d,c)
      end
      else
         Error
end; {Proc_E}
   Procedure Proc_R(p: integer, var t: integer);
      Var q1, q2, r1, r2: integer; {локальные переменные Proc_R}
      begin
         if ClassSymb = '+' then
         begin
            NextSymb;
            if ClassSymb = ClassId {текущий символ – идентификатор} then
            begin
               q1 := ValSymb;
               NextSymb;
               GetNew(r1);
               Output({+}, p, q1, r1);
               Proc_R(r1, t)
            end
         else
            Error;
         end
         else
         if ClassSymb = '*' then
         begin
            NextSymb;
            if ClassSymb = ClassId {текущий символ – идентификатор} then
            begin
               q2 := ValSymb;
               NextSymb;
               GetNew(r2);
               Output({*}, p, q2, r2);
               Proc_R(r2, t)
            end
         else
            Error;
         end
```

```
         else
            t := p
   end; {Proc_R}
begin
   { В начале анализа переменная ClassSymb содержит класс первого символа
     входной цепочки, а переменная ValSymb - значение этого символа }
   Proc_S;
       if ClassSymb = '⊥' then
          Access
       else
          Error
end {Recurs_Method_ATG};
```

На рис. 12.5 приведен список имен процедур со значениями фактических параметров в порядке их вызова при переводе входной цепочки $i_5 = i_1 + i_2 * i_3 \perp$ в цепочку тетрад. Справа в строке, соответствующей вызову процедуры `NextSymb`, изображена непрочитанная часть входной цепочки (текущий символ находится слева) непосредственно после вызова этой процедуры. □

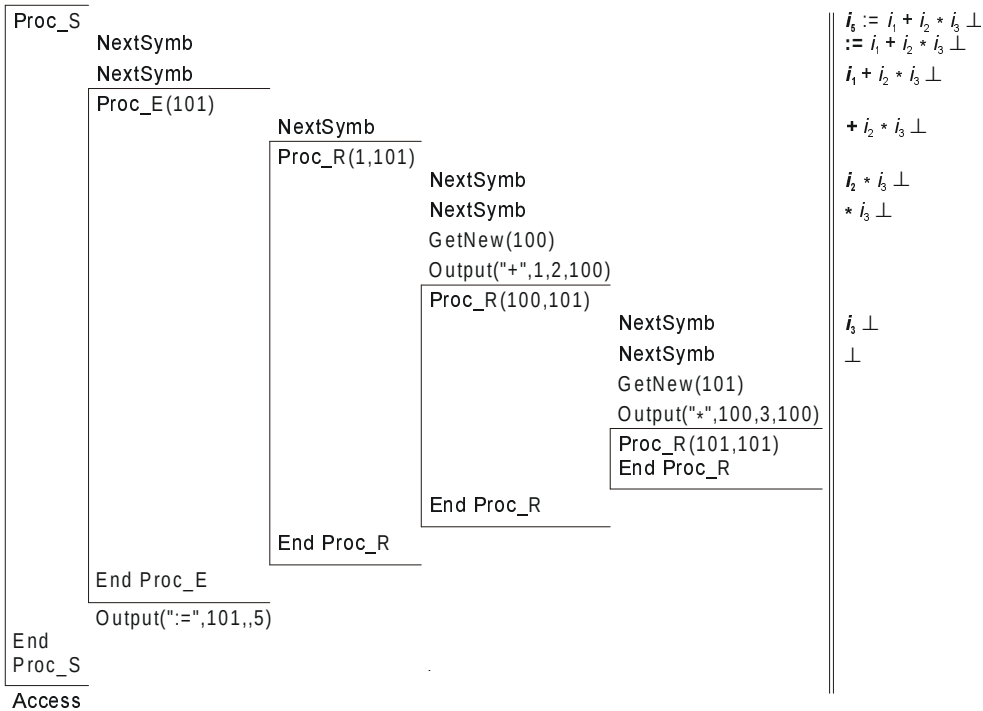


Рис. 12.5. Порядок вызова процедур при переводе цепочки $i_5 := i_1 + i_2 * i_3 \perp$

12.4. *S*-атрибутный ДМП-процессор

12.4.1. Математическая модель восходящего ДМП-процессора

Для любого восходящего синтаксического анализатора, рассматриваемого в данном учебнике, последовательность операций переноса и свертки, выполняемых при обработке допустимых входных цепочек, можно описать с помощью транслирующей грамматики. Входной для этой транслирующей грамматики является грамматика, на основе которой построен анализатор. Для получения транслирующей грамматики в самую крайнюю правую позицию каждого *i*-го правила входной грамматики вставляется операционный символ {*СВЕРТКА*, *i*}. Например, транслирующая грамматика, построенная по входной грамматике $G_0 = (\{E, T, P\}, \{i, +, *, (,)\}, P, E)$, где $P = \{E \to E + T, E \to T, T \to T * P, T \to P, P \to i, P \to (E)\}$, будет выглядеть следующим образом:

(1) $E \to E + T$ {*СВЕРТКА*, 1}

(2) $E \to T$ {*СВЕРТКА*, 2}

(3) $T \to T * P$ {*СВЕРТКА*, 3}

(4) $T \to P$ {*СВЕРТКА*, 4}

(5) $P \to i$ {*СВЕРТКА*, 5}

(6) $P \to (E)$ {*СВЕРТКА*, 6}

Если каждый входной символ в активной цепочке интерпретировать как представление операции *ПЕРЕНОС*, выполняемой в момент времени, когда этот символ является текущим входным символом, то активная цепочка в точности описывает последовательность операций переноса и свертки, выполняемых при обработке входной цепочки. Это объясняется тем, что каждая операция свертки выполняется сразу же после того, как локализована соответствующая *основа* (или первичная фраза), т. е. когда завершается обработка последнего символа в правой части правила вывода. Например, для входной цепочки $i + i * i$, разбор которой рассматривался в *разд. 9.4*, активная цепочка, порождаемая рассмотренной ранее транслирующей грамматикой, имеет вид:

$$i\{СВЕРТКА\_6\} + i\{СВЕРТКА\_6\} * i\{СВЕРТКА\_6\}\{СВЕРТКА\_3\}\{СВЕРТКА\_1\},$$

что полностью соответствует последовательности операций переноса и свертки, выполняемых при обработке входной цепочки.

Восходящий анализатор можно расширить действиями по выполнению перевода, если перевод определяется постфиксной транслирующей грамматикой. Модификация анализатора заключается в том, что операция свертки

расширяется действиями, определяемыми операционными символами соответствующего правила грамматики. Это можно сделать, т. к. при обработке входной цепочки момент времени выполнения свертки для каждого правила грамматики совпадает с моментом выполнения действий по переводу для этого правила. Например, для постфиксной транслирующей грамматики цепочечного перевода:

$E \rightarrow E + T \{+\}$

$E \rightarrow T$

$T \rightarrow T * P \{*\}$

$T \rightarrow P$

$P \rightarrow i \{i\}$

$P \rightarrow (E)$

операции свертки расширяются следующим образом: *СВЕРТКА*_1 будет обеспечивать выдачу операционного символа {+} в выходную строку, *СВЕРТКА*_3 — выдачу операционного символа {*}, а *СВЕРТКА*_6 — выдачу операционного символа {*i*}.

Синтаксический анализатор, дополненный формальными действиями по выполнению перевода, принято называть восходящим ДМП-процессором.

12.4.2. Реализация *S*-атрибутного ДМП-процессора

Восходящий ДМП-процессор можно легко преобразовать в *S*-атрибутный ДМП-процессор, реализующий атрибутный перевод, определяемый постфиксной *S*-атрибутной транслирующей грамматикой.

В *S*-атрибутном ДМП-процессоре каждый магазинный символ имеет конечное множество полей для представления атрибутов. Так же, как и в *L*-атрибутном ДМП-процессоре, примем, что магазинный символ с n атрибутами представляется в магазине $(n + 1)$-ой ячейками, верхняя из которых содержит имя символа, а остальные — поля для атрибутов. Поля магазинного символа, предназначенные для атрибутов, *заполняются* значениями атрибутов в момент *вталкивания символа в магазин* и не изменяются до момента выталкивания его из магазина.

В *S*-атрибутном ДМП-процессоре *операция переноса* расширяется таким образом, что значения атрибутов переносимого входного символа помещаются в соответствующие поля вталкиваемого при переносе магазинного символа.

При выполнении *операции свертки* для правила с номером i верхние символы магазина представляют собой правую часть i-го правила вывода входной грамматики, а поля магазинных символов содержат значения атрибутов соответствующих символов грамматики.

Расширенная операция свертки использует эти значения для вычисления значений всех атрибутов операционных символов, связанных с правилом вывода транслирующей грамматики, и значений всех атрибутов нетерминала из левой части правила. *Значения атрибутов операционных символов* используются для выдачи результатов в выходную ленту или выполнения других действий, определяемых этими символами. *Атрибуты нетерминала из левой части правила вывода* записываются в соответствующие поля магазинного символа, который соответствует этому нетерминалу и вталкивается в магазин во время свертки.

На рис. 12.6 изображена последовательность состояний магазина S-атрибутного ДМП-процессора, осуществляющего перевод цепочки $i_2 + i_3 \times i_5$ в последовательность тетрад со знаками операций СЛОЖИТЬ и УМНОЖИТЬ. S-атрибутная транслирующая грамматика, входной грамматикой которой является остовная грамматика G_{0S} (*см. разд. 9.5*), имеет вид:

E_p — СИНТЕЗИРОВАННЫЙ p

(1) $E_{p_2} \rightarrow E_{q_1} + E_{t_1}\ \{+\}_{q_2,\ t_2,\ p_1}$

$q_2 \leftarrow q_1$

$t_2 \leftarrow t_1$

$p_1, p_2 \leftarrow$ GETNEW

(3) $E_{p_2} \rightarrow E_{q_1} * E_{t_1}\ \{*\}_{q_2,\ t_2,\ p_1}$

$q_2 \leftarrow q_1$

$t_2 \leftarrow t_1$

$p_1, p_2 \leftarrow$ GETNEW

(5) $E_{p_2} \rightarrow i_{p_1}$

$p_2 \leftarrow p_1$

(6) $E_{p_2} \rightarrow (E_{p_1})$

$p_2 \leftarrow p_1$

На рис. 12.6, *а* приведено начальное состояние магазина. Рис. 12.6, *б* представляет результат операции *ПЕРЕНОС* входного символа i с атрибутом 2 в магазин, а рис. 12.6, *в* — результат выполнения операции *СВЕРТКА*, выполняемой в соответствии с правилом (5). Рис. 12.6, *г* и *д* иллюстрируют результат выполнения операции *ПЕРЕНОС* символов '+' и i_3 соответственно из входной строки в магазин, а рис. 12.6, *е* — результат выполнения операции *СВЕРТКА* символа i_3 в E_3. Аналогично выполняются операции переноса и свертки для символов '*' и i_5 (рис. 12.6, *ж*—*е*).

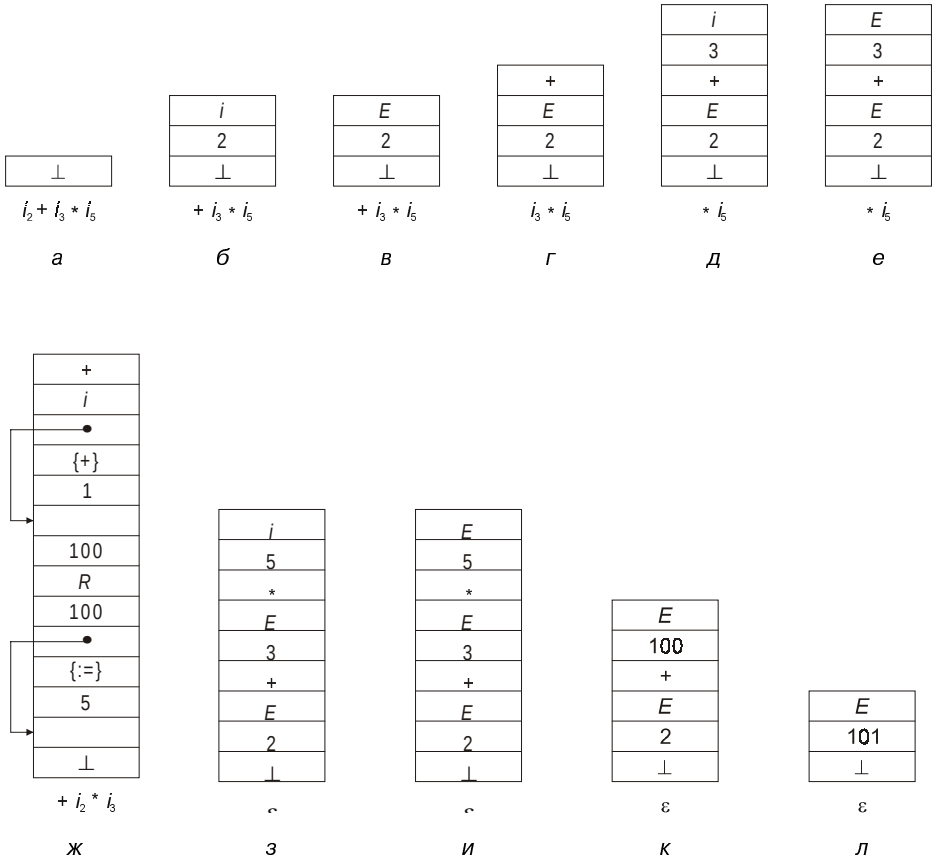


Рис. 12.6. Последовательность состояний магазина при обработке входной цепочки $i_2 + i_3 * i_5$

Рис. 12.6, *и* иллюстрирует результат выполнения операции *СВЕРТКА*, выполняемой в соответствии с правилом (1). При этом поля атрибутных символов из правой части этого правила используются для формирования тетрады:

*	3	5	100

,

которая выдается в выходную цепочку.

Далее выполняется операция *СВЕРТКА* по правилу (3) с выдачей тетрады:

+	2	100	101

(рис. 12.6, *к*—*л*).

Контрольные вопросы

1. Дайте определение L-атрибутной и S-атрибутной транслирующих грамматик.
2. Какое правило вычисления атрибутов называется копирующим?
3. В каком случае АТ-грамматика имеет форму простого присваивания? Приведите процедуру преобразования произвольной АТ-грамматики в форму простого присваивания.
4. Приведите алгоритм построения управляющей таблицы для транслирующей грамматики цепочечного перевода, входной грамматикой которой является $LL(1)$-грамматика.
5. Опишите, каким образом представляются в магазине и вычисляются унаследованные и синтезированные атрибуты символов грамматики в L-атрибутном ДМП-процессоре.
6. Приведите процедуру преобразования ДМП-процессора в L-атрибутный ДМП-процессор.
7. Как программируются процедуры в методе рекурсивного спуска, реализующего перевод, описываемый транслирующей грамматикой?
8. С какой целью осуществляется переименование имен атрибутов нетерминальных символов из левых частей правил вывода L-атрибутной транслирующей грамматики при реализации вывода L-атрибутного перевода методом рекурсивного спуска?
9. Как программируются процедуры в методе рекурсивного спуска, реализующего перевод, описываемый L-атрибутной транслирующей грамматикой?
10. Каким образом восходящий анализатор можно расширить действиями по выполнению перевода?
11. Опишите, каким образом представляются в магазине и вычисляются унаследованные и синтезированные атрибуты символов грамматики в S-атрибутном ДМП-процессоре.

Упражнения

1. Следующие правила вывода грамматики являются частью некоторой атрибутной грамматики. Атрибуты p, q и r — унаследованные, а s и t — синтезированные. Для каждого правила определите, от каких атрибутов могут зависеть правила вычисления p и r, чтобы это правило было L-атрибутным?
   1.1. $A_{s,\,q} \rightarrow a_u\ A_{t,\,p}\ B_r$.
   1.2. $A_{s,\,q} \rightarrow \{c\}_p\ b_u\ A_{t,\,r}$.
   1.3. $A_{s,\,q} \rightarrow c_u\ B_r\ A_{t,\,p}$.

2. Приведите следующие правила вывода АТ-грамматики к форме простого присваивания (имена унаследованных атрибутов начинаются с символа *i*, а имена синтезированных атрибутов — с символа *s*):

   2.1. $A_{s_1,\, i_1} \rightarrow E$

   $s_1 \leftarrow \sin(i_1)$

   2.2. $E_{s_1,\, i_1} \rightarrow A_{s_2}\, B_{s_3,\, i_2}\, C_{s_4,\, i_3}\, D_{s_5,\, i_4}$

   $i_2 \leftarrow i_1$

   $i_3 \leftarrow i_1 * i_1$

   $i_4 \leftarrow i_2 * i_3$

   $s_1 \leftarrow s_2 * s_2$

3. Преобразуйте *L*-атрибутную грамматику цепочечного перевода к форме простого присваивания и постройте для нее *L*-атрибутный ДМП-процессор.

   A_p СИНТЕЗИРОВАННЫЙ p

   B_p СИНТЕЗИРОВАННЫЙ p

   $\{b\}_p$ УНАСЛЕДОВАННЫЙ p

   (1) $S \rightarrow a_p\, A_q\, B_r\, A_s\, \{b\}_t$

   $r \leftarrow p + 2q$

   $t \leftarrow r + s$

   (2) $A_p \rightarrow a_q\, A_r\, B_s\, B_t\, B_u\, S$

   $s \leftarrow q + r$

   $t \leftarrow q * s - 4$

   $u, p \leftarrow r + t$

   (3) $A_p \rightarrow b$

   $p \leftarrow b$

   (4) $B_p \rightarrow a_q\, A_r\, \{b\}_t$

   $t \leftarrow r + q$

4. Для грамматики, заданной в упр. 2, постройте процессор методом рекурсивного спуска.

5. Для входной грамматики, описывающей синтаксис арифметических выражений, с правилами вывода

   $E \rightarrow (E + E)$

   $E \rightarrow (E * E)$

   $E \rightarrow i$

   где i — лексема, значением которой является указатель на элемент таблицы идентификаторов, определите постфиксную *S*-атрибутную грамматику, описывающую перевод этих выражений в цепочку тетрад с кодами операций: СЛОЖИТЬ и УМНОЖИТЬ, и постройте *S*-атрибутный ДМП-процессор, выполняющий заданный перевод.

Список литературы

1. Алгоритмический язык АЛГОЛ-60. Модифицированное сообщение / Пер. с англ. — М.: Мир, 1982.
2. Ахо А., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструменты. — М.: Издательский дом "Вильямс", 2001.
3. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Т. 1. — М.: Мир, 1978.
4. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Т. 2. — М.: Мир, 1978.
5. Ахо А. Индексные грамматики — расширение контекстно-свободных грамматик. / В сб. "Языки и автоматы". — М.: Мир, 1975.
6. Братчиков И. Л. Синтаксис языков программирования. — М.: Наука, 1975.
7. Вайнгартен Ф. Трансляция языков программирования. — М.: Мир, 1977.
8. Вебер Дж. Технология Java™ в подлиннике. — СПб.: БХВ-Петербург, 2001.
9. Гинзбург С. Математическая теория контекстно-свободных языков. — М.: Мир, 1977.
10. Гладкий А. В. Формальные грамматики и языки. — М.: Наука, 1973.
11. Грис Д. Конструирование компиляторов для цифровых вычислительных машин. — М.: Мир, 1975.
12. Гросс М., Лантен А. Теория формальных грамматик. — М.: Мир, 1971.
13. ГОСТ 23056-78 Язык программирования ФОРТРАН.
14. ГОСТ 23057-78 Язык программирования БАЗИСНЫЙ ФОРТРАН.
15. ГОСТ 27787-88 Язык программирования БЕЙСИК.
16. ГОСТ 27831-88 (ИСО 8652-87) Язык программирования АДА.
17. ГОСТ 27974-88 Язык программирования АЛГОЛ 68.
18. ГОСТ 27975-88 Язык программирования АЛГОЛ 68 расширенный.
19. ГОСТ 28140-89 Системы обработки информации. Язык программирования ПАСКАЛЬ.
20. Дейкстра Э. Дисциплина программирования. — М.: Мир, 1978.

21. Йенсен К., Вирт Н. Паскаль: Руководство для пользователя. / Пер. с англ. и предисл. Сальникова М. Л., Сальниковой Ю. В. — 1993.

22. Келли М., Спайс Н. Язык программирования ФОРТ / Пер. с англ. — М.: Радио и связь, 1993.

23. Кнут Д. Нисходящий синтаксический анализ. / Кибернетический сборник. — М.: Мир, 1978.

24. Кнут Д. Искусство программирования. Т. 1 Основные алгоритмы. / 3-е изд. — М.: Издательский дом "Вильямс", 2000.

25. Кнут Д. О переводе (трансляции) языков слева направо. / В сб. "Языки и автоматы". — М.: Мир.

26. Кнут Д. Семантика контекстно-свободных языков. / В сб. "Семантика языков программирования". — М.: Мир, 1980.

27. Кэнту М. Delphi 5 для профессионалов. — СПб.: Питер, 2001.

28. Лебедев В. Н. Введение в системы программирования. — М.: Статистика, 1975.

29. Льюис Ф., Розенкранц Д., Стирнз Р. Теоретические основы построения трансляторов. — М.: Мир, 1979.

30. Льюис Ф., Розенкранц Д., Стирнз Р. Атрибутные трансляции. / В сб. "Семантика языков программирования". — М.: Мир, 1980.

31. Маркотти М., Ледгард Х., Бохман Г. Формальные описания языков программирования. / В сб. "Семантика языков программирования". — М.: Мир, 1980.

32. Маурер У. Введение в программирование на языке ЛИСП / Пер. с англ. — М.: Мир, 1976.

33. Ноутон П., Шилдт Г. JavaTM2 / Пер. с англ. — СПб.: БХВ-Петербург, 2001.

34. Опалева Э. А., Самойленко В. П., Семенова О. Н. Формальные методы описания перевода. Учеб. пособие. — СПб.: СПбГЭТУ, 2000.

35. Опалева Э. А., Самойленко В. П. Формальные грамматики и распознающие автоматы. Учеб. пособие. — Л.: ЛЭТИ, 1991.

36. Опалева Э. А., Самойленко В. П., Семенова О. Н. Разработка языковых процессоров. Методические указания к курсовой работе. — СПб.: СПбГЭТУ, 2002.

37. Пратт Т., Зелковиц М. Языки программирования: реализация и разработка. — СПб.: Питер, 2001.

38. Рейуорд-Смит В. Дж. Теория формальных языков. Вводный курс. — М.: Радио и связь, 1988.

39. Себеста Р. У. Основные концепции языков программирования. / 5-е изд. Пер. с англ. — М.: Издательский дом "Вильямс", 2001.

40. Стобо Д. Язык программирования ПРОЛОГ / Пер. с англ. Волченкова Н. Г., Григорьева С. Г.; Под ред. Волченкова Н. Г. — М: Радио и связь, 1993.

41. Фельдман Дж., Грис Д. Системы построения трансляторов. / В сб. "Алгоритмы и алгоритмические языки", вып. 5. — М.: ВЦ АН СССР, 1971.

42. Фостер Дж. Автоматический синтаксический анализ. — М.: Мир, 1975.

43. Хоумер А., Крис У. Dynamic HTML. Справ. / Пер. с англ. — СПб.: Питер, 2000.

44. Хеллерман Х., Смит А. АПЛ/360: Программирование и применение / Пер. с англ. — М.: Машиностроение, 1982.

45. Хопгут Ф. Методы компиляции. — М.: Мир, 1972.

46. Хопкрофт Дж., Мотвани Р., Ульман Дж. Введение в теорию автоматов, языков и вычислений. / 2-е изд. Пер. с англ. — М.: Издательский дом "Вильямс", 2002.

47. Хомский Н. Три модели для описания языка. / Кибернетический сборник, вып. 2. — М.: ИЛ, 1962.

48. Хомский Н. Синтаксические структуры. / Новое в лингвистике, вып. 11. — М.: ИЛ, 1962.

49. Хомский Н. Формальные свойства грамматик. / Кибернетичекий сборник, новая серия, вып. 2. — М.: Мир, 1966.

50. Хювенен Э., Сеппенян Й. Мир Лиспа: В 2-х т. / Пер. с финск. — М.: Мир, 1990.

51. Шмидт В. Visual Basic 5.0. — М.: АБФ, 1997.

52. Эрли Дж. Эффективный алгоритм анализа контекстно-свободных языков / Языки и автоматы. — М.: Мир, 1975.

53. Эллис М., Строуструп В. Справочное руководство по языку программирования C++ с комментариями. — М.: Мир, 1992.

54. Языки программирования Ада, Си, Паскаль. Сравнение и оценка. / Под ред. Фьюэра А., Джехани Н./ Пер. с англ. Леонене И. А.; Под ред. [и предисл.] Леонаса В. В. — М.: Радио и связь, 1989.

55. Adobe System Inc., Postscript Language Reference Manual. — Addison-Wesley, Reading, MA, 1990.

56. American National Standard Programming Language PL/1. ANSY X3/53 — American National Standard Institute, New York, 1976.

57. American National Standard Programming Language C. ANSY X3/159 — American National Standard Institute, New York, 1989.

58. Appel A. W., McQueen D. B. Standard ML of New Jersy, in Third International Symp. on Programming Language Implementation and Logic programming, M. Wirsing (Ed.) — Springer-Verlag, New York, 1991.

59. Clocksin W. F., Mellish C. S. Programming in Prolog. — Epringer-Verlag, Berlin, 1987.

60. Gosling J., Joi B., Steel G. The Java Language Specification. — Addison-Wesley, Reading, MA, 1996.

61. Ingalls D. H. The Smalltalk-76 Programming System: Design and Implementation, ACM Symp. on the Principles of Programming Language. — January, 1995.

62. McCarthy J. LISP 1.5 Programmer's Manuel, 2nd edition. — MIT Press, Cambridge, MA — 1961.

63. Milner R., Tofte M., Harper R. The Definition of Standard ML. — MIT Press, Cambridge, MA — 1989.

64. Steel G. Common LISP: The Language, 2nd edition. — Digital Press, Bedford, MA, 1990.