Министерство образования и науки Российской Федерации
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

А.А. МАЛЯВКО

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

ФОРМАЛЬНЫЕ ЯЗЫКИ И МЕТОДЫ ТРАНСЛЯЦИИ

ЧАСТЬ 1

Утверждено
Редакционно-издательским советом университета
в качестве учебного пособия

НОВОСИБИРСК
2010

УДК 004.43(075.8)
М 219

Рецензенты: *А.В. Гунько*, канд. техн. наук, доц.;
Е.Л. Романов, канд. техн. наук, доц.

Работа подготовлена на кафедре
вычислительной техники для студентов IV курса АВТФ
(направление 230100 – Информатика и вычислительная техника)

Малявко А.А.

М 219 Системное программное обеспечение. Формальные языки и методы трансляции : учеб. пособие : в 3 ч. / А.А. Малявко. – Новосибирск: Изд-во НГТУ, 2010. – Ч. 1. – 102 с.

ISBN 978-5-7782-1429-3

В первой части рассмотрены процедурная и автоматная модели лексического анализа, изложены теоретические основы аппарата определения лексики (регулярные выражения) языков программирования, элементы теории конечных автоматов без памяти и методы ее практического применения для автоматизированного преобразования системы регулярных определений в лексический анализатор, способы организации информационных таблиц трансляторов, алгоритмы поиска в таблицах и пополнения таблиц.

Адресовано студентам старших курсов и аспирантам, а также преподавателям смежных дисциплин. Может быть полезно студентам и аспирантам ряда других технических специальностей, связанных с разработкой и использованием программного обеспечения.

УДК 004.43(075.8)

ISBN 978-5-7782-1429-3

ВВЕДЕНИЕ

Организация трансляторов – одно из направлений программирования, широко использующих формальные математические модели. Поэтому основная цель настоящего издания – дать сжатое изложение формальных методов описания лексики и синтаксиса языков программирования, сформулировать представление об основных алгоритмах лексического и синтаксического анализа.

Первая часть учебного пособия вводит в теорию формальных языков. Эта часть содержит описание основных механизмов процесса трансляции и раздел лексического анализа, включающий:

– определения основных понятий формальных языков и метаязыков, постановку задач акцепта и анализа;

– элементы теории конечных автоматов без памяти;

– методы преобразования совокупности правил описания лексики языка в детерминированный оптимальный распознаватель слов;

– методы организации таблиц слов и соответствующие алгоритмы поиска в таблицах / пополнения таблиц.

Во второй части пособия рассмотрены вопросы синтаксического анализа.

Третья часть посвящена семантическому анализу, генерации объектного кода и интерпретации.

Трансляторы: компиляторы и интерпретаторы

Все новые программы для компьютеров создаются с использованием уже существующих программ: текстовых редакторов, макропроцессоров (или препроцессоров), трансляторов, компоновщиков, отладчиков. На рис. В.1 показана более или менее типичная последовательность технологических операций по созданию и исполнению новой программы,

выполняемых с помощью уже существующих программ. Исполняемые программы показаны в виде блоков с двойными рамками, обрабатываемые ими данные – с одинарными.

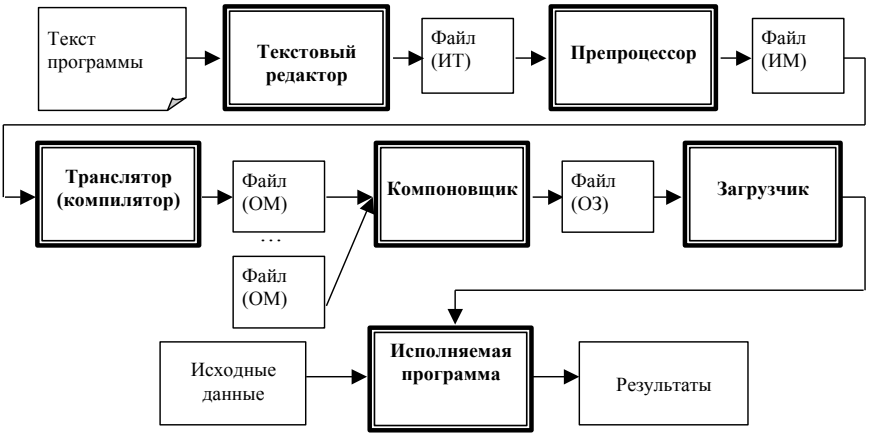


Рис. В.1. Технология компиляции: создание и исполнение программы

В показанной технологии каждый шаг выполняется однократно. На самом деле при создании любой реальной программы большинство из этих шагов, вероятнее всего, будет выполнено неоднократно. Кроме того, на рис. 1.1 не показан упомянутый ранее отладчик, без которого создание сколько-нибудь сложной программы трудно представить. Для удобства использования все программы, участвующие в процессе создания новых программ, обычно включаются в состав так называемых интегрированных оболочек (IDE – integrated development environment), облегчающих программисту взаимодействие с ними. Оболочка на рисунке тоже не показана. Естественно, и сама эта оболочка и все включенные в ее состав программы работают под управлением операционной системы компьютера.

Текст новой программы, написанной на языке программирования высокого уровня (например, Java или C#, или C++ …) для решения некоторой задачи, вводится с клавиатуры с использованием текстового редактора. Результат работы текстового редактора – исходный текст (ИТ) – запоминается в виде файла на диске. Этот файл затем может обрабатываться макропроцессором, реализующим макроподстановки и включение заголовочных файлов. Результатом этой обработки являет-

ся так называемый исходный модуль (ИМ). Далее транслятор осуществляет преобразование исходного модуля в так называемый объектный модуль (ОМ). Объектный модуль представляет собой файл программы, эквивалентной исходной, но содержащей смесь машинных команд с инструкциями компоновщику по «стыковке» с другими объектными модулями. Несколько объектных модулей, полученных в результате трансляции разных исходных модулей (выполнявшейся, возможно, на различных компьютерах в разных частях света и в разное время), преобразуется компоновщиком (другие названия – сборщик, линкер, редактор связей …) в файл образа задачи (ОЗ) и сохраняется им на диске. Теперь этот файл может быть использован независимо от тех программ, которые принимали участие в его создании, причем вполне вероятно – на других компьютерах. Загрузчик операционной системы при каждом запуске преобразует его в исполняемую программу, работающую под управлением и во взаимодействии с операционной системой. Исполняемая программа обрабатывает исходные данные (ИД) и формирует результаты решения задачи (Р).

Технология, показанная на рис. В.1 и называемая ***компиляцией*** (от слова «компилировать», т. е. собирать), отнюдь не является единственно возможной. Транслятор, реализующий в этой технологической цепочке перевод с одного языка на другой, называется, соответственно, компилятором. Показанный на рис. В.2 другой возможный способ получения результатов решения задачи путем прямого выполнения описанных в тексте исходной программы операций преобразования исходных данных, без создания и сохранения образа задачи называется ***интерпретацией***.

В этой технологии транслятор называется интерпретатором и выполняет функции как ввода/редактирования, макрообработки, собственно трансляции, так и пошагового исполнения (интерпретации) операций

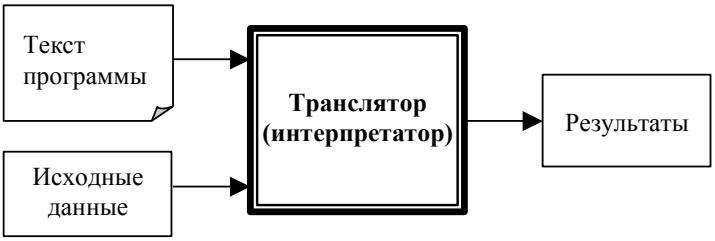


Рис. В.2. Технология интерпретации программы

по преобразованию исходных данных в результаты. Последовательность этих операций задана текстом исходной программы. Компоновщика в явном виде нет (неявно его функции также исполняет интерпретатор), сохраняемых файлов с объектными модулями или образом задачи также нет, следовательно, для запуска программы не используется загрузчик (напомним, что интерпретатор – это тоже исполняемая программа и для ее запуска загрузчик используется).

Широко применяется множество вариаций этих основных технологий, например:

− компиляция исходного текста программы в промежуточный псевдокод, сохраняемый на диске с последующим исполнением псевдокода интерпретатором (в настоящее время часто называемым виртуальной машиной);

− двухступенчатая (или более) компиляция с постепенным понижением уровня языка: язык высокого уровня – язык ассемблера – язык машинных команд.

В настоящем пособии рассмотрены теоретические основы процесса трансляции и методы автоматизации проектирования трансляторов. Дальнейшее изложение ориентировано на технологию компиляции Особенности реализаций как данной, так и других технологий при необходимости будут оговариваться особо.

В силу того, что транслятор – это инструмент для перевода текстов программ с одного формального языка на другой, определим основные понятия таких языков.

Элементарные понятия формальных языков

Простейшими элементами любого, в том числе формального, языка являются ***символы***. Все те символы, которые могут использоваться в данном языке для построения его слов, образуют множество (для множеств будем использовать обозначение { … }), называемое ***алфавитом*** языка. В алфавит языков программирования обычно входят все символы – как имеющие графическое изображение (буквы, цифры, …) и символы, так и не имеющие его (например, табуляция, возврат каретки, перевод строки, …). Обозначать произвольные символы принято малыми буквами латинского алфавита (либо просто использовать конкретный символ алфавита, например **%,** ^ или **5**).

Существуют формальные языки с более узким или, наоборот, более широким алфавитом. В качестве примера укажем язык двоичных чисел с алфавитом, содержащим ровно два символа {**0**, **1**}.

Цепочкой называется последовательность из нуля или более символов. Из символов любого алфавита можно образовать бесконечное множество цепочек. Приведем два примера цепочек, содержащих двоичные цифры: 100 010 и 10 101. Для обозначения произвольных цепочек принято использовать буквы греческого алфавита: **α, β, γ**, … **ω.**

Пустая цепочка, не содержащая ни одного символа, – очень важное понятие, которое в дальнейшем будет использоваться постоянно. Для пустой цепочки символов обычно используется обозначение **ε** (эпсилон), в некоторых источниках – **λ** (лямбда).

Формальным языком называется некоторое подмножество (возможно, бесконечное) бесконечного множества всех возможных цепочек символов алфавита. Способ образования этого подмножества, т. е. правила определения принадлежности цепочек к формальному языку, часто и называют этим языком. Цепочка символов называется ***правильной***, если она удовлетворяет этим правилам и, следовательно, принадлежит языку.

Существуют формальные языки, для которых любая цепочка, состоящая только из символов своего алфавита, является правильной. В качестве примера можно привести язык двоичных чисел или язык десятичных чисел с алфавитом {**0**,**1**,**2**,**3**,**4**,**5**,**6**,**7**,**8**,**9**}. Однако во всех языках программирования далеко не всякая цепочка символов алфавита языка является правильной, т. е. принадлежит этому языку.

Например, цепочка

class sampleOfEmptyClass{ }

является правильной программой языка Java (неважно, что в этой программе не определено никаких действий, по формальным критериям она правильна). Но если в этой цепочке переставить местами всего два символа (большую и маленькую буквы *c*), то получившаяся в результате перестановки цепочка

Class sampleOfEmptyclass{ }

уже не будет правильной в этом языке.

По аналогии с разговорными языками удобно считать, что любая правильная цепочка символов формального языка есть последовательность предложений, каждое из которых состоит из слов, а слова, в свою очередь, – из символов (рис. В.3).

Приведенная выше в качестве примера правильная цепочка символов языка Java может быть разложена на предложения и слова так, как показано на рис. В.4.

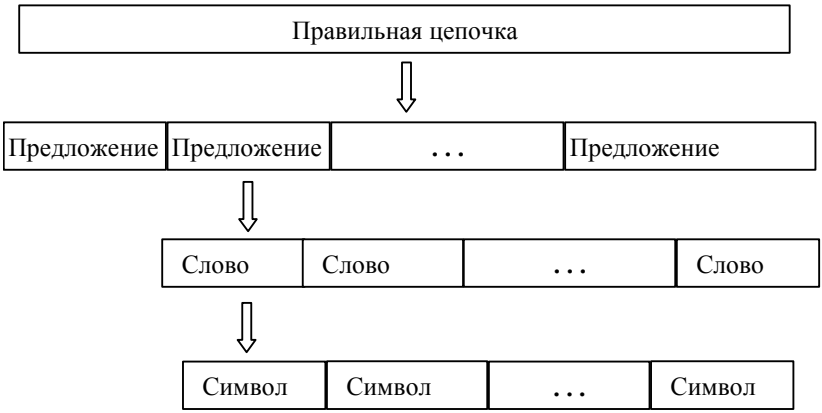


Рис. В.3. Структурирование правильных цепочек языка

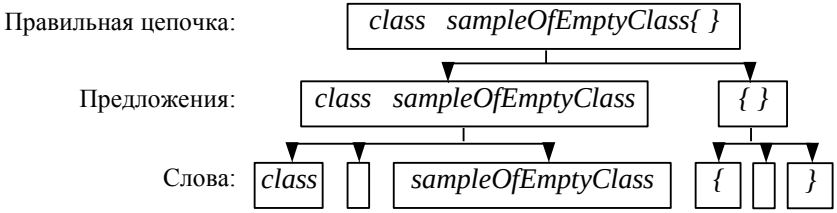


Рис. В.4. Пример разбиения цепочки на предложения и слова

Всю цепочку *Class sampleOfEmptyclass{ }* можно рассматривать как два предложения – заголовок объявления класса и тело класса.

Первое предложение состоит из ключевого слова *class*, за которым следует правильное слово, состоящее из трех пробельных символов, за которым, в свою очередь, следует слово *sampleOfEmptyClass* – имя объявляемого класса.

Второе предложение – тело класса (в данном случае пустое) также состоит из трех слов – открывающей фигурной скобки, одного пробела и закрывающей фигурной скобки.

Формальные языки, как и привычные разговорный или литературный языки, удобно определять с использованием трех систем правил: лексики, синтаксиса и семантики.

Совокупность правил, определяющих способы построения правильных слов из символов алфавита, есть ***лексика*** языка.

Совокупность правил образования предложений из слов и из более простых предложений есть ***синтаксис*** языка.

Взаимные связи между различными предложениями, а именно допустимость использования конкретных слов одновременно в нескольких различных предложениях, определяются совокупностью правил, называемых ***семантикой*** языка.

По мере необходимости для каждого этапа трансляции мы будем определять соответствующие системы правил, т. е. способы описания формальных языков и методы использования этих описаний как для собственно трансляции (перевода программ с одного языка на другой), так и для автоматизации построения трансляторов.

При описании любого языка приходится использовать либо тот же самый (как это делается при изучении русского языка в школе), либо некоторый другой язык. Языки, с помощью которых описывают (определяют) другие языки, принято называть метаязыками.

Для формальных языков, в частности языков программирования, точное описание необходимо как разработчикам транслятора, так и пользователям этого транслятора, т. е. программистам. Разработка программы становится бессмысленной, если программист и транслятор по-разному «понимают» одну и ту же конструкцию. Наличие точного описания – стандарта языка – позволяет программисту надеяться на получение желаемых результатов при разработке программы. Строгость и точность определения языка программирования становятся особо важными в следующих случаях:

- когда для одной аппаратно-программной платформы существует несколько различных трансляторов с одного и того же языка;
- при реализации одного и того же языка для одной аппаратной платформы, но под разные операционные системы;
- при реализации одного и того же языка для компьютеров с различной архитектурой.

Обычно такие случаи влекут за собой требование так называемой мобильности, т. е. возможности использования одних и тех же текстов программ с различными трансляторами, операционными системами и на различной аппаратуре.

Этапы процесса трансляции

Процесс трансляции программы обычно разделяют на четыре логически следующих друг за другом этапа: лексический анализ, синтаксический анализ, семантический анализ и генерация объектного кода

(рис. В.5). На каждом из этих этапов транслируемая программа преобразуется из одного промежуточного представления в другое.

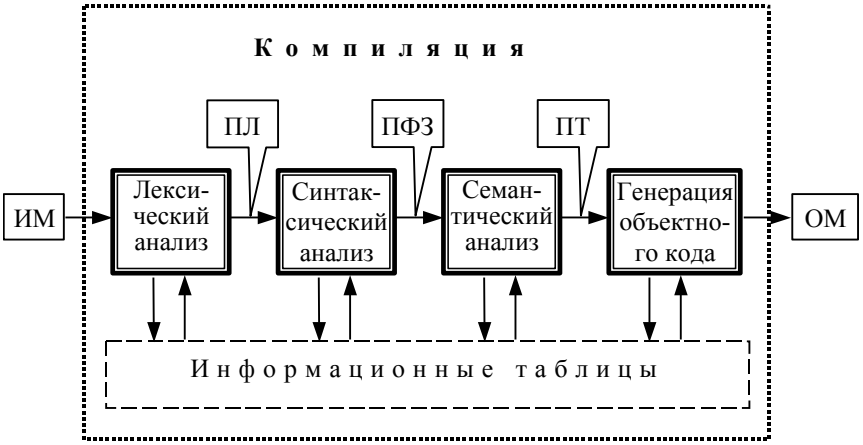


Рис. В.5. Логическая последовательность этапов компиляции

На этапе лексического анализа последовательность литер исходного модуля (ИМ) разбивается на слова языка, для каждого слова формируется его внутренний эквивалент – лексема (в некоторых источниках вместо термина «лексема» используется термин «токен»). Исходный модуль преобразуется из текстового представления в промежуточное представление, называемое последовательностью лексем (ПЛ). В результате лексического анализа создаются таблицы идентификаторов и констант как часть информационных таблиц транслятора.

На втором этапе, включающем синтаксический анализ, проверяется правильность предложений, образованных из слов и более простых предложений (одновременно определяются границы этих предложений). По мере обнаружения каждое предложение может преобразовываться в так называемую постфиксную запись (ПФЗ), которую также называют обратной польской записью. В этой форме записи знаки операций следуют после операндов, к которым эти операции должны применяться. Главное ее свойство состоит в том, что порядок появления знаков операций в ПФЗ совпадает с требуемой последовательностью их исполнения (следовательно, не нужны скобки для изменения этой последовательности, применяемые в привычной человеку инфиксной записи). При преобразовании условных операторов, операто-

ров цикла и других управляющих конструкций в ПФЗ транслятором могут создаваться новые слова (метки, знаки операций условного и безусловного перехода), которые заносятся в информационные таблицы.

В процессе семантического анализа (третьего этапа трансляции) проверяется применимость каждого знака операции к ее операндам. Постфиксная запись преобразуется в последовательность тетрад (ПТ) – так называемый псевдокод. Тетрада – это знак операции, наименования ее двух операндов и результата. На этапе семантического анализа, как правило, образуются новые слова для обозначения промежуточных результатов вычислений, которыми пополняются информационные таблицы.

На четвертом этапе псевдокод (с использованием накопленных в информационных таблицах сведений о программе) преобразуется в так называемый объектный модуль (ОМ) – окончательный результат процесса трансляции, но еще только полуфабрикат в процессе создания новой программы. При построении объектного модуля каждая тетрада заменяется ее машинным эквивалентом – одной или несколькими командами целевого процессора.

Приведем пример, поясняющий некоторые преобразования, выполняемые транслятором. Пусть в тексте программы на С-подобном языке содержится такой фрагмент:

*module = sqrt(a * a + b * b); //вычисление модуля вектора*

В результате лексического анализа будут выявлены последовательности литер, составляющих слова, эти слова будут преобразованы в лексемы, некоторые слова будут удалены и фрагмент примет вид

*module=sqrt(a*a+b*b);*

Комментарии и пробелы, призванные усилить зрительное восприятие текста, удалены, остались только значимые слова. Эти слова образуют правильное предложение – оператор присваивания. Заметим, что имя функции вычисления квадратного корня надо понимать как знак операции, аналогичный знакам «*», «+» и «=».Теперь этот фрагмент будет преобразован в постфиксную запись

*module a a * b b * + sqrt =*

В исходном фрагменте знаки операций были записаны в такой последовательности: *= sqrt * + *** («присвоить значение», «вычислить квадратный корень», «умножить», «сложить», «умножить»), однако

согласно общеизвестным правилам ясно, что выполнять их надо в таком порядке: * * + *sqrt* =(«умножить», «умножить», «сложить», «вычислить квадратный корень», «присвоить»). Именно этот порядок следования знаков операций выявлен и зафиксирован в постфиксной записи. Заметим, что в постфиксной записи не нужны скобки и ограничитель оператора присваивания – символ (слово) «точка с запятой».

Дальнейшее преобразование выполняется по ходу семантического анализа, выявляющего правомерность применения каждого знака операции к ее операндам. При этом оказывается необходимым явно назвать промежуточные результаты вычислений. Постфиксная запись превращается в последовательность тетрад:

*	*a*	*a*	*p1*
*	*b*	*b*	*p2*
+	*p2*	*p1*	*p3*
sqrt	*p3*	-	*p4*
=	*p4*	-	*module*

В этой последовательности (псевдокоде) точно определено, к каким именно операндам будет применяться каждая операция (в отличие от этого в вышеприведенной ПФЗ операндом знака операции *sqrt* является «результат сложения произведений *a* на *a* и *b* на *b*»). По ходу преобразования для каждой тетрады проверяется применимость знака операции к типам значений ее операндов и формируется тип данных результата операции.

Псевдокод может быть оптимизирован:

*	*a*	*a*	*p1*
*	*b*	*b*	*p2*
+	*p2*	*p1*	*p1*
sqrt	*p1*	-	*module*

При оптимизации уменьшилось количество как операций, так и переменных программы.

Далее псевдокод превращается генератором объектного кода компилятора в эквивалентную ему последовательность машинных команд. В интерпретаторе же осуществляется исполнение операций псевдокода над значениями операндов.

Реально выполняемая транслятором последовательность действий далеко не всегда совпадает с логической последовательностью этапов трансляции программы. Так, например, лексический анализатор обыч-

но работает по вызову из синтаксического анализатора и возвращает ему одну единственную лексему, или токен (рис. В.6). Следовательно последовательность лексем можно считать последовательностью результатов вызовов лексического анализатора. Таким образом, лексический анализ во времени протекает параллельно с синтаксическим. Как правило, семантический анализ реализуется в виде функций, вызываемых синтаксическим анализатором, и во времени также совмещается (а точнее – чередуется) с процессами синтаксического анализа.

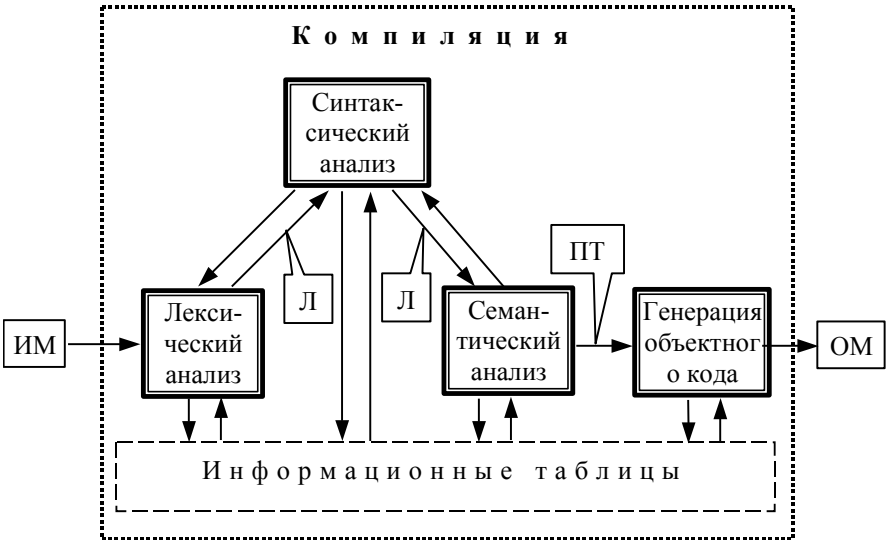


Рис. В.6. Реальная последовательность этапов компиляции

Однако генерация объектного кода обычно выполняется после полного завершения этапов лексического, синтаксического и семантического анализа. Причина этого состоит в том, что формировать объектный модуль имеет смысл только в том случае, если в программе не обнаружено ни лексических, ни синтаксических, ни семантических ошибок. При такой организации транслируемая программа (в исходном текстовом виде или в промежуточном представлении) полностью с начала до конца обрабатывается (проходится) два раза.

Первый проход выполняют лексический + синтаксический + семантический анализаторы, обрабатывающие исходный текст и формирующие последовательность тетрад – псевдокод.

Второй проход выполняет генератор объектного кода, преобразующий последовательность тетрад в объектный модуль.

В том случае, если компилятор выполняет оптимизацию формируемого объектного модуля, может потребоваться дополнительный проход, а при глубокой оптимизации – возможно, не один, а несколько.

Проектирование трансляторов

Создание транслятора для любого языка программирования – очень сложная задача. Существует большое количество инструментальных программных средств (пакетов), предназначенных для облегчения ее решения, т. е. для автоматизации проектирования трансляторов: Lex/Yacc, Flex/Bison, PCCTS, ANTLR, LLGEN, JavaCC и др. Такие пакеты осуществляют анализ формального описания языка программирования и преобразование этого описания в программу, являющуюся основой транслятора с этого языка. В некоторых пакетах задачи создания разных компонент транслятора функционально разделены. Например, Lex и Flex предназначены для автоматизации проектирования лексических, а Yacc и Bison – синтаксических анализаторов. Пользователю приходится прикладывать некоторые усилия для организации совместного функционирования результатов работы этих пакетов. В более поздних разработках сохранена возможность раздельного создания лексических и синтаксических анализаторов, но их корректное взаимодействие, как правило, обеспечивается средствами пакета и не требует участия пользователя.

Все пакеты позволяют пользователю решать задачи семантического анализа и генерации объектного кода или интерпретации путем включения дополнительной функциональности в синтаксический анализатор.

Многие современные трансляторы разработаны с использованием пакетов автоматизации проектирования. Поэтому изучение методов трансляции программ неотделимо от изучения методов создания трансляторов.

Здесь следует заметить, что методы и алгоритмы автоматизации проектирования трансляторов в принципе применимы и для разработки программ любого другого назначения. Действительно, транслятор – это программа, преобразующая исходные данные (текст программы на входном языке) в результаты (объектный модуль). В этом смысле

транслятор ничем не отличается от любой другой программы. Эффективные методы проектирования, разработанные для трансляторов, способны существенно сократить сроки и затраты на программирование при решении и других задач. Для этого требуется всего лишь формально определить язык исходных данных (его лексику, синтаксис и семантику) и использовать подходящий пакет автоматизации проектирования для получения программы, «понимающей» этот язык.

В настоящем учебном пособии для каждого этапа процесса трансляции рассматриваются способы определения соответствующих совокупностей правил – лексики, синтаксиса и семантики, т. е. формального описания языка программирования. Затем изучаются методы и алгоритмы преобразования формального описания языка в программную реализацию конечных автоматов, реализующих функции анализа и преобразования текста транслируемой программы. Рассматриваются алгоритмы функционирования акцепторов и анализаторов, методы оптимизации и генерации объектного кода. Изложение материала ведется в соответствии с ранее описанной логической последовательностью этапов процесса трансляции.

При рассмотрении прикладных аспектов используется учебный пакет автоматизации проектирования Вебтранслаб, созданный специально для визуализации технологий автоматизации разработки трансляторов и алгоритмов и методов анализа и преобразования текстов программ.

1. ЛЕКСИЧЕСКИЙ АНАЛИЗ

1.1. ПОСТАНОВКА ЗАДАЧИ

Начнем со следующей формулировки. Пусть дана цепочка, состоящая из символов алфавита некоторого языка. Необходимо выяснить, является ли она последовательностью правильных слов языка.

Решение задачи в такой постановке предполагает формирование результата как единственного бита информации, или, другими словами, связывания одного из двух логических значений «истина» или «ложь» с любой заданной цепочкой. Такая задача называется задачей лексического акцепта (от англ. accept – принимать). По существу, решение задачи лексического акцепта для всей цепочки символов сводится к циклическому повторению более простого действия – выяснению того, является ли начало цепочки правильным словом, и к «отбрасыванию» обнаруженного правильного слова, в результате чего цепочка укорачивается и в ее начале теперь находится следующее слово. До тех пор пока цепочка не пуста и ее начало есть правильное слово, процесс акцепта продолжается. Обнаружение такой последовательности символов, которая не является правильным словом, есть причина отказа от приема цепочки в целом, т. е. формирования для нее значения «ложь» в качестве результата. Исчерпание цепочки при условии, что все слова были правильными, есть основание для приема цепочки в целом, т. е. формирования значения «истина» в качестве результата работы.

Для последующих этапов трансляции важно не только (и не столько!) знать, содержит ли входная цепочка правильные слова. Значительно важнее, какие именно правильные слова обнаружены и в каком порядке они следуют друг за другом. По аналогии с естественными языками можно сказать, что правильные слова в предложении на фор-

мальном языке могут играть различные роли, наподобие подлежащих, сказуемых, дополнений и других членов предложения в естественных языках. Выяснение роли слова в предложении на любом, в том числе формальном, языке – задача более поздних этапов анализа, но для ее решения необходимо определить, какие роли данное слово может, в принципе, играть в предложениях языка. В естественном русском языке, например, никакой предлог не может играть роль подлежащего, так же как никакой глагол – роль дополнения, а вот роль сказуемого глагол играть может. Сейчас, по существу, мы использовали понятие групп слов, употребляя термины «предлог» и «глагол», но не указывая конкретно, какой именно предлог или какой именно глагол имеется в виду. Утверждение о том, какие роли могут, а какие не могут играть глаголы, относится ко всей группе слов.

Аналогичные утверждения справедливы и для формальных языков. Например, в С-подобных языках программирования в левой части оператора присваивания может находиться любой идентификатор, в то время как никакая константа не может играть эту роль. Однако в арифметических и логических выражениях как идентификаторы, так и константы могут быть использованы в качестве операндов разнообразных знаков операций. Константы и идентификаторы, таким образом, играют разные роли в предложениях языка.

Итак, по аналогии с естественными языками, для формальных языков необходимо разбить все правильные слова на группы в соответствии с той совокупностью ролей, которые эти слова могут играть в предложениях. Это разбиение осуществляется при создании языка, т. е. при формировании систем правил, описывающих его лексику, синтаксис и семантику. Количество различных групп слов и критерии отнесения каждого конкретного слова к той или иной группе зависят от множества факторов и не могут быть однозначно регламентированы. В зависимости от уровня рассмотрения (лексика, синтаксис или семантика) некоторые группы слов можно считать разбитыми на подгруппы. Фрагмент иерархии групп слов, типичной для многих языков программирования, показан на рис. 1.1.

Заметим, что все возможные группы слов можно отнести к одному из двух принципиально различных типов.

Предопределенными будем называть слова и, соответственно, группы слов, точный текст, смысл и возможные способы использования которых фиксируются в процессе разработки языка.

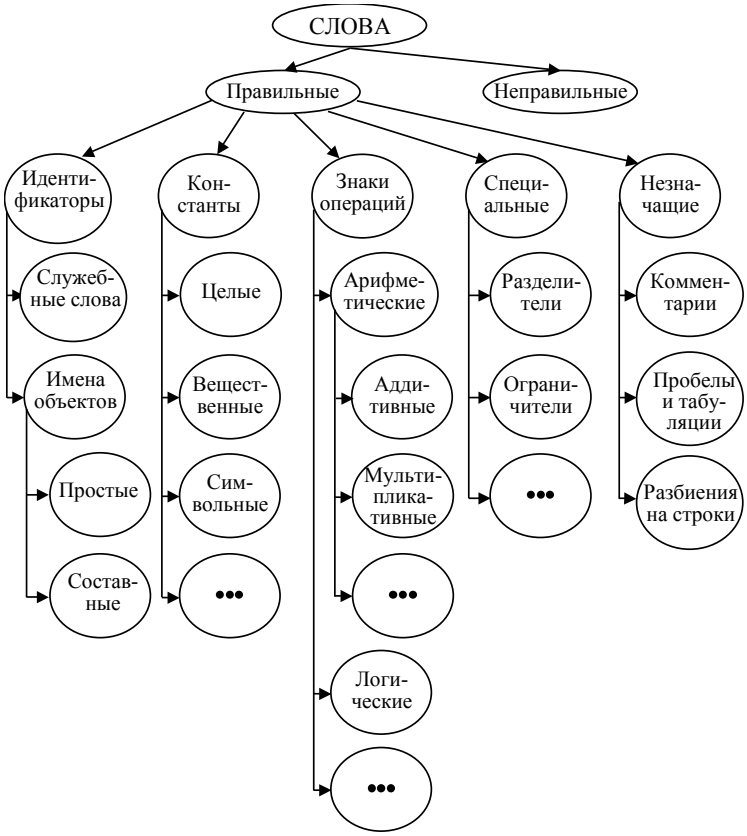


Рис. 1.1. Фрагмент иерархии групп слов в языках программирования

Так, например, в С-подобных языках предопределенным является слово, состоящее из одной литеры «;» и играющее роль ограничителя оператора. Другим примером является слово *else*. Эта цепочка из четырех литер по всем формальным признакам удовлетворяет правилам построения идентификаторов, но является служебным словом и не может быть использована в другом качестве. Однако известны языки, в которых способ использования слов зависит от окружающего их контекста (например, языки PL/1 или Алгол-68).

Перечень и смысл предопределенных слов языка, как правило, не может быть изменен в процессе трансляции программы.

Определяемыми называются слова, перечень и смысл которых выявляется именно в процессе трансляции данной программы. К определяемым словам обычно относятся идентификаторы и константы.

Задача определения группы или подгруппы (а следовательно, и перечня возможных ролей), к которой принадлежит каждое правильное слово в данной цепочке символов, должна решаться, очевидно, именно на этапе лексического анализа, т. е. тогда, когда выявляется, какая именно последовательность символов образует очередное правильное слово. В случае если группа содержит одно-единственное слово, на этом, т. е. на определении группы слов, функции этапа лексического анализа по обработке обнаруженного слова можно считать исчерпанными. Однако для групп, состоящих более чем из одного слова (а такие группы, безусловно, существуют, поскольку в любом языке в программе можно объявлять сколько угодно переменных), должна решаться задача сопоставления каждого обнаруженного слова с конкретным элементом группы. Результат этого сопоставления – уникальный в пределах группы номер слова – также следует вырабатывать на этапе лексического анализа путем поиска его текста в таблице, содержащей все слова данной группы. Для определяемых слов возможен отрицательный результат поиска, свидетельствующий о том, что это слово появилось в транслируемой программе впервые. В этом случае необходимо осуществлять пополнение таблицы для того, чтобы все последующие появления этого слова в тексте программы можно было отождествить с его первым вхождением.

Таблицы слов используются транслятором не только и не столько для учета слов. В них хранятся ***атрибуты*** слов, т. е. данные, необходимые для решения задач семантического анализа и для генерации объектного кода при компиляции. В интерпретаторах таблицы идентификаторов и констант используются для хранения текущих значений обрабатываемых данных.

Пара значений – код группы слов и номер слова в группе, которая формируется на этапе лексического анализа, является внутренним для транслятора эквивалентом слова и называется ***лексемой***. Для кода группы слов в литературе обычно используется специальный термин – ***токен***. Лексема более информативна, чем слово, поскольку и само слово и все атрибуты объекта, названного этим словом, можно извлечь из таблицы с помощью лексемы.

1.2. СТРУКТУРА И ФУНКЦИИ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА

Таким образом, лексический анализатор транслятора для каждого слова должен выполнять следующие функции.

1. Выявление цепочки символов, образующих правильное слово; определение токена, т. е. кода той группы слов, к которой это слово относится.

2. Выявление лексических ошибок, формирование диагностических сообщений, облегчающих поиск и устранение ошибок.

3. Поиск обнаруженного слова в таблице слов данной группы; пополнение таблицы определяемых слов при отсутствии слова в таблице; определение индекса строки как номера слова в группе.

4. Формирование лексемы как пары значений:

<токен, номер слова в группе>

Типичное разделение функций между компонентами лексического анализатора показано на рис. 1.2.

Здесь приведены логический состав и связи по данным (не по передачам управления) между компонентами лексического анализатора, выделенными на рисунке двойными рамками.

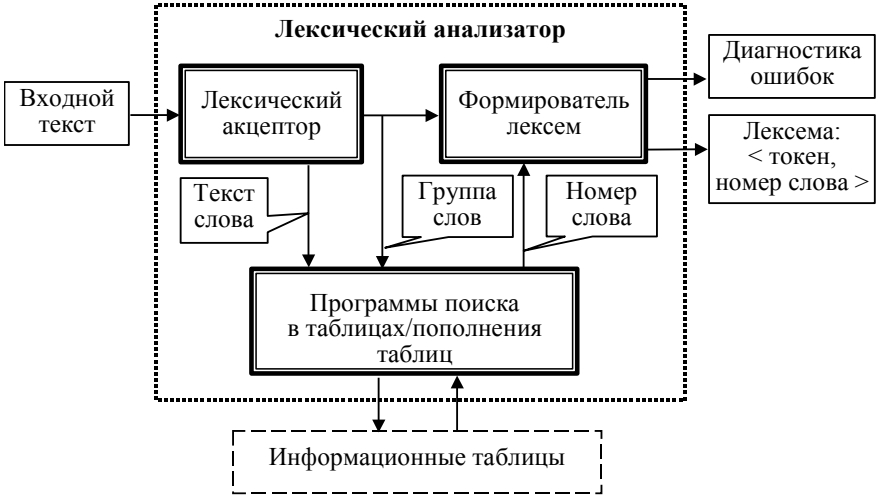


Рис. 1.2. Структура лексического анализатора

Предполагается, что лексический анализатор формирует одну лексему (или одно диагностическое сообщение об ошибке) в качестве результата одного запуска (вызова из синтаксического анализатора). Полное преобразование всего входного текста в последовательность лексем требует серии последовательных запусков анализатора до тех пор, пока не будет исчерпана входная цепочка символов. Начиная с этого момента анализатор при любом последующем запуске должен, очевидно, формировать специальную лексему, смысл которой эквивалентен понятию End of file.

Лексический акцептор осуществляет посимвольное чтение входного текста, формирование цепочки символов, составляющих текущее слово, и кода группы, к которой это слово относится (либо кода лексической ошибки, если обнаружена неправильная цепочка символов).

Если обнаружено правильное слово и группа, к которой оно относится, может содержать более одного слова, то осуществляется его поиск в соответствующей таблице. При обнаружении слова в таблице индекс строки, которая содержит это слово, передается формирователю лексем. При отсутствии слова в таблице реакция транслятора зависит от типа группы слов. Заметим, что для групп предопределенных слов в хорошо спроектированном трансляторе такая ситуация возникнуть не может. Лексический акцептор не может (точнее – не должен!) отнести к предопределенной группе слово, которого в ней нет. Поэтому далее эта ситуация не обсуждается. Отсутствие же только что обнаруженного слова в таблице определяемых слов есть нормальное явление и требует пополнения таблицы, т. е. занесения этого слова в таблицу. В результате пополнения образуется номер слова в группе, передаваемый формирователю лексем.

Код группы слов (токен) и для правильных слов – номер слова в группе используются формирователем лексем. При обнаружении лексической ошибки в конечном итоге должно быть сформировано диагностическое сообщение для пользователя транслятора с точным указанием позиции ошибочного символа во входном тексте. Диагностическое сообщение, возможно, будет формироваться вне лексического анализатора. В этом случае формирователь лексем должен возвратить лексему специального вида, информирующую синтаксический анализатор о зафиксированной лексической ошибке.

При обнаружении правильного слова формируется лексема как пара значений <токен, номер слова в группе>. Вследствие возможного существования подгрупп в некоторых группах, а также интерфейсных

соглашений с синтаксическим анализатором токены некоторых групп слов, вырабатываемые лексическим акцептором, могут затем преобразовываться формирователем лексем. Например, если общие правила образования слов группы «ключевые слова языка» совпадают с правилами образования слов группы «идентификаторы» (это характерно для многих языков), то лексический акцептор при обнаружении ключевого слова формирует токен идентификатора. Однако программа поиска в таблице и формирователь лексем должны заменить его на токен ключевого слова.

Существуют разные модели (т. е. способы реализации функций) лексического акцептора, в том числе – процедурная и автоматная. Рассмотрим вначале процедурную реализацию на примере лексического акцептора (и анализатора) объектно-ориентированного языка JavaScript, являющегося составной частью интерпретатора Rhino (носорог). Rhino написан на языке Java и является Open Source – программой, распространяемой на условиях лицензии MPL 1.1/GPL 2.0. Полные исходные тексты этого широко распространенного (встроенного в популярный браузер Mozilla Firefox) интерпретатора можно найти в Интернете по URL: https://developer.mozilla.org/en/Rhino (используя ссылку ftp://ftp.mozilla.org/pub/mozilla.org/js/rhino1_7R2.zip).

1.3. ПРОЦЕДУРНАЯ РЕАЛИЗАЦИЯ ЛЕКСИЧЕСКОГО АКЦЕПТОРА

Лексический анализ в интерпретаторе Rhino реализован в классе *TokenStream*. Код этого класса здесь приводится с некоторыми сокращениями, существо выполняемых действий описывается с помощью добавленных комментариев. Все оригинальные тексты программ, в том числе оригинальные комментарии, приведены курсивом, все добавленные комментарии – прямым шрифтом. Все удаленные фрагменты кода обозначены строкой:

… //опущено: <краткое описание удаленного фрагмента>

Метод *getToken* класса *TokenStream* – это собственно лексический акцептор интерпретатора Rhino. Он вызывается без параметров, читает литеры со входа, вызывая метод *getChar* , распознает очередное слово и возвращает не лексему, а ее часть – целочисленный код группы слов, или токен. (Отметим, что существует класс *Token*, содержащий только статические финальные поля и методы и позволяющий получать до-

полнительную информацию о словах языка по коду токена.) Для некоторых групп слов, таких как *identifier, keyword, number*, … метод *GetToken* запоминает слово в символьном массиве *stringBuffer* (частное поле класса *TokenStream*, см. в «Тексте слова» на рис. 1.2). При необходимости (как правило – в зависимости от того, какой *token* получен от метода *getToken)*, синтаксический анализатор интерпретатора Rhino может получить текст слова в строковом представлении, вызвав метод *getStringFromBuffer*.

Из метода *getToken* вызываются методы классов *Token, Kit, allStrings, ScriptRuntime* пакета Rhino и статические методы классов *Character, Double* пакета java.lang.

Вот код метода *getToken*:

```
…     //опущено: блок комментариев описания лицензии
package org.mozilla.javascript;
import java.io.*;
/**
 * This class implements the JavaScript scanner.
 *
 * It is based on the C source files jsscan.c and jsscan.h
 * in the jsref package.
 *
 * @see org.mozilla.javascript.Parser
 *
 * @author Mike McCabe
 * @author Brendan Eich
 */
class TokenStream {

//опущено: методы, прямо не связанные с изучаемым материалом

final int getToken() throws IOException {     //прочитать очередное слово
          //и вернуть код группы слов token
  int c;  //локальная переменная, хранящая код текущей литеры
retry:
  for (;;) {          //основной цикл чтения литер очередного слова
     for (;;) {       //подавить все незначащие символы кроме
        //EOF (EndOfFile) и EOL(EndOfLine)
        c = getChar();     //прочитать очередную литеру; заметим, что
        // литера возврата каретки '\r' подавляется методом getChar
```

```
        if (c == EOF_CHAR) {
            return Token.EOF;       //вернуть лексему EOF
        } else if (c == '\n') {
            dirtyLine = false;      //строка текста не содержала ничего,
        //кроме незначащих символов
            return Token.EOL;       //вернуть лексему EOL
        } else if (!isJSSpace(c)) { //частный метод isJSSpace возвращает
        //истину, если его аргумент – пробел (0x20), табуляция (0x9),
        //литера с одним из кодов 0xC, 0xB, 0xA0 или имеет категорию
        //Zs (SPACE_SEPARATOR) согласно спецификации Unicode
            if (c != '-') {     //возможно, обрабатывается конец комментария
        //в стиле HTML: -->
                dirtyLine = true;
            }
            break;
        }
    }
// identifier/keyword/instanceof? watch out for starting with a <backslash>
    boolean identifierStart;        //флажок начала идентификатора (или
      //служебного слова или instanceof)
    boolean isUnicodeEscapeStart = false;       //флажок начала
      //escape-последовательности
    if (c == '\\') {            //если первая литера – это \
      c = getChar();    //читать следующую литеру
      if (c == 'u') {       //и если это литера u
        identifierStart = true;  //то взвести флажки
        isUnicodeEscapeStart = true;
        stringBufferTop = 0;    //и подготовить символьный буфер
      //к накоплению литер слова
      } else {    //иначе сбросить флажки, возвратить прочитанную
      //литеру на вход
        identifierStart = false;
        ungetChar(c);
        c = '\\';           //и сделать вид, что со входа ничего не читалось
      }
    } else {        //если литера не «\», то проверить, может ли с нее
      //начинаться идентификатор, если да – установить флажок
      identifierStart = Character.isJavaIdentifierStart((char)c);
      if (identifierStart) {
```

```
        stringBufferTop = 0;    //подготовить буфер к приему слова
        addToString(c);         //и записать первую литеру в буфер
    }
}
if (identifierStart) {  //если была прочитана первая литера
    //идентификатора/ключевого слова/…
    boolean containsEscape = isUnicodeEscapeStart;
    for (;;) {              //цикл обработки остальных литер слова
        if (isUnicodeEscapeStart) {     //если было прочитано начало
    //escape-последовательности
            // strictly speaking we should probably push-back
            // all the bad characters if the <backslash>uXXXX
            // sequence is malformed. But since there isn't a
            // correct context(is there?) for a bad Unicode
            // escape sequence in an identifier, we can report an error here.
            int escapeVal = 0;  //переменная для Unicode-символа
            for (int i = 0; i != 4; ++i) {  //цикл чтения / 4 литер
                c = getChar();
                escapeVal = Kit.xDigitToInt(c, escapeVal);//преобразование
                // Next check takes care about c < 0 and bad escape
                if (escapeVal < 0) { break; }
            }
            if (escapeVal < 0) {            //если код отрицателен
                parser.addError("msg.invalid.escape");
                return Token.ERROR;     //то возврат ошибки
            }
            addToString(escapeVal);     //занесение символа в буфер
            isUnicodeEscapeStart = false;       //и сброс флажка
        } else {
            c = getChar();      //прочитать следующую литеру
            if (c == '\\') {        //если это обратный слэш «\»
                c = getChar();  //то прочитать еще одну литеру
                if (c == 'u') {     //внутри идентификатора она должна
    //быть литерой u
                    isUnicodeEscapeStart = true;  //установить флажки
                    containsEscape = true;
                } else {    //если это не литера u
                    parser.addError("msg.illegal.character");
                    return Token.ERROR;         //то вернуть ошибку
```

```
                }
            } else {
            if (c == EOF_CHAR || !Charater.isJavaIdentifierPart((char)c))
                {           //если текущая литера не принадлежит
                   break;  //текущему идентификатору, то выйти из цикла
                }
                addToString(c);    //добавить текущую литеру к буферу
             }
          }
        }
        ungetChar(c);      //возвратить на вход первую литеру слова,
        //следующего за идентификатором
        String str = getStringFromBuffer(); //получить слово как String
        if (!containsEscape) {       //были ли в слове
        //escape-последовательности? Если нет:
           // OPT we shouldn't have to make a string (object!) to
           // check if it's a keyword.
           // Return the corresponding token if it's a keyword
           int result = stringToKeyword(str);        //получить код
          //ключевого слова или Token.EOF, если слово не ключевое
           if (result != Token.EOF) {
//опущено: обработка ключевых слов, возможность употребления
//которых зависит от версии интерпретатора
           }
        }
        this.string = (String)allStrings.intern(str);    //вызывается метод
        //класса allStrings, проверяющий наличие прочитанного
        //идентификатора в рандомизированной таблице всех строк
        //обрабатываемой программы и добавляющий идентификатор
        //в таблицу при самом первом обнаружении его в тексте.
        return Token.NAME;        //код группы слов «идентификаторы»
    }
//конец обработки идентификаторов
    // is it a number?
    if (isDigit(c) || (c == '.' && isDigit(peekChar()))) {  //если текущая
        //литера – цифра или текущая литера – точка,
        //а следующая литера – цифра
        stringBufferTop = 0;        //подготовить буфер
        int base = 10;      //установить основание системы счисления = 10
```

```
if (c == '0') {       //если первая литера – цифра 0
   c = getChar();             //прочитать вторую литеру
   if (c == 'x' || c == 'X') {          //если это латинская x или X
      base = 16;       //то основание = 16
      c = getChar(); //прочитать первую литеру 16-ричного числа
  } else if (isDigit(c)) { //иначе, если после литеры 0 следует цифра
      base = 8;        //то основание = 8
   } else {            //иначе, запомнить лидирующий 0 в буфере
      addToString('0');
   }
}
if (base == 16) {             //если основание 16
   while (0 <= Kit.xDigitToInt(c, 0)) {      //то до тех пор,
//пока на входе 16-ричные цифры
      addToString(c);       //накапливать их в буфере
      c = getChar();        //читать следующую литеру
   }
} else {
  while('0' <= c && c <= '9') { //пока на входе десятичные цифры
      /*
       * We permit 08 and 09 as decimal numbers, which
       * makes our behavior a superset of the ECMA
       * numeric grammar.  We might not always be so
       * permissive, so we warn about it.
       */
      if (base == 8 && c >= '8') { //если основание = 8
//и очередная цифра > 7
       parser.addWarning("msg.bad.octal.literal",c =='8'?"8" : "9");
       base = 10; //добавить предупреждение и установить
//основание = 10
      }
      addToString(c);       //запоминать цифры в буфере
      c = getChar();        //читать следующую литеру
   }
}
boolean isInteger = true;             //флажок «целое» по умолчанию
if (base==10 && (c == '.' || c == 'e' || c == 'E')) {//если основание
//10 и очередная литера – это точка или буква e или буква E
   isInteger = false;        //сбросить флажок «целое»
```

```
            if (c == '.') {    //если после целой части точка
                do { //то все последующие цифры запомнить в буфере
                    addToString(c);
                    c = getChar();
                } while (isDigit(c));
            }
            if (c == 'e' || c == 'E') {          //если после целой или дробной
        //части следует e или E
                addToString(c);       //то признак экспоненциальной формы
        //запомнить в буфере
                c = getChar();        //прочитать следующую литеру
                if (c == '+' || c == '-') {       //если это знак экспоненты
                    addToString(c);   //то поместить его в буфер и прочитать
        //следующую литеру
                    c = getChar();
                }
                if (!isDigit(c)) {        //и проверить, что в экспоненте
        //есть хотя бы одна цифра
                    parser.addError("msg.missing.exponent");
                    return Token.ERROR;       //вернуть ошибку, если цифр нет
                }
                do {              //до тех пор, пока на входе цифры
                    addToString(c);   //запоминать их в буфере
                    c = getChar();      //и читать следующую литеру
                } while (isDigit(c));
            }
        }
        ungetChar(c);      //вернуть на вход литеру, следующую
        //после числового литерала
        String numString = getStringFromBuffer(); //получить строковое
        //представление литерала
        double dval;        //временная переменная для значения числа
//опущено: преобразование числового литерала во внутреннее
//представление числа
        this.number = dval;         //сохранить значение числа
        return Token.NUMBER;  //вернуть код группы слов «числа»
    }
        //конец обработки численных литералов
    // is it a string?
```

```
if (c == '"' || c == '\'') {       //если текущая литера – одиночный
   //апостроф или двойная кавычка
   // We attempt to accumulate a string the fast way, by
   // building it directly out of the reader.  But if there
   // are any escaped characters in the string, we revert to
   // building it out of a StringBuffer.
   int quoteChar = c;          // запомнить открывающую литеру
   //строки во временной переменной
   stringBufferTop = 0;//подготовить буфер для строкового литерала
   c = getChar();     //прочитать первую литеру литерала
strLoop: while (c != quoteChar) {    //до тех пор, пока текущая
   //литера не равна открывающей литере
      if (c == '\n' || c == EOF_CHAR) {       //если прочитан конец
   //строки или конец файла
         ungetChar(c);          //вернуть литеру на вход
         parser.addError("msg.unterminated.string.lit");
         return Token.ERROR;          //вернуть ошибку
      }
      if (c == '\\') {   //далее – обработка обратного слэша и
   //следующих(ей) за ним литер(ы)
         // We've hit an escaped character
         int escapeVal; //временная переменная для Unicode-символа
         c = getChar();          //прочитать следующую литеру
         switch (c) {  // сформировать новое значение переменной c
         case 'b': c = '\b'; break;       //забой
         case 'f': c = '\f'; break;       //вертикальная табуляция
         case 'n': c = '\n'; break;       //перевод строки
         case 'r': c = '\r'; break;       //возврат каретки
         case 't': c = '\t'; break;       //табуляция
         // \v a late addition to the ECMA spec,
         // it is not in Java, so use 0xb
         case 'v': c = 0xb; break;       //\v заменяется на 0xb согласно
   //спецификации ECMA
         case 'u':       //начало Unicode-литерала. Обработка похожа
   //на то, что делалось для числового литерала, но есть отличие:
   //если после \u не следует правильный Unicode-символ,
   //то вся последовательность, начиная с литеры u, запоминается
   //в буфере без преобразования
            // Get 4 hex digits; if the u escape is not
```

```
            // followed by 4 hex digits, use 'u' + the
            // literal character sequence that follows.
            int escapeStart = stringBufferTop;
            addToString('u');
            escapeVal = 0;
            for (int i = 0; i != 4; ++i) {
                c = getChar();
                escapeVal = Kit.xDigitToInt(c, escapeVal);
                if (escapeVal < 0) {       //если неверный Unicode-символ
                    continue strLoop;      //то продолжить внешний цикл
//обработки строкового литерала
                }
                addToString(c);
            }
            // prepare for replace of stored 'u' sequence by escape value
            stringBufferTop = escapeStart;     //если из предыдущего
//цикла выход выполнен не оператором continue, то поместить
//Unicode-символ в буфер вместо его литерала
            c = escapeVal;
            break;
        case 'x':       //обработка символьного литерала \xXX,
//приведено без доп. комментария
            // Get 2 hex digits, defaulting to 'x'+literal sequence, as above.
            c = getChar();
            escapeVal = Kit.xDigitToInt(c, 0);
            if (escapeVal < 0) {
                addToString('x');
                continue strLoop;
            } else {
                int c1 = c;
                c = getChar();
                escapeVal = Kit.xDigitToInt(c, escapeVal);
                if (escapeVal < 0) {
                    addToString('x');
                    addToString(c1);
                    continue strLoop;
                } else {            // got 2 hex digits
                    c = escapeVal;
                }
```

```
                }
                break;
            case '\n':      //не путать с 'n'
// Remove line terminator after escape to follow SpiderMonkey and C/C++
                c = getChar();      //читать следующую литеру, игнорируя
        //перевод строки после «\»
                continue strLoop;
            default:
                if ('0' <= c && c < '8') {   //обработка унаследованного из
        //языка C 8-ричного литерала вида\XXX
                    int val = c – '0';
                    c = getChar();
                    if ('0' <= c && c < '8') {
                        val = 8 * val + c – '0';
                        c = getChar();
                        if ('0' <= c && c < '8' && val <= 037) {
                            // c is 3rd char of octal sequence only
                            // if the resulting val <= 0377
                            val = 8 * val + c – '0';
                            c = getChar();
                        }
                    }
                    ungetChar(c);
                    c = val;
                }
            }
        }
        addToString(c);             //код литеры, возможно
    //сформированный выше, добавить в буфер
        c = getChar();              //прочитать следующую литеру
    }
    String str = getStringFromBuffer();         //получить строковое
    //представление литерала
    this.string = (String)allStrings.intern(str);    //добавить в таблицу
    return Token.STRING;    //код группы слов «строковый литерал»
}
if (c == '@') return Token.XMLATTR; //вернуть токен начала
      //XML – атрибута
switch (c) {            //этот переключатель обрабатывает
```

```
   //все остальные слова языка JavaScript
case ';':return Token.SEMI; //литера «;» – это отдельное слово языка
case '[': return Token.LB;    //литера «[» – также отдельное слово
case ']': return Token.RB;              // …
case '{': return Token.LC;              // …
case '}': return Token.RC;              // …
case '(': return Token.LP;              // …
case ')': return Token.RP;              // …
case ',': return Token.COMMA;           // …
case '?': return Token.HOOK;            // …
case ':':                  //с литеры «:» начинаются однолитерное слово :
   //и двухлитерное слово ::
   if (matchChar(':')) {         //метод matchChar читает со входа
   //следующую литеру и возвращает истину, если она совпадает
   //с его аргументом, иначе не делает ничего
      return Token.COLONCOLON;
   } else {
      return Token.COLON;
   }
case '.'://с точки начинаются слова DOT, DOTDOT и DOTQUERY
   if (matchChar('.')) {
      return Token.DOTDOT;
   } else if (matchChar('(')) {
      return Token.DOTQUERY;
   } else {
      return Token.DOT;
   }
case '|': //аналогично, с литеры «|» начинаются слова OR,
   //ASSIGN_BITOR и BITOR:
   if (matchChar('|')) {
      return Token.OR;
   } else if (matchChar('=')) {
      return Token.ASSIGN_BITOR;
   } else {
      return Token.BITOR;
   } //следующие ниже операторы case (вплоть до case  '!')
   //аналогичным образом обрабатывают одно-, двух- или
   //трехлитерные слова
case '^':
```

```
    if (matchChar('=')) {
        return Token.ASSIGN_BITXOR;
    } else {
        return Token.BITXOR;
    }
case '&':
    if (matchChar('&')) {
        return Token.AND;
    } else if (matchChar('=')) {
        return Token.ASSIGN_BITAND;
    } else {
        return Token.BITAND;
    }
case '=':
    if (matchChar('=')) {
        if (matchChar('='))
            return Token.SHEQ;
        else
            return Token.EQ;
    } else {
        return Token.ASSIGN;
    }
case '!':
    if (matchChar('=')) {
        if (matchChar('='))
            return Token.SHNE;
        else
            return Token.NE;
    } else {
        return Token.NOT;
    }
case '<':      //особая ситуация: с последовательности <!--
    //начинается HTML-комментарий, используемый для скрытия
    //JavaScript-кода от старых браузеров
    /* NB:treat HTML begin-comment as comment-till-eol */
    if (matchChar('!')) {
        if (matchChar('-')) {
            if (matchChar('-')) {
                skipLine();          //метод skipLine пропускает все литеры
```

```
    //входного потока вплоть до конца строки или конца файла,
    //что встретится раньше
                continue retry;
            }
            ungetCharIgnoreLineEnd('-');          //этот метод возвращает
    //на вход литеру, явно указываемую в качестве аргумента
    //(прочитанную предшествующим методом matchChar)
        }
        ungetCharIgnoreLineEnd('!');
    }
    if (matchChar('<')) {
        if (matchChar('=')) {
            return Token.ASSIGN_LSH;
        } else {
            return Token.LSH;
        }
    } else {
        if (matchChar('=')) {
            return Token.LE;
        } else {
            return Token.LT;
        }
    }
case '>':      //с литеры > начинаются одно-, двух-, трех-
    //и четырехлитерные слова языка
    if (matchChar('>')) {
        if (matchChar('>')) {
            if (matchChar('=')) {
                return Token.ASSIGN_URSH;
            } else {
                return Token.URSH;
            }
        } else {
            if (matchChar('=')) {
                return Token.ASSIGN_RSH;
            } else {
                return Token.RSH;
            }
        }
```

```
    } else {
        if (matchChar('=')) {
            return Token.GE;
        } else {
            return Token.GT;
        }
    }
case '*':
    if (matchChar('=')) {
        return Token.ASSIGN_MUL;
    } else {
        return Token.MUL;
    }
case '/':       //с литеры / кроме значащих слов могут начинаться
    //и однострочные, и блочные (возможно – многострочные)
    //комментарии; ниже следует анализ продолжения входной
    //цепочки с фиксацией возможных ошибок
    // is it a // comment?
    if (matchChar('/')) {
        skipLine();
        continue retry;
    }
    if (matchChar('*')) {
        boolean lookForSlash = false;
        for (;;) {
            c = getChar();
            if (c == EOF_CHAR) {
                parser.addError("msg.unterminated.comment");
                return Token.ERROR;
            } else if (c == '*') {
                lookForSlash = true;
            } else if (c == '/') {
                if (lookForSlash) {
                    continue retry;
                }
            } else {
                lookForSlash = false;
            }
        }
```

```
    }
    if (matchChar('=')) {
        return Token.ASSIGN_DIV;
    } else {
        return Token.DIV;
    }
case '%':     //далее, вплоть до case '-' следует обработка
    //одно- и двухлитерных слов
    if (matchChar('=')) {
        return Token.ASSIGN_MOD;
    } else {
        return Token.MOD;
    }
case '~':
    return Token.BITNOT;
case '+':
    if (matchChar('=')) {
        return Token.ASSIGN_ADD;
    } else if (matchChar('+')) {
        return Token.INC;
    } else {
        return Token.ADD;
    }
case '–':     //с литеры «–» могут начинаться знак операции
    //вычитания (–), присваивания с вычитанием (-=), декремента (--)
    //и конец HTML-комментария (-->)
    if (matchChar('=')) {
        c = Token.ASSIGN_SUB;
    } else if (matchChar('-')) {
        if (!dirtyLine) {
            // treat HTML end-comment after possible whitespace
            // after line start as comment-utill-eol
            if (matchChar('>')) {
                skipLine();
                continue retry;
            }
        }
        c = Token.DEC;
    } else {
```

```
            c = Token.SUB;
          }
          dirtyLine = true;
          return c;
        default:      //с любой из остальных литер никакое правильное
          //слово языка начинаться не может:
          parser.addError("msg.illegal.character");
          return Token.ERROR;
        }
      }
    }
    //опущено: некоторые поля и методы класса, прямо не связанные с изучаемым материалом
```

Лексический акцептор (метод *getToken* класса *TokenStream*) согласно процедурной реализации представляет собой основной бесконечный цикл, в теле которого:

− прежде всего, считываются и игнорируются последовательности пробельных литер, которые могут находиться перед любым значащим словом языка JavaScript:

− для определения того, что литера относится к пробельным, используется вызов статического метода *isJSSpace* этого же класса *TokenStream*;

− если в процессе обработки пробельных литер встречается перевод строки или входной поток завершается вообще, то метод *getToken* возвращает соответствующий токен (*Token.EOL* или *Token.EOF*);

− формируется или сбрасывается флажок *dirtyLine*, используемый далее в теле основного цикла для обработки возможных комментариев в стиле HTML, с помощью которых «закрывают» сценарии от показа пользователю устаревших веб-браузеров;

− далее по первой литере, не являющейся пробельной, выясняется, может ли с нее начинаться идентификатор (при этом учитывается возможность того, что первой литерой идентификатора может быть Unicode-символ вида *'\uXXXX'*), и если может, то:

− организуется чтение и накопление в символьном массиве *stringBuffer* (частное поле класса *TokenStream*) путем вызова частного метода *addToString* класса *TokenStream* последовательности литер, составляющих идентификатор;

− если идентификатор содержит Unicode-символы, то выполняется преобразование их внешнего представления во внутренние коды литер (путем вызова метода *xDigitToInt* класса *Kit* пакета Rhino), контролируется правильность этих Unicode-символов, осуществляется формирование диагностических сообщений и возврат токена *Token.ERROR* в случае обнаружения ошибок;

− первая же литера, которая не может принадлежать читаемому идентификатору, возвращается на вход, после чего:

- − символьный массив *stringBuffer* преобразуется в локальную строковую переменную *str*;
- − если идентификатор не содержал Unicode-символов, то вызывается частный метод *stringToKeyword,* которому передается переменная *str*;
- − этот метод проверяет, не является ли содержащийся в *str* идентификатор ключевым словом языка JavaScript, и возвращает токен ключевого слова, или *Token.EOF*;
- − если метод *stringToKeyword* вернул токен ключевого слова, то выполняется дополнительная обработка, зависящая от версии интерпретатора, в результате чего метод *getToken* возвращает токен или ключевого слова, или идентификатора (если в этой версии такое ключевое слово еще не использовалось), либо формирует диагностическое сообщение и возвращает *Token.ERROR;*
- − если же метод *stringToKeyword* вернул *Token.EOF*, то вызывается метод *intern* класса *allStrings*, обеспечивающий поиск обнаруженного идентификатора в таблице «всех строк», а при отсутствии такового – добавление его в эту таблицу. Отметим, что «таблица всех строк» организована как рандомизированная, с использованием мультипликативной функции рандомизации (см. разд. 1.6.3);
- − после этого метод *getToken* возвращает *Token.NAME* (токен идентификатора).

− первая непробельная литера может представлять собой начало численного литерала, для нее организуется:

- − чтение и обработка последующих литер;
- − преобразование литерала во внутреннее представление числа в соответствии с распознаваемым в процессе обработки основанием системы счисления и формой представления литерала (целая, вещественная, экспоненциальная);

− первая непробельная литера может представлять собой начало строкового литерала; для нее обеспечивается накопление составляющих литерал символов с обработкой Unicode- и escape-последовательностей вида *'\uXXXX', '\XXX', '\n' , '\t' , '\f'*, …;

− все остальные случаи обрабатываются переключателем, в теле которого распознаются:

− однолитерные слова наподобие: *'; ', ' [', '? ',*…;

− одно- или двухлитерные слова наподобие: *': ', ':: ', '. ', '.. ',*...

− одно-, двух- или трехлитерные слова, такие как: *'<', '<<', '<=', '<<=', '>', '>>', '>=', '>>=',* ...;

− комментарии строчные (начинающиеся с *'//'*), блочные (начинающиеся с *'/*'* и заканчивающиеся *'*/'*) и в стиле HTML (начинающиеся *'<!--'* и заканчивающиеся *'-->'*).

Разработка таких лексических акцепторов/анализаторов может показаться достаточно простой задачей, однако для этого подхода характерны некоторые трудности и проблемы.

1. Формальное определение лексики языка как части его стандарта не связано непосредственно с текстом программы. Как следствие любые изменения в лексических правилах могут потребовать значительных усилий по приведению текста программы в соответствие с ними.

2. Написание, отладка, верификация программы и доказательство отсутствия ошибок в ней представляют собой трудоемкую и практически не автоматизируемую работу.

Этих недостатков лишена автоматная модель лексического анализа, основанная на строгой теории конечных автоматов и регулярных выражений (определений).

1.4. АВТОМАТНАЯ МОДЕЛЬ ЛЕКСИЧЕСКОГО АКЦЕПТОРА

Автоматная модель основана на идее преобразования формального описания лексики входного языка в управляющую таблицу конечного автомата без памяти. Известны строго формальные способы такого преобразования и существует большое количество пакетов программ (Lex, Lex++, Flex и т.д.) для автоматизации процесса разработки лексических акцепторов.

Существо такого подхода к проектированию трансляторов, логическая структура получаемого в результате преобразования лексического

анализатора и взаимные связи его элементов по данным и по управлению отображены на рис. 1.3.

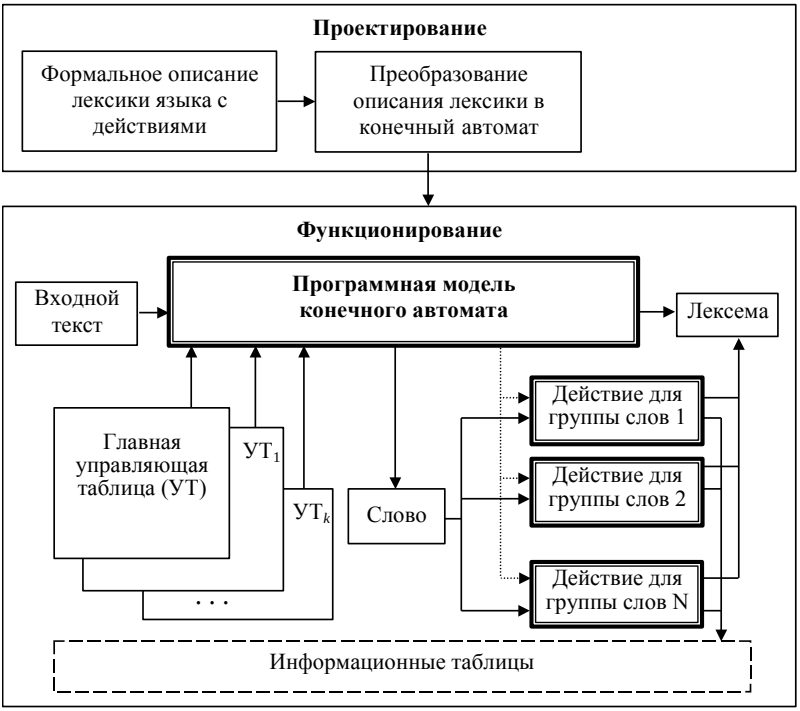


Рис. 1.3. Автоматная модель лексического анализатора

На стадии проектирования:

– человеком разрабатывается формальное описание лексики входного языка проектируемого транслятора с использованием метаязыка регулярных выражений;

– человеком разрабатываются функциональные расширения описания лексики в виде действий, сопоставляемых с моментом обнаружения правильного слова каждой группы и предназначенных для организации работы с информационными таблицами и для управления поведением автомата; действия пишутся на инструментальном языке программирования, т. е. том языке, на котором создается транслятор;

– компьютером (работающим под управлением пакета программ автоматизации проектирования трансляторов) осуществляется преобразование формального описания лексики с встроенными действиями в программную модель конечного автомата.

На стадии функционирования:

– программная модель конечного автомата, вызываемая внешней программой (синтаксическим анализатором), осуществляет распознавание очередного слова, сопровождающееся остановом автомата в соответствующем финальном состоянии;

– если этому финальному состоянию сопоставлено действие, то оно выполняется; результатом действия, в конечном счете, является сформированная лексема и, возможно, модификация состояния информационных таблиц транслятора. Именно так в автоматной модели реализуются функции поиска в таблицах/пополнения таблиц.

По ряду причин, обсуждаемых ниже, конечный автомат можно реализовывать в виде совокупности из главной управляющей таблицы и нескольких дополнительных управляющих таблиц, как это показано на рис. 1.3.

При запуске акцептора для обнаружения очередного слова программная модель всегда переключается на главную таблицу. Некоторые из встроенных в анализатор действий могут переключать автомат на требуемую дополнительную таблицу, обеспечивая сохранение состояния для последующего возврата во вспомогательном внутреннем стеке лексического анализатора. Возврат, т. е. восстановление сохраненного состояния автомата, производится в момент достижения финального состояния в любой дополнительной таблице, если действие для этого финального состояния не предусматривает оформления и выдачи лексемы. Возврат может быть управляемым из действия и производиться необязательно в порядке, обратном тому, в котором осуществлялись предшествующие переключения.

Использование таких механизмов управления автоматом позволяет обеспечивать, например, распознавание строковых литералов, однострочных и блочных комментариев и даже вложенных конструкций, характерных для языков типа XML, и существенно сокращать суммарный объем управляющих таблиц нескольких автоматов по сравнению с реализацией в виде единственного автомата. Подробнее организация функционирования автомата, управляемого несколькими таблицами, будет рассмотрена ниже.

Пока же приведем (в минимально необходимом объеме) некоторые теоретические основы конечных автоматов без памяти, затем рассмотрим метаязык регулярных выражений, используемый для описания лексики формальных языков, процедуры преобразования регулярных определений (именованных регулярных выражений) в конечный автомат без памяти и, наконец, охарактеризуем функционирование лексического акцептора, построенного согласно автоматной модели.

1.4.1. Конечные автоматы без памяти

Конечным автоматом без памяти называется математическая модель устройства, которое:

– имеет один вход, на входе в любой момент времени может находиться либо в точности один символ входного алфавита, либо ничего – пустая цепочка **ε;**

– не имеет выхода;

– функционирует в дискретном времени t_0, t_1, t_2, ...;

– имеет определенный набор рабочих состояний, среди которых выделено особое начальное (стартовое) состояние и некоторый набор состояний останова (финальных состояний);

– каждому финальному состоянию поставлена в соответствие группа правильных слов, возможно, не единственная;

– при запуске, т. е. в момент времени t_0 всегда оказывается в начальном состоянии, на входе в этот момент оказывается самый первый символ входной цепочки;

– в любой момент времени t_i по текущему состоянию и символу, находящемуся на входе, определяет номер рабочего или финального состояния, в котором автомат окажется в момент времени t_{i+1};

– если по каким-либо причинам номер следующего состояния не может быть определен, то автомат останавливается по ошибке в специальном состоянии;

– при любом переходе в рабочее состояние или специальное состояние останова по ошибке текущий входной символ заменяется следующим символом входной цепочки;

– при переходе в финальное состояние обнаружения правильного слова входной символ автомата не изменяется, т. е. этот переход выполняется по пустой цепочке **ε.**

Заметим, что применительно к программной реализации автоматов и для решения задачи распознавания последовательности слов переход по пустой цепочке в финальное состояние эквивалентен:

- чтению символа со входа;
- обнаружению того, что символ не принадлежит текущему слову;
- возврату символа на вход автомата.

Говорят, что входная цепочка отвергается автоматом, если в результате ее обработки он остановился в состоянии ошибки.

Говорят, что входная цепочка принимается автоматом, если в результате ее обработки он остановился в финальном состоянии обнаружения правильного слова.

Как следует из определения понятия автомата, он может быть задан совокупностью троек значений, определяющих переходы из одного состояния в другое по входному символу:

{текущее состояние; входной символ; следующее состояние}

Приведем пример такого определения автомата:

$$\{\boldsymbol{0},0,\boldsymbol{1}\}\ \{\boldsymbol{0},1,\boldsymbol{1}\}\ \{\boldsymbol{1},0,\boldsymbol{1}\}\ \{\boldsymbol{1},1,\boldsymbol{1}\}\ \{\boldsymbol{1},\varepsilon,\boldsymbol{2}\}$$

Состояния ***0*** и ***1*** являются рабочими, поскольку для них определены переходы в другие состояния по входным символам. Состояние ***2*** – финальное, поскольку для него не задано ни одного перехода.

При реализации в компьютерных программах конечные автоматы без памяти представляются либо графами состояний и переходов, либо управляющими таблицами. Мы будем практически в равной степени использовать эти два способа задания автоматов, поэтому сформулируем соответствующие понятия.

1.4.1.1. Граф состояний и переходов конечного автомата

Графом состояний и переходов (ГСП) конечного автомата без памяти (далее – конечного автомата) называется помеченный ориентированный граф, вершины которого сопоставлены состояниям, а дуги – переходам. Разметка вершин обычно производится целыми числами, обозначающими номера состояний. Имеется особая начальная вершина, в которую, как правило, не входит ни одна дуга. Существуют особые финальные вершины, из которых не может выходить ни одна дуга. Имеется одна или несколько рабочих вершин, в которые могут входить дуги и из которых могут выходить дуги.

Приведем пример графа состояний и переходов конечного автомата A_1, предназначенного для акцепта любого двоичного числа (рис. 1.4).

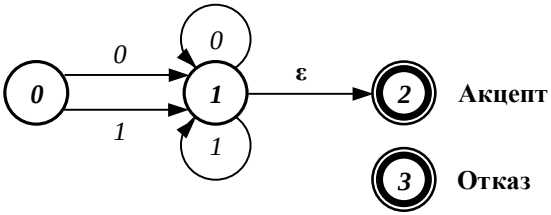


Рис. 1.4. Пример графа состояний и переходов

Разметка дуг графа производится с помощью указания одного из символов входного алфавита того языка, для распознавания слов которого построен данный автомат либо обозначения пустой цепочки символов **ε**. Если дуга помечена символом, то при переходе по этой дуге автомат заменяет входной символ следующим символом из входной цепочки. При переходе по дуге, помеченной обозначением пустой цепочки **ε**, символ на входе автомата не изменяется.

Автомат, граф состояний и переходов которого приведен на рис. 1.10, имеет начальное состояние номер ***0***, одно рабочее (номер ***1***) и два финальных состояния. Останов автомата в финальном состоянии с номером ***2*** соответствует акцепту входной цепочки символов. Действительно, если входная цепочка состоит только из символов *0* и *1* (двоичных цифр), то после запуска автомат из нулевого состояния по первой цифре перейдет в состояние номер ***1***, а на его входе окажется следующий символ. Далее по каждой двоичной цифре автомат будет осуществлять переходы из состояния ***1*** в состояние ***1*** до тех пор, пока не исчерпает всю входную цепочку.

Появление на входе автомата пустой цепочки **ε** приведет к переходу в финальное состояние номер ***2***, т. е. к останову автомата. Считается, что в этом случае автомат принимает входную цепочку символов, как правильную.

Финальное состояние отказа с номером ***3*** приведено на рис. 1.10 для того, чтобы определить реакцию конечного автомата при возможном появлении на его входе любого символа, отличающегося от символов *0* и *1*. В графе состояний и переходов нет дуг, помеченных такими символами. В соответствии с формулировкой понятия автомата он не может по такому входному символу определить следующее состоя-

ние. Поэтому предполагается, что появление любого такого символа приведет к останову автомата в состоянии ***3***.

Считается, что в этом случае автомат отвергает (не принимает) входную цепочку. Заметим, что данный автомат не принимает пустую входную цепочку в качестве двоичного числа.

В дальнейшем для упрощения изображения графов состояний и переходов будем использовать разметку дуг не в виде одиночного символа, а в виде множества символов. Это позволит заменить параллельные дуги (параллельными являются дуги, выходящие из одной вершины и ведущие в одну и ту же вершину) одной дугой.

При этом становится возможным явно указать переходы в финальное состояние по ошибке. Для этого в граф состояний и переходов можно добавить дуги, помеченные обозначениями вида «Прочие». Под этими дугами понимаются все те символы, которыми не помечена ни одна дуга, выходящая из данной вершины.

Приведем пример графа состояний и переходов автомата A_1, предназначенного для акцепта двоичных чисел (рис. 1.5). Дуги, ведущие в состояние *1*, помечены множествами символов {*0,1*}. Дуги, ведущие в состояние отказа, помечены обозначением «Прочие».

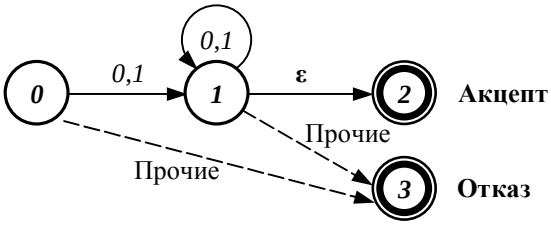


Рис. 1.5. Разметка дуг ГСП множествами символов

Можно считать, что любой конечный автомат имеет как минимум одно финальное состояние отказа (останова по ошибке). Далее такие одиночные финальные состояния отказа ни в графах состояний и переходов, ни в управляющих таблицах показаны не будут.

1.4.1.2. Управляющая таблица конечного автомата

Альтернативным способом определения конечного автомата без памяти является табличный способ. Автомат задают прямоугольной таблицей, строки которой обычно соответствуют состояниям, а столб-

цы – входным символам. Номера состояний и входные символы показывают в заголовках строк и столбцов, однако следует помнить, что заголовки строк и столбцов не являются элементами таблицы. В клетках таблицы указывают номера состояний перехода (из состояния, в строке которого находится клетка по символу, указанному в заголовке столбца). Приведем пример управляющей таблицы (УТ) автомата A_1, предназначенного для акцепта двоичных чисел.

Состояние	*0*	*1*	**ε**
0	1	1	
1	1	1	2
2	Финальное: акцепт		
3	Финальное: отказ		

Заштрихованные области фактически не являются элементами управляющей таблицы (см. каноническое определение конечного автомата без памяти) и приведены только для удобства. В дальнейшем эти области не будут выделяться штриховкой, но смысл их останется тем же. Рабочей зоной будем называть незаштрихованную часть таблицы, содержащую номера состояний перехода.

Если в рабочей зоне управляющей таблицы нет пустых клеток, то автомат называется ***полностью определенным***. В противном случае автомат называется ***не полностью определенным.***

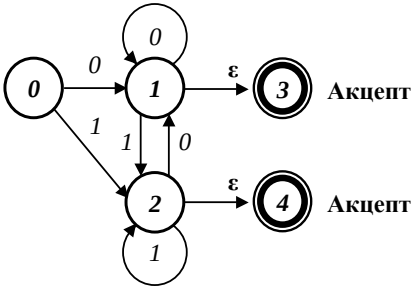
При задании автомата управляющей таблицей предполагается, что автомат оперирует с индексами строк и столбцов, «умеет» преобразовывать символы в индексы столбцов, «знает», какие состояния являются финальными, «умеет» переходить в финальное состояние отказа в тех случаях, когда на входе обнаруживается символ, которым не помечен ни один столбец таблицы или если текущая клетка пуста. Способы реализации этих «знаний» и «умений» будут рассмотрены ниже.

1.4.1.3. Отношение эквивалентности между автоматами

Пусть даны два автомата A_2 и A_3. Граф состояний и переходов и управляющая таблица автомата A_2 выглядят так, как показано на рис. 1.6.

Автомат A_2 принимает, так же как и автомат A_1, любые цепочки двоичных цифр, т. е. двоичные числа. Однако, в отличие от автомата A_1, у этого автомата два различных состояния акцепта с номерами ***3***

и ***4***. Останов автомата в состоянии ***3***, как легко можно видеть из графа состояний и переходов, свидетельствует о том, что входная цепочка содержит четное двоичное число, а в состоянии ***4*** – нечетное.

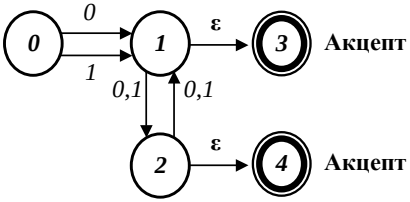


Состояние	*0*	*1*	ε
0	1	2	
1	1	2	3
2	1	2	4
3	Акцепт: четное		
4	Акцепт: нечетное		

Рис. 1.6. Граф состояний и переходов и УТ автомата A_2

Граф состояний и переходов и управляющая таблица автомата A_3 выглядят так, как показано на рис. 1.7.

Автомат A_3 также принимает любое двоичное число, но останов его в состоянии ***3*** свидетельствует о том, что правильная входная цепочка содержит нечетное количество символов, а в состоянии ***4*** – четное.



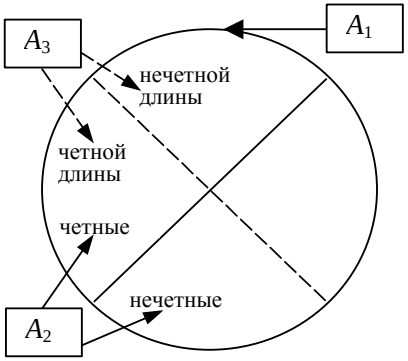
Состояние	*0*	*1*	ε
0	1	1	
1	2	2	3
2	1	1	4
3	Акцепт: нечетная длина		
4	Акцепт: четная длина		

Рис. 1.7. Граф состояний и переходов и УТ автомата A_3

Приведем финальные состояния останова автоматов A_1, A_2 и A_3 при обработке нескольких коротких правильных цепочек.

Справа в графической форме кругом показано множество всех двоичных чисел, принимаемых автоматом A_1. Разбиение этого множества на четные и нечетные числа показано сплошной диагональю, а на числа четной и нечетной длины – пунктирной диагональю.

Входная цепочка	A_1	A_2	A_3
0	***2***	***3***	***3***
1	***2***	***4***	***3***
00	***2***	***3***	***4***
01	***2***	***4***	***4***
10	***2***	***3***	***4***
11	***2***	***4***	***4***
001	***2***	***3***	***3***
010	***2***	***4***	***3***
011	***2***	***3***	***3***
100	***2***	***4***	***3***
101	***2***	***3***	***3***



Очевидно, что для любого автомата и любой входной цепочки может быть (по крайней мере, в принципе) определено финальное состояние, в котором автомат остановится при обработке данной цепочки. Таким образом, можно считать, что любой конечный автомат предназначен для разбиения бесконечного множества M всех возможных цепочек символов на непересекающиеся подмножества M_1, M_2, ... путем сопоставления с каждым из них одного из финальных состояний этого автомата. Такие подмножества будем называть группами слов в соответствии с теми целями, для которых мы рассматриваем понятия конечных автоматов.

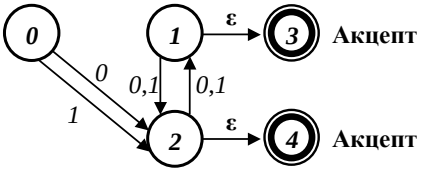
Если два автомата разбивают множество входных цепочек символов на одинаковые подмножества (независимо от того, каковы финальные состояния, сопоставленные с каждым подмножеством), то такие автоматы называются ***эквивалентными***.

Приведенные выше автоматы не являются попарно эквивалентными. Каждый из них разбивает множество всех возможных цепочек на различные подмножества.

Эквивалентные автоматы не всегда одинаковы.

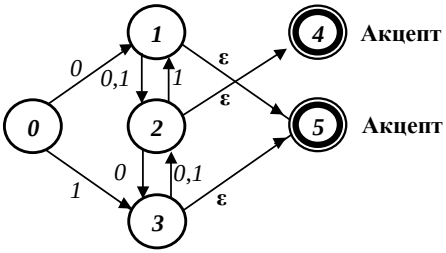
Пусть автомат A_4 задан графом состояний и переходов и управляющей таблицей, изображенными на рис. 1.8.

Легко видеть, что автомат A_4 эквивалентен автомату A_3, но их графы состояний и переходов и управляющие таблицы не совпадают. В данном случае различия могут показаться несущественными, поскольку отличаются только номера состояний. Более нагляден пример автомата A_5, эквивалентного A_3 (и A_4), но имеющего большее количество состояний (рис. 1.9).



Состояние	*0*	*1*	***ε***
0	2	2	
1	2	2	3
2	1	1	4
3	Акцепт: нечетная длина		
4	Акцепт: четная длина		

Рис. 1.8. Граф состояний и переходов и УТ автомата A_4



Состояние	*0*	*1*	**ε**
0	1	3	
1	2	2	5
2	1	3	4
3	2	2	5
4	Акцепт: четная длина		
5	Акцепт: нечетная длина		

Рис. 1.9. Граф состояний и переходов и УТ автомата A_5

Практически для любого требуемого разбиения цепочек на группы слов легко можно привести примеры эквивалентных автоматов, имеющих различное количество состояний.

Существование классов эквивалентных автоматов порождает постановку задачи нахождения автомата, оптимального в заданном смысле, например имеющего наименьшее количество клеток в рабочей зоне управляющей таблицы. Эта задача будет рассмотрена после того как мы определим так называемые недетерминированные автоматы и изучим некоторые их свойства.

1.4.1.4. Понятие истории работы конечного автомата

Историей работы конечного автомата без памяти для данной входной цепочки называется упорядоченная (по моменту дискретного времени) совокупность троек значений *{t,s,X}*, где: *t* – момент дискретного времени; *s* – текущий входной символ; *X* – текущее состояние.

Как уже отмечалось выше, для любой конечной по количеству символов входной цепочки автомат, запущенный в начальном состоянии, завершит работу за конечное число шагов, т. е. реализует конечную историю работы. Приведем пример истории работы автомата A_3, предназначенного для акцепта двоичных чисел и распознавания чисел, состоящих из четного/нечетного количества символов в качестве различных слов. Историю работы автомата представим в виде таблицы, столбцы которой содержат значения троек.

Вместо момента времени t_i укажем порядковый номер i такта работы автомата. Пусть дана входная цепочка *10110*, тогда история работы автомата A_3 будет выглядеть так:

Такт	0	1	2	3	4	5	6
Символ	*1*	*0*	*1*	*1*	*0*	***ε***	***ε***
Состояние	***0***	***1***	***2***	***1***	***2***	***1***	***3***

Останов автомата в состоянии ***3*** свидетельствует об акцепте цепочки, содержащей нечетное количество двоичных цифр. Для этой же цепочки история работы автомата A_5 будет выглядеть так:

Такт	0	1	2	3	4	5	6
Символ	1	0	1	1	0	***ε***	***ε***
Состояние	0	3	2	1	2	3	5

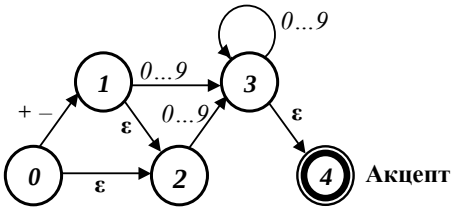
Автоматы A_3 и A_5 для любой конкретной входной цепочки будут реализовывать одни и те же истории работы при каждом запуске. Такие конечные автоматы называются ***детерминированными***.

1.4.1.5. Недетерминированные конечные автоматы без памяти

Детерминированность означает, что история работы автомата полностью определяется входной цепочкой символов. Если детерминированный автомат запустить несколько раз для одной и той же цепочки символов, то каждый раз автомат будет реализовывать одну и ту же историю работы.

Однако существуют и ***недетерминированные*** автоматы, которые могут отрабатывать разные последовательности переходов из состояния в состояние при различных запусках для одной и той же входной цепочки символов.

Недетерминированностью первого рода называется наличие хотя бы одного перехода по пустой цепочке символов **ε** в рабочее состояние. Простой пример автомата A_6 с такой недетерминированностью показан на рис. 1.10. Этот автомат получен в результате выполнения одного из шагов преобразования формального определения лексики языка целых десятичных чисел со знаком или без знака.



Состояние	+ –	*0...9*	ε
0	1		2
1		3	2
2		3	
3		3	4
5	Акцепт: число		

Рис. 1.10. ГСП и УТ недетерминированного автомата A_6

Автомат A_6, в отличие от ранее рассматривавшихся детерминированных автоматов, для одной и той же входной цепочки символов при разных запусках может реализовывать разные истории работы. Так, например, для цепочки «–25» одна из историй работы выглядит так:

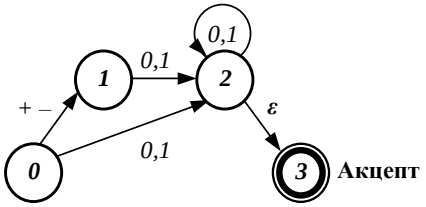
Такт	0	1	2	3
Символ	–	2	5	ε
Состояние	0	1	3	4

Возможна и такая история:

Такт	0	1	2	3	4
Символ	–	ε	2	5	***ε***
Состояние	0	1	2	3	4

Реализуя разные истории работы, автомат A_6 каждой входной цепочке ставит в соответствие всегда одно и то же финальное состояние. Это позволяет распространить определение отношения эквивалентности/неэквивалентности и на недетерминированные автоматы. Более того, если детерминированный и недетерминированный автоматы разбивают множество всех цепочек на одинаковые подмножества (группы слов), то их тоже можно считать эквивалентными.

Приведем пример детерминированного автомата A_7, эквивалентного автомату A_6 (рис. 1.11).



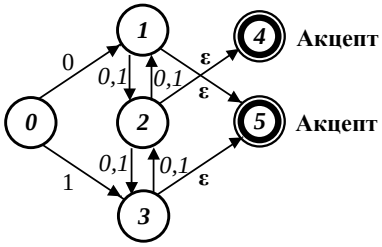
Состояние	*0*	*1*	+
0	2	2	1
1	2	2	
2	2	2	
3	Акцепт: число		

Рис. 1.11. Граф состояний и переходов и УТ автомата A_7

Недетерминированностью второго рода называется наличие в управляющей таблице клеток, содержащих два или более номера состояния (или столбцов, помеченных одним и тем же символом, но не являющихся идентичными). В графе состояний и переходов это соответствует дугам, выходящим из одной вершины, помеченным одним и тем же символом, но ведущим в разные вершины.

Недетерминированность второго рода, так же как и первого, порождает возможность реализации нескольких различных историй работы автомата для одной и той же входной цепочки при разных запусках.

Приведем пример автомата A_8, эквивалентного автоматам A_3, A_4 и A_5, но содержащего недетерминированность второго рода (рис. 1.12). Этот автомат, оказавшись в состоянии ***2***, может перейти по любому из символов *0* или *1* как в состояние ***1***, так и в состояние ***3***.



Состояние	*0*	*1*	ε
0	1	3	
1	2	2	5
2	1,3	1,3	4
3	2	2	5
4	*0*	*1*	***ε***
5	Финальное: акцепт		

Рис. 1.12. Граф состояний и переходов и УТ автомата A_8

Другое возможное представление управляющей таблицы автомата A_8, обладающего недетерминированностью второго рода, выглядит так:

Состояние	*0*	*0*	*1*	*1*	ε
0	1	1	3	3	
1	2	2	2	2	5
2	1	3	1	3	4
3	2	2	2	2	5
4	Финальное: акцепт				
5	Финальное: акцепт				
6	Финальное: отказ				

Для входной цепочки 10 110 этот автомат при разных запусках может реализовать разные истории работы, например такую:

Такт	0	1	2	3	4	7	
Символ	1	0	1	1	0	ε	
Состояние	0	3	2	1	2	3	5

или такую:

Такт	0	1	2	3	4	7	
Символ	1	0	1	1	0	ε	
Состояние	0	3	2	3	2	1	5

Несмотря на то что поведение автомата A_8 при разных запусках для одной и той же входной цепочки может быть различным, его финальное состояние для любой цепочки будет одним и тем же, независимо от того, какая история работы была им реализована.

Однако легко можно привести примеры недетерминированных автоматов, которые для одной и той же входной цепочки при разных запусках могут останавливаться в разных финальных состояниях. Такие автоматы мы рассматривать не будем.

Недетерминированность автомата как первого, так и второго рода может возникнуть в результате его построения формальными методами путем преобразования описания лексики языка. Кроме того, недетерминированность одного рода может возникнуть в процессе эквивалентных преобразований автомата при ликвидации недетерминированности другого рода.

1.4.1.6. Недостижимые, тупиковые и эквивалентные состояния

Недостижимым называется такое состояние автомата, в которое не существует ни одного пути из начального состояния в графе состояний и переходов.

Тупиковым называется состояние, из которого не существует ни одного пути в какое-либо финальное состояние акцепта. Как тупиковые, так и недостижимые состояния в определенном смысле являются лишними. Если удалить эти состояния вместе со всеми переходами – как входящими, так и исходящими из них, то будет получен автомат, эквивалентный исходному.

Эквивалентными называются такие два (или более) состояния, для которых строки в управляющей таблице либо полностью совпадают, либо различаются только переходами друг на друга. Любую пару эквивалентных состояний всегда можно заменить одним состоянием без потери функциональности автомата.

Приведем (рис. 1.13) пример автомата, эквивалентного автомату A_9, но содержащего недостижимые, тупиковые и эквивалентные состояния.

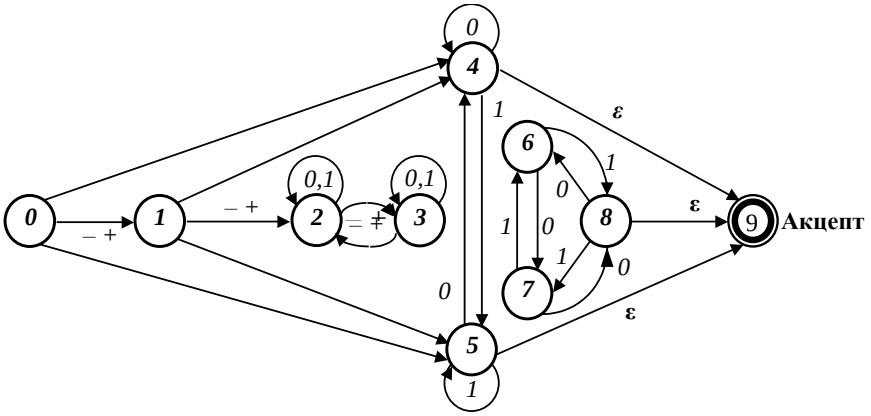


Рис. 1.13. Граф состояний и переходов автомата A_9

Управляющая таблица автомата A_9 выглядит так:

Состояние	*0*	*1*	–	+	ε
0	4	5	1	1	
1	4	5	2	2	
2	2	2	3	3	
3	3	3	2	2	
4	4	5			9
5	4	5			9
6	7	8			
7	8	7			
8	6	7			9
9	Финальное: акцепт				

Автомат A_9 имеет два тупиковых состояния с номерами ***2*** и ***3***, три недостижимых состояния с номерами ***6***, ***7*** и ***8***. Состояния ***4*** и ***5*** эквивалентны. Методы нахождения тупиковых, недостижимых и эквивалентных состояний будут рассмотрены ниже.

1.4.1.7. Оптимальность конечных автоматов без памяти

Теперь для любого множества эквивалентных автоматов можно определить понятие оптимального автомата. ***Оптимальным*** в множестве эквивалентных называется полностью определенный детерминированный автомат, управляющая таблица которого имеет наименьшее количество клеток (естественно, только в рабочей зоне).

Оптимальный автомат удовлетворяет следующим критериям.

1. Множества символов, помечающие столбцы управляющей таблицы, попарно не пересекаются.
2. Автомат не содержит недетерминированностей первого рода.
3. Автомат не содержит недетерминированностей второго рода.
4. Автомат не имеет тупиковых состояний.
5. Автомат не имеет недостижимых состояний.
6. Все рабочие состояния попарно не являются эквивалентными.
7. В рабочей зоне управляющей таблицы нет одинаковых столбцов.
8. В рабочей зоне управляющей таблицы нет пустых клеток.

Конечный автомат без памяти может быть преобразован в эквивалентный ему оптимальный автомат. Методы и алгоритмы такого

преобразования рассматриваются ниже, после изучения способов формального определения лексики языка и получения на этой основе недетерминированного неоптимального конечного автомата без памяти.

1.4.2. Способы определения лексики формальных языков

С точки зрения лексики любой язык (в том числе и формальный язык программирования) представляет собой множество всех правильных слов. Так, например, в русском языке есть слова: «так», «например», «в», «русском», «языке», «есть», «слова». Однако в русском языке нет слова «ьрюъгаыф» (оно не входит в множество правильных слов), хотя и представляет собой цепочку символов, принадлежащих алфавиту русского языка. Лексика естественных языков – это словарь, содержащий все допустимые слова (т. е. просто все множество правильных слов). Для формальных языков программирования, с одной стороны, лексику, в принципе, можно представить точно так же, а с другой стороны – это возможно только умозрительно, а на практике нереально и бессмысленно составлять словари, содержащие многие миллионы миллиардов слов вида «w21rcx7ainolaDqQ5». Вместо создания словарей для формального языка разрабатывается совокупность правил образования слов из символов, которая может быть использована для автоматического построения лексического анализатора слов данного языка.

Правила образования слов обычно формулируются на метаязыке регулярных выражений. Вначале мы рассмотрим основные понятия этого метаязыка, а затем – способ его использования для задания правил построения слов описываемого языка.

1.4.2.1. Регулярные выражения, основные понятия

Первичным называется регулярное выражение, порождающее (описывающее) цепочки, состоящие из единственного символа. К первичным относятся выражения вида:

– [<*произвольный символ*>], например: [*a*]. Говорят, что такое регулярное выражение порождает единственную цепочку, состоящую из этого символа, т. е.

$$[a] \rightarrow a$$

– [<*перечень символов*>], например [*aA*]. Такое выражение порождает цепочки, содержащие единственный символ из указанного перечня:

$$[aA] \rightarrow a$$

$$[aA] \rightarrow A$$

– [<*диапазон символов*>], например [*0–9*]. Это выражение можно считать сокращенной формой записи выражения вида [*0123456789*], а поэтому его смысл состоит в порождении цепочек, каждая из которых содержит единственную десятичную цифру. В одном первичном выражении может быть указано несколько диапазонов, например [*a–zA–Z*].

С последними двумя формами записи первичного регулярного выражения связаны три умолчания, о которых всегда нужно помнить. Во-первых, для указания диапазона необходимо точно знать таблицу кодирования символов. Знание таблицы избавит от попадания в диапазон «лишних» символов. Например, при использовании кодировки ASCII запись вида [*a–F*] ни в коем случае нельзя использовать для определения старших шестнадцатеричных цифр. Правильной будет запись [*a–fA–F*]. Во-вторых, считается, что после символа с максимальным значением кода следует символ с минимальным значением (в системе кодирования ASCII после символа с кодом 255 идет символ с кодом 0). В-третьих, символ дефиса (минуса) нельзя употреблять «внутри» перечня. Если этот метасимвол нужно определить и как символ алфавита описываемого языка, то в перечне он должен быть указан сразу после метасимвола [или непосредственно перед метасимволом]. Так, например, выражения [–+*/] или [+/*–] порождают каждое ровно четыре цепочки: –,+,*,/, а выражение [+–*/] в системе кодирования ASCII порождает ровно 256 цепочек длины 1, т. е. все символы алфавита. С учетом этих умолчаний обычно допускается записывать перечни и диапазоны последовательно без каких-либо разделителей между метасимволами [и]. Соответственно, выражения [*abk–orsx–z*] и [*abklmnorsxyz*] являются эквивалентными, т. е. порождают одно и то же множество цепочек символов.

Во многих диалектах языка регулярных выражений (обычно использующихся для поиска/замены подстрок в строках) допускается использование знака операции инверсии, применяемого к первичному выражению в виде

[^…]

Считается, что такое выражение порождает цепочки (длиной точно один символ) из символов, не входящих в множество, определенное первичным выражением [...]. Выражение [^*0–9*] можно понимать как «любой символ, не являющийся десятичной цифрой». Заметим, что учебный пакет Вебтранслаб при обработке регулярных определений не реализует инверсию множеств символов. Вместо этого может быть использовано специальное слово *other* (или *other+*, см. ниже операции над регулярными выражениями)

Иногда требуется оперировать с символами, не имеющими графического изображения, например символом табуляции, перевода каретки, новой строки и т. д. Для указания таких символов в первичных выражениях обычно используется техника, заимствованная из С-подобных языков. Для часто встречающихся символов используются общепринятые обозначения:

\t – символ табуляции

\n – символ новой строки

\r – символ перевода каретки

...

Аналогичный способ (экранирование следующего символа) необходим для представления таких символов, как \, [,], ", являющихся одновременно метасимволами языка регулярных выражений:

\\ – один символ \

[– символ [

] – символ]

\" – символ "

Простым называется произвольное регулярное выражение, заключенное в круглые скобки. Например, выражение ([*0–9*]) – простое выражение.

К первичным и простым (в некоторых диалектах – только к первичным) регулярным выражениям применяются следующие знаки операций, используемые для описания пустых цепочек (т. е. цепочек длины 0) или цепочек, длина которых больше единицы:

1) операция конкатенации, не имеющая знака операции: запись вида [/][*] порождает цепочку /* (начало комментария в С-подобных языках).

Во многих диалектах метаязыка регулярных выражений допускаются выражения вида <*произвольная строка символов*>, каждое из которых порождает цепочку, совпадающую с <*произвольная строка символов*>. Например, выражение "*ELSE*" считается эквивалентным выражению [*E*][*L*][*S*][*E*].

2) операция выбора (знак операции |): запись вида [*] | [/] порождает либо *, либо /. В некоторых случаях ее использование эквивалентно перечню [**/**] (или диапазону). Ниже будет приведен пример, который не может быть сведен ни к перечню, ни к диапазону;

3) операция «Пусто или в точности одно» (знак операции ?): запись вида [–+]? можно понимать как возможно отсутствующий знак числа. Это выражение порождает либо пустую цепочку, либо –, либо +;

4) операция «Одно или несколько» (знак операции +): запись вида [*0–9*]+ является типичным определением целой десятичной константы без знака (порождает непустые цепочки из десятичных цифр);

5) операция «Пусто, одно или несколько» (знак операции *): запись вида [*0–9a–zA–Z*]* является типичной частью определения идентификаторов и порождает произвольную, возможно пустую цепочку, состоящую из букв и/или цифр. Собственно определение идентификатора может выглядеть так:

[*a–zA–Z*][*0–9a–zA–Z*]*

Это означает, что цепочки, начинающиеся с буквы, за которой в произвольном порядке могут следовать буквы и/или цифры.

Приведем пример определения десятичной вещественной константы с возможно отсутствующим знаком:

[–+]? (([*0–9*]+ [.] [*0–9*]*) | ([*0–9*]* [.] [*0–9*]+))

Это конкатенация двух выражений, первое из которых [–+]? – описывает возможно отсутствующий знак константы, а второе есть выбор из описания константы с возможно отсутствующей дробной частью ([*0–9*]+[.][*0–9*]*) и описания константы с возможно отсутствующей целой частью ([*0–9*]*[.][*0–9*]+). Такое определение не допускает возможности порождения цепочек вида «.», «+.» и «–.» в качестве правильных вещественных констант.

Таким образом, отдельные регулярные выражения позволяют определять правила образования цепочек символов произвольной длины, т. е. слов, принадлежащих к одной группе слов формального языка.

1.4.2.2. Системы регулярных определений

Для полного описания лексики языка обычно используются именованные регулярные выражения, или регулярные определения:

<наименование группы слов> : <регулярное выражение>

Совокупность таких правил называется системой регулярных определений, задающих способы порождения нескольких групп слов. Приведем пример системы регулярных определений для групп слов, из которых может состоять оператор присваивания в С-подобных языках (в этом примере не описываются все группы слов языка С). В дальнейшем эту систему регулярных определений мы будем использовать под названием RESystem1 для иллюстрации изучаемых методов построения лексических анализаторов:

Ident	:	[*a*–z*A*–*Z*][*0*–*9a*–z*A*–*Z*]*
Const	:	[*0*–*9*]+([.][*0*–*9*]*)?
Const	:	[*0*–*9*]*[.][*0*–*9*]+
Formatting	:	[\r\n\t]+
OpSign	:	[–+*/]
Delimiter	:	[;]
AsSign	:	[=]

Во второй и третьей строках этой системы используется одно и то же имя группы слов для разных регулярных выражений. Такой способ определения допускается во многих диалектах языка регулярных определений. Таким образом, операция выбора в некоторых случаях может быть заменена повторным использованием наименования группы слов в системе регулярных определений.

Система регулярных определений может быть преобразована в оптимальный конечный автомат без памяти, способный распознавать правильные слова всех заданных групп. Каждой группе слов сопоставляется в точности одно финальное состояние, номер которого при останове автомата используется для идентификации той группы, в которую входит обнаруженное слово.

1.4.3. Преобразование системы регулярных определений в конечный автомат

Построение лексического акцептора для распознавания слов, заданных некоторой корректной системой регулярных определений, состоит в выполнении следующей последовательности действий (этапов):

– система регулярных определений преобразуется в недетерминированный неоптимальный и не полностью определенный конечный автомат без памяти (возможно – в несколько таких автоматов);

– каждый не полностью определенный недетерминированный неоптимальный конечный автомат преобразуется в оптимальный конечный автомат без памяти;

– управляющая таблица оптимального конечного автомата (автоматов) объединяется с программой, моделирующей поведение автомата (см. рис. 1.9); если система определений содержит действия, то каждое из них также встраивается в программную модель.

Подобная организация процесса построения лексического акцептора нужна для того, чтобы каждый из этих этапов по отдельности легко можно было формализовать для компьютерной реализации.

Рассмотрим методы реализации каждого из этапов создания лексического акцептора.

1.4.3.1. Преобразование системы регулярных определений в недетерминированный неоптимальный конечный автомат

Для этого преобразования вначале будем использовать представление конечного автомата без памяти в виде графа состояний и переходов. Построение конечного автомата начинается с образования начальной вершины с номером $\boldsymbol{0}$, общей для всех регулярных определений системы (рис. 1.14).

Для каждого именованного регулярного выражения $РВ_i$ образуются:

– промежуточная вершина $\boldsymbol{p_i}$;

– условная дуга перехода из начальной вершины графа в промежуточную вершину, помеченная всем регулярным определением $РВ_i$ (условность отражена прерывистостью дуги);

– финальная вершина $\boldsymbol{f_i}$, соответствующая акцепту слов данной группы;

– дуга перехода из промежуточной вершины $\boldsymbol{p_i}$ в финальную $\boldsymbol{f_i}$, помеченная обозначением пустой цепочки.

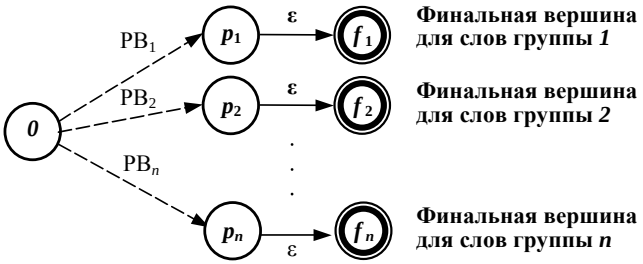


Рис. 1.14. Начальный вид графа состояний и переходов

Если для некоторой группы слов в системе определено несколько одинаково поименованных регулярных выражений, то для этой группы образуется единственная финальная вершина и строится несколько путей в эту вершину из начальной. Пусть для группы 2 задано два регулярных определения PB_2^1 и PB_2^2 (см. RESystem1, слова группы Const). В подобном случае граф автомата будет выглядеть следующим образом (рис. 1.15).

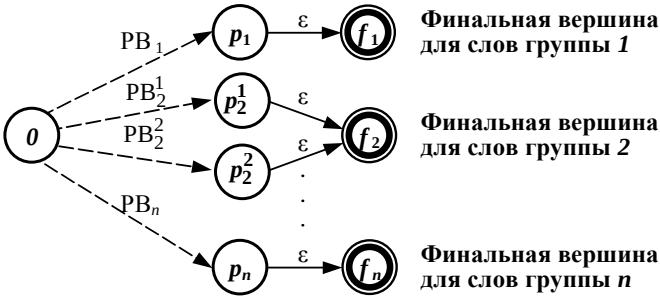
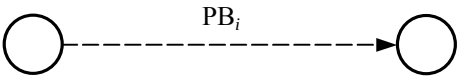


Рис. 1.15. Начальный вид графа состояний и переходов при множественном определении групп слов

Следует отметить, что из практических соображений любая система регулярных определений перед преобразованием явно или неявно должна быть расширена определением слова EndOfFile, образующего отдельную специальную группу. Этим словом заканчивается любая правильная программа на любом языке программирования.

Мы будем считать, что выражение, определяющее эту группу слов, автоматически добавляется преобразователем в заданную систему регулярных определений.

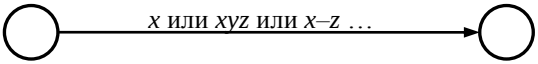
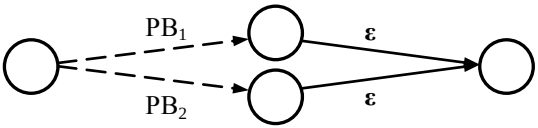
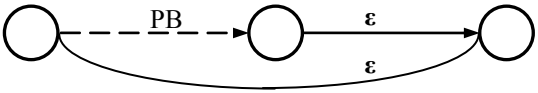
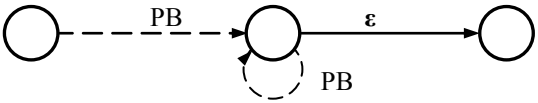
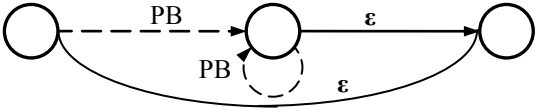
Затем каждый фрагмент графа, содержащий дугу условного перехода, т. е. имеющий вид:



Начальная вершина фрагмента Конечная вершина фрагмента

преобразуется согласно табл. 1.1 в другой фрагмент, включающий те же самые начальную и конечную вершины, но вместо одной условной дуги перехода появляется либо безусловная дуга, либо новые вершины и дуги.

Таблица 1.1

Вид выражения	Начальная вершина	Новый фрагмент	Конечная вершина
1. Первичное: [*x*] или [*xyz*] или [*x–z*] …			
2. $РВ_1$ $РВ_2$			
3. $РВ_1$ | $РВ_2$			
4. РВ?			
5. РВ+			
6. РВ*			

Процесс преобразования продолжается до тех пор, пока все дуги графа не окажутся помечены первичными регулярными выражениями (одиночными символами, перечнем или диапазоном символов) или обозначением пустой цепочки. Приведенные в таблице способы преобразования строго соответствуют очевидным правилам поведения автомата для распознавания символьных цепочек, определяемых каждым из способов образования сложного регулярного выражения из более простых.

В случае если регулярное выражение, помечающее условную дугу перехода, является первичным, условная дуга просто заменяется на дугу безусловного перехода с соответствующей разметкой.

В противном случае согласно правилам образования регулярных выражений любое $РВ_i$ можно представить как одно из выражений 2–6-й строк первого столбца таблицы и заменить условную дугу перехода исходного фрагмента на новый фрагмент из 3-го столбца таблицы с сохранением начальной и конечной вершин.

Появляющиеся условные дуги переходов будут помечены составными частями исходного регулярного выражения, т. е. более простыми подвыражениями. Основой любого регулярного выражения являются первичные выражения. Следовательно, для любого регулярного выражения, имеющего конечную длину, за конечное число шагов может быть получен граф состояний и переходов, не содержащий условных дуг переходов.

Пронумеруем в произвольном порядке все промежуточные вершины этого графа числами ***1,2,..,m***, затем все финальные вершины числами ***m+1,m+2,...,n*** и преобразуем граф состояний и переходов автомата в управляющую таблицу.

Вероятно, построенный автомат будет недетерминированным и неоптимальным. В качестве иллюстрации приведем граф состояний и переходов, построенный описанным выше способом по системе определений RESystem1 (рис. 1.16).

Особое внимание следует обратить на фрагмент графа состояний и переходов, обеспечивающий распознавание констант. Для этой группы слов в системе задано два различных регулярных определения:

Const : [*0–9*]+([.][*0–9*]*)?
Const : [*0–9*]*[.][*0–9*]+

Соответственно этим определениям в графе автомата образовано два фрагмента с общими начальной и финальной вершинами. Первое

выражение является конкатенацией двух более простых выражений **[***0–9***]+** и **([.][***0–9***]*)**?, поэтому в процессе преобразования условная дуга, помеченная этим выражением, была заменена двумя условными дугами, помеченными этими подвыражениями, и образована промежуточная вершина ***5***.

Затем для каждого из подвыражений по таблице соответствия условные дуги были заменены на нужные фрагменты. В результате образовались вершины ***4*** и ***6*** с соответствующими дугами. Продолжение этого процесса привело к образованию показанного графа (заметим, что на рис. 1.16 не показана финальная вершина ***18***, соответствующая акцепту слова EndOfFile).

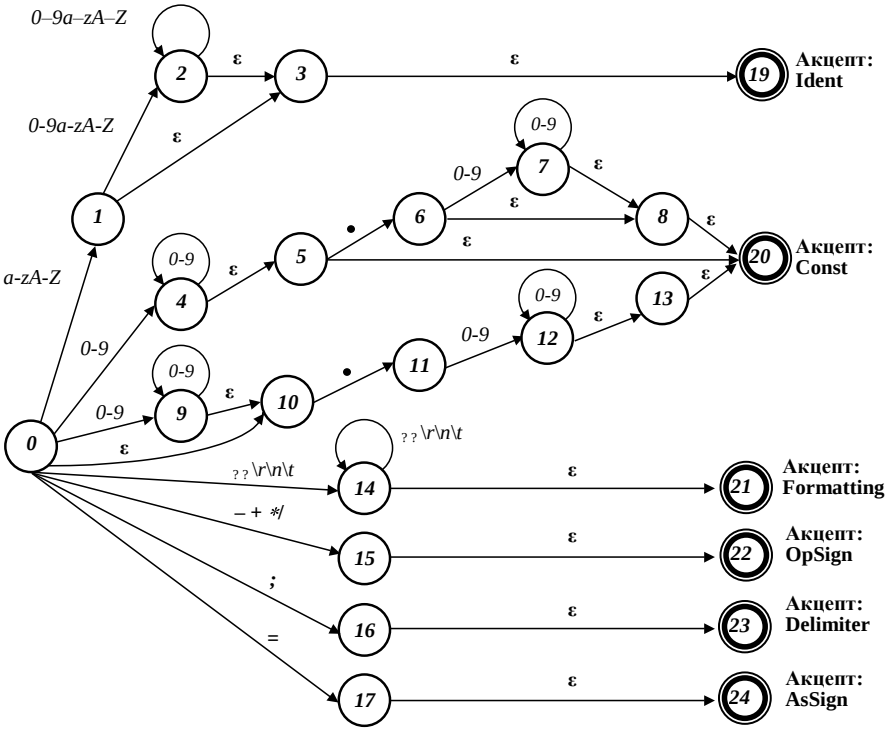


Рис. 1.16. Недетерминированный неоптимальный автомат, построенный по системе определений RESystem1

Управляющая таблица этого автомата дана в табл. 1.2.

Таблица 1.2

Состояние	*a–zA–Z*	*0–9a–zA–Z*	*0–9*	**.**	*\r\n\t*	*–+*/*	*;*	=	ε
0	1		4,9		14	15	16	17	18
1		2							3
2		2							3
3									19
4			4						5
5				6					20
6			7						8
7			7						8
8									20
9			9						10
10				11					
11			12						
12			12						13
13									20
14					14				21
15						15			22
16									23
17									24
18	Акцепт: EndOfFile								
19	Акцепт: Ident								
20	Акцепт: Const								
21	Акцепт: Formatting								
22	Акцепт: OpSign								
23	Акцепт: Delimiter								
24	Акцепт: AsSign								

Легко видеть, что это не полностью определенный недетерминированный, а следовательно, неоптимальный автомат. Для решения задач лексического анализа в трансляторах такой автомат должен быть преобразован в эквивалентный ему детерминированный оптимальный автомат. Рассмотрим методы этого преобразования.

1.4.3.2. Преобразование недетерминированного неоптимального автомата в детерминированный оптимальный

Критерии, которым должен удовлетворять оптимальный автомат, были сформулированы в пункте 1.4.1.7. Преобразование недетерминированного неоптимального автомата в детерминированный оптимальный можно осуществить путем последовательной проверки всех этих критериев. Если автомат не удовлетворяет проверяемому критерию, следует выполнить эквивалентное преобразование автомата, устраняющее нарушение этого критерия.

После того как будут проверены все критерии и выполнены все эквивалентные преобразования, автомат станет детерминированным оптимальным.

Приведем алгоритмы преобразований для каждого критерия.

Устранение пересечений множеств символов разметки столбцов управляющей таблицы (критерий 1)

Каждый столбец управляющей таблицы, имеющий в заголовке множество из $k > 1$ символов, следует заменить на k столбцов, поместив в их заголовки по одному символу из этого множества. При этом в таблице могут возникнуть столбцы, имеющие одинаковые заголовки. Затем объединить все одинаково помеченные столбцы. При этом могут возникнуть в явном виде недетерминированности второго рода. Добавить в таблицу по одному пустому столбцу для каждого символа из используемой таблицы кодировки, отсутствующему в заголовках существующих столбцов. В результате в управляющей таблице образуется ровно столько столбцов, сколько символов есть в используемой системе кодирования плюс один столбец, помеченный обозначением пустой цепочки символов (если, например, используется система кодирования ASCII, то управляющая таблица будет содержать 257 столбцов).

Применительно к представлению автомата графом состояний и переходов это преобразование состоит в том, что каждая дуга, помеченная более чем одним символом, превращается в k параллельных дуг, каждая из которых помечена в точности одним символом и ведет в ту же самую вершину, что и исходная дуга.

Устранение недетерминированностей (критерии 2 и 3)

Основная идея этого преобразования состоит в том, чтобы поставить в соответствие каждому подмножеству состояний (вершин в графе состояний и переходов) исходного автомата, в которых он может оказаться «одновременно» в результате недетерминированных переходов, в точности одно состояние (вершину графа) эквивалентного ему детерминированного автомата.

Если исходный автомат имеет n состояний, то множество всех непустых подмножеств его состояний включает ровно 2^n–1 подмножеств.

Начальному состоянию (вершине s_0) исходного автомата соответствует подмножество $\{s_0\}$, содержащее единственную начальную вершину и являющееся, по сути, начальной вершиной результата преобразования.

Остальные состояния детерминированного автомата и переходы между состояниями формируются путем выполнения следующего алгоритма, в котором используются отмеченные и неотмеченные множества вершин исходного автомата.

Шаг 1. Подготовка: формирование начального неотмеченного подмножества $\{s_0\}$.

Шаг 2. Основной цикл: организуется перебор и обработка неотмеченных подмножеств. Если ни одного такого подмножества нет – выход из основного цикла, переход к шагу 3. Каждое подмножество вершин после обработки помечается, т. е. исключается из перечня неотмеченных. Обработка неотмеченного подмножества заключается в следующих действиях.

Шаг 2.1. Построение ε-замыкания текущего подмножества вершин: в текущее подмножество включаются все те рабочие (не финальные) вершины, в которые есть дуги переходов, помеченные пустой цепочкой **ε** из какой бы то ни было вершины текущего подмножества. Процесс замыкания осуществляется путем перебора всех переходов по пустой цепочке **ε** из вершин текущего подмножества до тех пор, пока оно не перестанет изменяться или не будет исчерпан перечень таких переходов

Шаг 2.2. Поиск сформированного подмножества среди всех ранее обработанных (отмеченных) подмножеств вершин. Если в точности такого подмножества вершин еще не было образовано, то добавление его к перечню отмеченных подмножеств.

Шаг 2.3. Определение набора символов, помечающих все дуги переходов из вершин текущего подмножества (в том числе символа **ε**, помечающего дуги переходов в финальные состояния исходного автомата).

Шаг 2.4. Внутренний цикл: перебор всех символов набора. Для каждого из символов построенного на шаге 2.3 набора (обозначим такой символ как *x*) выполняется формирование неотмеченного подмножества вершин, в которые существуют дуги переходов из состояний текущего подмножества, помеченные этим символом. Каждое сформированное подмножество добавляется к перечню неотмеченных подмножеств, если его там еще нет. Формируется дуга перехода из текущего подмножества вершин во вновь сформированное подмножество, помеченная символом *x*.

Шаг 3. Завершение. Сопоставление каждому образованному в результате выполнения шагов 1 и 2 подмножеству вершин исходного графа в точности одной вершины нового графа. Если в рассматриваемом множестве есть хотя бы одна финальная вершина исходного автомата, то и поставленная ему в соответствие вершина нового графа считается финальной. Ассоциированная с финальной вершиной исходного графа группа слов теперь считается ассоциированной с финальной вершиной результата преобразования.

Автомат, полученный таким образом из исходного автомата, не будет содержать недетерминированностей ни первого, ни второго рода за счет:

– выполнения **ε**-замыкания каждого подмножества (шаг 2.1), в результате чего ликвидируются недетерминированности первого рода;

– формирования единого подмножества вершин с использованием всех дуг переходов по символу *x* из текущего подмножества на шаге 2.4, за счет чего ликвидируются недетерминированности второго рода.

Проиллюстрируем работу этого алгоритма на примере автомата, граф состояний и переходов которого показан на рис. 1.16.

Шаг 1. Начальное неотмеченное подмножество P_0 включает в себя вершину 0; множество неотмеченных подмножеств N содержит единственное подмножество P_0:

$$P_0 = \{0\}, \quad N = \{P_0\}$$

Шаг 2. Первое повторение основного цикла. Для обработки берем подмножество P_0 и применяем к нему операцию ε-замыкания:

$$P_0 = \{0, 10\}$$

Поскольку множество всех отмеченных подмножеств пока пусто, добавляем к нему подмножество P_0 и продолжаем его обработку, формируя набор символов, помечающих дуги, выходящие из вершин 0 и 10 исходного графа:

$$S_0 = \{a - zA - Z0 - 9.\sqcup \backslash r \backslash n \backslash t + */;= -\}$$

Для краткости здесь сохранены обозначения диапазонов символов вида $a - zA - Z0 - 9$. Теперь, перебирая все символы из множества S, строим различающиеся подмножества вершин, в которые ведут дуги из вершин 0 и 10 исходного графа. Таких неотмеченных подмножеств будет сформировано 7:

$$P_1 = \{1\},\ P_2 = \{4,9\},\ P_3 = \{11\},\ P_4 = \{14\},\ P_5 = \{15\},\ P_6 = \{16\},\ P_7 = \{17\}$$

Для каждого из них запомним перечень символов, помечающих дуги переходов из P_0.

Шаг 3. Второе повторение основного цикла. Для обработки берем подмножество P_1 (удаляя его из множества неотмеченных подмножеств) и применяем к нему операцию **ε**-замыкания:

$$P_1 = \{1, 3\}$$

В точности такого отмеченного подмножества нет, поэтому P_1 добавляется к множеству отмеченных подмножеств и для него формируется набор символов:

$$S_1 = \{a - zA - Z0 - 9\boldsymbol{\varepsilon}\}$$

Заметим, что в этот набор вошло обозначение пустой цепочки **ε**, поскольку из вершины *3* есть переход по пустой цепочке **ε** в финальную вершину.

Перебор всех символов этого набора позволит построить новые неотмеченные подмножества вершин:

$$P_8 = \{2\},\quad P_9 = \{19\}$$

Продолжение выполнения этого алгоритма (которое рекомендуется проделать читателю) в конечном итоге за конечное количество шагов приведет к образованию графа состояний и переходов детерминированного (но еще не оптимального) конечного автомата, показанного на рис. 1.17. Конечность количества шагов преобразования обусловлена тем, что множество всех подмножеств вершин конечного графа также конечно.

Для отображения связей этого автомата с исходным автоматом все вершины построенного графа помечены построенными в процессе

преобразования подмножествами вершин исходного графа, показанного на рис. 1.16.

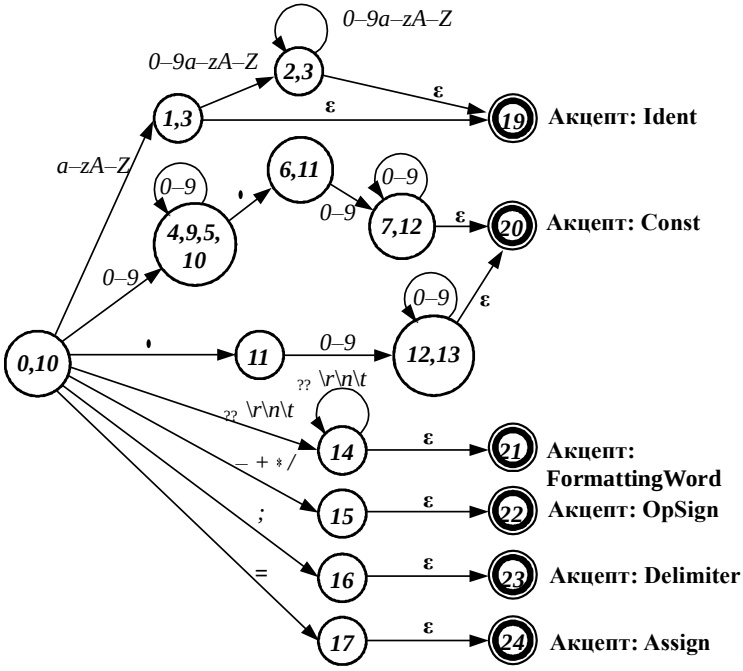


Рис. 1.17. Детерминированный неоптимальный автомат

Удаление тупиковых состояний (критерий 4)

Найти все тупиковые состояния, если они есть. Для этого необходимо определить списки достижимых финальных состояний для каждого рабочего состояния. Это производится следующим образом.

Т.1. С каждой строкой ***i*** управляющей таблицы сопоставляется список S_i (в начальный момент – пустой) достижимых финальных состояний, в которые есть переходы из данной строки.

Т.2. Начиная с нулевой, просматриваются все строки таблицы.

Т.3. В каждой строке (пусть ее номер ***n***) просматриваются все клетки.

Т.4. Если клетка не пуста и содержит номер финального состояния, то этот номер добавляется к списку S_n, если его нет в этом списке.

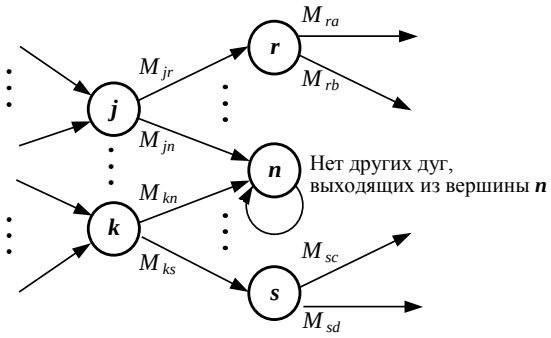
T.5. Если клетка не пуста и содержит номер рабочего состояния ***k*** (***k*** не равно ***n***), то все элементы списка S_k финальных состояний строки ***k*** добавляются к списку S_n финальных состояний строки *n* (естественно, если их нет в списке S_n).

T.6. Если была просмотрена последняя строка рабочей зоны и в процессе просмотра изменился хотя бы один список S_i, то нужно вернуться к шагу *T.2*, иначе процесс завершается.

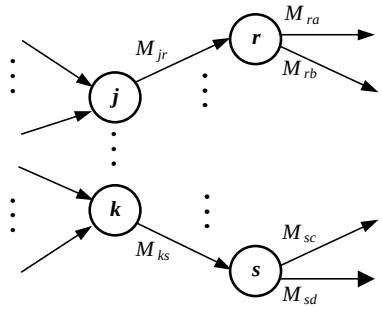
Все состояния, для которых списки достижимых финальных состояний пусты, являются тупиковыми и их следует удалить. Для этого необходимо:

- удалить из всех клеток таблицы все переходы в выявленные тупиковые состояния;
- удалить строки тупиковых состояний из управляющей таблицы.

В графе состояний и переходов это преобразование состоит в замене всех фрагментов вида



на фрагменты следующего вида



Очевидно, что в результате такого преобразования будет получен автомат, эквивалентный исходному, поскольку любая история работы преобразованного автомата может быть реализована исходным.

Удаление всех тупиковых состояний может привести к образованию эквивалентных состояний, которые в исходном автомате таковыми не являлись.

После удаления всех тупиковых состояний перейти к шагу 5.

Удаление недостижимых состояний (критерий 5)

Найти все недостижимые состояния, если они есть. Для этого производится следующее.

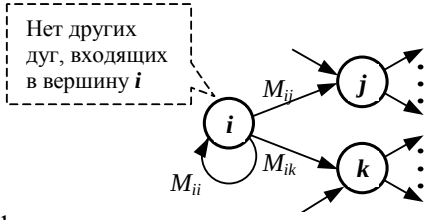
Н.1. С каждой строкой рабочей зоны управляющей таблицы (кроме нулевой строки) сопоставляется отметка «недостижима», а с нулевой строкой – «достижима».

Н.2. Начиная с нулевой просматриваются все «достижимые» строки.

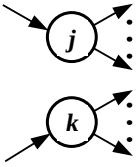
Н.3. В каждой строке просматриваются все клетки, и если в клетке находится номер рабочего состояния ***k***, то значение отметки строки ***k*** изменяется на значение «достижима».

Н.4. Если просмотрена последняя строка таблицы и хотя бы для одной строки отметка со значения «недостижима» изменилась на значение «достижима» – возвращение к шагу *Н*.2, иначе завершение процесса.

Все строки, отметка которых имеет значение «недостижима», следует удалить из таблицы. В графе состояний и переходов это преобразование состоит в замене всех фрагментов вида



на фрагменты вида



Очевидно, что в результате такого преобразования будет получен автомат, эквивалентный исходному, поскольку любая история работы преобразованного автомата может быть реализована и исходным автоматом, но не наоборот.

Удаление недостижимых состояний может привести к образованию эквивалентных состояний, которые в исходном автомате таковыми не являлись.

После удаления всех недостижимых состояний перейти к шагу 6.

Слияние эквивалентных состояний (критерий 6)

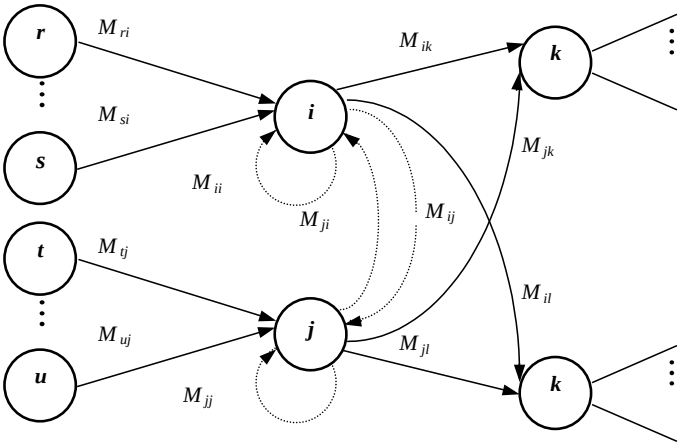
Для обнаружения таких состояний (если они есть) необходимо просмотреть все возможные пары строк рабочей зоны управляющей таблицы. Если найдется пара таких строк ***i*** и ***j***, содержимое которых полностью совпадает или различается в клетках одного и того же столбца только переходами внутри этой пары (т. е. парами переходов вида {***i*→*i*** и ***j*→*j***} или {***i*→*j*** и ***j*→*i***}, или {***i*→*j*** и ***j*→*j***} или {***i*→*i*** и ***j*→*i***}), то:

– удалить любую из них (например, строку ***j***) из таблицы;

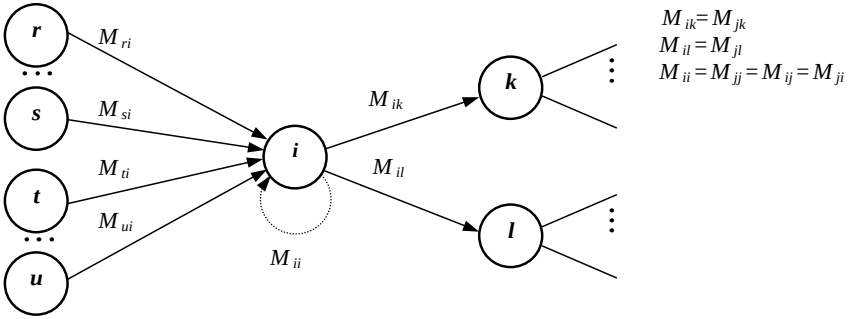
– во всех клетках остальных строк таблицы заменить переходы в удаленное состояние ***j*** переходами в оставленное состояние ***i***;

– вернуться к выполнению шага 6.

Заметим, что любое такое преобразование может привести только к нарушению критерия **6**. В графе состояний и переходов это преобразование состоит в замене фрагмента вида



на фрагмент вида



Не очень очевидно, но такое преобразование порождает автомат, эквивалентный исходному. Для доказательства этого достаточно просто отметить следующее.

1. Это преобразование не влияет как на истории работы до момента перехода в состояния ***i*** или ***j*** исходного автомата, так и на истории работы до момента перехода в состояния ***i*** автомата, полученного в результате преобразования.

2. Начиная с этих моментов для любой входной последовательности символов оба автомата обязательно реализуют совершенно одинаковые остатки историй работы.

После обнаружения и слияния всех множеств эквивалентных состояний перейти к шагу 7.

Слияние одинаковых столбцов (критерий 7)

Просмотреть попарно все столбцы управляющей таблицы. Если найдется пара одинаковых столбцов, помеченных множествами символов M_1 и M_2, то:

- заголовок одного из них заменить на объединение множеств M_1 и M_2;
- удалить другой столбец;
- вернуться к выполнению шага 7.

Это преобразование не может привести к нарушению ни одного из критериев. В графе состояний и переходов это преобразование состоит в объединении параллельных дуг и их разметок.

При использовании управляющей таблицы, столбцы которой помечены множествами символов, программа, моделирующая поведение

конечного автомата, обязана выполнять преобразование кода каждого входного символа в индекс столбца таблицы. Такое преобразование обычно производится с использованием вспомогательной одномерной таблицы, называемой транслитератором. Транслитератор – это одномерный целочисленный массив. Каждый элемент этого массива содержит индекс столбца управляющей таблицы для символа, код которого рассматривается как индекс элемента.

Формирование полностью определенного автомата (критерий 8)

В результате выполнения шагов 1–7 алгоритма будет получена управляющая таблица автомата (табл. 1.3), содержащая столбец, помеченный обозначением пустой цепочки. Некоторые клетки этой таблицы могут быть пусты. Каждая непустая клетка содержит единственный номер состояния перехода.

Таблица 1.3

Состояние	*a*–z*A*–*Z*	*0*–*9*	.	\r\n\t	–+*/	;	=	ε	Прочие
0	1	5	3	4	6	7	8	9	
1	1	1						10	
2		2						11	
3		2							
4				4				12	
5		5	2					11	
6								13	
7								14	
8								15	
9	Акцепт: EndOfFile								
10	Акцепт: Ident								
11	Акцепт: Const								
12	Акцепт: Formatting								
13	Акцепт: OpSign								
14	Акцепт: Delimiter								
15	Акцепт: AsSign								
16	Отказ: Error								

Рассмотрим для примера состояние управляющей таблицы автомата, построенного по системе регулярных определений RESystem1 в результате выполнения шагов 1–7. Строки управляющей таблицы могут быть пронумерованы не по порядку, поскольку некоторые состояния могли быть добавлены, а некоторые – удалены. Перенумеруем их последовательно, начиная с нуля, одновременно приводя в соответствие все переходы на те состояния, номера которых изменяются.

Граф состояний и переходов автомата изображен на рис. 1.18.

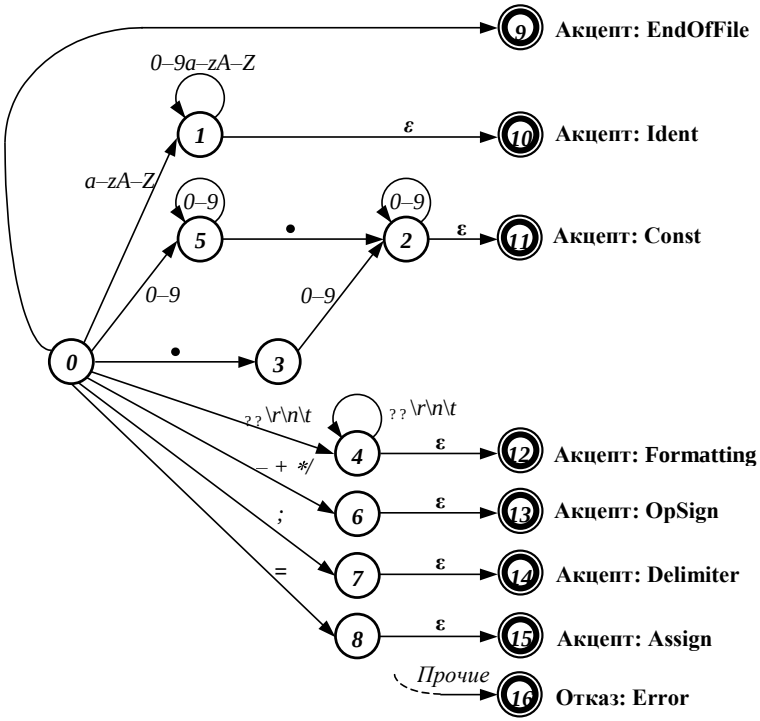


Рис. 1.18. Детерминированный оптимальный автомат, полученный из системы определений RESystem1

Для преобразования такого автомата в полностью определенный необходимо уяснить смысл понятия пустой цепочки символов, положенной в основу способа формирования графа состояний и переходов по системе регулярных определений. Пустую цепочку символов мож-

но (а в автоматной модели для не полностью определенных автоматов просто нужно) понимать как последовательность трех действий:

– чтение входного символа;

– анализ символа (проверка наличия перехода из данного состояния по данному символу);

– возврат символа на вход, если такого перехода нет.

Теперь ясно, что если в некоторой строке управляющей таблицы клетка в столбце, помеченном обозначением пустой цепочки символов, содержит номер состояния перехода, то этот номер следует занести во все пустые клетки данной строки. Тем самым будут реализованы переходы по пустой цепочке символов. Если же такая клетка пуста, то во все пустые клетки данной строки необходимо занести номер состояния останова по обнаружению лексической ошибки. В зависимости от целей, которые преследует построение автомата, можно предусмотреть либо единственное состояние останова по ошибке, либо множество таких состояний. В последнем случае номер состояния, в котором остановился автомат, может использоваться для формирования развернутых диагностических сообщений.

Таблица 1.4

Состояние	*A–zA–Z*	*0–9*	.	*\r\n\t*	*–+*/*	;	=	*EOF*	Прочие
0	1	5	3	4	6	7	8	9	16
1	1	1	10	10	10	10	10	10	16
2	11	2	11	11	11	11	11	11	16
3	16	2	16	16	16	16	16	16	16
4	12	12	12	4	12	12	12	12	16
5	11	5	2	11	11	11	11	11	16
6	13	13	13	13	13	13	13	13	16
7	14	14	14	14	14	14	14	14	16
8	15	15	15	15	15	15	15	15	16
9	Акцепт: EndOfFile								
10	Акцепт: Ident								
11	Акцепт: Const								
12	Акцепт: Formatting								
13	Акцепт: OpSign								
14	Акцепт: Delimiter								
15	Акцепт: AsSign								
16	Отказ: Error								

И, наконец, обозначение пустой цепочки в заголовке столбца необходимо заменить псевдосимволом конца файла (*EOF*), которое не следует путать со словом EndOfFile. Заполнив пустые клетки и переименовав столбец, помеченный пустой цепочкой, получим управляющую таблицу автомата (табл. 1.4).

Дополнительная оптимизация управляющей таблицы

Обратим внимание на рабочие состояния с номерами ***6***, ***7*** и ***8***, строки для которых содержат одни и те же переходы во всех клетках. Эти состояния существуют в рабочей зоне таблицы только потому, что согласно определению автомата любой переход в финальное состояние сопровождается возвратом символа на вход, а любой переход в рабочее состояние – чтением следующего символа. Если в составе каждой клетки рабочей зоны предусмотреть флажок, явным образом управляющий возвратом текущего символа на вход, то состояния ***6***, ***7*** и ***8*** можно ликвидировать. Вначале в рабочей зоне все переходы в финальные состояния акцепта слов помечаются флажком возврата, затем находятся и удаляются все состояния, содержащие в каждой клетке (за исключением столбца прочих символов) переход в одно финальное состояние акцепта. Все переходы на удаленные состояния заменяются переходами на соответствующие состояния акцепта (со сброшенным флажком возврата символа). Обозначим установленный флажок возврата минусом (–) и приведем окончательный вид управляющей таблицы автомата (табл. 1.5).

Состояния ***0***…***5*** являются рабочими, остальные – финальными (напомним, что хранить в памяти компьютера нужно только рабочую зону таблицы). Столбец прочих символов, содержащий совершенно одинаковые переходы в состояние останова по ошибке, также можно не хранить в памяти, но это уже зависит от реализации автомата в виде программной модели.

Дальнейшее уменьшение количества клеток рабочей зоны прямоугольной управляющей таблицы невозможно без потери функциональности автомата. Если это необходимо, то уменьшить объем занимаемой памяти можно путем организации хранения прямоугольной таблицы в виде совокупности линейных списков, содержащих упакованные строки. При упаковке строки таблицы представляются одномерными массивами, каждый элемент которых содержит адрес перехода и количество одинаковых столбцов. Например, строка 3 полученной таблицы могла бы быть представлена в виде такого списка: {13,1}{2,1}{13,7}.

Таблица 1.5

Состояние	*a–zA–Z*	*0–9*	**.**	\r\n\t	–+*/	*;*	=	*EOF*	Прочие
0	1	5	3	4	10	11	12	6	13
1	1	1	–7	–7	–7	–7	–7	7	13
2	–8	2	–8	–8	–8	–8	–8	8	13
3	13	2	13	13	13	13	13	13	13
4	–9	–9	–9	4	–9	–9	–9	9	13
5	–8	5	2	–8	–8	–8	–8	8	13
6	Акцепт: EndOfFile								
7	Акцепт: Ident								
8	Акцепт: Const								
9	Акцепт: Formatting								
10	Акцепт: OpSign								
11	Акцепт: Delimiter								
12	Акцепт: AsSign								
13	Отказ: Error								

Сокращения объема памяти, занимаемой управляющей таблицей автомата, можно добиться также специальными методами сжатия прямоугольных таблиц. Рассматривать такие методы мы не будем, отметим только, что любое уплотнение данных сопряжено с некоторым ростом затрат времени на работу с этими данными. Выбор оптимального варианта организации управляющих таблиц остается за разработчиком транслятора.

В принципе, существует возможность формирования нескольких автоматов на основе одной системы регулярных определений и управляемого переключения с одного автомата на другой в процессе функционирования лексического акцептора. Целью такого способа построения лексического акцептора обычно является реализация распознавания блочных вложенных комментариев (например – комментариев в стиле языка Си) на этапе лексического анализа.

В следующем пункте приводится пример системы регулярных определений, рассчитанной на преобразование не менее чем в три автомата и обеспечивающей распознавание как однострочных, так и блочных вложенных комментариев в стиле языка Си. При этом в качестве

побочного может быть достигнут эффект уменьшения суммарного объема управляющих таблиц нескольких автоматов по сравнению с таблицей единственного автомата для одной и той же совокупности групп распознаваемых слов.

Таким образом, мы рассмотрели методы и алгоритмы построения лексических акцепторов путем преобразования совокупности правил, описывающих лексику формального языка, в конечный автомат без памяти. Практическое применение этого подхода в трансляторах требует расширения понятия систем регулярных определений, для того чтобы получить на их основе полнофункциональные лексические анализаторы.

1.4.3.3. Включение действий в системы регулярных определений

Под действием понимается последовательность операторов на том языке программирования, на котором будет реализовываться программная модель конечного автомата (см. рис. 1.9), ассоциируемая с регулярным определением. Действия записываются в системе регулярных определений в качестве дополнительной колонки, например:

Автомат	**Имя группы слов**	**Регулярное выражение**	**Действие**
main	Ident	[*a–zA–Z*][*0–9a–zA–Z*]*	tables.PutToIdentTable(Lexem.textOfWord);
main	Const	([*0–9*]+([.][*0–9*]*)?	
main	Const	[*0–9*]*[.][*0–9*]+)	tables.PutToConstTable(Lexem.textOfWord);
main	Formatting	[\r\n\t]+	ignoreLastWord=true;
main	OpSign	[–+*/]	
main	Delimiter	[;]	
main	AsSign	[=]	
main	LineCmnt	[/][/]([^\r\n])*	ignoreLastWord=true;
main	BlockCmnt	[/][*]	ignoreLastWord=true; lStk.push(lexAcceptor); lexAcceptor=lexAcceptors [indexOfAutomat("body")];
body	BlockCmnt	[/][*]	ignoreLastWord=true; lStk.push(lexAcceptor); lexAcceptor=lexAcceptors [indexOfAutomat("body")];

body	BodyCmnt	other+	ignoreLastWord=true;

body	MayByEnd	[*]	ignoreLastWord=true;
			lStk.push(lexAcceptor);
			lexAcceptor=lexAcceptors
			[indexOfAutomat("test")];

test	EndCmnt	[/]	ignoreLastWord=true;
			lStk.pop();
			lexAcceptor=lStk.pop();

test	BodyCmnt	other	ignoreLastWord=true; lexAcceptor=lStk.pop();

В первом столбце указаны имена автоматов, ответственных за распознавание соответствующих групп слов. Данный пример системы регулярных определений ориентирован на синтаксис построителя лексических анализаторов учебного пакета автоматизации проектирования трансляторов Вебтранслаб.

По приведенной системе регулярных определений должны быть построены три автомата:

– основной автомат (с именем main) для обнаружения слов, относящихся к группам Ident, Const, OpSign, Delimiter, AsSign, Formatting, LineCmnt и BlockCmnt, и игнорирования, т. е. подавления слов трех последних групп;

– автомат с именем body, предназначенный для пропуска любых последовательностей символов, не содержащих начала вложенного комментария (последовательность символов **/***) и первого символа признака окончания комментария (символ ***);

– автомат с именем test, распознающий либо окончание комментария (символ / сразу после символа *), либо продолжение комментария (любой символ, отличный от символа / после *) и обеспечивающий необходимую реакцию на каждый из возможных случаев.

Совокупность операторов (действие), сопоставленная какому-либо регулярному определению, будет выполняться программной моделью конечного автомата в момент его перехода в финальное состояние акцепта любого слова данной группы. В вышеприведенной системе определений действия связаны не со всеми заданными группами слов. Вкратце рассмотрим их назначение.

При обнаружении автоматом любого идентификатора (константы) будет вызван метод PutToIdentTable (PutToConstTable) класса tables, по

названию которого можно сделать вывод о том, что обнаруженное слово будет занесено в таблицу идентификаторов/констант (естественно, если оно встретилось в тексте впервые). Детали этого процесса зависят от структуры информационных таблиц и взаимодействия лексического анализатора с другими компонентами транслятора и обсуждаются ниже.

В случае, если лексический акцептор обнаружил некоторую последовательность пробелов, знаков табуляции и/или символов завершения строки (такие последовательности нужны в текстах программ для удобного визуального форматирования), установка значения true флажка ignoreLastWord приведет к тому, что это слово не будет передано на дальнейшую обработку в синтаксический анализатор. Автомат «забудет» обнаруженную последовательность и займется чтением входного текста дальше в поисках следующего «значимого» слова. Флажок ignoreLastWord встроен в программную модель конечного автомата, формируемого построителем пакета Вебтранслаб. Забегая вперед, отметим, что на этапе лексического анализа аналогично подавляются и все комментарии. Строчные комментарии (группа LineCmnt) распознаются как одиночные слова. Обработка блочных комментариев более сложна, ее мы рассмотрим подробнее, однако для всех слов, из которых они фактически состоят, в системе регулярных определений также предусмотрена установка флажка ignoreLastWord в состояние true.

С группами слов OpSign, Delimiter и AsSign в приведенной системе регулярных определений не связано никаких действий. В этом примере предполагается, что для последующей обработки в синтаксическом анализаторе достаточно отнести обнаруженное слово к его группе.

При обнаружении начала блочного комментария (последовательность символов **/***) после подавления выдачи этого слова автомат лексического акцептора переключается с основной управляющей таблицы на вспомогательную таблицу автомата с именем body. Контекст исходной таблицы (индекс текущего автомата lexAcceptor) предварительно сохраняется в специальном стеке для последующего возврата после обработки всего комментария. Согласно системе регулярных определений, автомат body предназначен для обнаружения (и подавления) слов одной из трех групп:

– группа BlockCmnt, слово /* (начало вложенного комментария), по которому осуществляется повторное переключение на автомат body; сохранение в стеке контекста текущей таблицы гарантирует вос-

становление режима чтения комментария верхнего уровня после окончания вложенного;

– группа BodyCmnt, слово, состоящее из любых символов, кроме символа *; такие слова просто подавляются;

– группа MayByEnd, слово, состоящее из единственного символа *; поскольку этим символом может начинаться завершение блочного комментария, автомат после его подавления переключается на таблицу по имени testEnd для принятия окончательного решения.

Автомат testEnd просто проверяет единственный символ, следующий после символа *****. В случае если этим символом является / (или псевдосимвол *EOF*), т. е. обнаружено окончание блочного комментария, после его подавления с верхушки стека контекстов управляющих таблиц сбрасывается один элемент (естественно, это индекс автоамата body). Затем со стека снимается и используется для восстановления оставшийся индекс автомата. Если комментарий не был вложенным, то это индекс автомата main, и лексический акцептор возобновит выявление слов из входного текста вне комментария. Легко видеть, что если завершается вложенный комментарий, то во внутреннем стеке находится не менее трех элементов, причем оба верхних определяют контекст управляющей таблицы автомата body. Тогда описанный механизм приведет к возврату именно на эту управляющую таблицу, т. е. на продолжение обработки охватывающего комментария. В случае, если при переключении на автомат testEnd входным символом является что угодно, кроме символа / (и, естественно, псевдосимвола *EOF*). Этот символ игнорируется, и акцептор возвращается к управляющей таблице автомата body для продолжения чтения текущего комментария.

Отметим, что рассматриваемая здесь мультиавтоматная модель лексического акцептора есть очень простое решение задачи распознавания блочных вложенных комментариев. Невозможно решить такую задачу на этапе лексического анализа при использовании единственного конечного автомата без памяти. Перенос же этой функции (распознавания блочных вложенных комментариев) на этап синтаксического анализа сопряжен с возникновением очень больших трудностей вновь на этапе лексического анализа вследствие того, что внутри комментариев могут встречаться слова естественного языка, для распознавания которых не может быть создана обозримая система регулярных определений.

Как уже упоминалось выше, разбиение единой задачи лексического акцепта на несколько автоматов в качестве побочного эффекта дает

уменьшение суммарного объема таблиц. Приведем для иллюстрации управляющие таблицы автоматов.

Таблица автомата body

Состояние	/	*	*EOF*	Прочие
0	1	5	3	2
1	–6	4	3	2
2	–6	–6	3	2
3	Акцепт: EOF			
4	Акцепт: BlockCmnt {ignoreLastWord=true; lStk.push(lexAcceptor); lexAcceptor=lexAcceptors [indexOfAuto-mat("body")];}			
5	Акцепт: MayBeEnd {ignoreLastWord=true; lStk.push(lexAcceptor); lexAcceptor=lexAcceptors [indexOfAutomat("test")];}			
6	Акцепт: BodyCmnt {ignoreLastWord=true;}			

Таблица автомата testEnd

Состояние	/	*EOF*	Прочие
0	1	1	2
1	Акцепт: EndCmnt {ignoreLastWord=true; lStk.pop(); lexAccep-tor=lStk.pop();}		
2	Акцепт: BodyCmnt {ignoreLastWord=true; lexAccep-tor=lStk.pop();}		

Мы рассмотрели чисто внутренние функции лексического анализатора согласно мультиавтоматной модели, представленной на рис. 1.9. Теперь перейдем к изучению тех функций, способы выполнения которых зависят от взаимных связей лексического анализатора с другими компонентами транслятора. К ним относится работа с таблицами слов и формирование лексем.

1.5. ВЗАИМОДЕЙСТВИЕ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА С СИНТАКСИЧЕСКИМ И СЕМАНТИЧЕСКИМ АНАЛИЗАТОРАМИ

Лексический анализатор предназначен для преобразования входной последовательности символов в последовательность лексем, т. е. внутренних для транслятора эквивалентов слов. Любое слово есть просто

цепочка символов, однако, как уже отмечалось выше, для последующих этапов трансляции это исходное представление слов неудобно. Для них каждое слово нужно заменять внутренним представлением, облегчающим решение задач синтаксического и семантического анализа.

Действительно, синтаксическому анализатору совершенно безразлично, какой именно идентификатор появляется, например, в левой части оператора присваивания. Для него достаточно информации о том, что в этом месте входного текста находится слово из группы идентификаторов. Тогда как для семантического анализатора важно, какими свойствами обладает объект, названный этим идентификатором, для того чтобы проверить правильность данного оператора присваивания. Однако сам по себе текст идентификатора, взятый лексическим анализатором из текущего оператора программы, этой информации в себе не несет.

Вероятно, во входном тексте до рассматриваемого нами оператора присваивания встречалось объявление объекта, сопоставляющее его вид (простая переменная, массив, структура и т. д.), тип значения и класс памяти с данным идентификатором. Очевидно, что при обработке такого объявления семантическим анализатором все атрибуты объекта должны быть сохранены в таблице, а лексический анализатор при каждом появлении идентификатора объекта должен заменять его указателем на таблицу и указателем на конкретную строку в этой таблице – лексемой. Слова других групп обладают другими атрибутами.

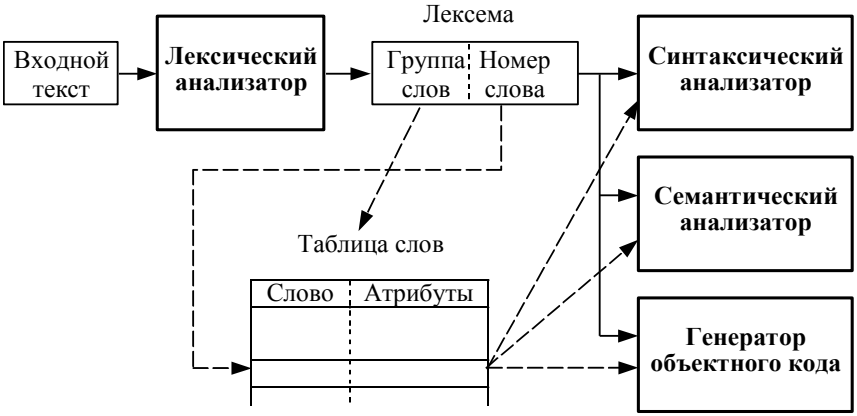


Рис. 1.19. Взаимосвязи компонент транслятора через информационные таблицы

Например, знаки операций могут характеризоваться количеством операндов, к которым они применяются, допустимыми типами значений операндов и типом значения вырабатываемого результата, приоритетом по отношению к другим операциям.

Таким образом, в качестве первого приближения зафиксируем, что:

– для каждой группы слов в трансляторе организуется отдельная таблица;

– строки таблиц содержат как минимум два поля – ключевое поле имени объекта (т. е. текста слова) и поле атрибутов объекта;

– лексический анализатор обеспечивает формирование лексемы на основании текста каждого обнаруженного слова;

– остальные компоненты транслятора используют нужные им элементы лексемы (например, синтаксическому анализатору во многих случаях достаточно указателя на таблицу, т. е. номера группы слов) и при необходимости – атрибуты объектов, выбираемые из таблиц, как это показано на рис. 1.19.

1.5.1. Учет областей видимости

Однако в эту простую схему придется внести существенные поправки, обусловленные свойствами современных языков программирования.

Возможность объявления нескольких различных объектов с одним и тем же именем и связанные с этим правила видимости объектов (или областей действительности объектов) вынуждают рассматривать таблицу идентификаторов как таблицу указателей списков типа «первый пришел – последний ушел», т. е. стеков.

При этом за лексическим анализатором сохраняются функции формирования таблиц слов как таковых, а в задачи синтаксического и семантического анализаторов – поддержание в требуемом виде всех списков.

В момент начала обработки любого блока необходимо модифицировать счетчик, учитывающий глубину вложенности блоков. При обработке любого объявления объекта с данным идентификатором должен создаваться (и помещаться в начало списка) элемент, содержащий текущую глубину блока, атрибуты объекта и указатель на тот элемент, который был первым в списке до этого момента. До тех пор пока не будет достигнут конец блока, по лексеме, в которую превращается

данный идентификатор, доступны атрибуты последнего объявленного объекта. Ранее объявленные объекты с таким же именем не видны. При обработке слова, обозначающего конец блока, семантический (или синтаксический) анализатор перед модификацией текущей глубины вложенности должен обеспечить удаление из всех списков таких элементов, у которых глубина вложенности не меньше, чем глубина покидаемого блока. Указатели, ведущие из этих элементов, переписываются в поле указателей таблицы, чем обеспечивается восстановление видимости ранее объявленных объектов. На рис. 1.20 приведена описанная схема взаимодействия составных частей транслятора.

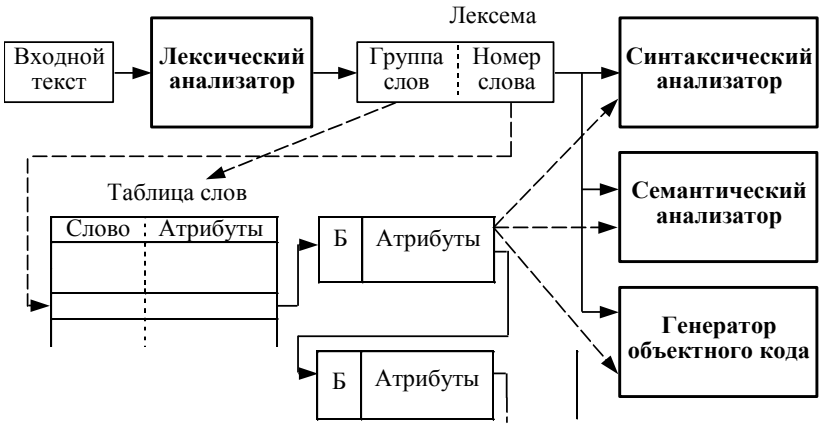


Рис. 1.20. Структура таблиц с учетом областей видимости

1.5.2. Пространства имен

Учет областей видимости не накладывает дополнительных требований на лексический анализатор. Однако такие свойства современных языков программирования, как возможность объявления указателей, структур, перечислений, объединений, классов (сложных структур данных), приводят к более радикальному изменению схемы взаимодействия составных частей транслятора.

Пусть, например, в тексте программы на языке Си встречается такой фрагмент:

```
int x;
struct {
```

```
    char x;
    int * y;
} z;
```

Объявление структуры с именем *z* создает, по существу, совершенно отдельное множество (пространство) имен элементов этой структуры, никак не пересекающееся с общим пространством имен. Элемент структуры с именем *x* не «закрывает» ранее объявленного целого *x*, хотя их идентификаторы совпадают.

Отсюда возникают как минимум два следствия.

Во-первых, для реализации множественных пространств имен в трансляторе вместо одной таблицы слов (для группы идентификаторов) должна строиться деревообразная структура. Некоторые элементы списков, связанных с основной таблицей идентификаторов, должны содержать вместо атрибутов объекта указатель на таблицу, содержащую идентификаторы элементов структуры. Такие элементы строятся для имен структур, объединений, классов.

На рис 1.21 показан один из возможных вариантов реализации нескольких пространств имен.

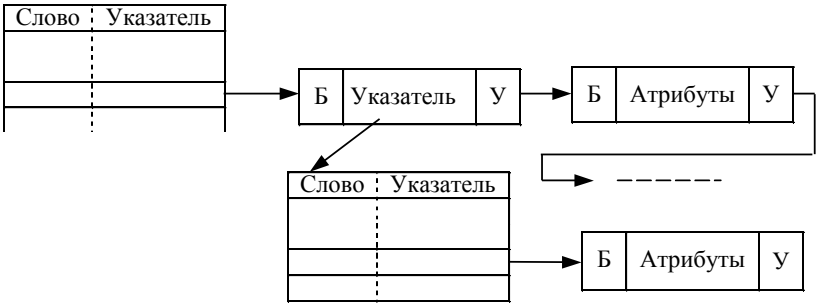


Рис. 1.21. Деревообразная структура таблиц для хранения идентификаторов вложенных структур данных

Во-вторых, перед разработчиком лексического анализатора возникает следующая проблема. Если, например, в тексте программы после вышеприведенного объявления появляется оператор присваивания:

$$x = z.x;$$

то один и тот же идентификатор *x*, очевидно, должен быть преобразован в совершенно разные лексемы, поскольку в левой и в правой частях этого оператора записаны имена разных объектов. Немедленно

возникает вопрос, каким образом лексический анализатор должен определять нужную таблицу для поиска слова и преобразования его в лексему. Можно, конечно, попытаться реализовать на этапе лексического анализа еще и элементы синтаксического для распознавания конструкций вида *x* и *z.x*.

Однако более логична такая реализация, когда синтаксический или семантический анализатор осуществляют управление лексическим анализатором путем переключения контекста поиска в результате обработки такого слова, как точка в вышеприведенном примере. Детали такого переключения контекста зависят от чисто технических аспектов реализации лексического анализатора. Тем не менее структура таблиц групп слов и возможность управления их контекстом со стороны других компонент транслятора существенно влияет на реализацию действий, встраиваемых в систему регулярных определений, на основе которой создается лексический анализатор. Рассмотрим теперь некоторые методы и алгоритмы операций над таблицами. Эти алгоритмы и составляют существо действий, встраиваемых в системы регулярных определений.

1.6. ПОИСК В ТАБЛИЦАХ / ПОПОЛНЕНИЕ ТАБЛИЦ

Рассмотрим таблицу как одномерный массив строк, каждая из которых содержит поле наименования объекта и атрибуты этого объекта либо указатель на элемент списка, содержащий атрибуты. Ранее мы уже упоминали, что существует два типа таблиц: предопределенные и пополняемые. Предопределенные используются для хранения информации об объектах, перечень и свойства которых заданы семантикой языка программирования и не могут изменяться в процессе трансляции программы. К таким объектам относятся базовые арифметические, логические и другие операции, управляющие операторы, всякого рода скобки, влияющие на последовательность выполнения операторов. Для таблиц предопределенных объектов не предусматривается возможность дополнения новых строк во время трансляции. Над этими таблицами может быть определена только функция поиска, возвращающая индекс или номер строки, содержащей заданное наименование объекта. Но для переменных и констант кроме операции поиска должна быть определена еще и операция дополнения (формирования новой строки), выполняемая в том случае, если поиск оказался безуспешным.

Для этапа лексического анализа существенно то, что абсолютно каждое слово, обнаруженное лексическим акцептором, приходится искать в соответствующей таблице и, как показывает практика создания трансляторов, временные затраты на поиск и дополнение могут оказаться существенными.

Рассмотрим некоторые возможные способы организации таблиц, соответствующие алгоритмы поиска и дополнения и их временные оценки, во многом определяющие качество транслятора.

1.6.1. Последовательная организация

В момент начала трансляции исходного текста таблица очищается. Это может быть выполнено, например, путем установки в ноль счетчика количества заполненных (актуальных) строк.

Поиск заключается в последовательном просмотре актуальных строк таблицы начиная с первой строки и сравнении заданного наименования объекта с содержимым поля наименования.

Если искомое наименование ранее было занесено в некоторую строку таблицы, то при просмотре этой строки результат сравнения окажется положительным. Просмотр в этот момент прекращается, а в качестве результата поиска возвращается номер найденной строки.

Но для того чтобы обнаружить отсутствие искомого наименования в таблице, придется просмотреть все до одной ранее заполненные строки. В этом случае в поле наименования первой строки незаполненной части таблицы заносится его текст и эта строка становится последней в заполненной части таблицы. Соответственно в качестве результата возвращается ее номер (индекс, адрес).

Оценим временные затраты на выполнение операций поиска и пополнения. Вначале рассмотрим поиск одного слова. В случае если искомого слова в таблице нет, время поиска прямо пропорционально количеству заполненных строк:

$$t_{\text{п}} \sim= N_{\text{т}}\, t_{\text{ср}}\,.$$

Здесь $t_{\text{п}}$ – время поиска; $N_{\text{т}}$ – количество актуальных строк в таблице к моменту начала поиска; $t_{\text{ср}}$ – время, затрачиваемое на сравнение двух текстовых значений.

Время пополнения для этого случая есть константа, величина которой не зависит от состояния таблицы, а определяется только алгоритмом формирования новой строки:

$$t_{\text{д}} = C.$$

Когда искомое слово уже находится в таблице, мы сделаем предположение, что соответствующая строка с равной вероятностью может быть первой, второй, …, последней в заполненной части таблицы. Тогда, очевидно, время поиска пропорционально:

$$t_{\text{п}} \sim= (N_{\text{т}}\, t_{\text{ср}}) / 2,$$

а время пополнения равно нулю:

$$t_{\text{д}} = 0.$$

При последовательной организации общие временные затраты на поиск в таблице прямо пропорциональны сумме произведений длины программы k_i, измеряемой в словах каждой группы, на количество слов в этой группе N_i, а время, затрачиваемое на пополнение, прямо пропорционально количеству объектов, определяемых в программе (т. е. количеству различных идентификаторов $N_{\text{ид}}$ и констант $N_{\text{к}}$):

$$T_{\text{п}} \sim = \sum (k_i * N_i),$$

$$T_{\text{д}} \sim = N_{\text{ид}} + N_{\text{к}}.$$

В формуле оценки общего времени поиска по существу скрывается почти квадратичный характер его зависимости от длины транслируемой программы в словах. Для небольших исходных текстов при высокой производительности современных компьютеров этим можно пренебречь. Однако для программ, содержащих большое количество слов (тысячи и десятки тысяч), затраты времени на трансляцию при последовательной организации таблиц могут оказаться слишком велики. Поэтому в трансляторах, как правило, применяются более совершенные алгоритмы поиска в таблицах/пополнения таблиц.

1.6.2. Упорядоченная организация

В момент начала трансляции исходного текста таблица пуста. Занесение новых строк производится таким образом, чтобы в актуальной части таблицы все строки были упорядочены либо по возрастанию, либо по убыванию содержимого поля наименования. При поиске операция сравнения искомого наименования с содержимым поля наименования некоторой строки возвращает в качестве результата одно из трех значений, которые можно интерпретировать следующим образом:

– «равно» – искомое слово найдено в данной строке;

– «выше» – искомое слово, возможно (но не обязательно), находится в одной из предшествующих строк;

– «ниже» – искомое слово, возможно (но не обязательно), находится в одной из последующих строк.

Собственно поиск производится следующим образом (так называемое деление пополам).

Шаг 1. Подготовка. Устанавливаются начальные значения двух переменных, которые будут ограничивать текущий диапазон поиска. Переменная «Низ» получает значение, равное номеру первой строки таблицы, а переменная «Верх» – номеру последней актуальной строки. Ясно, что если таблица пуста, то поиск сразу завершается с отрицательным результатом и записью номера первой строки таблицы в переменную «Середина» (см. ниже).

Шаг 2. Сравнение. Вычисляется номер средней строки в текущем диапазоне как результат целочисленного деления на 2 суммы значений переменных «Низ» и «Верх». Этот номер запоминается в качестве значения переменной «Середина» и искомое слово сравнивается с содержимым поля наименования средней строки. Если наименования совпали, то поиск завершается успешно, номер найденной строки возвращается в качестве результата, иначе запоминается результат сравнения («выше» или «ниже»).

Шаг 3. Проверка. Если значение переменной «Верх» равно значению переменной «Низ», то поиск завершается с отрицательным результатом. (Заметим, что значение переменной «Середина» нам еще понадобится, его надо каким-либо образом передать функции, выполняющей пополнение. К тому же в этот момент значения всех трех переменных равны.)

Шаг 4. Деление пополам. Если результат сравнения есть «выше», то переменная «Верх» получает значение, равное значению «Середина» минус 1, в противном случае переменная «Низ» получает значение, равное значению «Середина» плюс 1, и осуществляется переход к шагу 2.

Если поиск завершился неудачно, то номер строки последнего выполненного сравнения (значение переменной «Середина») указывает на то место, где должна находиться информация об этом слове. Однако в строке с этим номером находится информация о другом слове, попавшем в таблицу ранее. Поэтому операция пополнения должна выполнить следующие действия.

Шаг 1. Освобождение строки. Перенести на одну позицию вниз все актуальные строки таблицы начиная со строки с номером, равным

значению переменной «Середина». Увеличить на единицу счетчик актуальных строк.

Шаг 2. Пополнение. Занести в строку с номером «Середина» наименование искомого слова, сформировать, если это требуется, или очистить значения остальных полей этой строки. Возвратить значение переменной «Середина» в качестве результата.

Такое сочетание алгоритмов поиска и пополнения гарантирует, что строки в таблице всегда будут упорядочены по наименованию. А это, в свою очередь, ускорит поиск по сравнению с последовательной организацией таблицы. Приведем временные оценки. При отсутствии слова в таблице и текущем количестве актуальных строк $N_т$ время его поиска равно:

$$t_п \sim = \log_2 (N_т)\, t_{ср.}$$

Поскольку слово не найдено, осуществим его добавление, на которое придется израсходовать время, пропорциональное половине количества актуальных строк в таблице:

$$t_д \sim = (N_т * t_{пер}) / 2 + C,$$

где $t_{пер}$ – затраты времени на перенос одной строки; C – константа, определяемая способом формирования одной строки.

Если слово в таблице уже есть, то

$$t_п = (\log_2 (N_т) * t_{ср}) / x,$$

$$t_д = 0,$$

где величина $x > 1$ означает, что искомая строка может быть найдена при первом (с вероятностью $1/N_т$), втором (с вероятностью $2/N_т$), … последнем (с вероятностью 1, поскольку слово в таблице есть) сравнении.

Полное время поиска и пополнения при трансляции всей программы, содержащей k_1 слов первой группы (общее количество); k_2 слов второй группы, …; k_n слов последней группы, есть

$$T_п \sim= \sum (k_i \log_2 (N_i) / x_i).$$

При трансляции больших программ (для которых и актуальны эти оценки) временные затраты определяются прежде всего процессами поиска идентификаторов и констант. Поэтому приведем оценки времени поиска для этих двух групп слов:

$T_п \sim= \log_2 (N_{ид} + N_к) / 2$, если идентификаторы и константы хранятся в одной таблице;

$T_п$ ~= ($\log_2$ ($N_{ид}$) + $\log_2$ ($N_к$)) / 2, если идентификаторы и константы хранятся в разных таблицах;

Таким образом, вместо «почти» квадратичной зависимости времени поиска от длины программы в словах при последовательной организации таблиц упорядоченная организация характеризуется «почти» логарифмической зависимостью. Однако уменьшение затрат времени на поиск сопровождается некоторым увеличением затрат времени на пополнение таблиц. Эти затраты могут быть уменьшены путем организации дополнительной индексной таблицы. Если хорошо подумать, то окажется, что дополнительную индексную таблицу вводить необходимо, поскольку передвигать строки в основной таблице нельзя. Слово, однажды помещенное в таблицу и преобразованное в лексему с использованием номера строки, должно превращаться в ту же самую лексему при каждом его появлении во входном тексте независимо от того, пополнялась таблица другими словами или нет.

1.6.3. Рандомизированная организация

Рандомизированная организация характеризуется «почти» линейной зависимостью затрат времени на поиск от общей длины программы в словах и на пополнение – от количества различных слов. В начальный момент резервируется память под таблицу, фиксируется количество строк в ней (в течение времени трансляции одной программы этот параметр таблицы не может быть изменен, поэтому размер таблицы заведомо должен превышать количество различных идентификаторов и/или констант; для ускорения работы количество строк желательно выбирать равным степени двойки) и каждая строка помечается как пустая (неактуальная). Для пометки строк можно использовать специальные флажки или специальные (пустые или пробельные) значения в поле наименования слова. Количество актуальных строк можно не подсчитывать и не хранить.

Алгоритм поиска / пополнения заключается в следующем.

Шаг 1. Формирование индекса. Текстовое значение искомого слова с помощью функции рандомизации преобразуется в индекс – число, находящееся в диапазоне от нуля до N–1, где N – количество строк в таблице. О том, что такое функция рандомизации, скажем ниже. В данный момент выскажем только пожелание, чтобы эта функция тексты разных слов преобразовывала в разные значения индексов, но при каждом ее применении к одному и тому же слову выдавала один и тот же индекс.

Шаг 2. Проверка занятости. Проверяется актуальность строки с индексом, полученным на шаге 1. Если строка занята, то переход к шагу 3. Если же строка не занята, то искомого слова в таблице нет. Выполняется запись текста искомого слова в поле наименования этой строки, и если для индикации занятости используется отдельный флажок, то он устанавливается в соответствующее состояние. Индекс строки возвращается в качестве результата.

Шаг 3. Сравнение. Сравниваются искомое слово и поле наименования. Если они не совпадают, то переход к шагу 4. Иначе слово найдено, индекс строки возвращается в качестве результата.

Шаг 4. Разрешение коллизии. Строка занята, но не тем словом, которое ищется в данный момент (эта ситуация получила специальное название ***коллизии***). Известно много способов разрешения коллизий, мы опишем простейший. Начиная со строки, следующей за текущей строкой (и считая таблицу закольцованной так, что после строки с номером N–1 следует строка с номером 0), организуется простой последовательный поиск. В результате будет найдена либо строка, содержащая искомое слово, и возвращен ее индекс, либо пустая строка, в которую, как на шаге 2, будет записано искомое слово и ее индекс будет возвращен в качестве результата. При этом следует принять меры против зацикливания, которое в принципе может возникнуть, если в таблице не осталось ни одной свободной строки.

Заметим, что при такой организации таблиц качество транслятора существенно зависит от качества функции рандомизации. Действительно, если предположить, что эта функция выдает одну и ту же константу (например, 0) в качестве результата преобразования текста каждого слова, то поиск всегда будет просто последовательным с соответствующими временными затратами. Если, наоборот, для каждого уникального слова индекс как результат преобразования тоже будет уникален, то не возникнет ни одной коллизии, т. е. время каждого поиска и каждого пополнения будет равно константе, а представить более быстрый алгоритм уже нельзя.

Поэтому большое значение придается выбору функции рандомизации, т. е. псевдослучайного преобразования текстового значения в целочисленное, верхний предел которого задан размером таблицы. Существует, по-видимому, бесконечное количество способов такого преобразования. Мы рассмотрим только один, хорошо зарекомендовавший себя на практике.

Текст искомого слова есть последовательность символов, каждый из которых представлен небольшим (одно- или двухбайтным) целым числом. Размер значащей части этой последовательности ограничен стандартом языка или произволом разработчика компилятора. Если длина искомого слова превышает это ограничение, то возьмем только значащую часть последовательности, а если количество значащих символов меньше ограничения, то дополним текст слова необходимым количеством символов пробела. Теперь, ориентируясь на разрядность используемого при трансляции компьютера, разобьем полученную последовательность байтов на ряд подпоследовательностей, каждую из которых будем рассматривать как целое число. Например, если количество значащих символов в идентификаторах составляет 32, а компьютер умеет перемножать 4-байтные целые, то из текста слова можно сформировать восемь 32-разрядных целых чисел. Далее последовательно перемножим все эти числа, оставляя от каждого очередного произведения любые (но желательно средние) 32 бита из 64-битного произведения. И, наконец, из середины битового представления полученного результата вырежем n разрядов (исходя из соотношения $N \leq 2^n$). Если полученное значение окажется большим или равным количеству строк в таблице (а это возможно только в том случае, если это количество не есть степень двойки), то выполним операцию взятия модуля по N. Результат этой операции и есть индекс, возвращаемый для поиска.

Очевидно, что при многократном применении этой процедуры к тексту одного и того же слова будет формироваться один и тот же индекс. Для разных слов эта процедура может выдавать разные индексы, но возможны и случаи, когда индексы окажутся одинаковыми. Объясняется это тем, что количество всех возможных идентификаторов (при 32-символьном ограничении размера) равно:

51 456 614 975 406 020 687 179 378 939 503 261 597 730 789 013 233 371 509

или приблизительно $5.146 \cdot 10^{55}$, в то время, как разумное количество строк в таблице может находиться в значительно меньших пределах (32 768 – 131 072). Тогда на каждое значение индекса приходится, соответственно, от 10^{48} до 10^{50} различных идентификаторов.

Описанная нами мультипликативная функция рандомизации в среднем отличает очень низкую вероятность возникновения коллизий при условии, что количество заполненных строк не превышает 0.7 от

общего количества строк в таблице. Поскольку занятые строки неким псевдослучайным образом рассеяны по таблице, переход к последовательному поиску при коллизии можно рассматривать как просмотр очень короткой таблицы.

Приведем временные оценки для рандомизированной организации таблиц. Время поиска одного отсутствующего в таблице слова без возникновения коллизии есть константа:

$$t_{\text{п}} = C.$$

Время поиска одного отсутствующего слова при возникновении коллизии можно оценить как

$$t_{\text{п}} = C + n_i\, t_{\text{ср}},$$

где n_i – количество занятых строк во фрагменте таблицы, начинающемся со строки, где возникла коллизия. Можно считать, что n_i значительно меньше N.

Время дополнения одного слова в этих случаях есть константа:

$$t_{\text{д}} = C.$$

Время поиска одного ранее записанного в таблицу слова без возникновения коллизии есть константа:

$$t_{\text{п}} = C.$$

Время поиска одного ранее записанного в таблицу слова при возникновении коллизии можно оценить как

$$t_{\text{п}} = C + (n_i\, t_{\text{ср}}) / 2,$$

где n_i – количество занятых строк во фрагменте таблицы, начинающемся со строки, где возникла коллизия. Можно считать, что n_i значительно меньше N.

Время дополнения одного слова в этих случаях

$$t_{\text{д}} = 0.$$

Оценка общего времени поиска:

$$T_{\text{п}} \sim= k_{\text{н}}\, C + k_{\text{к}}\, (\text{C} + nt_{\text{ср}}),$$

где $k_{\text{н}}$ – количество поисков слов, для которых не возникает коллизия; $k_{\text{к}}$ – количество поисков слов, для которых коллизия возникает; n – средняя длина фрагмента, состоящего из последовательности занятых строк.

Общее время пополнения пропорционально количеству различных идентификаторов и констант в тексте транслируемой программы. Его можно оценить так же, как для последовательной организации таблиц:

$$T_д \sim= N_{ид} + N_к.$$

Таким образом, если количество коллизий $k_к$ значительно меньше количества различных слов, заносимых в таблицу (мы уже обсуждали условия, способствующие минимизации числа коллизий), рандомизированная организация информационных таблиц имеет наилучшие характеристики по сравнению с другими методами поиска / пополнения.

1.7. ФОРМИРОВАНИЕ ЛЕКСЕМ

Формирование лексем не является сколько-нибудь сложной задачей и состоит в объединении номера группы слов, получаемого от лексического акцептора с индексом строки в соответствующей таблице, выдаваемым функцией поиска/пополнения таблиц. Однако существуют типичные для многих языков программирования особенности, затрудняющие это простое действие. Например, во многих языках служебные слова являются идентификаторами по правилам их образования (т. е. с точки зрения лексического акцептора), но с точки зрения синтаксического и семантического анализаторов образуют, очевидно, совершенно отдельную группу слов. Это порождает некоторые затруднения при реализации лексического анализатора в целом, разрешать которые можно следующими способами.

1. При обнаружении любого идентификатора вначале производится поиск в таблице служебных слов. Если результат отрицателен, то поиск переключается на таблицу идентификаторов, пополнение предопределенной таблицы служебных слов не выполняется.

2. Таблица идентификаторов формируется как частично предопределенная, т. е. при разработке компилятора в нее заносятся строки, соответствующие служебным словам. С помощью специального значения некоторого атрибута такие строки размечаются как содержащие данные о словах другой группы. При успешном результате поиска формирователь лексем (см. рис. 1.3) проверяет значение этого атрибута и, если он соответствует служебному слову, изменяет номер группы слов, сформированный лексическим акцептором.

3. Каждое служебное слово выделяется в отдельную группу (без соответствующей предопределенной таблицы, которая попросту не

нужна для группы из одного слова). Естественно, что поиск запускается только для групп, содержащих более одного слова.

Подобная техника может применяться не только для групп «идентификаторы / служебные слова», но при необходимости и для таких групп / подгрупп слов, как «знаки операций / знаки арифметических операций / знаки логических операций / …».

ЛИТЕРАТУРА

1. *Ахо А., Сети Р., Ульман Д.* Компиляторы: принципы, технологии и инструменты. – М.: Изд. дом «Вильямс», 2001.
2. *Гордеев А.В., Молчанов А.Ю.* Системное программное обеспечение: учебник для вузов. – СПб.: Питер, 2001.
3. *Гавриков М.М., Иванченко А.Н., Гринченков Д.В.* Теоретические основы разработки и реализации языков программирования. – М.: Кнорус, 2010.
4. *Малявко А.А.* Теория формальных языков: учебное пособие. – Новосибирск: Изд-во НГТУ, 2001. – Ч. 1.
5. *Малявко* А.А. Теория *формальных* языков: учеб. пособие. – Новосибирск: Изд-во НГТУ, 2002. – Ч. 2.
6. *Малявко* А.А. Теория формальных языков: учеб. пособие. – Новосибирск: Изд-во НГТУ, 2004. – Ч. 3.
7. *Карпов* Ю. Г. Теория и технология программирования. Основы построения трансляторов: учеб. пособие. – СПб.: БХВ-Петербург, 2005.
8. *Пратт Т., Зелковиц М.* Языки программирования: реализация и разработка. – СПб.: Питер, 2001.
9. *Малявко А.А.* Системное программное обеспечение ЭВМ. Трансляторы / Методические указания. – Новосибирск: Изд-во НГТУ, 2006.
10. *Рейуорд-Смит В.Дж.* Теория формальных языков. Вводный курс. – М.: Радио и связь, 1988.
11. *Хантер Р*. Проектирование и конструирование компиляторов. – М.: Финансы и статистика, 1984.
12. *Льюис Ф., Розенкранц Д., Стирнз Р*. Теоретические основы проектирования компиляторов. – М.: Мир, 1979.
13. *Грис Д*. Конструирование компиляторов для цифровых вычислительных машин. – М.: Мир, 1985.

ОГЛАВЛЕНИЕ

Малявко Александр Антонович

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ
ФОРМАЛЬНЫЕ ЯЗЫКИ И МЕТОДЫ ТРАНСЛЯЦИИ

Учебное пособие

Редактор *Н.В. Городник*
Выпускающий редактор *И.П. Брованова*
Корректор *Л.Н. Киншт*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *В.Ф. Ноздрева*

Подписано в печать 01.07.2010. Формат 60×84 1/16. Бумага офсетная. 100 экз.
Уч.-изд. л. 6,04. Печ. л. 6,5. Изд. № 96. Заказ № . Цена договорная

Отпечатано в типографии
Новосибирского государственного технического университета
630092, г. Новосибирск, пр. К. Маркса, 20

ДИФФРЕНЦИАЛЬНАЯ ИМПЕДАНСНАЯ СПЕКТРОСКОПИЯ ТЕХНИЧЕСКИХ МАСЕЛ

Учебное пособие

Редактор *Н.В. Городник*
Выпускающий редактор *И.П. Брованова*
Корректор *Л.Н. Киншт*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *В.Ф. Ноздрева*

103

Подписано в печать 05.07.2010. Формат 60×84 1/16. Бумага офсетная. Тираж 800 экз.
Уч.-изд. л. 15,25. Печ. л. 15,5. Изд. № 133. Заказ № . Цена договорная

Отпечатано в типографии
Новосибирского государственного технического университета
630092, г. Новосибирск, пр. К. Маркса, 20