ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
УРАЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ им. А. М. ГОРЬКОГО

А. П. Замятин, А. М. Шур

ЯЗЫКИ, ГРАММАТИКИ, РАСПОЗНАВАТЕЛИ

Рекомендовано УМС по математике и механике
УМО по классическому университетскому образованию РФ
в качестве учебного пособия для студентов
высших учебных заведений, обучающихся по группе
математических направлений и специальностей

Екатеринбург
Издательство Уральского университета
2007

УДК 519.68+519.713+519.766.2
З269

Р е ц е н з е н т ы:

кафедра сетевых информационных систем Российского государственного профессионально-педагогического университета (заведующий кафедрой кандидат технических наук, доцент В. В. В ь ю х и н);

М. Ю. Х а ч а й, доктор физико-математических наук (Институт математики и механики УрО РАН)

Замятин А. П., Шур А. М.

З269 Языки, грамматики, распознаватели : Учебное пособие . – Екатеринбург : Изд-во Урал. ун-та , 2007. – 248 с.

ISBN 5-7996-0273-0

В пособии систематически изложены ряд важных разделов теории формальных языков и приложения этой теории к построению компиляторов. Пособие рассчитано на математически ориентированных читателей.

Для студентов математических направлений и специальностей, изучающих дисциплины «Лингвистические основы информатики», «Теория автоматов», «Языки и автоматы», «Дискретная математика».

УДК 519.68+519.713+519.766.2

ISBN 5-7996-0273-0

Введение

О чем эта книга?

В книге обсуждаются вопросы, связанные с синтаксическим и семантическим анализом формализованных языков. Под формализованными языками понимаются искусственные языки, имеющие четкий синтаксис и более или менее ясную семантику. Синтаксис чаще всего базируется на явно заданных правилах построения фраз языка, семантика описывается менее формально. Основные представители формализованных языков в книге – языки программирования. Последний термин понимается широко: это и собственно языки программирования (т. е. языки систем программирования), и языки запросов к базам данных, и языки разметки, и языки текстовых процессоров. Для примеров чаще всего используются конструкции языка Паскаль – не потому, что авторы очень его любят, а потому, что это универсальный учебный язык, отлично подходящий как раз для примеров.

Книга написана по материалам лекций по курсу «Лингвистические основы информатики», читавшемуся авторами студентам специальностей «Математика. Компьютерные науки» и «Компьютерная безопасность» математико-механического факультета УрГУ. Книгу можно рекомендовать и студентам других специальностей с углубленным изучением компьютерных наук, при условии понимания ее основной особенности: *книга рассчитана на «математически ориентированных» читателей, и в этом ее главное отличие от большинства учебников по теории компиляции.* Авторами сделан значительный акцент на доказательность общих утверждений. Теорем, для которых вместо доказательства приводится ссылка на источник, где это доказательство можно прочитать, совсем мало, эти доказательства громоздки и не связаны впрямую с приводимыми в книге алгоритмами. Причин для такого акцента две. Во-первых,

читатели, которым адресована книга, не приучены принимать математические утверждения и алгоритмы на веру, а требуют четкого обоснования. Во-вторых, именно в процессе математического доказательства наиболее полно раскрываются обсуждаемые понятия. Мы надеемся, что, прочитав эту книгу, читатель сможет понять не только то, *как* работает тот или иной алгоритм, но и *почему* он работает (и, что еще важнее, почему он *не* работает в той или иной ситуации).

Что такое компилятор?

Краткий ответ на вопрос, вынесенный в заголовок, по-видимому, может дать любой из читателей этой книги. Компилятор – это компьютерная программа, производящая перевод текста, написанного на одном (исходном) компьютерном языке в «эквивалентный» текст на другом (целевом) компьютерном языке. Попробуем заглянуть чуть глубже. Исходный и целевой языки обычно имеют совершенно разную структуру (и даже природу), что исключает возможность перевода «слово в слово». Как же происходит перевод?

Для понимания этого проще всего посмотреть, что нужно переводчику, чтобы качественно перевести фразу с одного естественного языка на другой. Переводчик должен выявить структуру фразы, определить значение каждого слова в ней, выделить основную мысль, уловить все оттенки смысла, игру слов и т. д. Это *стадия анализа*, по итогам которой формируется «внутреннее представление» данной фразы в голове переводчика. Далее переводчик переходит ко второй стадии – *стадии реализации*, на которой он должен облечь имеющееся «внутреннее представление» в форму фразы того языка, на который выполняется перевод, максимально точно передав семантику (смысл) фразы. Формализация естественных языков – очень трудная задача, поэтому компьютерный перевод пока качественно несоизмерим с «человеческим», даже когда речь идет о технических, а не о художественных текстах. Искусственные языки, в частности компьютерные, намного проще естественных и к тому же имеют более четкое и строгое описание синтаксиса, т. е. правил построения фраз. Поэтому оказывается возможным строго формализовать как правила анализа, так и правила реализации и в полной мере воспользоваться вычислительной мощью компьютера.

Первый компилятор был написан Грейс Хоппер в 1952 году для языка программирования A-0. Компилятор, по структуре похожий

на современный, впервые был построен группой Джона Бэкуса в 1957 году (для компании IBM и языка FORTRAN). В 1960–1970-е годы произошел качественный скачок теории формальных языков, который привел к появлению новых, более мощных методов компиляции и позволил конструировать языки с гораздо более сложным синтаксисом.

Процесс обработки исходного текста компилятором состоит из тех же двух стадий, что и работа переводчика, – стадий анализа и реализации. В английском языке для них используются термины «front end» и «back end» соответственно. На стадии анализа производится проверка исходного текста («программы») на корректность, т. е. на соответствие синтаксическим и семантическим правилам языка, на котором эта программа написана, а также выяснение ее структуры и перевод ее в некоторое «внутреннее представление». На стадии реализации производится оптимизация внутреннего представления и генерация кода на целевом языке с учетом особенностей оборудования, операционной системы и т. д. Результатом реализации могут быть исполняемый код (программы или запроса к базе данных), программа на другом языке, эквивалентная исходной, значение формулы, web-страница или, к примеру, оригинал-макет книги, которую вы сейчас держите в руках.

Отметим, что стадия анализа присутствует всегда, а стадия реализации в некоторых простых случаях может отсутствовать: например, вычисление значения формулы в табличном редакторе может быть произведено параллельно с проверкой этой формулы на корректность записи. В других случаях, наоборот, роль заключительной стадии очень велика, поскольку эффективность работы исполняемого кода зависит от учета многих факторов, например, таких, как особенности процессора, объем кэш-памяти, тип файловой системы или язык, на котором написана база данных, запрос к которой нужно выполнить.

В основе и стадии анализа, и стадии реализации лежат глубокие математические результаты, но на этом сходство стадий заканчивается. Основное их отличие можно сформулировать очень коротко: на стадии анализа производится *точное* решение задач *распознавания*, а на стадии реализации – *приближенное* решение задач *оптимизации*. Точные методы решения оптимизационных задач оказываются неприменимыми, иначе работа компилятора не будет укладываться ни в какие пространственно-временные рамки, ведь ему, в частности, приходится решать несколько взаимосвязанных NP-полных задач.

В предлагаемой книге рассматривается только стадия анализа. В основе работы всех блоков компилятора, осуществляющих анализ, лежат фундаментальные результаты теории формальных языков. Понимание этих результатов и вытекающих из них алгоритмов необходимо для квалифицированного программиста, даже если он никогда в жизни не столкнется с задачей написания компилятора.

Языки, грамматики, распознаватели – немного истории

Основные математические объекты и связи между ними, рассматриваемые в этой книге, укладываются в простую схему, приведенную на рис. 0.1.

Рис.0.1. *Математические объекты и связи в этой книге*

Едва ли возможно определить, в работах каких математиков впервые изучались слова – конечные последовательности элементов фиксированного множества и формальные языки, т. е. множества слов. Пожалуй, первым ученым, систематически исследовавшим данную тематику, был норвежский математик Аксель Туэ (1863–1922). Бурное развитие теории формальных языков началось в 1950-е годы, во многом благодаря начавшейся компьютерной эре. В этой книге мы осветим лишь малую часть данной теории, в чем несложно убедиться, например, обратившись к оглавлению трехтомного справочника [Hand].

«Грамматическая» модель описания естественных языков была впервые предложена американским лингвистом и психологом Ноамом Хомским (р. 1928) в середине 1950-х годов. Так, Хомский ввел понятия контекстно-свободной грамматики и контекстно-

свободного языка, которые в дальнейшем сыграли ключевую роль в описании синтаксиса уже искусственных языков. В 1963 году в соавторстве с французским математиком Марселем Шютценберже (1920–1996) Хомский описал иерархию классов грамматик и соответствующих им классов формальных языков.

Подобные модели в математике возникали и раньше. Еще в 1912 году Туэ предложил «переписывающие системы» для вывода одних строк символов из других. В 1948 году русский математик А. А. Марков-младший (1903–1979) ввел понятие «нормального алгорифма», основой которого является переписывающая система, как универсальную вычислительную модель, эквивалентную машине Тьюринга.

Исторически первую модель *распознавателя* формальных языков предложил в 1936 году английский математик Алан Тьюринг (1912–1954). Машина Тьюринга могла не только распознавать строки символов, но и преобразовывать одни строки в другие. На ее основе Тьюринг спроектировал один из первых в мире компьютеров. Кстати, термин «компьютер» в его современном значении впервые употребил именно Тьюринг.

Что касается распознавателей, изучаемых в данной книге, то понятие конечного автомата ввел американский логик и математик Стефен Клини (1909–1994). Понятие МП-автомата в явном виде впервые сформулировал Хомский в 1962 году, хотя устройства с лентой памяти, доступ к которой организован по принципу стека, были независимо описаны разными авторами еще в середине 1950-х годов.

Математические результаты, приведенные в этой книге, получены в 1950–1970-х годах. С тех пор они переместились с переднего края науки в лекционные аудитории и университетские учебники. При этом ресурсы оптимизации доказательств и способов подачи материала далеко не исчерпаны. Авторы надеются, что им удалось внести свой скромный вклад в эту работу.

О других учебниках

Математического учебника, компактно и адекватно отображающего весь имеющийся в данной книге материал, просто нет, иначе у авторов не было бы необходимости писать данную книгу. Достаточно сложна для чтения и немного путает читателя устаревшими обозначениями безусловно ценная книга [Ги]. Ярко написанная книга [Сал] представляет лишь часть необходимого материала, к тому же она

давно стала библиографической редкостью. Трехтомный [Hand] – прекрасный справочник, но в нем отсутствует большинство доказательств. Хорошо написан, но пока практически недоступен учебник по теории автоматов [Sak]. Авторы рекомендуют ознакомиться с отличным курсом лекций по автоматам и языкам [Kar] (на английском языке). В частности, в нем можно найти доказательства практически всех утверждений первых трех глав книги. Естественно, как и большинство курсов лекций, он доступен только в электронном виде.

С учебниками по теории компиляторов дело обстоит существенно лучше. Среди таких учебников, доступных на русском языке, в первую очередь нужно отметить книгу [АСУ]. Вышедшая в свет в 1986 году (в русском переводе – в 2001 году), она по-прежнему остается базовым учебником по данному предмету на факультетах компьютерных наук большинства зарубежных университетов. В конце 2006 года вышло второе, расширенное и переработанное, издание. Как и все современные учебники на эту тему, обсуждаемая книга ориентирована на алгоритмы и примеры и практически не содержит доказательств (к чести авторов, надо отметить, что нередко они приводят поясняющие рассуждения, которые вдумчивый читатель вполне может переработать в доказательство).

Более раннее детище Ахо и Ульмана – двухтомник [АУ1], [АУ2] – занимает промежуточное положение между учебником по теории формальных языков и учебником по компиляторам. В частности, из него можно почерпнуть доказательства основных результатов, относящихся к методам синтаксического анализа.

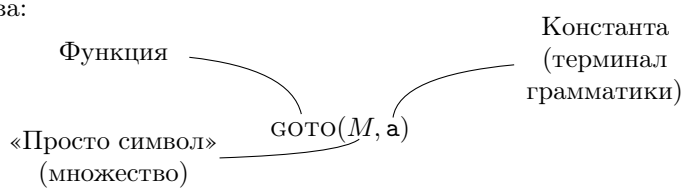
Два недавно вышедших учебника [Ка] и [ОС] ориентированы на студентов технических вузов. Они содержат изложение основных методов анализа с содержательными примерами, но доказательства в них полностью отсутствуют.

Немного особняком стоит учебник [СТ], не имеющий русского перевода. В нем очень подробно и хорошо описан «back end» современного компилятора, чего нет, пожалуй, ни в одном другом учебнике.

И наконец, еще одна рекомендация читателям: поищите на сайте какого-нибудь известного западного университета материалы к читаемым курсам. Немного усилий – и вы станете обладателем комплекта иллюстрирующего материала (например, PowerPoint-презентаций) курса «Compiler design». Изучение этого комплекта принесет вам несомненную пользу.

Структура книги и особенности изложения

Учебник состоит из трех частей, разбитых на 11 глав, и содержит в общей сложности 57 параграфов. Нумерация математических утверждений, алгоритмов, примеров, таблиц, рисунков и формул внутри каждой части своя, перед номером ставится только номер главы. Все основные алгоритмы записаны при помощи псевдокода. Чтобы улучшить читаемость формул (в которых обычно присутствуют объекты разной природы), авторы приняли решение использовать внутри формул наряду с обычным математическим шрифтом еще два:



Рекомендуется «сквозное» чтение книги. Однако возможно чтение отдельных частей и даже отдельных глав: ссылки на материал других глав везде прописаны явно, что позволяет без труда отыскать в книге все сведения, необходимые для понимания материала конкретной главы. Кроме того, книга снабжена указателем обозначений и подробным предметным указателем.

В первой части книги закладывается математический фундамент для построения лексического, синтаксического и семантического анализаторов, реализующих стадию анализа в компиляторе. Краткое содержание этой части прекрасно иллюстрируется при помощи рис. 0.1.

Почти вся вторая часть (кроме небольшой главы 1 о лексическом анализе) посвящена важнейшему этапу компиляции – синтаксическому анализу. Подробно, с большим количеством примеров разобраны и нисходящие, и восходящие методы анализа (LL-анализ, метод рекурсивного спуска, анализ на основе отношений простого и операторного предшествования, LR-анализ).

В третьей части рассмотрены вопросы семантического анализа: построение и анализ атрибутных грамматик, возможность совмещения семантического анализа с синтаксическим анализом, построение внутреннего («промежуточного») представления исходной программы. Отдельная глава посвящена семантическим проверкам, выполняемым компилятором, на примере проверки типов.

Для чтения книги необходимо знание «джентльменского набора»

из стандартного курса дискретной математики – множества, бинарные отношения, основы математической логики и теории графов. Предполагается также, что читатель знает хотя бы один алгоритмический язык программирования и знаком с базовыми алгоритмами, такими как алгоритмы поиска.

Авторы благодарят Ю. В. Гамзову и В. Ю. Попова за многочисленные ценные замечания, высказанные при обсуждении материала книги. Огромное спасибо Юхани Карюмяки за любезно предоставленную возможность пользоваться записями его лекций по курсу «Языки и автоматы» при подготовке книги. Отдельная благодарность всем студентам, которые своими вопросами на лекциях и ответами на экзаменах помогли авторам понять, что и как следует писать в этой книге.

Часть 1.

Языки и способы их задания

Основная цель данной части – определиться с пониманием термина «язык» и способами задания языков. Языки будут задаваться, как правило, одним из двух способов: распознавателями или порождающими грамматиками. Иногда в качестве средства задания языка могут использоваться операции над языками, в частности языки будут задаваться регулярными выражениями. В данной части содержится также некоторый материал об эквивалентных преобразованиях способов задания, т. е. преобразованиях, сохраняющих задаваемый язык. К таким преобразованиям относятся: преобразования, ведущие к приведенному конечному автомату (гл. 2), устранение недостижимых и непроизводящих символов при переходе к приведенной грамматике, устранение аннулирующих правил вывода при переходе к ε-свободной грамматике (гл. 4) и ряд других преобразований.

Глава 1. Языки и операции над ними

Данная глава является вводной ко всей книге. В ней приводится одна из распространенных формализаций (интуитивного) понятия «язык». Вводятся примеры языков, отражающих некоторые конструкции языков программирования. Эти примеры будут использоваться на протяжении всей книги. Глава содержит также определения операций над языками. Вводятся как традиционные операции над языками: булевы операции, умножение, итерация, подстановка, так и ряд более специфических операций.

§ 1. Понятие языка

Попытаемся ответить на вопрос, что такое язык, дать точное (математическое) определение этого понятия. Если взять толковый словарь, то там можно найти несколько значений (словарных статей) слова «язык». Среди прочих будет и значение этого слова как средства общения, средства передачи информации. Именно в этом смысле мы будем рассматривать термин «язык».

Традиционная лингвистика предлагает рассматривать понятие языка по крайней мере с двух точек зрения, в двух планах: в плане содержания и в плане выражения. План содержания – это то, *что* можно выразить средствами языка, а план выражения – это то, *как* это можно сделать. Говоря математическим языком, план содержания есть множество смыслов (значений), которые можно выразить в языке, а план выражения есть множество выражений (множество фраз) языка, которыми выражается тот или иной смысл. Будем считать, что есть еще две функции, одна из которых смыслу ставит в соответствие одну или несколько фраз, имеющих этот смысл. Другая функция фразе языка ставит в соответствие значение (одно или несколько) этой фразы. Все понятия, относящиеся к плану содержания, принято объединять термином *семантика*, понятия, относящиеся к плану выражения, – термином *синтаксис*.

Рассмотрим примеры.

В качестве первого примера возьмем *язык логики высказываний*. Выражением (фразой) этого языка будет формула логики высказываний, например, $F = X_1 \& X_2 \& X_3 \to X_4$. Смысл (значение) этой фразы определяется интерпретацией – функцией, которая атомарным формулам, входящим в данную формулу (в данном случае выражениям X_1, X_2, X_3 и X_4), сопоставляет некоторые высказывания. Применив к этим высказываниям имеющиеся в формуле логические операции, получим высказывание, являющееся значением всей формулы. Возьмем интерпретацию, которая атомарным формулам X_1, X_2, X_3 и X_4 ставит в соответствие высказывания

X_1: «функция $f(x)$ непрерывна на отрезке $[a, b]$»,
X_2: «функция $f(x)$ дифференцируема на интервале (a, b)»,
X_3: «функция $f(x)$ в точках a и b принимает равные значения»,
X_4: «существует точка c из интервала (a, b) такая, что $f'(c) = 0$».

Тогда значением формулы F будет высказывание, известное в математическом анализе как теорема Ролля.

Вторым примером будет язык программирования *Паскаль*. Вы-

ражением (фразой) языка Паскаль является синтаксически правильная программа, написанная на этом языке. Смысл (значение) выражений языка в этом случае определяется компилятором Паскаля и представляет собой вычислительный процесс, происходящий в компьютере при выполнении откомпилированной программы.

В качестве третьего примера рассмотрим *русский язык*. Как и многие естественные языки, русский язык можно рассматривать на нескольких уровнях. На одном уровне выражением языка можно считать слово, на другом – (синтаксически правильное) предложение русского языка. Отметим, что возможны и другие уровни рассмотрения русского языка – как более «мелкие», так и более «крупные». Множество смыслов русского языка, как любого развитого естественного языка, в полном объеме определить невозможно, так как для человека, для которого данный язык является родным, множество смыслов языка – это «все», это весь смысловой универсум. Функции, которые по выражению языка определяют его значение или, наоборот, некоторый смысл облекают в форму предложения, формируются в процессе изучения языка и фактически являются для человека «встроенными».

Мы рассмотрели примеры из трех больших классов языков: это классы логико-математических языков, языков программирования и естественных языков. Эти языки являются предметом изучения многих научных направлений в математике, информатике и лингвистике. Можно привести и «менее значительные» примеры. В качестве четвертого примера возьмем *язык химических формул*. Выражениями этого языка являются химические формулы, например H_2SO_4. Множеством значений (смыслов) языка химических формул является множество химических веществ. Так, значением приведенного выше выражения является, как известно, серная кислота.

Приведенные примеры показывают, что задача формулирования точного определения понятия языка является очень сложной и вообще едва ли выполнимой. Фактически мы ограничимся более или менее приемлемым вариантом. Попробуем понять, каковы же общие моменты в приведенных выше примерах. Такие моменты есть, и они касаются способа построения выражений. Во всех случаях фиксируется некоторый конечный набор исходных объектов. Выражением языка является некоторая (не любая) последовательность этих объектов. Это наблюдение приводит к следующим определениям.

Определение. *Алфавитом* называется непустое конечное множе-

ство. *Цепочкой* (или *словом*, или *строкой*) над алфавитом Σ называется конечная последовательность элементов Σ. Такие последовательности принято записывать без запятой.

Например, если $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, то `abba`, `aab`, `a` – цепочки над Σ. *Длина* цепочки w – это количество символов в ней, обозначаемое через $|w|$. *Пустую цепочку* – цепочку нулевой длины – обозначим через ε. Множество всех цепочек над Σ обозначается через Σ^*, а множество всех непустых цепочек над Σ – через Σ^+.

Мы готовы ввести основное определение параграфа.

Определение. *Языком* над Σ называется подмножество в Σ^*.

Таким образом, язык – это множество цепочек. Данное определение отражает только план выражения языка и в общем соответствует содержанию книги: в основном здесь изучаются синтаксические понятия, относящиеся к языку. Семантика языка будет рассматриваться в ограниченном виде.

Приведем примеры языков, которые отражают некоторые конструкции языков программирования. Эти примеры будут использоваться во всей книге.

Начнем с *языка цепочек сбалансированных скобок* (или короче – *скобочного языка*) LB. Алфавит языка состоит из двух символов [и]. Разумеется, понятно, какие цепочки принадлежат языку LB, а какие не принадлежат. Например, цепочки [], [][], [[]] принадлежат LB, а цепочки []][, []] не принадлежат этому языку. Тем не менее нам понадобится строгое определение сбалансированной цепочки скобок. Наиболее удобен рекурсивный вариант определения.

Определение. 1. Пустая цепочка является сбалансированной.
2. Если цепочки скобок u и v сбалансированы, то цепочки $[u]$ и uv также сбалансированы.
3. Других сбалансированных цепочек скобок нет.

В качестве второго примера рассмотрим *язык списков*, который будем обозначать через LL. Алфавитом этого языка будет множество $\{\,\mathtt{a},\ ;\ ,\ [\ ,\ ]\,\}$. Определимся с тем, что мы будем понимать под списком.

Определение. 1. Цепочка `a` является списком.
2. Если цепочки u и v – списки, то цепочки $[u]$ и $u;v$ также являются списками.
3. Других списков нет.

Из определения видно, что цепочки [a]; a , [a; a] , [[a]] являются списками, т. е. принадлежат языку LL, а цепочки [ ] , [a] не являются.

Третьим в этом списке примеров рассмотрим *язык описаний типов* LD. Это язык над алфавитом { i, : , ; , real, int }. Цепочки этого языка будут иметь следующий вид: вначале идет (непустой) список букв i, разделенных точкой с запятой, затем двоеточие и заканчивает цепочку один из символов real или int. Примеры описаний: i; i; i : real, i : int. Разумеется, язык LD лишь приблизительно отражает грамматические конструкции, используемые при описаниях типов в реальных языках программирования. Тем не менее он полезен для учебных примеров.

В качестве четвертого примера возьмем *язык двоичных чисел* LN. Алфавит состоит из 0, 1 и десятичной точки. Цепочки языка LN имеют вид u, $u.v$ или $.v$, где u и v – произвольные последовательности нулей и единиц. Заметим, что в языке LN есть цепочки, целая часть которых начинается нулем, а также цепочки, дробная часть которых нулем заканчивается. Если дополнительно потребовать, чтобы целая часть начиналась единицей, а дробная единицей заканчивалась, то получится язык, который мы будем обозначать через LN'.

Список примеров языков завершим рассмотрением *языка арифметических выражений* LA. Этот язык будет содержать арифметические выражения, построенные из символа x, который символизирует переменную или константу, знаков операций $+$ и $*$ с обычным приоритетом и скобок. Нам понадобится строгое

Определение. 1. Цепочка x – арифметическое выражение.
2. Если цепочки u и v – арифметические выражения, то цепочки $u + v$, $u * v$, (u) также являются арифметическими выражениями.
3. Других арифметических выражений нет.

Приведем примеры цепочек из языка LA: x + x * x, (x + x) * x, (x).

В заключение параграфа введем ряд полезных для дальнейшего понятий, относящихся к цепочкам.

Определение. Пусть w – цепочка над некоторым алфавитом. Если $w = uv$ для некоторых (возможно, пустых) цепочек u и v, то u называется *префиксом* w, а v – *суффиксом* w. Префикс (суффикс) цепочки w называется *собственным*, если он отличен от w и ε.

Например, если w = abc, то цепочки abc, ab, a, ε – префиксы цепочки w, а цепочки abc, bc, c, ε – ее суффиксы. Подчеркнем, что w и ε являются как префиксами, так и суффиксами цепочки w.

§ 2. Операции над языками

В этом параграфе мы введем основные операции над языками, необходимые в дальнейшем.

Пусть K, L и M – языки над алфавитом Σ. Отметим прежде всего, что будем использовать обычные теоретико-множественные операции: пересечение, объединение, дополнение и разность. Напомним, что

$$\begin{aligned} K \cap L &= \{w \in \Sigma^* |\ w \in K \text{ и } w \in L\}, \\ K \cup L &= \{w \in \Sigma^* |\ w \in K \text{ или } w \in L\}, \\ \overline{K} &= \{w \in \Sigma^* |\ w \notin K\}, \\ K \backslash L &= \{w \in \Sigma^* |\ w \in K \text{ и } w \notin L\}. \end{aligned}$$

Определение. *Произведением* языков K и L называется язык

$$KL = \{uv \in \Sigma^* |\ u \in K \text{ и } v \in L\}\ .$$

Например, для языков $K = \{\mathtt{ab}, \mathtt{abc}\}$ и $L = \{\mathtt{cba}, \mathtt{ba}\}$ получим $KL = \{\mathtt{abcba}, \mathtt{abba}, \mathtt{abccba}\}$. Пример, в частности, показывает, что операция умножения некоммутативна. С другой стороны, легко проверить, что эта операция ассоциативна, а язык $\{\varepsilon\}$ является нейтральным элементом. Таким образом, можно говорить о *степени* языка и положить дополнительно $L^0 = \{\varepsilon\}$.

Определение. *Итерацией* языка L называется язык $L^* = \bigcup\limits_{i=0}^{\infty} L^i$.

Например, если $L = \{\mathtt{a}, \mathtt{ab}\}$, то $L^* = \{\varepsilon, \mathtt{a}, \mathtt{ab}, \mathtt{a}^2, \mathtt{a}^2\mathtt{b}, \mathtt{aba}, \mathtt{abab}, \mathtt{a}^3, \ldots\}$, но $\mathtt{ba} \notin L^*$.

Свойства теоретико-множественных операций для языков совпадают со свойствами этих операций для произвольных множеств, поэтому здесь не приводятся. Отметим ряд свойств, в которых «участвуют» произведение и итерация. Условимся, что приоритет умножения выше приоритета всех теоретико-множественных операций, приоритет итерации выше, чем приоритет умножения.

Пусть K, L и M – языки над одним и тем же алфавитом. Тогда справедливы следующие утверждения:

1. $K(L \cup M) = KL \cup KM$,
2. $K(L \cap M) \subseteq KL \cap KM$,
3. $(L \cup M)^* \supseteq L^* \cup M^*$,
4. $(L \cap M)^* \subseteq L^* \cap M^*$.

Докажем, например, второе утверждение. Пусть $x \in K(L \cap M)$. Тогда $x = uv$, где $u \in K$, $v \in L \cap M$. Имеем $v \in L$ и $v \in M$. Отсюда следует, что $uv \in KL$ и $uv \in KM$. Следовательно, $x \in KL \cap KM$. Мы доказали, что $K(L \cap M) \subseteq KL \cap KM$. Обратное включение неверно. Возьмем, например, $K = \{\varepsilon, \mathtt{b}\}$, $L = \{\mathtt{ab}\}$, $M = \{\mathtt{bab}\}$. Тогда $K(L \cap M) = \emptyset$, $KL \cap KM = \{\mathtt{bab}\}$.

Введем еще одну операцию над языками, которую будем называть *подстановкой*. Пусть $\Sigma = \{\mathtt{a}_1, \ldots, \mathtt{a}_n\}$ и Δ – два алфавита, τ – отображение, которое каждому элементу $\mathtt{b} \in \Sigma$ ставит в соответствие язык над Δ, т. е. $\tau(\mathtt{b}) \subseteq \Delta^*$. Рассмотрим два расширения отображения τ. Сначала расширим это отображение на Σ^*, положив $\tau(\mathtt{b}_1\mathtt{b}_2 \ldots \mathtt{b}_k) = \tau(\mathtt{b}_1)\tau(\mathtt{b}_2) \ldots \tau(\mathtt{b}_k)$ и $\tau(\varepsilon) = \{\varepsilon\}$. Далее расширим полученное отображение на множество языков над Σ. Если L – язык над Σ, то определим $\tau(L) = \bigcup\limits_{w \in L} \tau(w)$. Язык $\tau(L)$ будет языком над Δ, он является *результатом действия подстановки* τ на язык L.

Убедимся в том, что определенные ранее операции объединения, произведения и итерации являются частными случаями подстановки. Пусть $\Sigma = \{\mathtt{a}_1, \mathtt{a}_2\}$, L_1 и L_2 – языки над Δ, $\tau(\mathtt{a}_1) = L_1$, $\tau(\mathtt{a}_2) = L_2$. Тогда легко проверить, что $\tau(\{\mathtt{a}_1\mathtt{a}_2\}) = L_1L_2$, $\tau(\{\mathtt{a}_1, \mathtt{a}_2\}) = L_1 \cup L_2$ и $\tau(\{\mathtt{a}_1\}^*) = L_1^*$.

Операции над языками будем использовать для задания языков. Пусть $F(x_1, x_2, \ldots, x_n)$ – выражение, построенное из переменных x_1, x_2, …, x_n, знаков операций и скобок, и L_1, L_2, …, L_n – некоторые языки над Σ. Тогда говорят, что язык $F(L_1, L_2, \ldots, L_n)$ задается выражением $F(x_1, x_2, \ldots, x_n)$. Например, если $F(x_1, x_2) = x_1(x_2 \cup x_3)^*$, $L_1 = \{\mathtt{A}, \mathtt{B}, \ldots, \mathtt{Z}, \mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$, $L_2 = \{0, 1, \ldots, 9\}$, то язык $F(L_1, L_2)$ состоит из цепочек, составленных из букв и цифр и начинающихся с буквы, т. е. $F(L_1, L_2)$ – множество имен языка программирования типа Паскаль. Следовательно, множество имен задается выражением $x_1(x_2 \cup x_3)^*$.

Кроме приведенных выше «традиционных» операций над языками, в задачах будут использоваться еще и операции, приведенные ниже. Пусть L – язык над Σ, $\mathtt{a}$ – элемент из Σ. Тогда

$$\begin{aligned} L\backslash\mathtt{a} &= \{w \in \Sigma^* \mid w\mathtt{a} \in L\}, \\ \mathtt{a}\backslash L &= \{w \in \Sigma^* \mid \mathtt{a}w \in L\}, \\ init(L) &= \{w \in \Sigma^* \mid \exists x \in \Sigma^* : wx \in L\}, \\ min(L) &= \{w \in L \mid \forall x, y \in \Sigma^+ : w = xy \Rightarrow x \notin L\}. \end{aligned}$$

Первые две операции называются соответственно *правым* и *ле-*

вым делением на $\mathtt{a}$. Префиксное замыкание $init$ строит множество всех префиксов цепочек из L. Наконец, $min(L)$ – множество цепочек из L, никакой собственный префикс которых не принадлежит L. Приведем пример. Пусть $L = \{\mathtt{aba}, \mathtt{aaba}, \mathtt{abab}, \mathtt{aa}, \mathtt{bbb}\}$. Тогда $L\backslash\mathtt{a} = \{\mathtt{ab}, \mathtt{aab}, \mathtt{a}\}$, $\mathtt{a}\backslash L = \{\mathtt{ba}, \mathtt{aba}, \mathtt{bab}, \mathtt{a}\}$, $min(L) = \{\mathtt{aba}, \mathtt{aa}, \mathtt{bbb}\}$, $init(L) = \{\varepsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{bb}, \mathtt{aab}, \mathtt{aba}, \mathtt{bbb}, \mathtt{aaba}, \mathtt{abab}\}$.

Кроме операций над языками мы будем рассматривать отображения специального вида.

Определение. Отображение $\phi : \Sigma^* \to \Delta^*$ называется *гомоморфизмом*, если $\phi(uv) = \phi(u)\phi(v)$ для любых $u, v \in \Sigma^*$. Язык $\phi(L) = \{\phi(w) \mid w \in L\}$ называется *гомоморфным образом* языка L (при гомоморфизме ϕ).

Например, если $L = \{\mathtt{aba}, \mathtt{aa}, \mathtt{bbb}\}$, $\phi(\mathtt{a}) = \mathtt{ac}$, $\phi(\mathtt{b}) = \mathtt{ca}$, то $\phi(L) = \{\mathtt{accaac}, \mathtt{acac}, \mathtt{cacaca}\}$.

Очевидно, что для задания гомоморфизма достаточно указать образы всех букв из Σ. Отметим, что гомоморфизм можно рассматривать как частный случай подстановки, при котором образ любой буквы – это язык из одного слова.

Задачи

1. Пусть K, L и M – языки над алфавитом Σ. Доказать следующие утверждения:

а) $K(L \cup M) = KL \cup KM$, б) $(K \cup L)^* \supseteq K^* \cup L^*$,
в) $(K \cap L)^* \subseteq K^* \cap L^*$, г) $(K \cup L)^* = (K^* \cup L^*)^*$.

Можно ли в пунктах «б» и «в» включение заменить на равенство?
2. Пусть $A = \{\mathtt{a}\}$, $B = \{\mathtt{b}\}$, $C = \{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\}$. Выразить следующие языки через A, B и C с помощью операций объединения, произведения, итерации и подстановки:

а) $L_1 = \{\mathtt{a}^i\mathtt{b} \mid i \in \mathbb{N}\}$; б) $L_2 = \{\mathtt{a}^i\mathtt{b}^j \mid i, j \in \mathbb{N} \text{ и } i \le j\}$;
в) $L_3 = \{\mathtt{a}^i\mathtt{b}^j \mid i, j \in \mathbb{N} \text{ и } i < j\}$; г) $L_4 = \{\mathtt{a}^i\mathtt{b}^j \mid i, j \in \mathbb{N} \text{ и } i \ne j\}$;
д) $L_5 = \{\mathtt{a}^{2i}\mathtt{b} \mid i \in \mathbb{N}\}$; е) $L_6 = \{\mathtt{a}^i\mathtt{b}^{2i} \mid i \in \mathbb{N}\}$;
ж) $L_7 = \{\mathtt{a}^{2i}\mathtt{b}^{2i} \mid i \in \mathbb{N}\}$; з) $L_8 = \{\mathtt{a}^i\mathtt{b}^{2j} \mid i, j \in \mathbb{N} \text{ и } i \le j\}$.

3. Проверить следующие равенства:

а) $(L\backslash\mathtt{a})\{\mathtt{a}\} = L$; б) $(L\{\mathtt{a}\})\backslash\mathtt{a} = L$;
в) $\mathtt{a}\backslash(L\backslash\mathtt{a}) = (\mathtt{a}\backslash L)\backslash\mathtt{a}$; г) $min(min(L)) = min(L)$;

д) $\{\mathtt{a}\}min(L) = min(\{\mathtt{a}\}L)$;
е) $\{\mathtt{a}\}init(L) = init(\{\mathtt{a}\}L)\backslash\{\varepsilon\}$;
ж) $init(L \cup M) = init(L) \cup init(M)$;
з) $init(L \cap M) = init(L) \cap init(M)$;
и) $min(L \cup M) = min(L) \cup min(M)$;
к) $min(L \cap M) = min(L) \cap min(M)$.

Глава 2. Распознаватели

Распознавателем языка L над алфавитом Σ называется алгоритм (или физическое устройство), который по произвольной цепочке $w \in \Sigma^*$ определяет, принадлежит ли w языку L или не принадлежит. Иными словами, распознаватель вычисляет предикат «$w \in L$». В теории алгоритмов говорят, что распознаватель «решает проблему вхождения в L», а в теории формальных языков – что он «распознает L». Последним термином мы и будем пользоваться. Отметим, что распознаватель в процессе работы исследует определенные структурные свойства цепочки w и может быть использован для получения информации об этих свойствах.

Функционально распознаватели могут быть организованы поразному. Мы рассматриваем только два принципиально различных типа распознавателей, наиболее важных с точки зрения обработки формализованных языков: конечные автоматы и автоматы с магазинной памятью. Ввиду направленности этой книги на формализованные языки и, более узко, на языки программирования мы не коснемся здесь *универсальных* распознавателей, таких как машина Тьюринга.

Каждому типу распознавателей соответствует класс языков, распознаваемых устройствами этого типа. В данной главе подробно рассматривается класс языков, распознаваемых конечными автоматами. Класс языков, распознаваемых автоматами с магазинной памятью, изучается в гл. 3, хотя сами автоматы вводятся в текущей главе.

§ 3. Конечные автоматы

Определение. *Детерминированным конечным автоматом* (сокращенно ДКА) называется пятерка $\mathcal{A} = (Q, \Sigma, \delta, \mathtt{q}_0, F)$, где Q – непустое конечное множество *состояний*, Σ – конечный алфавит, $\mathtt{q}_0 \in Q$ –

начальное состояние, $F \subseteq Q$ – множество *заключительных* (или *допускающих*) состояний, $\delta : Q \times \Sigma \to Q$ – (всюду определенная) *функция переходов*.

Слово «автомат» в этом и следующем параграфе используется как синоним ДКА. В § 5 будут введены два типа *недетерминированных* конечных автоматов, обобщающих ДКА.

На автомат можно смотреть как на физическое устройство, состоящее из устройства управления и входной ленты (рис. 2.1). Входная лента разделена на ячейки и в ячейках ленты записаны символы из Σ (по одному в ячейке). В каждый момент времени устройство управления находится в некотором состоянии и «видит» одну ячейку ленты. Устройство считывает символ из просматриваемой ячейки, определяет свое новое состояние (по текущему состоянию и прочтенному символу), переходит в это состояние и сдвигается на одну ячейку ленты вправо.

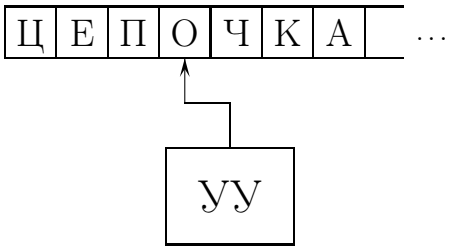


Рис. 2.1. *Схематическое устройство ДКА*

ДКА, как и любой распознаватель, работает дискретно, *тактами*. Такт работы ДКА состоит в переходе из текущего состояния $\mathtt{q}$ при текущем входном символе $\mathtt{a}$ в состояние $\mathtt{q}' = \delta(\mathtt{q}, \mathtt{a})$. Далее под «переходом» в автомате мы всегда понимаем тройку вида $(\mathtt{q}, \mathtt{a}, \mathtt{q}')$. Для записи последовательности тактов работы автомата удобно доопределить функцию переходов на всем множестве $Q \times \Sigma^*$ естественным образом:

$$\delta(\mathtt{q}, \varepsilon) = \mathtt{q}, \qquad \delta(\mathtt{q}, u\mathtt{a}) = \delta(\delta(\mathtt{q}, u), \mathtt{a}).$$

Таким образом, $\delta(\mathtt{q}, w)$ – это состояние, в которое придет ДКА, если, находясь в состоянии $\mathtt{q}$, обработает в результате последовательности тактов цепочку w. Автомат начинает работу, обозревая самую

левую ячейку ленты и находясь в состоянии $\mathtt{q}_0$. Поскольку δ – всюду определенная функция, то автомат «просмотрит» всю цепочку, записанную на входной ленте.

Определение. ДКА $\mathcal{A} = (Q, \Sigma, \delta, \mathtt{q}_0, F)$ *распознает* (или *допускает*) цепочку $w \in \Sigma^*$, если $\delta(\mathtt{q}, w) \in F$. Языком, *распознаваемым* $\mathcal{A}$ (обозначается через $L(\mathcal{A})$), называется множество всех цепочек, распознаваемых этим автоматом.

Будем использовать два способа задания конечного автомата: расширенную таблицу переходов и диаграмму переходов. Расширенная таблица переходов представляет собой таблицу значений функции переходов δ, первая строка которой соответствует начальному состоянию, а заключительные состояния помечены единицами в дополнительном столбце (табл. 2.1). *Диаграммой переходов* называется ориентированный граф, вершины которого – состояния автомата, а дуги помечены элементами алфавита. Данный граф содержит дугу из $\mathtt{q}$ в $\mathtt{q}'$, помеченную символом $\mathtt{x}$, если и только если $\delta(\mathtt{q}, \mathtt{x}) = \mathtt{q}'$. Начальное состояние будет обозначаться «двойным кружком», а заключительные состояния – «жирным кружком» (рис. 2.2). Поясним оба способа на примере.

Пример 2.1. Пусть $\mathcal{A}_1$=$(Q, \Sigma, \delta, \mathtt{q}_0, F)$, Q=$\{\mathtt{q}_0, \mathtt{q}_1, \mathtt{q}_2, \mathtt{q}_3\}$, Σ=$\{\mathtt{a}, \mathtt{b}\}$, $F = \{\mathtt{q}_2\}$, а δ задана при помощи табл. 2.1, *а*. Значение $\delta(\mathtt{q}, \mathtt{x})$ – элемент этой таблицы, находящийся в клетке с координатами $[\mathtt{q}, \mathtt{x}]$. Диаграмма переходов автомата $\mathcal{A}_1$ приведена на рис. 2.2.

Таблица 2.1. *Расширенные таблицы переходов автоматов $\mathcal{A}_1$ (а), $\mathcal{A}_2$ (б) и $\mathcal{A}_3$ (в)*

	a	b	F
$\mathtt{q}_0$	$\mathtt{q}_0$	$\mathtt{q}_1$	0
$\mathtt{q}_1$	$\mathtt{q}_2$	$\mathtt{q}_3$	0
$\mathtt{q}_2$	$\mathtt{q}_2$	$\mathtt{q}_3$	1
$\mathtt{q}_3$	$\mathtt{q}_3$	$\mathtt{q}_3$	0

а

	a	b	F
$\mathtt{q}_0$	$\mathtt{q}_0$	$\mathtt{q}_1$	0
$\mathtt{q}_1$	$\mathtt{q}_2$	$\mathtt{q}_3$	0
$\mathtt{q}_2$	$\mathtt{q}_2$	$\mathtt{q}_3$	1
$\mathtt{q}_3$	$\mathtt{q}_3$	$\mathtt{q}_3$	0
$\mathtt{q}_4$	$\mathtt{q}_4$	$\mathtt{q}_4$	0

б

	a	b	F
$\mathtt{q}_0$	$\mathtt{q}_0$	$\mathtt{q}_1$	0
$\mathtt{q}_1$	$\mathtt{q}_2$	$\mathtt{q}_3$	0
$\mathtt{q}_2$	$\mathtt{q}_2$	$\mathtt{q}_4$	1
$\mathtt{q}_3$	$\mathtt{q}_3$	$\mathtt{q}_4$	0
$\mathtt{q}_4$	$\mathtt{q}_3$	$\mathtt{q}_4$	0

в

Цепочки $\mathtt{aba}, \mathtt{a}^2\mathtt{ba}^3$ распознаются этим автоматом, а цепочки $\mathtt{a}^2, \mathtt{abab}$ не распознаются. Нетрудно понять, что

$$L(\mathcal{A}_1) = \{\mathtt{ba}^k \mid k \in \mathbb{N}\} \cup \{\mathtt{a}^n\mathtt{ba}^k \mid n, k \in \mathbb{N}\}.$$

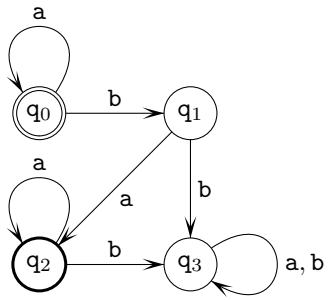
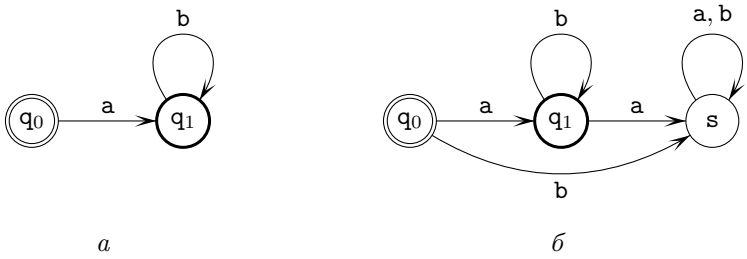


Рис. 2.2. *Диаграмма переходов автомата* $\mathcal{A}_1$

Замечание 2.1. Часто в определении ДКА функцию переходов считают *частичной*, а ДКА со всюду определенной функцией переходов считают частным случаем и называют *полным* ДКА (мы же будем называть ДКА с частичной функцией переходов *неполным*). Если неполный автомат не может выполнить очередной такт, то он останавливается (и входная цепочка при этом не распознается). Доопределение функции переходов на произвольные слова производится так же, как это проделано выше, при этом определение распознаваемой цепочки не изменяется. Нетрудно видеть, что всякий неполный ДКА можно доопределить до полного с сохранением распознаваемого языка. Для этого достаточно ввести одно новое незаключительное состояние s и значения всех неопределенных переходов (включая переходы из s) положить равными s (рис. 2.3).



а *б*

Рис. 2.3. *Неполный (а) и полный (б) автоматы для языка* $\{\mathtt{ab}^n \mid n \geq 0\}$

Замечание 2.2. Для произвольной цепочки w и произвольного состояния q автомата $\mathcal{A}$ в диаграмме переходов $\mathcal{A}$ существует един-

ственный путь, начинающийся в $\mathtt{q}$ и помеченный w. Для произвольной цепочки w и произвольного состояния $\mathtt{q}$ неполного автомата $\mathcal{A}$ в диаграмме переходов $\mathcal{A}$ существует *не более одного* пути из $\mathtt{q}$, помеченного w.

§ 4. Приведенные конечные автоматы

Ясно, что язык может распознаваться не одним автоматом. Например, если таблицу переходов автомата $\mathcal{A}_1$ немного изменить (табл. 2.1, *б* и *в*), то мы получим автоматы $\mathcal{A}_2$ и $\mathcal{A}_3$, распознающие тот же самый язык $L(\mathcal{A}_1)$. Эти два автомата в определенном смысле «хуже» автомата $\mathcal{A}_1$. Действительно, автомат $\mathcal{A}_2$ содержит состояние $\mathtt{q}_4$, в которое он никогда не перейдет при анализе цепочки. Такое состояние является, следовательно, «лишним». В автомате $\mathcal{A}_3$ состояния $\mathtt{q}_3$ и $\mathtt{q}_4$ при анализе любой цепочки «ведут себя одинаково», и поэтому одно из них также является «лишним». Возникает естественный вопрос: *как для автомата $\mathcal{A}$ найти автомат $\mathcal{B}$, минимальный по числу состояний и распознающий тот же язык, что и $\mathcal{A}$?* Прежде чем ответить на него, дадим точные определения. Через $\mathcal{A}^{\mathtt{q}}$ будем обозначать автомат, полученный из автомата $\mathcal{A}$ заменой начального состояния на состояние $\mathtt{q}$.

Определение. Состояние $\mathtt{s}$ автомата $\mathcal{A}$ называется *достижимым*, если существует цепочка w такая, что $\delta(\mathtt{q}_0, w) = \mathtt{s}$, и *недостижимым* в противном случае. Состояния $\mathtt{s}$ и $\mathtt{t}$ автомата $\mathcal{A}$ *эквивалентны* (обозначается $\mathtt{s} \sim \mathtt{t}$), если $L(\mathcal{A}^{\mathtt{s}}) = L(\mathcal{A}^{\mathtt{t}})$. Автомат называется *приведенным*, если он не имеет (различных) эквивалентных состояний и любое его состояние достижимо.

Определение. Автомат $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, \mathtt{q}_1, F_1)$ изоморфен автомату $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, \mathtt{q}_2, F_2)$, если существует биекция $h : Q_1 \to Q_2$, сохраняющая переходы, т. е. такая, что $h(\delta_1(\mathtt{q}, \mathtt{a})) = \delta_2(h(\mathtt{q}), \mathtt{a})$ для произвольных $\mathtt{q} \in Q_1$, $\mathtt{a} \in \Sigma$.

Замечание 2.3. Автоматы $\mathcal{A}_1$ и $\mathcal{A}_2$ изоморфны тогда и только тогда, когда их диаграммы переходов изоморфны как помеченные орграфы.

Определение. Два автомата *эквивалентны*, если они имеют общий алфавит и распознают один и тот же язык[†].

[†]Автоматы, имеющие разные алфавиты, нельзя считать эквивалентными, так как множества допустимых входов для этих автоматов различны.

Замечание 2.4. Понятие эквивалентности состояний можно рассматривать в более широком контексте, положив, что состояние $\mathtt{s}$ автомата $\mathcal{A}_1$ и состояние $\mathtt{t}$ автомата $\mathcal{A}_2$ эквивалентны, если выполняется условие $L(\mathcal{A}_1^{\mathtt{s}}) = L(\mathcal{A}_2^{\mathtt{t}})$. В частности, автоматы $\mathcal{A}_1$ и $\mathcal{A}_2$ с общим алфавитом эквивалентны тогда и только тогда, когда эквивалентны их начальные состояния.

Лемма 2.1. *Пусть* $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, \mathtt{q}_1, F_1)$ *и* $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, \mathtt{q}_2, F_2)$ *– ДКА с общим алфавитом,* $\mathtt{s} \in Q_1$, $\mathtt{t} \in Q_2$ *и* $\mathtt{s} \sim \mathtt{t}$. *Тогда для любого* $\mathtt{a} \in \Sigma$ *выполняется равенство* $\delta_1(\mathtt{s}, \mathtt{a}) \sim \delta_2(\mathtt{t}, \mathtt{a})$.

Доказательство. Пусть $w \in \mathcal{A}_1^{\delta_1(\mathtt{s},\mathtt{a})}$, т. е. $\delta_1(\delta_1(\mathtt{s}, \mathtt{a}), w) \in F_1$. По определению имеем

$$\delta_1(\delta_1(\mathtt{s}, \mathtt{a}), w) = \delta_1(\mathtt{s}, \mathtt{a}w),$$

откуда $\mathtt{a}w \in \mathcal{A}_1^{\mathtt{s}}$. Поскольку $L(\mathcal{A}_1^{\mathtt{s}}) = L(\mathcal{A}_2^{\mathtt{t}})$, выполнено условие

$$\delta_2(\mathtt{t}, \mathtt{a}w) = \delta_2(\delta_2(\mathtt{t}, \mathtt{a}), w) \in F_2,$$

т. е. $w \in \mathcal{A}_2^{\delta_2(\mathtt{t},\mathtt{a})}$ (рис. 2.4); в силу симметрии, $\mathcal{A}_1^{\delta_1(\mathtt{s},\mathtt{a})} = \mathcal{A}_2^{\delta_2(\mathtt{t},\mathtt{a})}$.

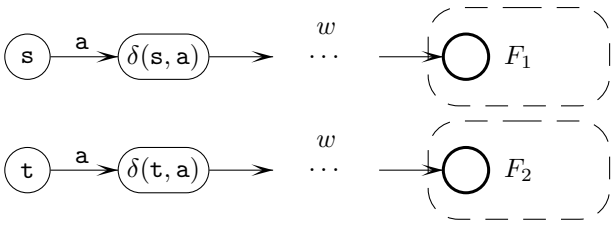


Рис. 2.4. *Иллюстрация стабильности отношения* $\sim$

□

Описанное в лемме 2.1 свойство отношения $\sim$ называется *стабильностью*. Используя это свойство, мы можем дать ответ на вопрос о минимальном автомате.

Теорема 2.1. *Пусть* $\mathcal{A}$ *– произвольный автомат. Среди автоматов, эквивалентных* $\mathcal{A}$, *существует единственный с точностью до изоморфизма приведенный автомат* $\mathcal{B}$. *Количество состояний автомата* $\mathcal{B}$ *строго меньше количества состояний любого другого автомата, эквивалентного* $\mathcal{A}$.

Доказательство. Вначале докажем существование приведенного автомата, эквивалентного автомату $\mathcal{A}$.

Множество достижимых состояний данного автомата найти просто. Достаточно, к примеру, запустить поиск в глубину или в ширину из начального состояния (таблица переходов предоставляет данные в формате, удобном для такого поиска). Удалив недостижимые состояния, мы очевидно получаем автомат, эквивалентный исходному. Поэтому далее предполагается, что $\mathcal{A} = (Q, \Sigma, \delta, \mathtt{q}_0, F)$ – ДКА, все состояния которого достижимы. Поскольку $\sim$ очевидно является отношением эквивалентности на Q, рассмотрим соответствующее разбиение $Q = K_1 \cup \ldots \cup K_m$. Определим ДКА $R(\mathcal{A}) = (K, \Sigma, \phi, K_{\mathtt{q}_0}, F_K)$ следующим образом: $K = \{K_1, \ldots, K_m\}$, $K_{\mathtt{q}_0}$ – класс, содержащий состояние $\mathtt{q}_0$, множество F_K состоит из всех классов, содержащих только состояния из F, а

$$\phi(K_i, \mathtt{a}) = K_j \iff \exists \mathtt{q} \in K_i \ (\delta(\mathtt{q}, \mathtt{a}) \in K_j)\,. \tag{2.1}$$

Автомат $R(\mathcal{A})$ определен корректно, так как из леммы 2.1 следует (при $\mathcal{A}_1 = \mathcal{A}_2 = \mathcal{A}$), что условие (2.1) однозначно задает функцию переходов. Действительно, если взять другую вершину $\mathtt{q}' \in K_i$, то $\mathtt{q}' \sim \mathtt{q}$ и $\delta(\mathtt{q}', \mathtt{a}) \in K_j$ ввиду стабильности отношения $\sim$.

Заметим, что $\mathtt{s} \in F$ тогда и только тогда, когда $\varepsilon \in \mathcal{A}^{\mathtt{s}}$. Следовательно, каждый класс K_i содержит либо только заключительные, либо только незаключительные состояния. В частности, все заключительные состояния принадлежат классам из F_K.

Из стабильности отношения $\sim$ следует, что для $\mathtt{q} \in K_i$, $\mathtt{q}' \in K_j$ равенство $\delta(\mathtt{q}, \mathtt{a}) = \mathtt{q}'$ влечет $\phi(K_i, \mathtt{a}) = K_j$. По индукции это свойство выполняется не только для букв, но и для произвольного слова $w \in \Sigma^*$. Обратное тоже верно: если $\phi(K_i, w) = K_j$, то для любого $\mathtt{q} \in K_i$ имеем $\delta(\mathtt{q}, w) \in K_j$. Следовательно, условие $\delta(\mathtt{q}_0, w) \in F$ выполняется тогда и только тогда, когда $\phi(K_{\mathtt{q}_0}, w) \in F_K$, т. е. языки, распознаваемые автоматами $\mathcal{A}$ и $R(\mathcal{A})$, совпадают.

Осталось проверить, что автомат $R(\mathcal{A})$ приведенный. Все его состояния достижимы, поскольку в автомате $\mathcal{A}$ нет недостижимых состояний; в качестве слова, по которому данное состояние K_i может быть достигнуто из $K_{\mathtt{q}_0}$, можно взять любое слово, по которому некоторая вершина из K_i достигается в автомате $\mathcal{A}$. Далее, возьмем два произвольных состояния $K_i \neq K_j$ автомата $R(\mathcal{A})$ и докажем, что они не эквивалентны. Пусть $\mathtt{q} \in K_i$, $\mathtt{q}' \in K_j$. Поскольку $\mathtt{q} \not\sim \mathtt{q}'$, существует слово w такое, что ровно одно из состояний $\delta(\mathtt{q}, w)$, $\delta(\mathtt{q}', w)$ автомата $\mathcal{A}$ является заключительным. Тогда ровно одно из состояний

$\phi(K_i, w)$, $\phi(K_j, w)$ автомата $R(\mathcal{A})$ заключительное, откуда следует неэквивалентность K_i и K_j.

Теперь докажем единственность приведенного автомата, эквивалентного данному. Рассмотрим эквивалентные приведенные автоматы $\mathcal{B}_1 = (Q_1, \Sigma, \delta_1, \mathtt{q}_1, F_1)$ и $\mathcal{B}_2 = (Q_2, \Sigma, \delta_2, \mathtt{q}_2, F_2)$ и докажем, что они изоморфны. Состояния $\mathtt{q}_1$ и $\mathtt{q}_2$ эквивалентны (замечание 2.4). Возьмем произвольное состояние $\mathtt{s} \in Q_1$. По определению приведенного автомата оно достижимо, т. е. существует цепочка w такая, что $\delta_1(\mathtt{q}_1, w) = \mathtt{s}$. Положим $\mathtt{t} = \delta_2(\mathtt{q}_2, w)$. Снова пользуясь стабильностью отношения $\sim$, устанавливаем, что $\mathtt{s} \sim \mathtt{t}$.

Итак, каждое состояние автомата $\mathcal{B}_1$ имеет эквивалентное состояние в автомате $\mathcal{B}_2$, и притом единственное, так как $\mathcal{B}_2$ не содержит различных эквивалентных состояний по определению. Следовательно, мы имеем биекцию между множествами Q_1 и Q_2. Эта биекция сохраняет переходы в силу стабильности отношения $\sim$, т. е. является изоморфизмом автоматов $\mathcal{B}_1$ и $\mathcal{B}_2$.

Последнее утверждение теоремы следует из того факта, что если автомат $\mathcal{A}$ не является приведенным, то эквивалентный ему приведенный автомат $R(\mathcal{A})$ по построению имеет строго меньшее количество состояний, чем $\mathcal{A}$. Теорема доказана. □

Из доказательства теоремы следует, что для получения приведенного автомата, эквивалентного данному, достаточно уметь эффективно строить отношение $\sim$. Мы будем строить не само отношение, а соответствующее разбиение.

Будем говорить, что слово w является *разделяющим* для состояний $\mathtt{s}$ и $\mathtt{t}$ автомата $\mathcal{A}$, если w принадлежит ровно одному из языков $L(\mathcal{A}^{\mathtt{s}})$, $L(\mathcal{A}^{\mathtt{t}})$. Наименьшую длину разделяющего слова обозначим через $sep(\mathtt{s}, \mathtt{t})$ (если $\mathtt{s} \sim \mathtt{t}$, то $sep(\mathtt{s}, \mathtt{t}) = \infty$).

Лемма 2.2. *Пусть $\mathcal{A} = (Q, \Sigma, \delta, \mathtt{q}_0, F)$ – ДКА, $k \in \mathbb{N}$. Тогда либо $sep(\mathtt{s}, \mathtt{t}) = k$ для некоторых $\mathtt{s}, \mathtt{t} \in Q$, либо $sep(\mathtt{s}, \mathtt{t}) < k$ для любых неэквивалентных $\mathtt{s}, \mathtt{t} \in Q$.*

Доказательство. Утверждение леммы вытекает из следующего наблюдения: слово $\mathtt{a}w$ разделяет состояния $\mathtt{s}$ и $\mathtt{t}$ тогда и только тогда, когда w разделяет состояния $\delta(\mathtt{s}, \mathtt{a})$ и $\delta(\mathtt{t}, \mathtt{a})$ (рис. 2.5). □

Алгоритм 2.1. *Построение классов эквивалентных состояний.*
Вход. ДКА $\mathcal{A} = (Q, \Sigma, \delta, \mathtt{q}_0, F)$.

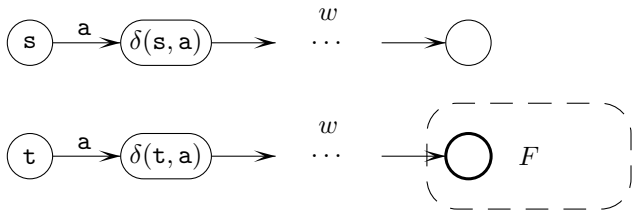


Рис. 2.5. *Разделяющие слова*

Выход. Множество $P = \{K_1, \ldots, K_m\}$ классов разбиения Q на эквивалентные состояния.

1. $P_0 \leftarrow \{F, Q\backslash F\}$; $n \leftarrow 0$
2. $P_{n+1} \leftarrow \bigcup_{K \in P_n} \text{SEPARATE}(P_n, K)$
3. **если** $(P_n \neq P_{n+1})$
4. $\quad n \leftarrow n + 1$; **перейти на** 2
5. **иначе** $P \leftarrow P_n$

Используемая в строке 2 функция $\text{SEPARATE}(P, K)$ возвращает разбиение класса K разбиения P на подклассы в соответствии со следующим условием: *два состояния* $\mathtt{s}$ *и* $\mathtt{t}$ *попадают в один подкласс тогда и только тогда, когда для любого* $\mathtt{a} \in \Sigma$ *состояния* $\delta(\mathtt{s}, \mathtt{a})$ *и* $\delta(\mathtt{t}, \mathtt{a})$ *находятся в одном классе разбиения* P.

Докажем корректность алгоритма. Пусть n – минимальное число такое, что состояния $\mathtt{s}$ и $\mathtt{t}$ принадлежат разным классам разбиения P_n. Покажем, что $sep(\mathtt{s}, \mathtt{t}) = n$. База индукции: $n = 0$. Очевидно, что состояние из F и состояние из $Q\backslash F$ разделяются пустым словом. Проверим шаг индукции. По выбору числа n состояния $\mathtt{s}$ и $\mathtt{t}$ принадлежат одному классу разбиения P_{n-1}. Следовательно, для любого $\mathtt{a} \in \Sigma$ состояния $\delta(\mathtt{s}, \mathtt{a})$ и $\delta(\mathtt{t}, \mathtt{a})$ находятся в одном классе разбиения P_{n-2}. В то же время для некоторого $\mathtt{a} \in \Sigma$ эти состояния принадлежат разным классам P_{n-1}. Значит, к состояниям $\delta(\mathtt{s}, \mathtt{a})$ и $\delta(\mathtt{t}, \mathtt{a})$ применимо предположение индукции и $sep(\delta(\mathtt{s}, \mathtt{a}), \delta(\mathtt{t}, \mathtt{a})) = n - 1$. Но тогда $sep(\mathtt{s}, \mathtt{t}) = n$ (ср. доказательство леммы 2.2).

Таким образом, если алгоритм разделяет два состояния, то они не эквивалентны. Если алгоритм закончил работу, положив $P = P_n$, то $P_{n+1} = P_n$, т. е. не существует состояний с условием $sep(\mathtt{s}, \mathtt{t}) = n{+}1$. Согласно лемме 2.2 все неэквивалентные состояния уже разделены.

Корректность алгоритма доказана. Заметим, что число итераций цикла ограничено сверху количеством состояний автомата.

Пример 2.2. Построим классы эквивалентных состояний для автомата $\mathcal{A}_4$, заданного табл. 2.2.

Таблица 2.2. *Таблица переходов автомата* $\mathcal{A}_4$

	a	b	F
0	2	4	0
1	7	4	0
2	3	6	0
3	3	1	0
4	7	1	0
5	2	7	1
6	7	3	0
7	7	5	1

Исходное разбиение имеет вид $P_0 = \{\ \{0,1,2,3,4,6\}, \{5,7\}\ \}$.

При первой итерации первый класс разобьется на подклассы $\{0,2,3\}$ (переходы по любой букве приводят в первый класс разбиения P_0) и $\{1,4,6\}$ (переход по a во второй, а по b – в первый класс разбиения P_0). Второй класс разобьется на два одноэлементных подкласса. Получаем $P_1 = \{\ \{0,2,3\}, \{1,4,6\}, \{5\}, \{7\}\ \}$.

При второй итерации произойдет только одно изменение: из класса $\{1,4,6\}$ выделится состояние 4 (благодаря переходу по b). Вследствие этого на третьем шаге разделится класс $\{0,2,3\}$, и это будет последним изменением разбиения. В итоге

$$P = P_3 = \{\ \{0,3\}, \{2\}, \{1,4\}, \{6\}, \{5\}, \{7\}\ \}.$$

Замечание 2.5. Алгоритм 2.1 может быть реализован на практике следующим образом. Для хранения текущего класса каждого состояния автомата используем статический массив $Class$ длины $|Q|$, а множество классов сформируем в виде очереди. Основным шагом является разбиение класса K (первого в очереди) на подклассы, производимое следующим образом. В конец очереди добавляется новый класс и в него переносится произвольный элемент из K. Далее, пока K не пуст, берем его произвольный элемент q, удаляем из K

и сравниваем с одним представителем каждого из вновь построенных классов. Если для представителя $\mathtt{r}$ некоторого из этих классов выполняется $Class[\delta(\mathtt{q},\mathtt{a})] = Class[\delta(\mathtt{r},\mathtt{a})]$ для любого $\mathtt{a} \in \Sigma$, то $\mathtt{q}$ помещаем в данный класс. Если же $\mathtt{q}$ нельзя поместить ни в один из вновь созданных классов, то создаем для $\mathtt{q}$ новый класс в конце очереди. Когда K становится пустым, удаляем его, после чего проходим по вновь созданным классам и обновляем для их элементов значения массива $Class$.

Пусть счетчик num хранит число классов. Условие остановки алгоритма 2.1 эквивалентно тому, что значение num останется неизменным в течение num шагов. Для контроля этого условия удобно использовать счетчик $fails$, значение которого увеличивается на 1, если на очередном шаге значение num не изменилось, и обнуляется, если это значение увеличилось. При условии $num = fails$ алгоритм заканчивает работу.

§ 5. Недетерминированные конечные автоматы

Определение. *Недетерминированным конечным автоматом* (сокращенно НКА) называется пятерка $\mathcal{B}$=$(Q, \Sigma, \delta, Q_0, F)$, где Q, Σ, F – те же, что и в случае ДКА, Q_0 – множество начальных состояний, а $\delta:\ Q \times \Sigma \to 2^Q$ – недетерминированная функция переходов.

Замечание 2.6. Функция переходов δ может быть эквивалентно определена условием $\delta \subseteq Q \times \Sigma \times Q$, т. е. δ есть *произвольное множество переходов*. Такое определение, в частности, показывает, что любой помеченный орграф является диаграммой переходов некоторого НКА (элементы δ, т. е. тройки вида $(\mathtt{q}, \mathtt{a}, \mathtt{q}')$, суть помеченные ребра орграфа).

На НКА можно смотреть, аналогично ДКА, как на физическое устройство. Разница состоит в следующем. НКА, находящийся в начале такта в состоянии q и обозревающий ячейку с символом a, может перейти в *любое* состояние из множества $\delta(\mathtt{q}, \mathtt{a})$. Если $\delta(\mathtt{q}, \mathtt{a}) = \emptyset$, то автомат останавливается. Как и для ДКА, доопределим функцию переходов на произвольных словах $w \in \Sigma^*$ естественным образом:

$$\delta(\mathtt{q}, \varepsilon) = \mathtt{q}, \qquad \delta(\mathtt{q}, u\mathtt{a}) = \bigcup_{\mathtt{r} \in \delta(\mathtt{q}, u)} \delta(\mathtt{r}, \mathtt{a}).$$

Определение. НКА $\mathcal{B} = (Q, \Sigma, \delta, Q_0, F)$ *распознает* (или *допускает*) цепочку $w \in \Sigma^*$, если $\big(\bigcup\limits_{\mathtt{q} \in Q_0} \delta(\mathtt{q}, w)\big) \cap F \neq \emptyset$. (Другими слова-

ми, *существует* последовательность тактов автомата $\mathcal{B}$, в результате которой он просмотрит всю цепочку w и перейдет из некоторого начального состояния в некоторое заключительное.) Языком, *распознаваемым* $\mathcal{B}$ (обозначается через $L(\mathcal{B})$), называется множество всех цепочек, допускаемых этим автоматом.

Для задания НКА можно использовать расширенную таблицу переходов и диаграмму переходов. Клетки таблицы содержат списки состояний (возможно, пустые); в еще одном дополнительном столбце помечены начальные вершины. В диаграмме переходов из одной вершины может выходить несколько (или ни одной) дуг, помеченных одним и тем же входным символом.

Пример 2.3. Рассмотрим НКА $\mathcal{B}$, заданный табл. 2.3.

Таблица 2.3. *Расширенная таблица переходов НКА* $\mathcal{B}$

	a	b	Q_0	F
$\mathtt{q}_0$	$\mathtt{q}_0$,$\mathtt{q}_1$	$\mathtt{q}_0$	1	0
$\mathtt{q}_1$		$\mathtt{q}_2$	0	0
$\mathtt{q}_2$	$\mathtt{q}_2$	$\mathtt{q}_1$	0	1

Цепочка $\mathtt{a}^2\mathtt{b}$ допускается этим автоматом. Действительно, имеем $\delta(\mathtt{q}_0, \mathtt{a}) = \delta(\mathtt{q}_0, \mathtt{a}^2) = \{\mathtt{q}_0, \mathtt{q}_1\}$, $\delta(\mathtt{q}_0, \mathtt{a}^2\mathtt{b}) = \{\mathtt{q}_0, \mathtt{q}_2\}$, $\{\mathtt{q}_0, \mathtt{q}_2\} \cap \mathtt{F} \neq \emptyset$. В то же время для цепочки abba вычисляем $\delta(\mathtt{q}_0, \mathtt{abba}) = \{\mathtt{q}_0, \mathtt{q}_1\}$, т. е. $\mathcal{B}$ не распознает эту цепочку. Диаграмма переходов автомата $\mathcal{B}$ изображена на рис. 2.6.

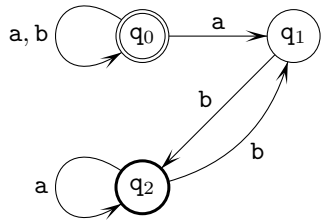


Рис. 2.6. *Диаграмма переходов НКА* $\mathcal{B}$

Распознаваемость цепочки w означает существование на диаграмме пути из $\mathtt{q}_0$ в $\mathtt{q}_2$, помеченного w.

Ясно, что ДКА является частным случаем НКА. Следовательно, класс языков, распознаваемых ДКА, содержится в классе языков, распознаваемых НКА.

Теорема 2.2 (Рабина–Скотта). *Класс языков, распознаваемых НКА, совпадает с классом языков, распознаваемых ДКА.*

Доказательство. Возьмем произвольный НКА $\mathcal{B} = (Q, \Sigma, \delta, Q_0, F)$. Для доказательства теоремы достаточно показать, что существует ДКА $\mathcal{A}$ такой, что $L(\mathcal{A}) = L(\mathcal{B})$.

Возьмем автомат $\mathcal{A} = (2^Q, \Sigma, \delta', Q_0, F')$, где

$$\delta'(P, \mathtt{a}) = \bigcup_{\mathtt{q} \in P} \delta(\mathtt{q}, \mathtt{a}), \qquad F' = \{P \in 2^Q \mid P \cap F \neq \emptyset\},$$

и докажем, что этот автомат – искомый. Для этого индукцией по длине произвольной цепочки w покажем, что

$$\delta'(P, w) = \bigcup_{\mathtt{q} \in P} \delta(\mathtt{q}, w).$$

База индукции ($|w| = 0$) немедленно следует из определений. Рассмотрим шаг индукции. Пусть $w = u\mathtt{a}$:

$$\delta'(P, u\mathtt{a}) = \delta'(\delta'(P, u), \mathtt{a}) = [\texttt{по предположению индукции}]$$

$$= \delta'(\bigcup_{\mathtt{q} \in P} \delta(\mathtt{q}, u), \mathtt{a}) = \bigcup_{\mathtt{r} \in \bigcup\limits_{\mathtt{q} \in P} \delta(\mathtt{q}, u)} \delta(\mathtt{r}, \mathtt{a}) = \bigcup_{\mathtt{q} \in P} \bigcup_{\mathtt{r} \in \delta(\mathtt{q}, u)} \delta(\mathtt{r}, \mathtt{a}) = \bigcup_{\mathtt{q} \in P} \delta(\mathtt{q}, u\mathtt{a}).$$

Шаг индукции доказан. Осталось заметить, что

$$L(\mathcal{A}) = \{w \in \Sigma^* \mid \delta'(Q_0, w) \in F'\} = \{w \in \Sigma^* \mid \delta'(Q_0, w) \cap F \neq \emptyset\} =$$

$$= \{w \in \Sigma^* \mid \bigcup_{\mathtt{q} \in Q_0} \delta(\mathtt{q}, w) \cap F \neq \emptyset\} = L(\mathcal{B}).$$

Теорема доказана. □

Приведенный в доказательстве теоремы способ получения ДКА из НКА называется «построением подмножеств». Его эффективная реализация, позволяющая сразу получать ДКА без недостижимых состояний, приведена в алгоритме 2.2.

Алгоритм 2.2. *Переход от НКА к ДКА построением подмножеств.*
Вход. НКА $\mathcal{B} = (Q, \Sigma, \delta, Q_0, F)$.
Выход. Эквивалентный ДКА $\mathcal{A} = (Q', \Sigma, \delta', Q_0, F')$ без недостижимых состояний.

1. **для каждого** $P \subseteq Q$ $label(P) \leftarrow 0$
2. $Q' \leftarrow \{Q_0\}$
3. **пока** $(\exists P \in Q' : label(P) = 0)$, **повторять**
4. **для каждого** $\mathtt{a} \in \Sigma$
5. $\delta'(P, \mathtt{a}) \leftarrow \bigcup\limits_{\mathtt{q} \in P} \delta(\mathtt{q}, \mathtt{a})$
6. $Q' \leftarrow Q' \cup \{\delta'(P, \mathtt{a})\}$
7. $label(P) \leftarrow 1$
8. $F' \leftarrow \{P \in Q' \mid P \cap F \neq \emptyset\}$

Поскольку каждое состояние в Q', кроме начального, получено переходом из ранее построенного состояния, то все состояния $\mathcal{A}$ очевидно достижимы.

Пример 2.4. Проиллюстрируем построение автомата $\mathcal{A}$ для автомата $\mathcal{B}$ из примера 2.3. Автомат $\mathcal{A}$ задан расширенной таблицей переходов; порядок строк в ней соответствует порядку появления состояний в Q'.

Таблица 2.4. *Результат «детерминирования» НКА* $\mathcal{B}$

	a	b	F
$\mathtt{q}_0$	$\mathtt{q}_0, \mathtt{q}_1$	$\mathtt{q}_0$	0
$\mathtt{q}_0, \mathtt{q}_1$	$\mathtt{q}_0, \mathtt{q}_1$	$\mathtt{q}_0, \mathtt{q}_2$	0
$\mathtt{q}_0, \mathtt{q}_2$	$\mathtt{q}_0, \mathtt{q}_1, \mathtt{q}_2$	$\mathtt{q}_0, \mathtt{q}_1$	1
$\mathtt{q}_0, \mathtt{q}_1, \mathtt{q}_2$	$\mathtt{q}_0, \mathtt{q}_1, \mathtt{q}_2$	$\mathtt{q}_0, \mathtt{q}_1, \mathtt{q}_2$	1

Введем еще один вид автомата, обобщающий как ДКА, так и НКА.

Определение. *Недетерминированным конечным автоматом с* ε*-переходами* (ε-НКА) называется пятерка $\mathcal{B} = (Q, \Sigma, \delta, Q_0, F)$, где Q, Σ, F, Q_0 – те же, что и в случае НКА, а $\delta : \ Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ – недетерминированная функция переходов.

Основным отличием ε-НКА от рассмотренных ранее конечных автоматов является возможность не сдвигаться по входной цепочке по окончании некоторых тактов (на таком такте автомат «читает пустое слово»). Соответственно диаграмма переходов ε-НКА может содержать дуги, помеченные ε. В дальнейшем мы увидим, что ε-НКА очень удобно строить по заданному языку.

Как и рассмотренные ранее виды конечных автоматов, ε-НКА распознает цепочку w, если и только если в диаграмме переходов существует путь из некоторого начального состояния в некоторое заключительное, помеченный w. Определение распознаваемости через обобщение функции переходов на произвольные слова будет дано ниже.

Замечание 2.7. Произвольный ε-НКА эквивалентен ε-НКА с единственным начальным и единственным заключительным состоянием. Действительно, достаточно добавить к исходному автомату два состояния, $\mathtt{q}_0$ и $\mathtt{f}$, как показано на рис. 2.7; ε-НКА вида $(Q, \Sigma, \delta, \mathtt{q}_0, \mathtt{f})$ мы будем называть *нормальными*. В дальнейшем рассматриваются только нормальные ε-НКА.

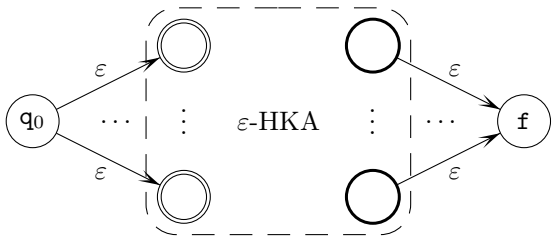


Рис. 2.7. *Построение нормального ε-НКА*

Пусть $\mathcal{B} = (Q, \Sigma, \delta, \mathtt{q}_0, \mathtt{f})$ – нормальный ε-НКА. Рассмотрим бинарное отношение ρ на множестве состояний:

$$(\mathtt{q}, \mathtt{p}) \in \rho \Longleftrightarrow \mathtt{p} \in \delta(\mathtt{q}, \varepsilon).$$

Заметим, что пара $(\mathtt{q}, \mathtt{p})$ лежит в рефлексивно-транзитивном замыкании ρ^* отношения ρ тогда и только тогда, когда автомат может перейти из состояния $\mathtt{q}$ в состояние $\mathtt{p}$, не сдвигаясь по входной цепочке. Множество состояний $\mathtt{p}$ таких, что $(\mathtt{q}, \mathtt{p}) \in \rho^*$, будем называть *замыканием* $\mathtt{q}$ и обозначать $Clo(\mathtt{q})$. Замыкание множества состояний P

зададим равенством $Clo(P) = \bigcup\limits_{\mathtt{q}\in P} Clo(\mathtt{q})$. Определим «обобщенную» функцию переходов для произвольных слов:

$$\bar{\delta}(\mathtt{q},\varepsilon) = Clo(\mathtt{q}), \qquad \bar{\delta}(\mathtt{q},u\mathtt{a}) = \bigcup_{\mathtt{r}\in\bar{\delta}(\mathtt{q},u)} Clo(\delta(\mathtt{r},\mathtt{a})).$$

Легко заметить, что $\mathtt{r} \in \bar{\delta}(\mathtt{q},w)$ тогда и только тогда, когда в диаграмме автомата $\mathcal{B}$ существует путь из $\mathtt{q}$ в $\mathtt{r}$, помеченный w. Таким образом, распознаваемость цепочки w автоматом $\mathcal{B}$ можно эквивалентно определить условием $\mathtt{f} \in \bar{\delta}(\mathtt{q}_0,w)$.

Теорема 2.3. *Класс языков, распознаваемых ε-НКА, совпадает с классом языков, распознаваемых НКА.*

Доказательство. Рассмотрим произвольный нормальный ε-НКА $\mathcal{B} = (Q,\Sigma,\delta,\mathtt{q}_0,\mathtt{f})$ и построим НКА $\mathcal{B}' = (Q,\Sigma,\delta',Q_0,\mathtt{f})$ такой, что $Q_0 = Clo(\mathtt{q}_0)$, $\delta'(\mathtt{q},\mathtt{a}) = \bar{\delta}(\mathtt{q},\mathtt{a})$. Сравним пути, помеченные цепочкой $w = \mathtt{a}_1\ldots\mathtt{a}_n$, в соответствующих диаграммах переходов (рис. 2.8). Из определения функции $\bar{\delta}$ следует, что путь в $\mathcal{B}$ (дуги прямыми линиями) существует тогда и только тогда, когда существует путь в $\mathcal{B}'$ (дуги пунктиром снизу). Тем самым множества цепочек, распознаваемых $\mathcal{B}$ и $\mathcal{B}'$, совпадают, т. е. автоматы $\mathcal{B}$ и $\mathcal{B}'$ эквивалентны. Теорема доказана. □

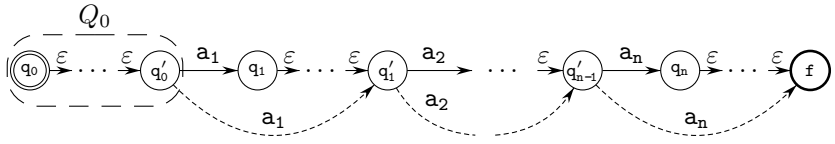


Рис. 2.8. *Сравнение путей в ε-НКА и «обычном» НКА*

Алгоритм 2.2 можно немного видоизменить для построения ДКА, эквивалентного данному ε-НКА. Заметим, что замыкание $Clo(\mathtt{q})$ легко вычисляется любым поиском из состояния $\mathtt{q}$.

Алгоритм 2.3. *Построение ДКА по ε-НКА.*
Вход. ε-НКА $\mathcal{B} = (Q,\Sigma,\delta,\mathtt{q}_0,F)$.
Выход. Эквивалентный ДКА $\mathcal{A} = (Q',\Sigma,\delta',Q_0,F')$ без недостижимых состояний.

1. **для каждого** $P \subseteq Q \;\; label(P) \leftarrow 0$
2. $Q_0 \leftarrow Clo(\mathtt{q}_0)$; $Q' \leftarrow \{Clo(\mathtt{q}_0)\}$
3. **пока** $(\exists P \in Q' : label(P) = 0)$, **повторять**
4. **для каждого** $\mathtt{a} \in \Sigma$
5. $\delta'(P, \mathtt{a}) \leftarrow \bigcup\limits_{\mathtt{q} \in P} \bar{\delta}(\mathtt{q}, \mathtt{a})$
6. $Q' \leftarrow Q' \cup \{\delta'(P, \mathtt{a})\}$
7. $label(P) \leftarrow 1$
8. $F' \leftarrow \{P \in Q' \mid P \cap F \neq \emptyset\}$

§ 6. Операции над языками, распознаваемыми конечными автоматами

Обозначим через $\mathbb{A}$ класс всех языков над фиксированным алфавитом Σ, распознаваемых конечными автоматами. Естественно возникает вопрос о замкнутости класса $\mathbb{A}$ относительно операций над языками. Ответ на этот вопрос содержится в следующем утверждении.

Теорема 2.4. *Класс $\mathbb{A}$ замкнут относительно объединения, пересечения, дополнения, произведения и итерации.*

Доказательство. Рассмотрим произвольные нормальные ε-НКА $\mathcal{B}_1 = (Q_1, \Sigma, \delta_1, \mathtt{q}_1, \mathtt{f}_1)$ и $\mathcal{B}_2 = (Q_2, \Sigma, \delta_2, \mathtt{q}_2, \mathtt{f}_2)$, считая, что $Q_1 \cap Q_2 = \emptyset$. Пусть $L_1 = L(\mathcal{B}_1)$, $L_2 = L(\mathcal{B}_2)$. Тогда язык $L_1 \cup L_2$ распознается автоматом $\mathcal{B}_3 = (Q_1 \cup Q_2 \cup \{\mathtt{q}_0, \mathtt{f}\}, \Sigma, \delta_3, \mathtt{q}_0, \mathtt{f})$, получаемым «параллельным соединением» автоматов $\mathcal{B}_1$ и $\mathcal{B}_2$ (рис. 2.9). В самом деле, существование в диаграмме переходов $\mathcal{B}_3$ пути из $\mathtt{q}_0$ в $\mathtt{f}$, помеченного словом w, равносильно существованию пути, помеченного этим словом, либо в автомате $\mathcal{B}_1$ из $\mathtt{q}_1$ в $\mathtt{f}_1$, либо в автомате $\mathcal{B}_2$ из $\mathtt{q}_2$ в $\mathtt{f}_2$; отсюда $L(\mathcal{B}_3) = L(\mathcal{B}_1) \cup L(\mathcal{B}_2)$. Аналогичные рассуждения справедливы для автомата $\mathcal{B}_4$, распознающего язык $L_1 L_2$ и представленного на рис. 2.10, а также для автомата $\mathcal{B}_5$, распознающего язык L_1^* (рис. 2.11). Таким образом, замкнутость класса языков $\mathbb{A}$ относительно объединения, произведения и итерации доказана. Отметим, что автоматы $\mathcal{B}_3$, $\mathcal{B}_4$ и $\mathcal{B}_5$ также являются нормальными ε-НКА.

Теперь заметим, что если язык L распознается детерминированным конечным автоматом $\mathcal{A} = (Q, \Sigma, \delta, \mathtt{q}_0, F)$, то его дополнение $\bar{L}$ распознается ДКА $\bar{\mathcal{A}} = (Q, \Sigma, \delta, \mathtt{q}_0, Q \backslash F)$. Замкнутость относительно пересечения следует из замкнутости относительно объединения и дополнения. Теорема доказана. □

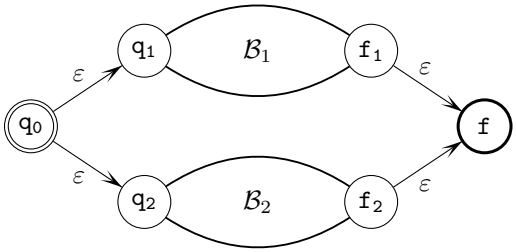


Рис. 2.9. *«Параллельное соединение» нормальных ε-НКА*

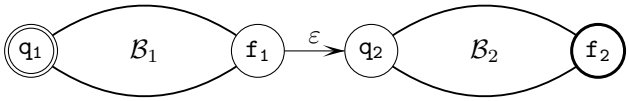


Рис. 2.10. *«Последовательное соединение» нормальных ε-НКА*

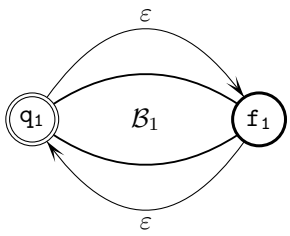


Рис. 2.11. *Итерация нормального ε-НКА*

Предложение 2.1. *Любой конечный язык распознаваем конечным автоматом.*

Доказательство. Автоматы, распознающие языки $\emptyset$, $\{\varepsilon\}$ и $\{\mathtt{a}\}$, приведены на рис. 2.12, *а–в*.

а *б* *в*

Рис. 2.12. *Автоматы, распознающие простейшие языки*

Язык $\{w\}$, состоящий из одной цепочки, либо совпадает с $\{\varepsilon\}$, либо является конечным произведением языков вида $\{\mathtt{a}\}$. Любой конечный язык либо совпадает с $\emptyset$, либо является конечным объединением языков вида $\{w\}$. Таким образом, предложение следует из теоремы 2.4. □

§ 7. Регулярные языки. Теорема Клини

В этом параграфе доказывается основополагающий результат теории конечных автоматов, утверждающий совпадение класса «автоматных» языков $\mathbb{A}$ с классом языков, задаваемых при помощи *регулярных* операций – объединения, произведения и итерации.

Определение. Язык называется *регулярным*, если он может быть получен из конечных языков при помощи конечного числа операций объединения, произведения и итерации.

Нетрудно понять, что словосочетание «конечных языков» в определении регулярности можно заменить на «языков вида $\emptyset$, $\{\varepsilon\}$ и $\{\mathtt{a}\}$, где $\mathtt{a}$ – буква». Класс регулярных языков обозначим через $\mathbb{R}$.

Теорема 2.5 (Клини). *Класс регулярных языков совпадает с классом языков, распознаваемых конечными автоматами.*

Доказательство. Из предложения 2.1 и теоремы 2.4 немедленно следует, что всякий регулярный язык распознается некоторым конечным автоматом, т. е. $\mathbb{R} \subseteq \mathbb{A}$. Для доказательства обратного включения нужно установить, что всякий язык, распознаваемый конечным автоматом, является регулярным. В качестве класса автоматов возьмем неполные ДКА (см. замечание 2.1). В этом замечании установлена эквивалентность каждого неполного ДКА «обычному» ДКА. Поскольку «обычный» ДКА является частным случаем неполного, отсюда следует, что неполные ДКА распознают в точности класс языков $\mathbb{A}$. Доказательство проведем индукцией по числу переходов автомата $\mathcal{A} = (Q, \Sigma, \delta, \mathtt{q}_0, F)$, распознающего язык L.

База индукции. Если функция переходов нигде не определена, то автомат не в состоянии проработать ни одного такта; очевидно, что $L(\mathcal{A}) = \{\varepsilon\}$, если $\mathtt{q}_0 \in F$, и $L(\mathcal{A}) = \emptyset$ в противном случае. Предположим теперь, что в $\mathcal{A}$ определен единственный переход $\delta(\mathtt{q}, \mathtt{a}) = \mathtt{r}$ и разберем возможные варианты. Если $\mathtt{q} \neq \mathtt{q}_0$, то $\mathcal{A}$ снова не выполнит ни одного такта и мы получаем случай, разобранный выше. Для $\mathtt{q} = \mathtt{q}_0$ все возможные случаи сведены в табл. 2.5.

Таблица 2.5. *Возможные случаи для ДКА с единственным переходом*

$\mathtt{r} = \mathtt{q}_0$	$\mathtt{q} \in F$	$\mathtt{r} \in F$	$L(\mathcal{A})$
Нет	Нет	Нет	$\emptyset$
Нет	Нет	Да	$\{\mathtt{a}\}$
Нет	Да	Нет	$\{\varepsilon\}$
Нет	Да	Да	$\{\varepsilon, \mathtt{a}\}$
Да	Нет	Нет	$\emptyset$
Да	Да	Да	$\{\mathtt{a}\}^*$

Языки $\emptyset$, $\{\mathtt{a}\}$, $\{\varepsilon\}$, $\{\varepsilon, \mathtt{a}\}$ и $\{\mathtt{a}\}^*$ являются регулярными. База индукции рассмотрена.

Шаг индукции. Пусть автомат $\mathcal{A} = (Q, \Sigma, \delta, \mathtt{q}_0, F)$ имеет k переходов, $k > 1$. Зафиксируем переход $\delta(\mathtt{q}, \mathtt{a}) = \mathtt{r}$. Через δ' обозначим функцию переходов на $Q \times \Sigma$, значение которой не определено на паре $(\mathtt{q}, \mathtt{a})$ и совпадает со значением δ на всех остальных парах. Рассмотрим автоматы

$$\begin{aligned}
\mathcal{A}_0 &= (Q, \Sigma, \delta', \mathtt{q}_0, F),\\
\mathcal{A}_1 &= (Q, \Sigma, \delta', \mathtt{q}_0, \{\mathtt{q}\}),\\
\mathcal{A}_2 &= (Q, \Sigma, \delta', \mathtt{r}, \{\mathtt{q}\}),\\
\mathcal{A}_3 &= (Q, \Sigma, \delta', \mathtt{r}, F).
\end{aligned}$$

Данные автоматы имеют $(k-1)$ переходов, откуда языки $L(\mathcal{A}_0)$, $L(\mathcal{A}_1)$, $L(\mathcal{A}_2)$ и $L(\mathcal{A}_3)$ регулярны по предположению индукции. Доказав, что

$$L = L(\mathcal{A}) = L(\mathcal{A}_0) \cup L(\mathcal{A}_1)\mathtt{a}[L(\mathcal{A}_2)\mathtt{a}]^* L(\mathcal{A}_3), \tag{2.2}$$

мы покажем регулярность языка L и завершим доказательство теоремы.

Возьмем цепочку w из L. Если автомат, проработав на w, ни разу не воспользовался переходом $\delta(\mathtt{q}, \mathtt{a}) = \mathtt{r}$, то $w \in L(\mathcal{A}_0)$. Если же этот переход использовался, то цепочку можно представить в виде $w = w_0\mathtt{a}w_1\mathtt{a}\ldots w_{n-1}\mathtt{a}w_n$ $(n \geq 1)$, где все указанные буквы $\mathtt{a}$ соответствуют переходам из $\mathtt{q}$ в $\mathtt{r}$ (рис. 2.13).

Получаем $w_0 \in L(\mathcal{A}_1)$, $w_1, \ldots, w_{n-1} \in L(\mathcal{A}_2)$, $w_n \in L(\mathcal{A}_3)$, откуда $w \in L(\mathcal{A}_1)\mathtt{a}[L(\mathcal{A}_2)\mathtt{a}]^* L(\mathcal{A}_3)$. Тем самым мы показали включение слева направо в формуле (2.2).

Обратное включение практически очевидно. Так, $L(\mathcal{A}_0) \subseteq L$ по определению, а если $w \in L(\mathcal{A}_1)\mathtt{a}[L(\mathcal{A}_2)\mathtt{a}]^* L(\mathcal{A}_3)$, то w можно записать в виде $w_0\mathtt{a}w_1\mathtt{a}\ldots w_{n-1}\mathtt{a}w_n$, где $n \geq 1$, $w_0 \in L(\mathcal{A}_1)$,

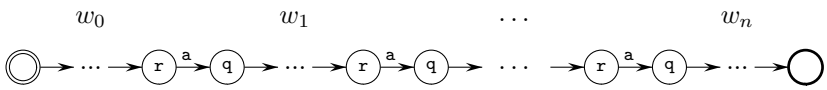


Рис. 2.13. *Шаг индукции*

$w_1, \ldots, w_{n-1} \in L(\mathcal{A}_2)$, $w_n \in L(\mathcal{A}_3)$, откуда $w \in L$ (рис. 2.13). Итак, равенство (2.2) доказано. □

Итак, класс языков, распознаваемых конечными автоматами, совпадает с классом регулярных языков. Регулярные языки обычно задаются *регулярными выражениями*.

Определение. Пусть Σ – произвольный алфавит.
1. Символы $\emptyset$ и ε, а также буквы из Σ являются регулярными выражениями над Σ.
2. Выражения $r|s$, $(r)\cdot(s)$, $(r)^*$, где r и s – регулярные выражения над Σ, являются регулярными выражениями над Σ.
3. Других регулярных выражений над Σ нет.

Ответ на вопрос о том, в каком смысле регулярное выражение *определяет* язык, в общем-то, ясен. Тем не менее дадим уточнение. Регулярные выражения $\emptyset$, ε и $\mathtt{x}$, где $\mathtt{x} \in \Sigma$, определяют соответственно языки $\emptyset$, $\{\varepsilon\}$ и $\{\mathtt{x}\}$. Если R и S – языки, определяемые регулярными выражениями r и s, то выражения $r|s$, $(r)\cdot(s)$ и $(r)^*$ определяют соответственно языки $R \cup S$, RS и R^*. Знак «|», общеупотребительный в значении «или», когда речь идет о перечислении альтернатив, в нашем случае естественно соответствует объединению языков. Язык, определяемый регулярным выражением r, будем обозначать через $L(r)$. Таким образом, класс $\mathbb{R}$ можно рассматривать как класс языков, определяемых регулярными выражениями.

Чтобы уменьшить число скобок в записи регулярных выражений, условимся о приоритете операций над языками. Считается, что итерация имеет наивысший приоритет, а объединение – самый низкий. Кроме того, можно пользоваться тем, что объединение и умножение – ассоциативные, а итерация – идемпотентная операция (т. е. $(L^*)^* = L^*$ для любого языка L). Символы $\emptyset$, ε и буквы не будем заключать в скобки, а знак операции умножения (т. е. точку) чаще всего будем опускать. Эти соглашения позволяют, например, регулярное выражение $(((\mathtt{a}) \cup (\mathtt{b})) \cup ((\mathtt{b}) \cdot (\mathtt{c})))^*$ записать как $(\mathtt{a} \cup \mathtt{b} \cup \mathtt{bc})^*$.

Приведем пример. Пусть $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $r = \mathtt{aa}^*\mathtt{b}^2 \cup \mathtt{bb}^*\mathtt{a}$, $s = (\mathtt{a} \cup \mathtt{b})^*\mathtt{a}$. Тогда $L(r) = \{\mathtt{a}^k\mathtt{b}^2 \mid k \in \mathbb{N}\} \cup \{\mathtt{b}^l\mathtt{a} \mid l \in \mathbb{N}\}$, $L(s) = \{w\mathtt{a} \mid w \in \Sigma^*\}$.

В завершение параграфа обсудим вопрос о том, как по регулярному выражению r построить конечный автомат, распознающий язык $L(r)$. Удобнее всего строить нормальный ε-НКА. Для этого достаточно воспользоваться результатами предложения 2.1 и теоремы 2.4 (см. рис. 2.9–2.12). Чтобы получить ДКА, можно затем использовать алгоритм 2.3.

Пример 2.5. На рис. 2.14 изображен нормальный ε-НКА, распознающий язык, заданный регулярным выражением $r = \mathtt{a}(\mathtt{a} \cup \mathtt{b})^*\mathtt{ab}$. Автомат построен в соответствии с изложенным выше способом.

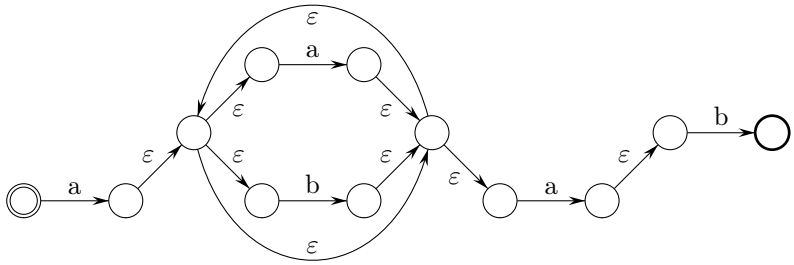


Рис. 2.14. *Нормальный ε-НКА, распознающий язык* $L(\mathtt{a}(\mathtt{a} \cup \mathtt{b})^*\mathtt{ab})$

§ 8. Фрагменты языков программирования, распознаваемые конечными автоматами

Практически в каждом языке есть понятие *имени* или *идентификатора*. С синтаксической точки зрения имя чаще всего представляет собой последовательность букв и цифр, начинающуюся с буквы[†], т. е. множество имен задается регулярным выражением

$$\texttt{<имя>} = (\mathtt{a}|\mathtt{b}|\ldots|\mathtt{z})(\mathtt{a}|\mathtt{b}|\ldots|\mathtt{z}|\mathtt{0}|\mathtt{1}|\ldots|\mathtt{9})^*$$

и поэтому распознается конечным автоматом (для простоты мы полагаем, что алфавит языка содержит только строчные латинские буквы). Если ввести предварительно регулярные выражения

$$\texttt{<буква>} = \mathtt{a}|\mathtt{b}|\ldots|\mathtt{z} \quad \text{и} \quad \texttt{<цифра>} = 0|1|\ldots|9\ ,$$

[†] «Альтернативный» синтаксис имени обычно такой: именем считается непустая цепочка, начинающаяся с предписанного символа, например символа $.

то регулярное выражение для множества имен будет выглядеть как

$$\texttt{<имя>} = \texttt{<буква>}(\texttt{<буква>} \mid \texttt{<цифра>})^*$$

Также просто усмотреть, что множество целых чисел без знака распознается конечным автоматом, поскольку задается регулярным выражением[†]

$$\texttt{<целое\_без\_знака>} = \texttt{<цифра>}(\texttt{<цифра>})^*$$

Чтобы определить регулярным выражением число без знака, надо предусмотреть (возможную) дробную часть и (возможный) порядок. Соответствующее регулярное выражение будет иметь вид

$$\texttt{<число\_без\_знака>} = \texttt{<целое\_без\_знака>}(.\texttt{<целое\_без\_знака>} \mid \varepsilon)\cdot$$
$$\cdot(E(+|-|\varepsilon)\texttt{<целое\_без\_знака>} \mid \varepsilon)$$

Регулярное выражение для *констант* в языке Паскаль имеет следующий вид:

$$\texttt{<константа>} =$$
$$= ((+|-|\varepsilon)\texttt{<число\_без\_знака>}|\texttt{<имя>}) \mid \texttt{'}(\texttt{<буква>} \mid \texttt{<цифра>})^*\texttt{'}$$

Отметим еще понятие *простого типа* из того же языка Паскаль. Оно задается регулярным выражением

$$\texttt{<простой\_тип>} =$$
$$= \texttt{<имя>} \mid [\texttt{<имя>}(,\texttt{<имя>})^*] \mid \texttt{<константа>..<константа>}$$

В этом выражении пришлось круглые скобки заменить на квадратные, так как круглые используются для обозначения порядка операций. Если убрать рекурсию из определения понятия «список полей», то можно написать регулярное выражение для понятия «тип».

Приведенными примерами (и близкими к ним) фактически исчерпываются фрагменты языков программирования, распознаваемые конечными автоматами. Более сложные конструкции, например, такие, как выражения, уже не могут быть заданы конечными автоматами. Это будет показано в начале следующего параграфа.

Замечание 2.8. Было бы в высшей степени ошибочно сделать на основании последнего абзаца вывод о незначительной роли конечных автоматов в программировании. Их непосредственная роль в процессе компиляции программ (см. ч. 2, гл. 1) действительно невелика, но они играют очень важную вспомогательную роль при построении компилятора (ч. 2, гл. 4). К тому же «центр» применения автоматов в программировании совсем не здесь – они давно применя-

[†]Мы немного упростили ситуацию. На самом деле такое `<целое_без_знака>` используется ниже в дробной части и экспоненте числа, а для целой части числа нужно дополнительно учесть, что она не начинается с 0, если не равна 0.

ются при спецификации, отладке, тестировании и документировании поведения программ (например, при помощи автоматов удобно описывать «многооконный» режим работы). В последнее время активно развиваются подходы к программированию, основанные на понятии «состояния» вычислительного процесса; конечные преобразователи («автоматы с выходом») играют в этих подходах ключевую роль.

§ 9. Автоматы с магазинной памятью

Изученные выше конечные автоматы обладают довольно слабыми распознающими возможностями. Продемонстрируем это на следующем классическом примере.

Предложение 2.2. *Скобочный язык LB не является регулярным.*

Доказательство. Предположим противное: пусть LB распознается конечным автоматом $\mathcal{A}$. Количество состояний автомата $\mathcal{A}$ обозначим через n. Поскольку $\mathcal{A}$ распознает все правильные расстановки скобок, он распознает расстановку

$$w = \underbrace{[\ [\ \ldots\ [}_{n\text{ скобок}}\ \underbrace{]\ ]\ \ldots\ ]}_{n\text{ скобок}}\ .$$

Автомат начинает чтение цепочки w в начальном состоянии $\mathtt{q}_0$ и в процессе чтения n открывающих скобок последовательно попадает в n состояний, которые мы обозначим через $\mathtt{q}_1, \ldots, \mathtt{q}_n$. Поскольку в автомате только n состояний, то найдутся такие числа i и j, что $0 \le i < j \le n$ и $\mathtt{q}_i = \mathtt{q}_j$. С учетом этого замечания путь в автомате, помеченный цепочкой w, можно представить как на рис. 2.15. Но тогда $\mathcal{A}$, очевидно, распознает некорректную расстановку скобок

$$w' = \underbrace{[\ [\ \ldots\ [}_{n-j+i\text{ скобок}}\ \underbrace{]\ ]\ \ldots\ ]}_{n\text{ скобок}},$$

так как путь, помеченный этой цепочкой, получается из пути на рис. 2.15 удалением цикла у вершины $\mathtt{q}_i = \mathtt{q}_j$. Тем самым $\mathcal{A}$ не распознает LB. Полученное противоречие завершает доказательство. □

Из этого предложения следует, что язык арифметических выражений LA, как и любой другой язык выражений со скобками, не является регулярным, поскольку никакой конечный автомат не в состоянии распознавать без ошибок правильные расстановки скобок.

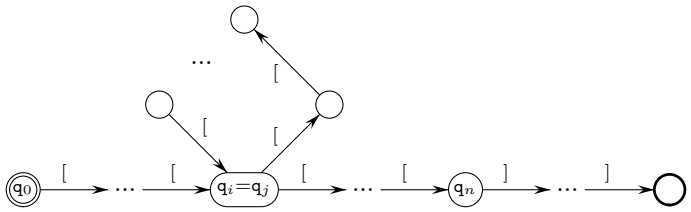


Рис.2.15. *Путь, помеченный цепочкой w*

Замечание 2.9. Именно скобки представляют основную сложность распознавания выражений. Если из описания языка LA убрать скобки, то получившийся язык распознается конечным автоматом с двумя состояниями. Проверку этого простого факта мы оставляем читателю.

Язык арифметических выражений с точки зрения сложности его цепочек является очень простым по сравнению, скажем, с языками программирования. Это означает, что для проверки синтаксической корректности программ необходим более сильный распознаватель, нежели конечный автомат. В качестве такого распознавателя обычно используется *автомат с магазинной памятью* (сокращенно МП-автомат, МПА; в оригинале – push-down automaton, PDA). Понятие МП-автомата является одним из центральных в данной книге. Мы начнем его обсуждение с модели МПА как физического устройства (рис. 2.16).

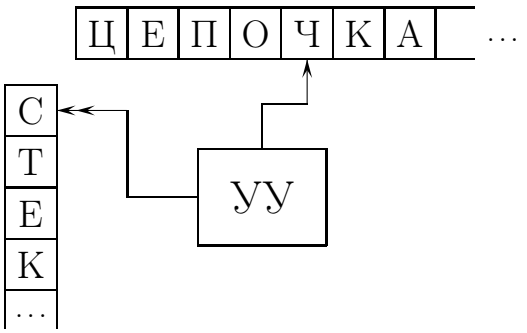


Рис.2.16. *Схема устройства МП-автомата. Для цепочки допускается только чтение, а для стека – чтение и запись*

Как и конечный автомат, МПА содержит устройство управления, которое в каждый момент времени находится в некотором состоянии (из конечного набора), и входную ленту, предназначенную для посимвольного чтения слева направо входной цепочки. Но в отличие от КА добавляется еще одна лента – лента памяти (иногда называемая *магазином*). Как и входная лента, лента памяти потенциально неограничена в одну сторону, но предназначена как для считывания, так и для записи. Доступ к ней организован по принципу стека: «последний вошел – первый вышел», все операции могут производиться только с текущей вершиной стека. В дальнейшем мы всегда говорим об этой ленте как о *стеке автомата*. Алфавит, используемый для работы со стеком, в общем случае не совпадает со входным алфавитом.

Опишем функционирование МП-автомата. Как и конечный автомат, МПА работает тактами. В ходе каждого такта устройство управления анализирует

свое текущее состояние $\mathtt{q}$,
обозреваемый входной символ $\mathtt{a}$,
символ B на вершине стека

и принимает решение о том,

в какое состояние $\mathtt{q}'$ перейти,
какую цепочку γ записать в стек вместо символа B,
следует ли сдвинуть указатель вправо по входной ленте ($^{\rightarrow}$)
или оставить его на месте ($\_$).

Данный процесс мы будем рассматривать как выполнение автоматом *команды*

$$(\mathtt{q}, \mathtt{a}, B) \to (\mathtt{q}', \gamma, ^{\rightarrow}) \quad \text{или} \quad (\mathtt{q}, \mathtt{a}, B) \to (\mathtt{q}', \gamma, \_) \tag{2.3}$$

соответственно.

Автомат *допускает* цепочку, записанную на входной ленте, если он дошел до ее конца и оказался при этом в одном из заключительных состояний. «Недопуск» цепочки происходит в одном из двух случаев: если автомат «дочитал» цепочку, но не пришел в заключительное состояние, либо если он не «дочитал» ее до конца (способность не сдвигаться вправо по входной ленте означает возможность зацикливания автомата, даже если считать, что автомат имеет команду для любой возможной тройки $(\mathtt{q}, \mathtt{a}, B)$).

Теперь дадим формальное

Определение. Автоматом с магазинной памятью (МП-автоматом) называется семерка $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, \mathtt{q}_0, F, \gamma_0)$, где Q – конечное множество состояний, Σ – конечный *входной* алфавит, Γ – конечный *стековый* алфавит, $\mathtt{q}_0 \in Q$ – начальное состояние, $F \subseteq Q$ – множество заключительных состояний, $\gamma_0 \in \Gamma^*$ – начальное содержимое стека, а δ – конечное множество команд вида (2.3).

Для удобства формализации обработки цепочек МП-автоматом принято считать, что входной и стековый алфавит содержат по специальному символу-ограничителю. Символ $\dashv$ – «символ конца строки» – размещается на входной ленте непосредственно справа от входной цепочки (и не встречается внутри самой цепочки). Считается, что указатель на входной ленте видит этот символ тогда и только тогда, когда прочтение входной цепочки завершено. Символ ∇ – «символ дна стека» – размещается под нижним символом в стеке (и более нигде не встречается). Когда стек пуст, его «верхним символом» является как раз ∇. На таком такте при записи в стек символ дна остается на своем месте, т. е. при выполнении команды с левой частью $(\mathtt{q}, \mathtt{a}, \nabla)$ цепочка символов γ будет записана в стек «поверх» символа ∇.

Определение. Пусть МП-автомат $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, \mathtt{q}_0, F, \gamma_0)$ зафиксирован. *Конфигурацией* $\mathcal{M}$ называется тройка $[\mathtt{q}, w, \gamma]$, где $\mathtt{q} \in Q$ – текущее состояние автомата, $w \in \Sigma^*$ – суффикс входной цепочки, начинающийся в позиции указателя, $\gamma \in \Gamma^*$ – содержимое стека (начиная с верхнего символа, не считая символа дна). Тот факт, что после выполнения команды в конфигурации $[\mathtt{q}, w, \gamma]$ автомат перешел в конфигурацию $[\mathtt{q}', w', \gamma']$, мы записываем в виде $[\mathtt{q}, w, \gamma] \models [\mathtt{q}', w', \gamma']$ и говорим, что вторая конфигурация *следует* за первой. Запись $[\mathtt{q}, w, \gamma] \models^* [\mathtt{q}', w', \gamma']$ означает, что автомат может перейти из первой конфигурации во вторую за конечное число (в том числе нуль) шагов.

Пример 2.6. Автомат, находящийся в конфигурации $[\mathtt{q}, \mathtt{a}w, B\gamma]$, на очередном такте может выполнить только команду с левой частью $(\mathtt{q}, \mathtt{a}, B)$. Если такая команда имеет вид $(\mathtt{q}, \mathtt{a}, B) \to (\mathtt{q}', \gamma', \rightarrow)$, то имеем $[\mathtt{q}, \mathtt{a}w, B\gamma] \models [\mathtt{q}', w, \gamma'\gamma]$; если же правая часть команды равна $(\mathtt{q}', \gamma', \_)$, то получим $[\mathtt{q}, \mathtt{a}w, B\gamma] \models [\mathtt{q}', \mathtt{a}w, \gamma'\gamma]$.

Автомат, находящийся в конфигурации $[\mathtt{q}, \mathtt{a}w, \varepsilon]$, может выполнить только команду с левой частью $(\mathtt{q}, \mathtt{a}, \nabla)$. При этом следующая конфигурация будет иметь вид $[\mathtt{q}', w, \gamma']$ или $[\mathtt{q}', \mathtt{a}w, \gamma']$.

Определение. МП-автомат $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, \mathtt{q}_0, F, \gamma_0)$ *распознает* (или *допускает*) цепочку $w \in \Sigma^*$, если $[\mathtt{q}_0, w, \gamma_0] \models^* [\mathtt{f}, \varepsilon, \gamma]$ для некоторых $\mathtt{f} \in F$, $\gamma \in \Gamma^*$. Язык, состоящий из всех цепочек, распознаваемых $\mathcal{M}$, называется языком, *распознаваемым* этим автоматом, и обозначается $L(\mathcal{M})$. Два МП-автомата *эквивалентны*, если они распознают один и тот же язык.

Из приведенного выше описания МПА видно, что можно рассматривать как детерминированную, так и недетерминированную версию такого автомата. Детерминированный МПА (ДМПА) в любой конфигурации имеет единственную возможность для продолжения работы: либо он переходит в строго определенную следующую конфигурацию, либо – в отсутствие такой конфигурации – останавливается. Следовательно, такой автомат имеет не более одной команды для каждой тройки $(\mathtt{q}, \mathtt{a}, B)$. Для некоторых троек команды могут отсутствовать[†], т. е. автомат является *неполным* – ср. замечание 2.1 о неполных ДКА. Множество команд ДМПА можно представлять в более привычном виде (частичной) функции переходов

$$\delta : Q \times \Sigma \times \Gamma \longrightarrow Q \times \Gamma^* \times \{\rightarrow, \_\}.$$

Недетерминированный МП-автомат (НМПА) может иметь несколько команд с одинаковой левой частью; тем самым для некоторых его конфигураций следующая конфигурация определяется не единственным образом. Данное нами формальное определение МПА не накладывает специальных ограничений на систему команд[‡], а следовательно задает именно НМПА. Отметим, что систему команд НМПА можно рассматривать как функцию переходов

$$\delta : Q \times \Sigma \times \Gamma \longrightarrow (2^{Q \times \Gamma^* \times \{\rightarrow, \_\}})_{fin},$$

где индекс $_{fin}$ указывает на рассмотрение только конечных подмножеств.

Что касается распознающих возможностей детерминированных и недетерминированных МПА, то ситуация здесь резко отличается

[†]Отсутствие команды равносильно (с точки зрения распознавания цепочек) наличию команды «ничего не делать», имеющей вид $(\mathtt{q}, \mathtt{a}, B) \rightarrow (\mathtt{q}, B, \_)$. Далее мы предполагаем без ограничения общности, что команд «ничего не делать» в автомате нет.

[‡]Кроме требования конечности δ. Это требование автоматически выполняется для детерминированного автомата, поскольку число команд не превосходит мощности конечного множества $Q \times \Sigma \times \Gamma$, но должно быть оговорено явно для недетерминированного автомата ввиду бесконечности множества Γ^*.

от внешне похожей ситуации с конечными автоматами. Класс языков, распознаваемых НМПА, как мы увидим в дальнейшем, строго больше класса языков, распознаваемых ДМПА. Более того, класс языков, распознаваемых НМПА, совпадает с одним из самых известных и важных классов в теории формальных языков. Этот класс допускает простую и эффективную характеризацию, в то время как его подкласс, состоящий из языков, распознаваемых ДМПА, охарактеризовать существенно сложнее. Вообще, для решения теоретических вопросов удобно рассматривать именно НМПА. Наоборот, с точки зрения практической применимости НМПА почти бесполезны, а ДМПА очень важны. Связано это с отсутствием эффективного алгоритма для моделирования работы первых и наличием такого (очевидного) алгоритма для моделирования работы вторых.

Для моделирования работы ДМПА будем хранить множество команд в трехмерной *управляющей таблице*, проиндексированной множествами Q, Σ и Γ: тройка индексов $(\mathtt{q}, \mathtt{a}, B)$ рассматривается как левая часть команды, а в клетке с этими индексами находится ее правая часть. Все операции по переходу из текущей конфигурации автомата в следующую очевидно требуют константного времени. Эффективному моделированию НМПА препятствует тот факт, что «текущих» конфигураций НМПА может быть сколь угодно много[†]. Поскольку в данной книге нас интересует в первую очередь практическое применение автоматов, мы в дальнейшем будем почти всегда рассматривать ДМПА и термин «МП-автомат» использовать в значении «ДМПА», если иное не оговорено явно.

Замечание 2.10. Для лучшей читаемости управляющей таблицы мы договоримся не писать в клетке новое состояние, если оно совпадает с текущим, а также опускать знак _ (отсутствие сдвига по входной ленте).

Пример 2.7. Построим МП-автомат, который распознает скобочный язык LB. Имеем $\Sigma = \{\,[,]\,\}$. Нам потребуются два состояния, начальное q и заключительное f и единственный стековый символ [, при помощи которого мы будем запоминать незакрытые левые скобки. Автомат начинает работу с пустым стеком. Управляющая таблица приведена в табл. 2.6.

Легко видеть, что в любой конфигурации автомата стек содер-

[†]В отличие от НМПА, НКА можно моделировать эффективно, поскольку для конечного автомата количество «текущих конфигураций» ограничено числом состояний.

Таблица 2.6. *Управляющая таблица МП-автомата для языка LB*

		[	]	⊣
q :	[	[[, →	ε, →	
	∇	[, →		f, ε
f :	[			
	∇			

жит в точности все незакрытые левые скобки в прочитанной части цепочки. Цепочка распознается автоматом тогда и только тогда, когда в момент обнаружения ограничителя ⊣ стек пуст, т. е. все скобки являются «парными». При обнаружении лишней правой скобки автомат останавливается, не прочитав цепочку до конца, так как клетка $(\mathtt{q},], \nabla)$ пуста. Если по завершении чтения цепочки останутся незакрытые левые скобки, то автомат останется в незаключительном состоянии q.

В приведенном примере состояние f с «содержательной» точки зрения не нужно вообще – соответствующая часть таблицы пуста, и вместо перехода в состояние f в клетке $(\mathtt{q}, \dashv, \nabla)$ достаточно было бы поместить указание «допустить цепочку». Такая ситуация, как мы увидим, является общей (хотя далеко не для каждого автомата это очевидно). В связи с этим представляется целесообразным использование в МП-автоматах специальной *команды «допустить цепочку»*, которую мы будем записывать в виде $(\mathtt{q}, \mathtt{a}, B) \to \checkmark$.

Замечание 2.11. Добавление команды допуска к множеству возможных команд не расширяет распознающие возможности МП-автоматов, т. е. любой МП-автомат с командами допуска эквивалентен МП-автомату без таких команд. В самом деле, можно добавить к исходному автомату еще одно заключительное состояние, в котором независимо от символов на входе и в стеке выполняется только операция сдвига по входной цепочке. Заменив все команды допуска на команды перехода в новое состояние, мы получим эквивалентный автомат без команд допуска.

При конструировании МП-автоматов в данной книге мы будем постоянно пользоваться этой дополнительной командой. Как мы увидим в дальнейшем, использование команды допуска позволяет всегда в случае НМПА, и достаточно часто в случае ДМПА,

обойтись автоматами с одним состоянием, тем самым делая моделирование МП-автоматов более эффективным как по используемой памяти, так и по времени работы. Рассмотрим еще один пример.

Пример 2.8. МП-автомат, задаваемый таблицей 2.7, начинает работу с пустым стеком и распознает язык $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$. Состояние q является начальным и единственным заключительным.

Таблица 2.7. *Управляющая таблица МП-автомата для языка* $\{a^n b^n | n{\geq}0\}$

		a	b	⊣
q :	a	aa, $\rightarrow$	r, ε, $\rightarrow$	
	▽	a, $\rightarrow$		
r :	a		ε, $\rightarrow$	
	▽			q, ε

В отличие от предыдущего примера оба состояния этого автомата играют нетривиальную роль: переход в r означает, что символы a больше появляться в цепочке не должны. Тем не менее, введя один дополнительный стековый символ, мы можем модифицировать автомат, обойдясь единственным состоянием (и единственным применением специальной команды «допустить цепочку»). Результат модификации показан в табл. 2.8. Поскольку состояние автомата единственно, оно нигде в таблице не фигурирует. В начале работы в стеке находится символ X. Он снимается со стека, когда найдена первая буква b; после этого автомат не может прочитать больше ни одной буквы a.

Таблица 2.8. *Модификация управляющей таблицы МП-автомата для языка* $\{a^n b^n \mid n \geq 0\}$

	a	b	⊣
a		ε, $\rightarrow$	
X	Xa, $\rightarrow$	ε	ε
▽			✓

В заключение параграфа покажем, что НМПА распознаю́т более широкий класс языков, чем ДМПА. Для произвольной цепочки $w = \mathtt{a}_1 \ldots \mathtt{a}_n$ через $\overleftarrow{w}$ обозначим цепочку $\mathtt{a}_n \ldots \mathtt{a}_1$.

Предложение 2.3. *Если алфавит Σ содержит более одной буквы, то язык $\{w\overleftarrow{w} \mid w \in \Sigma^*\}$ распознается недетерминированным МП-автоматом, но не распознается детерминированным МП-автоматом.*

Доказательство. Вначале построим НМПА, распознающий указанный язык. Возьмем произвольный символ $X \notin \Sigma$. Множество $\Sigma \cup \{X\}$ будет стековым алфавитом, а X – начальным содержимым стека. Автомат имеет единственное состояние, а его управляющая таблица схематично представлена в табл. 2.9. Здесь $\mathtt{x}$ означает произвольный элемент из Σ, $\mathtt{y}$ – произвольный элемент из Σ, отличный от $\mathtt{x}$. Автомат является недетерминированным, так как паре $(\mathtt{x}, X)$ соответствуют две различные команды.

Таблица 2.9. *Управляющая таблица НМПА для языка* $\{w\overleftarrow{w} \mid w \in \Sigma^*\}$

	$\mathtt{x}$	$\dashv$
X	$X\mathtt{x}, \overrightarrow{\ }$ ε	ε
$\mathtt{x}$	$\varepsilon, \overrightarrow{\ }$	
$\mathtt{y}$		
∇		✓

Обработка цепочки построенным автоматом, очевидно, происходит в два этапа – накопления символов в стеке (пока верхушкой стека является символ X) и удаления символов из стека (после того как X снят со стека). Если к тому моменту, когда автомат выбрал команду $(\mathtt{x}, X) \to \varepsilon$, он обработал префикс u входной цепочки, то содержимое стека в этот момент равно, очевидно, $\overleftarrow{u}$. На второй стадии автомат может выполнить очередной такт тогда и только тогда, когда очередной символ входа совпадает с текущим верхним символом в стеке (клетка $(\mathtt{x}, \mathtt{x})$ содержит команду, а клетка $(\mathtt{x}, \mathtt{y})$ пуста). Следовательно, цепочка будет допущена в том и только в том случае, когда ее часть, оставшаяся необработанной к моменту снятия X со стека, равна $\overleftarrow{u}$. Мы показали, что построенный автомат распознает язык $\{w\overleftarrow{w} \mid w \in \Sigma^*\}$. Отметим, что недетерминизм в данном случае является существенным – автомат «угадывает» местонахождение середины цепочки, длина которой неизвестна.

Теперь покажем, что никакой ДМПА не может распознать данный язык. В самом деле, поскольку алфавит содержит более од-

ной буквы, МПА, распознающий все цепочки вида $w\overleftarrow{w}$ и только их, должен побуквенно сравнивать вторую половину цепочки с первой. Значит, при обработке первой половины цепочки вся информация о прочитанных символах накапливается в стеке, а при обработке второй половины – извлекается из него для сравнения. Пусть некоторый ДМПА распознает цепочку $w\mathtt{aa}\overleftarrow{w}$ и после обработки ее префикса $w\mathtt{a}$ находится в состоянии `q` с символом B на вершине стека. Тогда у этого ДМПА есть команда с левой частью $(\mathtt{q}, \mathtt{a}, B)$, указывающая на переход к сравнению обработанной и необработанной частей цепочки. Поскольку команда с такой левой частью единственна, автомат не сможет распознать цепочку $w\mathtt{aaaa}\overleftarrow{w}$, принадлежащую рассматриваемому языку, так как приступит к сравнению уже после прочтения префикса $w\mathtt{a}$. Доказательство теоремы завершено. □

§ 10. Фрагменты языков программирования, распознаваемые МП-автоматами

При помощи МП-автоматов можно распознать бóльшую часть конструкций языков программирования. Подробно это обсуждается во второй части книги. Здесь мы ограничимся двумя примерами.

Первый из примеров – это МП-автомат $\mathcal{A}_1$, распознающий язык списков LL. Автомат имеет единственное состояние; его управляющая таблица приведена ниже (табл. 2.10). В начале работы стек содержит единственный символ L.

Таблица 2.10. *Управляющая таблица МП-автомата* $\mathcal{A}_1$

	a	;	[	]	⊣
L	$M,\rightarrow$		$L]M,\rightarrow$		
M		$LM,\rightarrow$		ε	ε
]				$\varepsilon,\rightarrow$	
∇					✓

Проиллюстрируем работу МП-автомата $\mathcal{A}_1$ по распознаванию цепочки `a;[a;a;a]` (табл. 2.11). Символ $\diamond$ отделяет обработанную часть входной цепочки от необработанной.

Из табл. 2.11 видно, что автомат распознает цепочку `a;[a;a;a]`. Доказать, что язык, распознаваемый автоматом $\mathcal{A}_1$, равен LL, имея

Таблица 2.11. *Протокол обработки цепочки* `a;[a;a;a]` *автоматом* $\mathcal{A}_1$

Такт	Позиция указателя	Содержимое стека
1	◇a;[a;a;a]⊣	$L\nabla$
2	a◇;[a;a;a]⊣	$M\nabla$
3	a;◇[a;a;a]⊣	$L\nabla$
4	a;[◇a;a;a]⊣	$L]M\nabla$
5	a;[a◇;a;a]⊣	$M]M\nabla$
6	a;[a;◇a;a]⊣	$L]M\nabla$
7	a;[a;a◇;a]⊣	$M]M\nabla$
8	a;[a;a;◇a]⊣	$L]M\nabla$
9	a;[a;a;a◇]⊣	$M]M\nabla$
10	a;[a;a;a◇]⊣	$]M\nabla$
11	a;[a;a;a]◇⊣	$M\nabla$
12	a;[a;a;a]◇⊣	∇

только управляющую таблицу автомата, довольно трудно. Это утверждение будет следовать из результатов ч. 2, гл. 2 этой книги. Там же будет указан алгоритм построения управляющей таблицы.

Второй пример – МП-автомат $\mathcal{A}_2$, распознающий язык арифметических выражений LA. Этот автомат также имеет единственное состояние, а его управляющая таблица приведена в табл. 2.12. В начале работы стек содержит единственный символ E.

Таблица 2.12. *Управляющая таблица МП-автомата* $\mathcal{A}_2$

	x	+	*	(	)	⊣
E	TE'			TE'		
E'		$TE',\rightarrow$			ε	ε
T	FT'			FT'		
T'		ε	$FT',\rightarrow$		ε	ε
F	$\varepsilon,\rightarrow$			$E),\rightarrow$		
)					$\varepsilon,\rightarrow$	
∇						✓

Если обработать при помощи автомата $\mathcal{A}_2$ цепочку `(x+x)*x` (табл. 2.13), то можно убедиться, что она распознаваема. Как и

в предыдущем примере, равенство $L(\mathcal{A}_2) = LA$ затруднительно доказать непосредственно; мы покажем этот факт в ч. 2, гл. 2.

Таблица 2.13. *Протокол обработки цепочки* `(x+x)*x` *автоматом* $\mathcal{A}_2$

Такт	Позиция указателя	Содержимое стека
1	◇`(x+x)*x`⊣	$E\nabla$
2	◇`(x+x)*x`⊣	$TE'\nabla$
3	◇`(x+x)*x`⊣	$FT'E'\nabla$
4	`(`◇`x+x)*x`⊣	$E)T'E'\nabla$
5	`(`◇`x+x)*x`⊣	$TE')T'E'\nabla$
6	`(`◇`x+x)*x`⊣	$FT'E')T'E'\nabla$
7	`(x`◇`+x)*x`⊣	$T'E')T'E'\nabla$
8	`(x`◇`+x)*x`⊣	$E')T'E'\nabla$
9	`(x+`◇`x)*x`⊣	$TE')T'E'\nabla$
10	`(x+`◇`x)*x`⊣	$FT'E')T'E'\nabla$
11	`(x+x`◇`)*x`⊣	$T'E')T'E'\nabla$
12	`(x+x`◇`)*x`⊣	$E')T'E'\nabla$
13	`(x+x`◇`)*x`⊣	$)T'E'\nabla$
14	`(x+x)`◇`*x`⊣	$T'E'\nabla$
15	`(x+x)*`◇`x`⊣	$FT'E'\nabla$
16	`(x+x)*x`◇⊣	$T'E'\nabla$
17	`(x+x)*x`◇⊣	$E'\nabla$
18	`(x+x)*x`◇⊣	∇

Задачи

1. Конечный автомат задан расширенной таблицей переходов. Начертить диаграмму переходов, задающую этот автомат (табл. 2.14).

Таблица 2.14. *Таблицы переходов (к задаче 1)*

	a	b	F
q_0	q_2	q_1	0
q_1	q_2	q_1	0
q_2	q_3	q_3	1
q_3	q_3	q_3	0

а

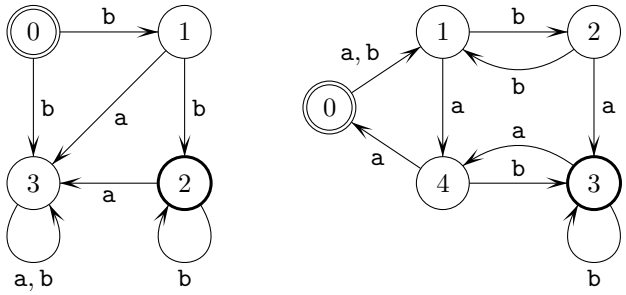
	a	b	F
q_0	q_0	q_3	0
q_1	q_3	q_1	0
q_2	q_0	q_3	1
q_3	q_2	q_3	0
q_4	q_4	q_4	0

б

	a	b	F
q_0	q_1	q_3	0
q_1	q_4	q_2	0
q_2	q_3	q_3	1
q_3	q_3	q_3	0
q_4	q_2	q_3	0

в

2. Конечный автомат задан диаграммой переходов. Составить таблицу переходов, задающую этот автомат (рис. 2.17).



а *б*

Рис. 2.17. *Диаграммы переходов (к задаче 2)*

3. Для каждого из автоматов, указанных в задачах 1 и 2, найти:
а) самую короткую цепочку, допускаемую автоматом;
б) две другие цепочки, допускаемые автоматом;
в) две цепочки длины 4, не допускаемые автоматом.

4. Построить приведенный автомат, эквивалентный заданному таблицей 2.15, *а–в*.

Таблица 2.15. *Таблицы переходов (к задаче 4)*

	a	b	F
0	0	2	0
1	6	3	1
2	5	4	0
3	0	3	1
4	0	3	0
5	6	5	1
6	6	2	0

а

	a	b	F
0	3	0	1
1	4	0	1
2	3	4	0
3	1	5	0
4	0	6	0
5	0	3	1
6	1	4	0

б

	a	b	F
0	5	2	0
1	6	2	0
2	0	4	0
3	3	5	0
4	6	2	1
5	3	0	1
6	3	1	1

в

5. Построить приведенный автомат, распознающий множество:
а) $\{(\mathtt{ab})^n \mid n \in \mathbb{N}\}$;
б) $\{\mathtt{ab}^2\mathtt{a}^n \mid n \in \mathbb{N}\}$;
в) $\{\mathtt{ab}^n \mid n \in \mathbb{N}\} \cup \{\mathtt{aba}^n \mid n \in \mathbb{N}\}$;

г) $\{\mathtt{and}, \mathtt{or}, \mathtt{not}, \mathtt{false}, \mathtt{true}\}$;
д) двоичных чисел с плавающей точкой;
е) двоичных слов четной длины;
ж) двоичных слов с четным числом и нулей, и единиц;
з) двоичных слов, не содержащих подслова 01;
и) двоичных слов, не содержащих подпоследовательности 101.

6. Для каждого из автоматов задач 1 и 2 задать регулярным выражением язык, распознаваемый этим автоматом.

7. Построить ДКА, эквивалентный данному НКА (табл. 2.16).

Таблица 2.16. *Таблицы переходов (к задаче 7)*

	a	b	F
0	1		0
1	0	0, 2	0
2			1

а

	a	b	F
0	0, 1		0
1		1, 2	0
2	1		1

б

8. Построить стандартный ε-НКА, распознающий язык, заданный регулярным выражением. По ε-НКА построить приведенный ДКА, используя алгоритм 2.3:

а) $(\mathtt{a}^2 \cup \mathtt{aba} \cup \mathtt{ba})^*$; б) $\mathtt{a}^*\mathtt{b} \cup \mathtt{bb}^*\mathtt{a}$;
в) $(\mathtt{a} \cup \mathtt{b})^* \cup ((\mathtt{ac})^2)^*$; г) $\mathtt{a}^*(\mathtt{a}^2\mathtt{b} \cup \mathtt{bb}^*\mathtt{a} \cup \mathtt{b}^2)^*$.

9*. Пусть L – регулярный язык. Проверить, обязательно ли регулярны следующие языки:

а) $init(L)$; б) $sub(L) = \{w \in \Sigma^* |\ \exists x, y \in \Sigma^* : xwy \in L\}$;
в) $L\backslash\mathtt{a}$; г) $u\backslash L = \{w \in \Sigma^* |\ uw \in L\}$;
д) $min(L)$; е) $rev(L) = \{\mathtt{a}_n \dots \mathtt{a}_1 \in \Sigma^* | \mathtt{a}_1 \dots \mathtt{a}_n \in L\}$.

10. Доказать, что следующие языки нерегулярны:

а) $\{\mathtt{a}^i\mathtt{b}\mathtt{a}^i |\ i \in \mathbb{N}\}$; б) $\{\mathtt{a}^i |\ i -$ простое число$\}$;
в) $\{\mathtt{a}^{2^i} |\ i \in \mathbb{N}\}$; г) $\{ww \,|\, w \in \Sigma^*\}$.

11*. Пусть L – регулярный язык и $S(L) = \{|w| \ | \ w \in L\}$. Доказать, что $S(L)$ – объединение конечного числа (возможно бесконечных) арифметических прогрессий.

12. Какие из цепочек $\mathtt{a}^3$, $\mathtt{ba}^2\mathtt{b}^2$, $\mathtt{b}^3\mathtt{ab}^2$, $\mathtt{b}^3\mathtt{a}^2\mathtt{b}^2\mathtt{a}$ допускаются МП-автоматом с управляющей таблицей, приведенной в табл. 2.17, *а*? Автомат начинает работу с пустым стеком.

Таблица 2.17. *Управляющие таблицы МП-автоматов для задач 12, 13*

	a	b	⊣
A	$AA, \rightarrow$		
B	$C, \rightarrow$	ε	
C	$CC, \rightarrow$	$\varepsilon^{\rightarrow}$	
∇	$A, \rightarrow$	$B, \rightarrow$	✓

а

	a	b	c	⊣
D	$DD, \rightarrow$		ε	
E		ε	$\varepsilon^{\rightarrow}$	
∇	$D, \rightarrow$	$E, \rightarrow$		✓

б

13. Управляющая таблица МП-автомата приведена в табл. 2.17, *б*. Найти цепочки, допускаемые этим автоматом:

а) цепочку длины, большей 4, не содержащую букву b;

б) цепочку длины, большей 5, с тремя переменами букв a и c;

в) цепочку длины, большей 3, содержащую букву b.

Для каждой из этих цепочек построить таблицы, иллюстрирующие работу автомата, аналогичные табл. 2.11. В начале работы стек пуст.

14. Построить МП-автоматы, допускающие следующие языки:

а) $\{\mathtt{a}^l\mathtt{b}^m | \, l > m > 0\}$;

б) $\{\mathtt{a}^l\mathtt{b}^m | \, m > l > 0\}$;

в) $\{\mathtt{a}^l\mathtt{b}^l\mathtt{a}^m\mathtt{b}^m | \, l, m \geq 0\}$;

г) $\{\mathtt{a}^l\mathtt{b}^{2l} | \, l > 0\}$;

д) $\{\mathtt{a}\mathtt{b}^l | \, l > 0\} \cup \{\mathtt{a}^m\mathtt{b}^{2m} | \, m > 0\}$.

15. Какой из языков $L_1 = \{\mathtt{a}^n\mathtt{b}^n | \, n > 0\}$, $L_2 = \{\mathtt{a}^l\mathtt{c}\mathtt{b}^m | \, l, m > 0\}$ или $L_3 = \{\mathtt{a}^n\mathtt{c}\mathtt{b}^n | \, n > 0\}$ допускается МП-автоматом с управляющей таблицей, приведенной в табл. 2.18, *а*? В начале работы стек содержит символ D.

Таблица 2.18. *Управляющая таблица МП-автомата для задачи 15*

	a	b	c	⊣
D	$ED, \rightarrow$		$F, \rightarrow$	
E		$\varepsilon^{\rightarrow}$		
F		ε		
∇				✓

Глава 3. Контекстно-свободные грамматики

Данная глава посвящена основному способу задания используемых на практике формализованных языков – контекстно-свободным грамматикам. Контекстно-свободные грамматики представляют значительный теоретический интерес и широко используются в практике программирования. Глава начинается с обсуждения более широкого понятия – понятия порождающей грамматики.

§ 11. Порождающие грамматики

Определение. *Порождающей грамматикой* (или просто *грамматикой*) называется четверка $G = (\Sigma, \Gamma, P, S)$, где Σ – *основной* (или *терминальный*) алфавит, элементы которого называются *терминалами*, Γ – *вспомогательный* (или *нетерминальный*) алфавит, элементы которого называются *нетерминалами*, $S \in \Gamma$ – выделенный нетерминал, называемый *аксиомой*, P – множество формальных выражений вида $\alpha \to \beta$, где $\alpha, \beta \in (\Sigma \cup \Gamma)^*$ и α содержит хотя бы один нетерминал, называемых *правилами вывода*.

Приведем пример: пусть $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $\Gamma = \{S, D\}$ (S – аксиома), $P = \{S \to \mathtt{a}S D\mathtt{a}, S \to \mathtt{aba}, \mathtt{a}D \to D\mathtt{a}, \mathtt{b}D \to \mathtt{bb}\}$. Полученную грамматику обозначим через $G_{3,1}$[†].

Грамматика, как было сказано выше, – способ задания языка. Дадим соответствующие определения.

Определение. Пусть $G = (\Sigma, \Gamma, P, S)$ – грамматика, $\gamma, \delta \in (\Sigma \cup \Gamma)^*$. Цепочка δ *непосредственно выводима в* G из цепочки γ (обозначается $\gamma \Rightarrow_G \delta$), если существуют такие цепочки $\alpha, \beta, \gamma_1, \gamma_2 \in (\Sigma \cup \Gamma)^*$, что $\gamma = \gamma_1\alpha\gamma_2$, $\delta = \gamma_1\beta\gamma_2$ и $\alpha \to \beta \in P$.

Когда понятно, о какой грамматике идет речь, мы будем писать просто $\gamma \Rightarrow \delta$. Приведем примеры непосредственной выводимости в грамматике $G_{3,1}$: $\mathtt{a}S D\mathtt{a} \Rightarrow \mathtt{aaba}D\mathtt{a}$ ($\gamma_1 = \mathtt{a}$, $\gamma_2 = D\mathtt{a}$, $S \to \mathtt{aba} \in P$); $\mathtt{a}D\mathtt{b}S\mathtt{a} \Rightarrow D\mathtt{ab}S\mathtt{a}$ ($\gamma_1 = \varepsilon$, $\gamma_2 = \mathtt{b}S\mathtt{a}$, $\mathtt{a}D \to D\mathtt{a} \in P$); $S \Rightarrow \mathtt{aba}$ ($\gamma_1 = \gamma_2 = \varepsilon$, $S \to \mathtt{aba} \in P$).

Определение. Пусть $G = (\Sigma, \Gamma, P, S)$ – грамматика, $\gamma, \delta \in (\Sigma \cup \Gamma)^*$. Цепочка δ *выводима в* G из цепочки γ (обозначается $\gamma \Rightarrow^*_G \delta$), если

[†]Для грамматик, используемых в книге «локально», мы будем применять здесь и далее двойную индексацию (номер главы, номер примера).

существуют число $n \geq 0$ и цепочки η_0, η_1,..., η_n такие, что

$$\gamma = \eta_0 \Rightarrow \eta_1 \Rightarrow \eta_2 \ldots \Rightarrow \eta_n = \delta \ .$$

Последовательность цепочек $\eta_0, \eta_1, \ldots, \eta_n$ называется *выводом*, число n – *длиной* вывода.

Замечание 3.1. Отношение выводимости в G есть рефлексивно-транзитивное замыкание отношения непосредственной выводимости в G, этим и объясняется выбор обозначения. Мы будем также использовать транзитивное замыкание непосредственной выводимости $\Rightarrow_G^+$ (отношение «нетривиальной» выводимости в G).

Пример: $S \Rightarrow_{G_{3,1}}^* \mathtt{a}^2\mathtt{b}^2\mathtt{a}^2$; соответствующий вывод имеет длину 4:

$$S \Rightarrow \mathtt{a}S D\mathtt{a} \Rightarrow \mathtt{aaba}D\mathtt{a} \Rightarrow \mathtt{aab}D\mathtt{aa} \Rightarrow \mathtt{aabbaa}.$$

Введем основное определение этого параграфа.

Определение. *Языком, порождаемым грамматикой* G (обозначается через $L(G)$), называется множество цепочек над основным алфавитом, выводимых из аксиомы грамматики.

Из приведенных выше примеров следует, что $\mathtt{aba}, \mathtt{a}^2\mathtt{b}^2\mathtt{a}^2 \in L(G_{3,1})$. Нетрудно показать, что $L(G_{3,1}) = \{\mathtt{a}^n\mathtt{b}^n\mathtt{a}^n \mid n \in \mathbb{N}\}$.

Грамматика порождает язык над своим основным алфавитом. Какую же лингвистическую роль играет вспомогательный алфавит? Для того чтобы ответить на этот вопрос, рассмотрим еще один пример порождающей грамматики, которую будем обозначать через $G_{3,2}$. Пусть вспомогательный алфавит грамматики $G_{3,2}$ состоит из элементов <S>, <A>, <B>, символ <S> является аксиомой, основной алфавит содержит строчные буквы латинского алфавита и десятичные цифры. Правилами вывода этой грамматики являются следующие правила: <S>→<S><A>, <S>→<S><B>, <S>→<A>, <A>→ a, <A>→ b,..., <A>→ z, <B>→ 0, <B>→ 1,..., <B>→ 9. Легко видеть, что язык, порожденный грамматикой $G_{3,2}$, состоит из последовательностей букв и цифр, начинающихся с буквы, т. е. $L(G_{3,2})$ – множество имен языка программирования типа Паскаль. Этот пример показывает, что элементы вспомогательного алфавита обозначают грамматические классы языка, а именно, элемент <S> – класс имен, <A> – класс букв, <B> – класс цифр.

На примере грамматики $G_{3,2}$ сделаем еще одно полезное наблюдение. Мы видим, что у этой грамматики есть довольно много пра-

вил с одной и той же левой частью. Такие правила вывода называются *альтернативами*. Множество правил $\{A \to \beta_1,\ A \to \beta_2, \ldots,\ A \to \beta_k\}$ чаще всего записывают в виде «обобщенного» правила $A \to \beta_1 \,|\, \beta_2 \,|\, \ldots \,|\, \beta_k$ (ср. запись регулярных выражений, § 7). Этот способ записи позволит представить 39 правил грамматики $G_{3,2}$ в виде трех обобщенных правил:

<S> → <S><A> | <S><B> | <A>,
<A> → a | b | ... | z,
<B> → 0 | 1 | ... | 9.

Чтобы упростить обозначения, *порождающую грамматику будем задавать только множеством правил вывода*. Условимся об обозначениях, которые будем применять по умолчанию при задании (и обсуждении) порождающих грамматик:

1) терминалы обозначаем малыми буквами начала латиницы;

2) нетерминалы – большими буквами начала латиницы или строками символов в угловых скобках; в левой части первого правила вывода стоит аксиома;

3) произвольные грамматические символы из $\Sigma \cup \Gamma$ – большими буквами конца латиницы;

4) цепочки терминалов – малыми буквами конца латиницы;

5) цепочки произвольных грамматических символов из $\Sigma \cup \Gamma$ – малыми греческими буквами.

Во всех случаях можно использовать индексы.

§ 12. Примеры порождающих грамматик

В предыдущем параграфе были рассмотрены две грамматики и порождаемые ими языки, причем первыми появлялись грамматики, а затем языки. В этом параграфе мы продолжим список примеров грамматик, но порядок появления грамматик и языков будет другой. Мы будем ставить вопросы о существовании грамматик, порождающих те или иные языки.

Вначале рассмотрим язык $L = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \,|\, n \in \mathbb{N}\}$, который похож на язык, порожденный грамматикой $G_{3,1}$. Напишем грамматику для этого языка (будем ее обозначать через $G_{3,3}$), используя в качестве аналога грамматику $G_{3,1}$. В последней грамматике $(n{-}1)$-кратное

применение правила $S \to \mathtt{a}S D\mathtt{a}$ и последующее применение правила $S \to \mathtt{aba}$ дает цепочку $\mathtt{a}^n\mathtt{b}(\mathtt{a}D)^{n-1}\mathtt{a}$. Правило вывода $\mathtt{a}D \to D\mathtt{a}$ дает возможность переставлять $\mathtt{a}$ и D и группировать символы D после $\mathtt{b}$. Затем с помощью правила $\mathtt{b}D \to \mathtt{bb}$ ближайший к $\mathtt{b}$ символ D превращается в $\mathtt{b}$.

Построение грамматики $G_{3,3}$ начнем с двух правил: $S \to \mathtt{a}SBC$ и $S \to \mathtt{ab}C$. Применение первого правила $(n-1)$ раз и последующее применение второго правила даст цепочку $\mathtt{a}^n\mathtt{b}C^nB^{n-1}$. Добавим правила $CB \to BC$ и $\mathtt{b}B \to \mathtt{bb}$. Первое обеспечивает перестановку C и B, второе – превращение B в $\mathtt{b}$ (если слева стоит $\mathtt{b}$). После применения этих правил получается цепочка $\mathtt{a}^n\mathtt{b}^nC^n$. Осталось добавить правила $\mathtt{b}C \to \mathtt{cc}$ и $\mathtt{c}C \to \mathtt{cc}$, с помощью которых из C^n можно получить $\mathtt{c}^n$. Итак,

$$G_{3,3} = \{S \to \mathtt{a}SBC \,|\, \mathtt{ab}C, CB \to BC, \mathtt{b}B \to \mathtt{bb}, \mathtt{b}C \to \mathtt{cc}, \mathtt{c}C \to \mathtt{cc}\}.$$

В качестве второго языка рассмотрим язык $L \in \{\mathtt{a}, \mathtt{b}\}^*$, состоящий из всех слов с равным количеством $\mathtt{a}$ и $\mathtt{b}$.

Одной из наиболее простых грамматик, порождающих этот язык, будет, по-видимому, следующая грамматика. Сначала запишем правила вывода $S \to ASB$ и $S \to \varepsilon$, позволяющие из аксиомы получить цепочки вида A^nB^n. Затем «обеспечим возможность» перестановки букв A и B с помощью правила $AB \to BA$. Осталось добавить правила $A \to \mathtt{a}$, $B \to \mathtt{b}$. В итоге получаем грамматику

$$G_{3,4} = \{S \to ASB \,|\, \varepsilon, AB \to BA, A \to \mathtt{a}, B \to \mathtt{b}\}.$$

Теперь построим другую грамматику, порождающую тот же язык, основанную на рекурсии. Если цепочка $w \in L$ начинается с $\mathtt{a}$, то в оставшейся ее части число букв $\mathtt{b}$ на единицу больше числа букв $\mathtt{a}$. Пусть B – вспомогательный символ, из которого выводятся цепочки с таким свойством. Аналогично, пусть A – вспомогательный символ, из которого выводятся цепочки, содержащие букв $\mathtt{a}$ на единицу больше, чем букв $\mathtt{b}$. Отразим сказанное в правилах вывода $S \to \mathtt{a}B$, $S \to \mathtt{b}A$. Рассмотрим теперь цепочку v, выводимую из B. Если v начинается с буквы $\mathtt{b}$, то в оставшейся части цепочки число букв $\mathtt{a}$ равно числу букв $\mathtt{b}$, т. е. оставшаяся часть цепочки выводима из S. Добавим поэтому правило $B \to \mathtt{b}S$. Если же v начинается с буквы $\mathtt{a}$, то в оставшейся части цепочки v число букв $\mathtt{b}$ на две больше числа букв $\mathtt{a}$. Отразим этот факт в правиле $B \to \mathtt{a}BB$. Рассуждая

аналогичным образом для цепочки, выводимой из A, включим в список правил вывода правила $A \to a\mathrm{S}$, $A \to \mathtt{b}AA$. Осталось учесть, что L содержит пустую цепочку, добавив правило $S \to \varepsilon$. Результатом будет грамматика

$$G_{3,5} = \{S \to \mathtt{a}B \,|\, \mathtt{b}A \,|\, \varepsilon, B \to \mathtt{b}S \,|\, \mathtt{a}BB, A \to \mathtt{a}S \,|\, \mathtt{b}AA\}.$$

Параграф завершим построением грамматики, порождающей язык $L = \{w\mathtt{c}w \,|\, w \in \{\mathtt{a}, \mathtt{b}\}^*\}$.

Цепочки этого языка – копии произвольной цепочки над алфавитом $\{\mathtt{a}, \mathtt{b}\}$, разделенные символом $\mathtt{c}$. Грамматика для этого языка строится следующим образом. Сначала правилами вывода $S \to \mathtt{a}AS$, $S \to \mathtt{b}BS$ и $S \to \mathtt{c}$ порождается фрагмент $w\mathtt{c}$, причем вместе с каждым символом $\mathtt{a}$ или $\mathtt{b}$ записывается его «дубль» – нетерминал A или B соответственно. После этого символы A и B переводятся в правую часть цепочки правилами вывода $A\mathtt{a} \to \mathtt{a}A$, $A\mathtt{b} \to \mathtt{b}A$, $B\mathtt{a} \to \mathtt{a}B$, $B\mathtt{b} \to \mathtt{b}B$. Взаимное расположение символов A и B остается прежним. В результате можно будет получить цепочку вида $wW\mathtt{c}$, где слово W – то же самое, что w, только записанное «большими буквами». Осталось превратить символы A и B соответственно в $\mathtt{a}$ и $\mathtt{b}$ сразу после разделителя $\mathtt{c}$. Это можно сделать, используя правила вывода $A\mathtt{c} \to \mathtt{ca}$ и $B\mathtt{c} \to \mathtt{cb}$ и двигаясь от конца слова W к началу. Собрав вместе все написанные правила, получим грамматику

$$G_{3,6} = \{S \to \mathtt{a}AS \,|\, \mathtt{b}BS \,|\, \mathtt{c}, A\mathtt{a} \to \mathtt{a}A, A\mathtt{b} \to \mathtt{b}A, B\mathtt{a} \to \mathtt{a}B, B\mathtt{b} \to \mathtt{b}B, Ac \to \mathtt{ca}, Bc \to \mathtt{cb}\}.$$

§ 13. Классы грамматик и классы языков

Уже из приведенных в двух предыдущих параграфах примеров можно сделать вывод о том, что грамматики представляют собой достаточно универсальную модель представления языков. Возникают два естественных вопроса:

1) какие языки могут быть порождены грамматиками?

2) какие классы языков будут получаться, если накладывать различные ограничения на грамматики?

В этом параграфе мы кратко осветим оба вопроса. Более подробно см., напр., [Сал].

Поскольку языки – это множества, к ним применимы все понятия, используемые для множеств в теории алгоритмов. Так, язык L – *рекурсивный*, если существует алгоритм[†], определяющий по любому слову w принадлежность w к L (т. е. для L существует распознаватель). Язык M – *рекурсивно-перечислимый*, если существует алгоритм, перечисляющий (в общем случае за бесконечное время) все слова из M и только их. Поскольку по грамматике легко построить перечисляющий алгоритм (например, алгоритм строит в ширину дерево всех возможных выводов), то любой язык, порожденный грамматикой, рекурсивно перечислим. Обратное утверждение совсем не очевидно, но тоже верно. Тем самым любой язык, который в принципе можно задать «конечным образом», можно задать порождающей грамматикой, т. е. грамматики действительно представляют собой наиболее универсальную модель описания языков! Отметим, что поскольку класс рекурсивно-перечислимых языков содержит класс рекурсивных языков, грамматикой можно задать любой язык, для которого существует распознаватель, в частности, языки, распознаваемые конечными автоматами и МП-автоматами.

При рассмотрении некоторых естественных ограничений на вид правил вывода в грамматиках получаются классы грамматик (и порождаемые ими классы языков), образующие так называемую *иерархию Хомского*. Самая широкая ступень этой иерархии образована классом всех грамматик и классом рекурсивно-перечислимых языков.

Определение. Грамматика называется *контекстно-зависимой* (соответственно *контекстно-свободной*, *праволинейной*), если каждое правило вывода в ней имеет вид $\alpha A\beta \to \alpha\gamma\beta$ (соответственно $A \to \alpha$, $A \to \mathtt{a}B$).

Вторую ступень иерархии формируют контекстно-зависимые грамматики и порождаемые ими контекстно-зависимые языки. Выразительных возможностей таких грамматик с запасом хватает для практически используемых формализованных языков, в частности для языков программирования. Однако отсутствие удобных распознавателей и сложность решения многих алгоритмических проблем делают этот класс грамматик неудобным для практического применения.

[†]Согласно тезису Тьюринга любой алгоритм реализуется универсальным вычислителем – машиной Тьюринга.

Третья ступень иерархии образована контекстно-свободными грамматиками и языками, обсуждению которых посвящена примерно половина объема этой книги. Контекстно-свободные языки гораздо проще контекстно-зависимых с точки зрения решения алгоритмических проблем, для них существуют удобные распознаватели (это МП-автоматы, как мы увидим в дальнейшем), а главное – контекстно-свободные грамматики способны выразить бóльшую часть синтаксиса языков программирования.

Последняя ступень образована праволинейными грамматиками и порождаемыми ими регулярными языками (см. задачу 2). Об этих языках достаточно сказано в гл. 2.

§ 14. Контекстно-свободные грамматики и языки

Начнем с важного уточнения определения контекстно-свободного языка, показывающего, что такие языки могут порождаться не только контекстно-свободными грамматиками.

Определение. Язык называется контекстно-свободным, если *существует* контекстно-свободная грамматика, его порождающая.

В дальнейшем мы будем пользоваться сокращениями *КС-язык* и *КС-грамматика*. Нам также удобно ввести отдельный термин для цепочки, которая принадлежит выводу некоторой терминальной цепочки из аксиомы КС-грамматики. Такие цепочки будем называть *формами*.

Грамматики $G_{3,2}$ и $G_{3,5}$, введенные в предыдущих параграфах, являются контекстно-свободными. Следовательно, порождаемые этими грамматиками языки также являются контекстно-свободными. В примере 2.8 рассматривался язык $\{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$, который не распознается конечными автоматами, но распознается МП-автоматом. Этот язык порождается КС-грамматикой $G_{3,7} = \{S \to \mathtt{a}S\mathtt{b} \mid \mathtt{ab}\}$. В § 17 нам встретится язык $\{\mathtt{a}^n\mathtt{b}^n\mathtt{a}^m \mid n, m \in \mathbb{N}\}$. Он также является контекстно-свободным, так как порождается КС-грамматикой $G_{3,8} = \{S \to CD \mid \mathtt{a}C\mathtt{b} \mid \mathtt{ab}, D \to \mathtt{a}D \mid \mathtt{a}\}$. Естественно задать вопрос, является ли язык $\{\mathtt{a}^n\mathtt{b}^n\mathtt{a}^n \mid n \in \mathbb{N}\}$ контекстно-свободным. Грамматика $G_{3,1}$, порождающая этот язык, контекстно-свободной не является. Однако в принципе может существовать и КС-грамматика, порождающая этот язык. Ответ на поставленный вопрос мы дадим в § 15.

Вывод в КС-грамматике удобно представлять при помощи дерева, называемого *деревом вывода*. Это понятие рассмотрим сначала

на примере. Пусть $G_{3,9} = \{S \to \mathtt{a}SD\mathtt{a} \,|\, \mathtt{bc}, D \to \mathtt{a}E \,|\, \mathtt{a}EF, E \to D \,|\, \mathtt{c}, F \to \mathtt{b}\}$ и $w = \mathtt{abcaacba}$. Цепочка w выводима в $G_{3,9}$:

$$\begin{aligned} S \Rightarrow \mathtt{a}SD\mathtt{a} \Rightarrow \mathtt{abc}D\mathtt{a} \Rightarrow \mathtt{abca}EF\mathtt{a} \Rightarrow \mathtt{abca}DF\mathtt{a} \Rightarrow \\ \Rightarrow \mathtt{abcaa}EF\mathtt{a} \Rightarrow \mathtt{abcaac}F\mathtt{a} \Rightarrow \mathtt{abcaacba} = w \ . \end{aligned} \tag{3.1}$$

Построим по выводу (3.1) корневое дерево, узлы которого помечены грамматическими символами, следующим образом (рис. 3.1). Вначале дерево состоит только из корня, помеченного аксиомой. Применению правила $S \to \mathtt{a}SD\mathtt{a}$ на первом шаге вывода соответствует добавление в дерево четырех сыновей корня, помеченных (слева направо) символами a, S, D, a; в результате получается дерево T_1. На втором шаге вывода применялось правило $S \to \mathtt{bc}$: добавим два потомка с метками b и c к листу дерева T_1, помеченному S, получая дерево T_2. Последовательно обработав подобным образом все шаги вывода (3.1), получим дерево T_3, которое и является деревом вывода (цепочки w в грамматике $G_{3,9}$). Если обойти все листья дерева T_3 слева направо, то мы прочитаем в точности цепочку w.

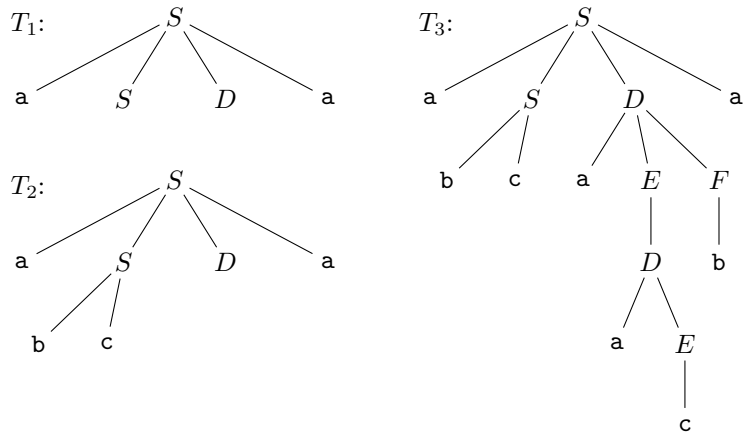


Рис. 3.1. *Построение дерева вывода*

Перейдем к точным определениям.

Определение. Корневое дерево называется *упорядоченным*, если на множестве его узлов задано отношение линейного порядка $\lhd$, удовлетворяющее двум условиям:

1) если узел y является потомком узла x, то $x \lhd y$;

2) если узел y является братом узла x и $x \lhd y$, то $z \lhd y$ для любого потомка z узла x.

О порядке $\lhd$ на любом множестве попарно несмежных узлов упорядоченного дерева (в частности, на множестве братьев) мы говорим как о «порядке слева направо» и изображаем упорядоченные деревья соответствующим образом.

Определение. *Деревом вывода* цепочки $w \in \Sigma^*$ в грамматике $G = \{\Sigma, \Gamma, P, S\}$ называется упорядоченное дерево, узлы которого помечены символами из множества $\Sigma \cup \Gamma \cup \{\varepsilon\}$ так, что

1) корень дерева помечен аксиомой, внутренние узлы – нетерминалами, листья – терминалами или ε, причем узлы с меткой ε не имеют братьев;

2) если $y_1 \lhd y_2 \lhd \ldots \lhd y_k$ – все сыновья узла x в дереве, $Y_1, \ldots, Y_k, X$ – метки упомянутых узлов, то $X \to Y_1 \ldots Y_k \in P$;

3) если $z_1 \lhd z_2 \lhd \ldots \lhd z_m$ – все листья дерева, $a_1, \ldots, a_m$ – их метки, то $w = a_1 \ldots a_m$.

Определение. Поддерево T' дерева вывода T называется *стандартным*, если корень T лежит в T' и для любого узла $x \in T'$ либо x является листом T', либо все сыновья x в дереве T также лежат в T'.

Например, на рис. 3.1 деревья T_1 и T_2 являются стандартными поддеревьями дерева вывода T_3.

Определение. Вывод $S \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n = w$ в грамматике G *представлен* деревом вывода T, если в T существуют стандартные поддеревья $T_1 \subset \ldots \subset T_n$ такие, что для каждого $i = 1, \ldots, n$ метки листьев поддерева T_i, упорядоченных слева направо, образуют форму α_i. (Мы также будем говорить, что вывод формы α_i представлен деревом T_i.)

Очевидно, что вывод (3.1) представлен деревом T_3 на рис. 3.1. Но из определения понятно, что одно и то же дерево может представлять различные выводы. Укажем еще два вывода цепочки w в грамматике $G_{3,9}$:

$$\begin{gathered} S \Rightarrow \mathtt{a}S D\mathtt{a} \Rightarrow \mathtt{a}S\mathtt{a}EF\mathtt{a} \Rightarrow \mathtt{a}S\mathtt{a}E\mathtt{ba} \Rightarrow \mathtt{a}S\mathtt{a}D\mathtt{ba} \Rightarrow \\ \Rightarrow \mathtt{a}S\mathtt{aa}E\mathtt{ba} \Rightarrow \mathtt{a}S\mathtt{aacba} \Rightarrow \mathtt{abcaacba} = w\ . \end{gathered} \tag{3.2}$$

$$S \Rightarrow \mathtt{a}SD\mathtt{a} \Rightarrow \mathtt{abc}D\mathtt{a} \Rightarrow \mathtt{abca}E\mathtt{a} \Rightarrow \mathtt{abca}D\mathtt{a} \Rightarrow \\ \Rightarrow \mathtt{abcaa}EF\mathtt{a} \Rightarrow \mathtt{abcaac}F\mathtt{a} \Rightarrow \mathtt{abcaacba} = w \ . \tag{3.3}$$

Легко убедиться, что вывод (3.2) представлен тем же деревом T_3, что и вывод (3.1), в то время как для вывода (3.3) дерево будет другим (рис. 3.2).

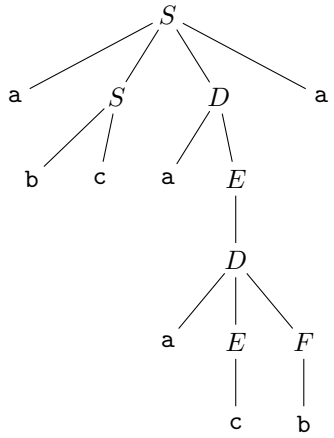


Рис. 3.2. *Другое дерево вывода цепочки w в грамматике $G_{3,9}$*

Среди всех выводов, представленных деревом T_3 на рис. 3.1, мы не случайно выбрали выводы (3.1) и (3.2) – такие выводы считаются «каноническими». В выводе (3.1) правила применялись каждый раз к самому левому нетерминалу. Такой вывод называется *левосторонним* (или *левым*). Наоборот, в выводе (3.2) правила применялись к самому правому нетерминалу. Вывод с таким свойством называется *правосторонним* (или *правым*). Формы, принадлежащие таким выводам, будем называть соответственно l- и r-формами.

Замечание 3.2. Каждому дереву вывода соответствует единственный левый и единственный правый вывод цепочки.

Замечание 3.3. Основная роль дерева вывода состоит в том, что оно связывает *синтаксис* и *семантику* выводимой цепочки (которая, к примеру, может быть фразой естественного языка или – что интересует нас больше – компьютерной программой). Несколько упрощая ситуацию, можно сказать, что семантика компьютерной программы – это *алгоритм* решения задачи, а дерево вывода

описывает структуру программы и тем самым указывает порядок выполнения машинных операций, необходимый для реализации алгоритма.

Определение. Грамматика G называется *однозначной*, если каждая цепочка из $L(G)$ представляется в G единственным деревом вывода, и *неоднозначной* в противном случае, т. е. если в $L(G)$ найдется цепочка, представленная в G двумя различными деревьями вывода.

Как было установлено выше, грамматика $G_{3,9}$ неоднозначна. То, что неоднозначность – отрицательное свойство грамматики, вытекает из замечания 3.3: цепочка, представленная двумя деревьями вывода, – это программа, которая кодирует две различные последовательности машинных инструкций, одна из которых наверняка реализует не тот алгоритм, который имел в виду программист.

Пример 3.1. Грамматика $G_{3,10} = \{E \to E{+}E \mid E{*}E \mid \mathtt{0} \mid \mathtt{1} \mid \ldots \mid \mathtt{9}\}$ порождает арифметические выражения над числами от 0 до 9. Семантика цепочки в данной грамматике – это значение арифметического выражения, которое определяется операндами, операторами и порядком выполнения операций в выражении. Неоднозначность грамматики $G_{3,10}$ (рис. 3.3) приводит к невозможности однозначно определить значение выражения и, как следствие, к непригодности данной грамматики для практического задания арифметических выражений.

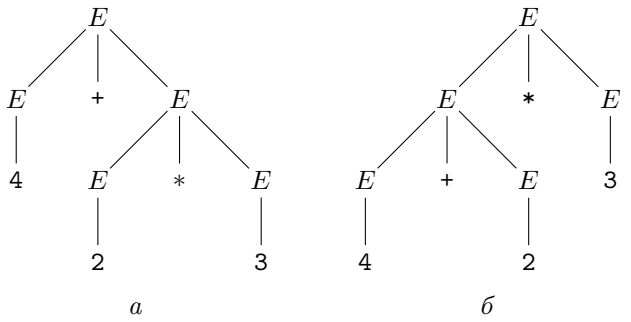


Рис.3.3. *Два дерева вывода, представляющих выражение* $\mathtt{4}+\mathtt{2}*\mathtt{3}$. *Дерево «а» показывает, что сложение должно применяться к результату умножения, а дерево «б» – что в умножении участвует результат сложения*

Неоднозначность не только плохое, но и трудноопределимое свой-

ство грамматики. Известный теоретический результат гласит, что *не существует алгоритма, определяющего по произвольной КС-грамматике, однозначна она или нет.* Это означает, что однозначность каждой грамматики надо проверять «индивидуально».

Перенесем понятие «неоднозначность» с грамматики на язык. КС-язык будем называть *однозначным*, если существует однозначная грамматика, его порождающая, и *неоднозначным* в противном случае, т. е. если *любая* КС-грамматика, порождающая этот язык, является неоднозначной. Из этого определения можно предположить, что неоднозначные КС-языки являются довольно экзотическими объектами. Однако это не так. Неоднозначным является следующий язык, содержащий цепочки довольно простой структуры:

$$L = \{\mathtt{a}^k\mathtt{b}^k\mathtt{c}^l\mathtt{d}^l \mid k,l \in \mathbb{N}\} \cup \{\mathtt{a}^k\mathtt{b}^l\mathtt{c}^l\mathtt{d}^k \mid k,l \in \mathbb{N}\}\,.$$

Язык L является контекстно-свободным, так как порождается, в частности, КС-грамматикой

$$\begin{aligned} G = \{&S \to AB \mid C, A \to \mathtt{a}A\mathtt{b} \mid \mathtt{ab}, B \to \mathtt{c}B\mathtt{d} \mid \mathtt{cd},\\ &C \to \mathtt{a}C\mathtt{d} \mid \mathtt{a}D\mathtt{d}, D \to \mathtt{b}D\mathtt{c} \mid \mathtt{bc}\}\ . \end{aligned}$$

Цепочка $\mathtt{a}^2\mathtt{b}^2\mathtt{c}^2\mathtt{d}^2$, принадлежащая языку L, имеет в этой грамматике два дерева вывода (рис. 3.4). Более того, известно, что в любой КС-грамматике, порождающей L, найдутся цепочки вида $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n\mathtt{d}^n$, представимые не единственным деревом вывода. Доказательство мы здесь не приводим, см., напр., [Ги].

§ 15. Примеры КС-грамматик и языков

В предыдущем параграфе были приведены несложные и в основном формальные примеры контекстно-свободных грамматик и языков. В этом параграфе сосредоточим внимание на языках, отражающих те или иные конструкции языков программирования (см. § 1), и на грамматиках, порождающих эти языки.

Пример 3.2. Рассмотрим скобочный язык LB. Если определение сбалансированной цепочки скобок «перевести» на язык КС-грамматик, то получится грамматика $GB_1 = \{S \to \varepsilon \mid [S] \mid SS\}$.

Ясно, что $L(GB_1) = LB$. Однако грамматика GB_1 обладает существенным недостатком – она неоднозначна. «Виновником» этого

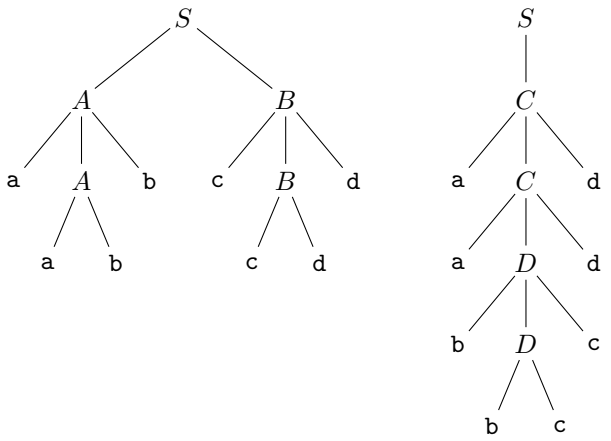


Рис. 3.4. *Два дерева вывода для строки* $\mathtt{a}^2\mathtt{b}^2\mathtt{c}^2\mathtt{d}^2$ *в грамматике* G

является третье правило вывода: нетрудно проверить, что любая цепочка представима различными деревьями вывода.

«Склеив» два последних правила, получим грамматику, лишенную этого недостатка: $GB_2 = \{S \to \varepsilon \,|\, S[S]\}$. Поскольку в грамматике GB_2 выполняется $S \Rightarrow^* \varepsilon$, $S \Rightarrow^* [S]$ и $S \Rightarrow^* [S][S]$, она также порождает язык LB.

Грамматика GB_2 содержит правило вывода вида $A \to A\alpha$. Такие правила называются *леворекурсивными*. Их наличие в грамматике делает неприменимыми некоторые методы синтаксического анализа цепочек, как мы увидим в ч. 2. В данном случае левую рекурсию легко заменить на двойственную правую, получая грамматику $GB_3 = \{S \to \varepsilon \,|\, [S]S\}$ (наличие *праворекурсивных* правил упомянутыми методами синтаксического анализа допускается).

Пример 3.3. Построим грамматики, порождающие язык списков LL. Непосредственно из определения списка, приведенного в § 1, получаем неоднозначную грамматику $GL_1 = \{S \to \mathtt{a} \,|\, [S] \,|\, S; S\}$.

Цепочка $\mathtt{a};\mathtt{a};\mathtt{a}$ представима в этой грамматике двумя различными деревьями вывода. Введением дополнительного нетерминала можно избавиться от неоднозначности; приведем леворекурсивный (GL_2) и праворекурсивный (GL_3) варианты грамматики:

$$GL_2 = \{S \to S; L \,|\, L,\ L \to \mathtt{a} \,|\, [S]\}\,,\ GL_3 = \{S \to L; S \,|\, L,\ L \to \mathtt{a} \,|\, [S]\}\,.$$

Равенства $L(GL_2) = L(GL_3) = LL$ читателю предлагается проверить самостоятельно.

Пример 3.4. Язык описаний типов LD порождается грамматикой

$$GD = \{D \to L{:}T,\ L \to L;\mathtt{i} \,|\, \mathtt{i},\ T \to \mathtt{real} \,|\, \mathtt{int}\}\ .$$

Пример 3.5. Язык двоичных чисел LN порождается грамматикой

$$GN_1 = \{S \to L \,|\, .L \,|\, L.L,\ L \to LB \,|\, B,\ B \to \mathtt{0} \,|\, \mathtt{1}\}\ .$$

Если потребовать, чтобы целая часть двоичного числа всегда начиналась единицей, а дробная единицей заканчивалась (соответствующий язык обозначался в § 1 через LN'), то надо несколько изменить грамматику:

$$GN_2 = \{S \to L \,|\, .R \,|\, L.R,\ L \to L\mathtt{0} \,|\, L\mathtt{1} \,|\, \mathtt{1},\ R \to \mathtt{0}R \,|\, \mathtt{1}R \,|\, \mathtt{1}\}\ .$$

Пример 3.6. Завершим параграф рассмотрением языка арифметических выражений LA. Если, как и выше, порождающую грамматику составить, повторяя определение из § 1, то получим неоднозначную грамматику $GA_1 = \{E{\to}E{+}E \,|\, E{*}E \,|\, (E) \,|\, \mathtt{x}\}$ (ср. пример 3.1).

Составим для языка LA однозначную грамматику. Неоднозначность грамматики GA_1 обусловлена отсутствием указаний на порядок выполнения арифметических операций, т. е. на *приоритет* операций. Как мы знаем, действия в скобках должны выполняться раньше действий за скобками, умножение – раньше сложения, а одноименные операции – в любом порядке, так как они ассоциативны. Таким образом, арифметическое выражение – это сумма одного или более слагаемых, каждое из которых – произведение одного или более множителей, каждый из которых есть буква x или арифметическое выражение, заключенное в скобки. Кроме того, для составления однозначной грамматики примем «естественный» порядок выполнения одноименных операций – слева направо. Приведенная структура арифметического выражения отражена в грамматике GA_2, содержащей, помимо нетерминала E для выражений, нетерминалы T (для слагаемых) и F (для множителей):

$$GA_2 = \{E \to E{+}T \,|\, T,\ T \to T{*}F \,|\, F,\ F \to (E) \,|\, \mathtt{x}\}\ .$$

Замечание 3.4. Грамматика GA_2 содержит леворекурсивные правила $E{\to}E{+}T$ и $T{\to}T{*}F$. Для их устранения можно, как в предыдущих примерах, просто поменять местами нетерминалы в правых

частях таких правил. Такая замена не повлияет на синтаксис выводимых цепочек (по-прежнему порождаться будет язык LA), но повлияет на семантику: одноименные операции станут выполняться не слева направо, а справа налево. Общий алгоритм устранения левой рекурсии с сохранением семантики будет приведен в следующей главе.

§ 16. Теорема о накачке

В этом параграфе мы докажем одно из фундаментальных свойств КС-языков. Довольно часто можно показать, что тот или иной язык не является контекстно-свободным, установив отсутствие у него данного свойства.

Теорема 3.1 (о накачке). *Для любого КС-языка L существуют натуральные числа n и m такие, что любая цепочка $w \in L$ с условием $|w| > n$ представима в виде $w = xuzvy$, где*

1) $uv \neq \varepsilon$,

2) $|uzv| \leq m$,

3) $xu^k zv^k y \in L$ для любого натурального k.

Доказательство. Если язык L является конечным, то теорема очевидно справедлива, поэтому далее мы считаем, что L бесконечен. Зафиксируем произвольную КС-грамматику $G = (\Sigma, \Gamma, P, S)$, порождающую L. Для каждой цепочки $w \in L$ можно выбрать дерево вывода в G с минимальным числом вершин. Далее рассматриваются только такие минимальные деревья.

Рассмотрим все минимальные деревья высоты не более $|\Gamma|$ (их конечное число) и в качестве n возьмем максимальную длину цепочки, вывод которой представлен таким деревом. Кроме того, положим $m = M^{|\Gamma|+1}$, где M – наибольшая длина правой части правила в P.

Пусть $w \in L$ и $|w| > n$. Рассмотрим минимальное дерево вывода T цепочки w. Его высота больше $|\Gamma|$ по выбору n, а значит, в самом длинном пути от корня до листа найдутся две вершины, помеченные одним и тем же нетерминалом. Пусть этот путь имеет вид $s_0, \ldots, s_r$, а узлы s_i и s_j (где $i < j$) помечены нетерминалом A, рис. 3.5. Без ограничения общности можно считать, что $r - i \leq |\Gamma| + 1$, поскольку среди любых последовательных $|\Gamma| + 1$ внутренних узлов пути есть два одинаково помеченных.

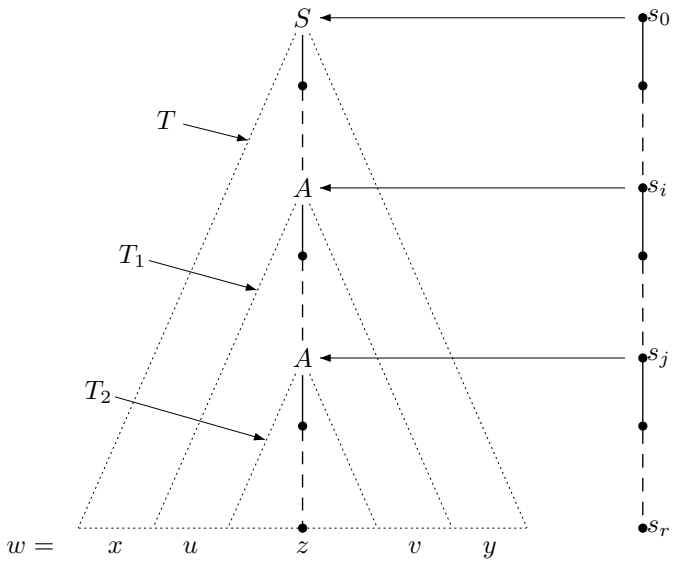


Рис. 3.5. *«Высокое» дерево вывода и его поддеревья*

Наша цель – доказать, что разбиение $w = xuzvy$, указанное на рис. 3.5, удовлетворяет всем условиям теоремы. Для этого введем в рассмотрение следующие поддеревья дерева T:

T_1, состоящее из узла s_i и всех его потомков,

T_2, состоящее из узла s_j и всех его потомков,

T', получаемое из T удалением всех потомков узла s_i,

T'_1, получаемое из T_1 удалением всех потомков узла s_j.

Дерево T представляет вывод $S \Rightarrow^* w$, а его стандартное поддерево T' – вывод $S \Rightarrow^* xAy$. Дерево T_1 представляет вывод $A \Rightarrow^* uzv$, а его стандартное поддерево T'_1 – вывод $A \Rightarrow^* uAv$. Наконец, дерево T_2 представляет вывод $A \Rightarrow^* z$. Следовательно, $xu^k zv^k y \in L$ для любого k:

$$S \Rightarrow^* xAy \Rightarrow^* xuAvy \Rightarrow^* xu^2Av^2y \Rightarrow^* \ldots \Rightarrow^* xu^kAv^ky \Rightarrow^* xu^kzv^ky.$$

Проверим, что $uv \neq \varepsilon$. Действительно, в противном случае T'_1 есть дерево вывода $A \Rightarrow^* A$. Но тогда вершины s_i и s_j можно отождествить, получая дерево вывода цепочки w, содержащее меньше вершин, нежели дерево T. Это противоречит выбору T.

Осталось показать, что $|uzv| \leq m$. Высота дерева T_1 равна $r - i$ (если бы в этом дереве существовал путь более длинный, чем путь $s_i, \ldots, s_r$, то путь $s_0, \ldots, s_r$ не был бы самым длинным в дереве T). Следовательно, в T_1 не более M^{r-i} листьев, а длина цепочки uzv очевидно не больше количества листьев. Из условия $r - i \leq |\Gamma| + 1$ и определения m получаем требуемое неравенство. Доказательство завершено. □

В качестве примеров применения теоремы 3.1 докажем, что два важных языка, упоминавшихся в § 12, не являются контекстно-свободными.

Пример 3.7. Язык $L = \{\mathtt{a}^n\mathtt{b}^n\mathtt{a}^n \mid n \in \mathbb{N}\}$ не является контекстно-свободным. Предположим противное: L – КС-язык. Тогда по теореме 3.1 для некоторого r выполняется $\mathtt{a}^r\mathtt{b}^r\mathtt{a}^r = xuzvy$, где, в частности, $|uv| > 0$ и $xu^2zv^2y \in L$.

Слово $\mathtt{a}^r\mathtt{b}^r\mathtt{a}^r$ состоит из блоков $\mathtt{a}^r$, $\mathtt{b}^r$, и $\mathtt{a}^r$; рассмотрим расположение цепочек u и v относительно этих трех блоков. Если u или v имеет непустое пересечение с двумя блоками, то цепочка xu^2zv^2y содержит более двух перемен букв, т. е. не имеет вида $\mathtt{a}^s\mathtt{b}^s\mathtt{a}^s$. Значит, и u, и v могут иметь непустое пересечение лишь с одним блоком. Отсюда следует, что при переходе от цепочки $xuzvy$ к цепочке xu^2zv^2y хотя бы один из трех блоков не увеличился в размерах. Но в силу условия $|uv| > 0$ хотя бы один из блоков увеличился в размерах, т. е. полученная цепочка не имеет вида $\mathtt{a}^s\mathtt{b}^s\mathtt{a}^s$. Данное противоречие доказывает, что L не является КС-языком.

Так как язык $\{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$ является КС-языком, то из приведенного примера можно сделать неформальный вывод о том, что контекстно-свободные грамматики «умеют сравнивать» количество объектов двух видов, но не трех.

Пример 3.8. Язык $L = \{w\mathtt{c}w \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$ не является контекстно-свободным. Предположим, что L является КС-языком. Рассмотрим цепочку $\omega = \mathtt{a}^l\mathtt{b}^l\mathtt{c}\mathtt{a}^l\mathtt{b}^l \in L$, где l – число, превосходящее числа n и m из формулировки теоремы 3.1. В силу этой теоремы данная цепочка представима в виде $xuzvy$, где, в частности, $0 < |uv| < l$ и $\omega' = xu^2zv^2y \in L$.

Изучим расположение цепочек u и v в слове ω. Ни u, ни v не может содержать буквы $\mathtt{c}$, в противном случае цепочка ω' будет содержать две буквы $\mathtt{c}$ и не будет принадлежать L. Далее, u и v не могут располагаться по одну сторону от буквы $\mathtt{c}$, так как в этом

случае в цепочке ω' с одной стороны от буквы $\mathtt{c}$ будет больше букв, чем с другой. Так как $|uv| < l$, то остается единственный вариант расположения u и v: цепочка u располагается в блоке $\mathtt{b}^l$ слева от $\mathtt{c}$, а цепочка v – в блоке $\mathtt{a}^l$ справа от $\mathtt{c}$. Но в этом случае цепочка ω' очевидно содержит разное количество букв $\mathtt{a}$ (как и букв $\mathtt{b}$) слева и справа от буквы $\mathtt{c}$. Итак, мы получили противоречие с условием $\omega' \in L$, из которого можем заключить, что язык L не контекстно-свободен.

Теорема о накачке применима и для получения результатов другого сорта. Например, с ее помощью можно эффективно охарактеризовать КС-языки над алфавитом из одной буквы. Дадим необходимое для этого

Определение. Множество $M \in \mathbb{N}_0$ называется *периодическим*, если существуют натуральные числа n_0 (индекс) и d (период) такие, что для любого $n \geq n_0$ условие $n \in M$ влечет $n+d \in M$.

Теорема 3.2. *Следующие условия эквивалентны для языка L над алфавитом $\{\mathtt{a}\}$:*

1) L – контекстно-свободный язык;

2) L – регулярный язык;

3) $M = \{n \in \mathbb{N} \mid \mathtt{a}^n \in L\}$ – периодическое множество.

Доказательство. $1 \Rightarrow 3$. Пусть $L \subseteq \{\mathtt{a}\}^*$ – КС-язык, m и n – соответствующие константы из теоремы 3.1. Умножение цепочек, составленных из букв $\mathtt{a}$, коммутативно. Следовательно, для каждой цепочки $w \in L$ такой, что $|w| \geq n$, можно записать $w = xuzvy = xzy\mathtt{a}^j$, где $0 < j \leq m$ и для любого $r \geq 1$ цепочка $xzy\mathtt{a}^{rj}$ принадлежит L. Тем самым за любой цепочкой из L длины не меньше n следует бесконечная последовательность цепочек из L, длины которых образуют арифметическую прогрессию с разностью, не превосходящей m. Положим $d = m!$; тогда для всякой цепочки $w \in L$ с условием $|w| \geq n$ гарантировано, что $w\mathtt{a}^d \in L$. Это означает, что число n можно взять в качестве индекса, а число d – в качестве периода множества M.

$3 \Rightarrow 2$. Пусть множество M бесконечно (конечный язык очевидно регулярен), а числа n_0 и d являются соответственно его индексом и периодом. Для каждого i такого, что $0 \leq i < d$, определим минимальное число k_i, удовлетворяющее условиям $k_i \in M$, $k_i \geq n_0$ и $k_i \equiv i \pmod{d}$. (Если чисел с такими свойствами нет, будем считать k_i не определенным.) Хотя бы одно из чисел k_i определено ввиду бесконечности M. Через k обозначим максимум из всех k_i. Построим

ДКА, распознающий L (рис. 3.6; метки ребер на рисунке не указаны, так как все они очевидно равны $\mathtt{a}$). Нетрудно видеть, что из начальной вершины 0 до вершины n при $n < k$ читается только слово $\mathtt{a}^n$, а при $n \geq k$ – множество всех слов вида $\mathtt{a}^{n+md}$, $m \geq 0$. С учетом определения k множество слов, читаемых до данной вершины, либо целиком лежит в L, либо целиком не лежит в L. В первом случае вершину n пометим как заключительную, во втором – помечать не будем. Требуемый автомат построен. Значит, язык L регулярен по теореме Клини.

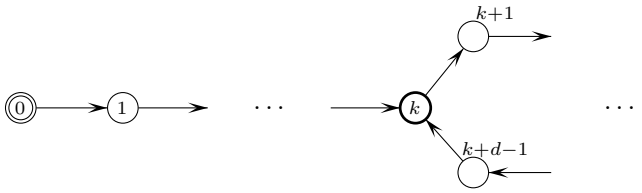


Рис. 3.6. *Построение ДКА по периодическому множеству*

С учетом очевидной импликации $2 \Rightarrow 1$ теорема доказана. □

Одно из важных следствий этой теоремы мы получим в следующем параграфе.

§ 17. Операции над КС-языками

Параграф посвящен вопросу о замкнутости класса КС-языков относительно операций (необходимые определения см. в гл. 1). Вначале докажем замкнутость этого класса относительно подстановок; приводимая ниже теорема позволяет получить ряд следствий о замкнутости класса относительно других операций.

Теорема 3.3. *Пусть τ – подстановка из произвольного конечного алфавита Σ в произвольный конечный алфавит Δ такая, что для любой буквы $\mathtt{a} \in \Sigma$ язык $\tau(\mathtt{a})$ является контекстно-свободным. Тогда если $L \in \Sigma^*$ – КС-язык, то $\tau(L)$ также является КС-языком.*

Доказательство. Пусть $\Sigma = \{\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n\}$, язык L порождается грамматикой $G = (\Sigma, \Gamma, P, S)$ и для каждого $i = 1, \ldots, n$ язык $\tau(\mathtt{a}_i)$

порождается грамматикой $G_i = (\Delta, \Gamma_i, P_i, S_i)$. Без ограничения общности можно считать, что множества нетерминалов рассматриваемых грамматик попарно не пересекаются.

Возьмем грамматику $H = (\Delta, \Gamma', P', S)$, где $\Gamma' = \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_n$, а множество правил P' определено следующим образом. Каждому правилу вывода $A \to \beta$ грамматики G поставим в соответствие правило $A \to \beta'$, где цепочка β' получается из цепочки β заменой каждого символа $\mathtt{a}_i$ на символ S_i, т. е. на аксиому грамматики G_i $(1 \le i \le n)$. Положим

$$P' = P_1 \cup \ldots \cup P_n \cup \{A \to \beta' \mid (A \to \beta) \in P\} .$$

Докажем, что $L(H) = \tau(L)$. Вначале предположим, что $w \in L(H)$ и T – какое-нибудь дерево вывода цепочки w в H (рис. 3.7). Если внутренний узел x дерева помечен нетерминалом из Γ_i, то все его потомки получены применением правил грамматики G_i, т. е. их метки принадлежат $\Gamma_i \cup \Delta$. Поскольку корень дерева T помечен нетерминалом из Γ, то в T имеется стандартное поддерево T_1, все внутренние узлы которого имеют метки из Γ, а все листья – метки из $\bigcup_{i=1}^{n} \Gamma_i \cup \Delta$. Но если $A \Rightarrow_H \alpha X \beta$, где $A \in \Gamma$, $X \notin \Gamma$, то X является аксиомой одной из грамматик G_i по определению P'. Следовательно, все листья дерева T_1 помечены аксиомами грамматик G_i, а значит, T_1 представляет вывод $S \Rightarrow_H^* S_{i_1} \ldots S_{i_k}$. Из определения P' получаем, что $S \Rightarrow_G^* \mathtt{a}_{i_1} \ldots \mathtt{a}_{i_k}$, т. е. $u = \mathtt{a}_{i_1} \ldots \mathtt{a}_{i_k} \in L$. Далее, имеем $S \Rightarrow_H^* S_{i_1} \ldots S_{i_k} \Rightarrow_H^* w_1 \ldots w_k = w$, где $S_{i_j} \Rightarrow_{G_{i_j}}^* w_j$, т. е. $w_j \in \tau(\mathtt{a}_{i_j})$. В итоге $w \in \tau(\mathtt{a}_{i_1}) \ldots \tau(\mathtt{a}_{i_k}) = \tau(u) \in \tau(L)$.

Теперь докажем обратное включение: пусть $w \in \tau(L)$. Возьмем цепочку $u = \mathtt{a}_{i_1} \ldots \mathtt{a}_{i_k} \in L(G)$ такую, что $w \in \tau(u) = \tau(\mathtt{a}_{i_1}) \ldots \tau(\mathtt{a}_{i_k})$. Можно записать $w = w_1 \ldots w_k$, где для каждого j цепочка w_j принадлежит $\tau(\mathtt{a}_{i_j})$. Но тогда $S \Rightarrow_G^* u$ и $S_{i_j} \Rightarrow_{G_{i_j}}^* w_j$ для всех j. Отсюда $S \Rightarrow_H^* S_{i_1} \ldots S_{i_k} \Rightarrow_H^* w_1 \ldots w_k = w$, т. е. $w \in L(H)$. □

В § 2 показано, что объединение, произведение и итерация языков получаются применением подстановки $\tau(\mathtt{a}_1) = L_1$, $\tau(\mathtt{a}_2) = L_2$ к (КС-)языкам $\{\mathtt{a}_1, \mathtt{a}_2\}$, $\{\mathtt{a}_1\mathtt{a}_2\}$ и $\{\mathtt{a}_1\}^*$ соответственно. Таким образом, получаем

Следствие 3.1. *Класс КС-языков замкнут относительно объединения, произведения и итерации.*

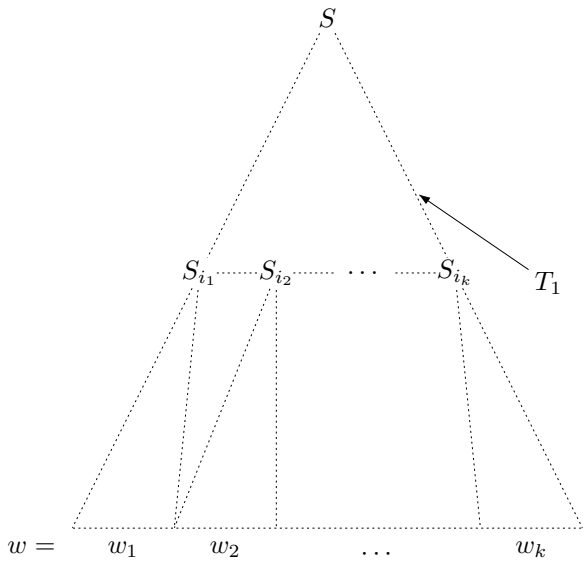


Рис. 3.7. *Дерево вывода T цепочки w в грамматике H*

Произвольный гомоморфизм из Σ^* в Δ^* можно рассматривать как подстановку τ такую, что для любого $\mathtt{a} \in \Sigma$ язык $\tau(\mathtt{a})$ состоит ровно из одной цепочки над Δ, т. е. является КС-языком. Значит, справедливо

Следствие 3.2. *Класс КС-языков замкнут относительно перехода к гомоморфным образам.*

Из следствия 3.2 можно заключить, что язык $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \in \mathbb{N}\}$ не контекстно-свободен, так как согласно примеру 3.7 его гомоморфный образ $\{\mathtt{a}^n\mathtt{b}^n\mathtt{a}^n \mid n \in \mathbb{N}\}$ не является КС-языком.

Объединяя последнее следствие с теоремой 3.2, получаем

Следствие 3.3. *Если L – КС-язык, то множество $\{|w| \mid w \in L\}$ является периодическим.*

Доказательство. Рассмотрим гомоморфизм f, который каждую букву, встречающуюся в словах из L, отображает в букву $\mathtt{a}$. Язык $f(L)$ контекстно-свободен по следствию 3.2. Поскольку f сохраняет длины слов, требуемое утверждение немедленно вытекает из теоремы 3.2. □

Предложение 3.1. *Класс КС-языков не замкнут относительно пересечения и дополнения.*

Доказательство. Рассмотрим языки $L_1 = \{\mathtt{a}^n\mathtt{b}^n\mathtt{a}^m \mid n, m \in \mathbb{N}\}$ и $L_2 = \{\mathtt{a}^m\mathtt{b}^n\mathtt{a}^n \mid n, m \in \mathbb{N}\}$. Эти языки являются контекстно-свободными по следствию 3.1, так как представляют собой произведения КС-языка $\{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$ на КС-язык $\mathtt{a}^*$ (соответственно слева и справа). Но пересечением этих языков является язык $\{\mathtt{a}^n\mathtt{b}^n\mathtt{a}^n \mid n \in \mathbb{N}\}$, который не принадлежит классу КС-языков (см. пример 3.7). Итак, класс КС-языков не замкнут относительно пересечения. Пересечение выражается через объединение и дополнение: $L_1 \cap L_2 = \overline{\overline{L}_1 \cup \overline{L}_2}$. Замкнутость относительно объединения доказана ранее, следовательно, класс КС-языков не замкнут относительно дополнения. □

Тем не менее для КС-языков выполняется следующий ограниченный вариант замкнутости относительно пересечения.

Теорема 3.4. *Пересечение КС-языка с регулярным языком является КС-языком.*

Доказательство. Пусть K – КС-язык, порождаемый грамматикой $G = (\Sigma, \Gamma, P, S)$, а M – регулярный язык, распознаваемый детерминированным конечным автоматом $\mathcal{A} = (Q, \Sigma, \delta, \mathtt{q}_0, Q_f)$. Для каждого состояния $\mathtt{f} \in Q_f$ рассмотрим ДКА $\mathcal{A}_\mathtt{f} = (Q, \Sigma, \delta, \mathtt{q}_0, \{\mathtt{f}\})$. Очевидно, что $M = L(\mathcal{A}) = \bigcup\limits_{\mathtt{f}\in Q_f} L(\mathcal{A}_\mathtt{f})$, откуда

$$K \cap M = K \cap \bigcup_{\mathtt{f}\in Q_f} L(\mathcal{A}_\mathtt{f}) = \bigcup_{\mathtt{f}\in Q_f} K \cap L(\mathcal{A}_\mathtt{f}) \ .$$

Поскольку класс КС-языков замкнут относительно объединения, достаточно доказать теорему для случая, когда Q_f состоит из единственного состояния. Положим $Q_f = \{\mathtt{f}\}$.

Для доказательства нам понадобится один результат о преобразованиях грамматик из гл. 4, а именно теорема 4.1 (§ 21). В силу этой теоремы достаточно ограничиться рассмотрением случая, когда грамматика G не содержит правил вывода вида $A \to \varepsilon$.

Для порождения языка $K \cap M$ рассмотрим контекстно-свободную грамматику $H = (\Sigma, \Gamma', P', S')$, где $\Gamma' = Q \times (\Gamma \cup \Sigma) \times Q$, $S' = (\mathtt{q}_0, S, \mathtt{f})$, а множество P' состоит из правил двух типов:

1) если $(A \to X_1X_2\ldots X_k) \in P$, то для каждого набора состояний $\mathtt{q}, \mathtt{r}, \mathtt{p}_1, \ldots, \mathtt{p}_{k-1} \in Q$ множество P' содержит правило вывода $(\mathtt{q}, A, \mathtt{r}) \to (\mathtt{q}, X_1, \mathtt{p}_1)(\mathtt{p}_1, X_2, \mathtt{p}_2)\ldots(\mathtt{p}_{k-1}, X_k, \mathtt{r})$;

2) если $\mathtt{a} \in \Sigma$ и $\delta(\mathtt{q}, \mathtt{a}) = \mathtt{r}$, то P' содержит правило $(\mathtt{q}, \mathtt{a}, \mathtt{r}) \to \mathtt{a}$.

Поскольку правила второго вида порождают только листья дерева вывода, то можно считать, что при выводе терминальной цепочки из аксиомы грамматики H вначале выполняются правила первого вида, а затем – правила второго вида, т. е. вывод имеет вид

$$(\mathtt{q}_0, S, \mathtt{f}) \Rightarrow_H^* (\mathtt{q}_0, \mathtt{a}_1, \mathtt{p}_1)(\mathtt{p}_1, \mathtt{a}_2, \mathtt{p}_2)\ldots(\mathtt{p}_{k-1}, \mathtt{a}_k, \mathtt{f}) \Rightarrow_H^* \mathtt{a}_1\mathtt{a}_2\ldots\mathtt{a}_k\ .$$

По определению грамматики H первая из стрелок в этом выводе имеет место тогда и только тогда, когда $S \Rightarrow_G^* \mathtt{a}_1\mathtt{a}_2\ldots\mathtt{a}_k$, т. е. $\mathtt{a}_1\mathtt{a}_2\ldots\mathtt{a}_k \in K$. Вторая из стрелок выполнена тогда и только тогда, когда $\delta(\mathtt{q}_0, \mathtt{a}_1\mathtt{a}_2\ldots\mathtt{a}_k) = \mathtt{f}$, т. е. $\mathtt{a}_1\mathtt{a}_2\ldots\mathtt{a}_k \in M$. В итоге терминальная цепочка выводима в H тогда и только тогда, когда она принадлежит пересечению языков K и M. Получаем $K \cap M = L(H)$, что и требовалось. □

В заключение параграфа приведем теорему, которая является аналогом теоремы Клини для регулярных языков. Предварительно сформулируем определение, обобщающее определение правильного скобочного языка LB.

Определение. Язык над алфавитом $\{\mathtt{a}_1, \ldots, \mathtt{a}_n, \mathtt{b}_1, \ldots, \mathtt{b}_n\}$, порождаемый грамматикой $\{S \to \varepsilon \mid \mathtt{a}_1S\mathtt{b}_1S \mid \ldots \mid \mathtt{a}_nS\mathtt{b}_nS\}$, называется *языком Дика*.

Теорема 3.5. *Для любого КС-языка L над алфавитом Σ найдутся алфавит Δ, регулярный язык R и язык Дика D над Δ такие, что $L = \phi(R \cap D)$, где ϕ – некоторый гомоморфизм из Δ^* в Σ^*.*

Доказательство этой теоремы можно найти в книге [Ги]. Из теоремы 3.4 и следствия 3.2 немедленно следует и обратное утверждение – о том, что гомоморфный образ пересечения регулярного языка и языка Дика является КС-языком. Следовательно, *любой КС-язык может быть получен из конечных языков и языков Дика с помощью объединения, произведения, итерации, пересечения с регулярными языками и перехода к гомоморфным образам.*

§ 18. КС-грамматики и МП-автоматы

Естественно поставить вопрос о сравнении класса КС-языков и класса языков, распознаваемых МП-автоматами. Ответ на этот вопрос частично содержится в следующем утверждении.

Теорема 3.6 (о распознавании КС-языков). *Для любого КС-языка существует распознающий его НМПА с единственным состоянием и единственной командой допуска* $(\dashv, \nabla) \to \checkmark$.

Доказательство. Поскольку искомый МП-автомат имеет только одно состояние, в записи его команд и конфигураций состояние указывать не будем. Пусть $G = (\Sigma, \Gamma, P, S)$ – КС-грамматика, порождающая язык L, а $\mathcal{M}$ – НМПА с единственным состоянием, основным алфавитом Σ, стековым алфавитом $\Gamma \cup \Sigma$, начальным содержимым стека S и системой команд δ, состоящей из команд трех типов:

(1) для любых $\mathtt{a} \in \Sigma$, $B \in \Gamma$ и любого правила $(B \to \gamma) \in P$ есть команда $(\mathtt{a}, B) \to (\gamma, \_)$;

(2) для любого $\mathtt{a} \in \Sigma$ есть команда $(\mathtt{a}, \mathtt{a}) \to (\varepsilon, {}^{\rightarrow})$;

(3) есть команда допуска $(\dashv, \nabla) \to \checkmark$.

Покажем, что $\mathcal{M}$ распознает язык L. Пусть $w \in L$, тогда в G существует левосторонний вывод

$$S \Rightarrow u_1 B_1 \alpha_1 \Rightarrow u_1 u_2 B_2 \alpha_2 \Rightarrow \ldots \Rightarrow u_1 ... u_{n-1} B_{n-1} \alpha_{n-1} \Rightarrow u_1 ... u_n = w,$$

где $u_i \in \Sigma^*$, $B_i \in \Gamma$, $\alpha_i \in (\Gamma \cup \Sigma)^*$ для всех i. Тогда в автомате $\mathcal{M}$ можно реализовать такую последовательность конфигураций:

$$[w, S] = [u_1 u_2 \ldots u_n, S] \models [u_1 u_2 \ldots u_n, u_1 B_1 \alpha_1] \models^* [u_2 \ldots u_n, B_1 \alpha_1] \models^* \ldots$$
$$\models^* [u_n, B_{n-1} \alpha_{n-1}] \models [u_n, u_n] \models^* [\varepsilon, \varepsilon] \ .$$

В конфигурации $[\varepsilon, \varepsilon]$ применима команда допуска, т. е. $w \in L(\mathcal{M})$.

Теперь предположим, что $w \in L(\mathcal{M})$. Поскольку команда допуска единственна, из конфигурации $[w, S]$ можно перейти в конфигурацию $[\varepsilon, \varepsilon]$ за конечное число шагов. Обозначим это число через m и выберем $\mathtt{a}_1, \ldots, \mathtt{a}_m \in \Sigma \cup \{\varepsilon\}$ так, что $w = \mathtt{a}_1 \ldots \mathtt{a}_m$, и

$$[w, S] = [\mathtt{a}_1 \ldots \mathtt{a}_m, \gamma_1] \models [\mathtt{a}_2 \ldots \mathtt{a}_m, \gamma_2] \models \ldots \models [\mathtt{a}_m, \gamma_m] \models [\varepsilon, \varepsilon],$$

где $\gamma_1, \ldots, \gamma_m$ – некоторые строки стековых символов, причем $\gamma_1 = S$. Из определения команд автомата $\mathcal{M}$ следует, что

– каждая строка γ_i непуста (иначе невозможен i-й шаг);

– если на i-м шаге применяется команда вида (2), то γ_i начинается с терминала $\mathtt{a}_i$, причем $\gamma_i = \mathtt{a}_i\gamma_{i+1}$;

– если на i-м шаге применяется команда вида (1), то γ_i начинается с нетерминала, $\mathtt{a}_i = \varepsilon$, и если $\gamma_i = B_i\beta_i$, то $\gamma_{i+1} = \alpha_i\beta_i$, где $(B_i \to \alpha_i) \in P$.

Пусть команды вида (1) применялись только на шагах с номерами $k_1, \ldots, k_n$, где $1 = k_1 < \ldots < k_n$. Тогда можно построить вывод в грамматике G:

$$S \Rightarrow \alpha_1 = \mathtt{a}_1 \ldots \mathtt{a}_{k_2-1} B_{k_2}\beta_{k_2} \Rightarrow$$
$$\Rightarrow \mathtt{a}_1 \ldots \mathtt{a}_{k_2-1}\alpha_{k_2}\beta_{k_2} = \mathtt{a}_1 \ldots \mathtt{a}_{k_2-1}\mathtt{a}_{k_2} \ldots \mathtt{a}_{k_3-1} B_{k_3}\beta_{k_3} \Rightarrow \ldots \Rightarrow$$
$$\Rightarrow \mathtt{a}_1 \ldots \mathtt{a}_{k_{n-1}-1}\alpha_{k_{n-1}}\beta_{k_{n-1}} = \mathtt{a}_1 \ldots \mathtt{a}_{k_n-1} B_{k_n}\beta_{k_n} \Rightarrow$$
$$\Rightarrow \mathtt{a}_1 \ldots \mathtt{a}_{k_n-1}\alpha_{k_n}\beta_{k_n} = \mathtt{a}_1 \ldots \mathtt{a}_m = w \ .$$

Следовательно, $w \in L$. Теорема доказана. □

Справедлива и существенно более сложная обратная теорема.

Теорема 3.7. *Язык, распознаваемый НМПА, является контекстно-свободным.*

Доказательство этой теоремы приводится, например, в [Ги] и [Kar]. Идея конструирования грамматики по автомату напоминает идею доказательства теоремы 3.4, но ее реализация гораздо более трудоемка.

Из теорем 3.6 и 3.7 вытекает

Следствие 3.4. *Класс языков, распознаваемых НМПА, совпадает с классом контекстно-свободных языков.*

Принимая во внимание предложение 2.3, получаем

Следствие 3.5. *Класс языков, распознаваемых ДМПА, является собственным подклассом класса контекстно-свободных языков.*

Характеристика этого собственного подкласса в терминах грамматик буден дана в ч. 2, гл. 4.

§ 19. КС-грамматики и языки программирования

В этом параграфе мы рассмотрим вопрос об описании синтаксиса языков программирования с помощью КС-грамматик. Ответ

на этот вопрос, вообще говоря, несложен: традиционно синтаксис языков программирования описывается так называемыми *формулами Бэкуса–Наура*, которые фактически являются правилами вывода контекстно-свободной грамматики. Формулы Бэкуса–Наура отличаются от правил вывода контекстно-свободной грамматики лишь тем, что в роли «стрелки» в правилах вывода используется знак ::= и что используются фигурные скобки, означающие повторение нуль или более раз. Формула вида $A ::= \{\beta\}$ очевидно заменяет собой два правила вывода: $A \to \beta$ и $A \to \varepsilon$. Подобное описание языка Паскаль можно найти в [ЙВ], языков C^{++} и C$^\sharp$ – в [АСУ].

Обозначим буквой G КС-грамматику языка программирования (например, Паскаля), построенную по формулам Бэкуса–Наура. Всякая синтаксически правильная программа, написанная на Паскале, выводима в грамматике G. *Обратное утверждение неверно.* Дело в том, что синтаксис фактически любого языка программирования, в том числе Паскаля, содержит ряд *контекстных* условий, которые не могут быть выражены правилами КС-грамматик (или формулами Бэкуса–Наура).

В следующем примере приведены несколько контекстных условий для языка Паскаль, аналоги которых есть во многих языках программирования.

Пример 3.9. 1. *Совпадение типов.* В операторе присваивания `<переменная>:=<выражение>` требуется, чтобы тип переменной совпадал с типом выражения. Цепочка

`program a(b); var x : boolean; begin x := 3.14 end.`

будет выводима в грамматике G, но не является синтаксически правильной программой. Можно показать, что данное синтаксическое условие невыразимо правилами вывода никакой КС-грамматики, т. е. является контекстным условием.

2. *Описание имен.* В Паскале требуется существование и единственность описания каждого используемого имени. В то же время в грамматике G будут выводимы цепочки вида

`program a(b); var x : real; begin y := 3.14 end.`
`program a(b); var x : real; var x : boolean; begin x := 3.14 end.`

Упростим ситуацию и рассмотрим цепочку вида $w\mathtt{c}w$. Первое вхождение цепочки w в $w\mathtt{c}w$ можно мыслить как объявление имени, а

второе – как его использование (`c` – это все, что находится в программе между объявлением и использованием данного имени). В примере 3.8 было доказано, что язык $L = \{w\mathtt{c}w \mid w \in \{\mathtt{a},\mathtt{b}\}^*\}$ не является КС-языком. Этот пример подтверждает, что требования описания имен являются контекстными.

3. *Корректность вызова.* Необходимо соответствие формальных параметров в описании процедуры и фактических параметров в операторе вызова процедуры. Это условие также является контекстным. Моделирующий его не контекстно-свободный язык читателю предлагается придумать самостоятельно.

Как уже отмечалось, приведенные выше примеры синтаксических условий не могут быть выражены средствами КС-грамматик. При расширении множества используемых грамматик до класса контекстно-зависимых грамматик, способного выразить используемые контекстные условия, возникают проблемы ввиду отсутствия эффективного распознавателя. Поэтому при проектировании компиляторов разработчики довольно часто поступают следующим образом. Основная часть синтаксиса описывается КС-грамматикой, и применяются инструментальные средства, автоматизирующие разработку *синтаксического анализатора* (фактически МП-автомата) для проверки соответствия программы заложенным в грамматику требованиям языка. Синтаксическим анализаторам – одной из важнейших частей компилятора – посвящен основной объем материала ч. 2 этой книги. Контекстные условия анализируются на следующей стадии компиляции, стадии семантических проверок. Этой стадии посвящена последняя, третья часть книги.

Задачи

1. КС-грамматика называется *праволинейной*, если каждое ее правило вывода имеет один из трех видов: $A \to \mathtt{x}B$, $A \to \mathtt{x}$, $A \to \varepsilon$, где A и B – нетерминалы, `x` – терминал. Найти праволинейные грамматики, порождающие языки, состоящие из

а) непустых цепочек, составленных из букв латиницы и десятичных цифр и начинающихся с буквы;

б) непустых цепочек длины не более 6, составленных из букв латиницы и десятичных цифр и начинающихся с букв `i` и `j`;

в) двоичных цепочек, имеющих четную длину;

г) двоичных цепочек, имеющих четное число и нулей, и единиц.

2*. Доказать, что класс языков, порождаемых праволинейными грамматиками (см. задачу 1), совпадает с классом регулярных языков.

3. Найти КС-грамматики, порождающие следующие языки:

а) множество цепочек, составленных из равного количества нулей и единиц;

б) $\{\mathtt{a}^n\mathtt{b}^m \mid n, m \in \mathbb{N}_0, n \le m \le 2n\}$;

в) множество формул логики высказываний;

г)* $\{\mathtt{a}^n\mathtt{b}^m\mathtt{c}^k \mid n, m, k \in \mathbb{N}_0,\ n \ne m$ или $m \ne k\}$;

д)* $\{uv \mid u, v \in \{\mathtt{a}, \mathtt{b}\}^*,\ |u| = |v|, u \ne v\}$.

4. Найти грамматики, порождающие следующие языки:

а) $\{\mathtt{a}^n\mathtt{b}^{n^2} \mid n \in \mathbb{N}_0\}$;

б) множество десятичных записей квадратов натуральных чисел;

в)* $\{ww \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$;

г) множество цепочек, составленных из равного количества букв a, b и c.

5. Доказать, что КС-грамматика $\{S \to \mathtt{a}A\mathtt{b}, \mathtt{a}A \to \mathtt{aa}A\mathtt{b}, A \to \varepsilon\}$ порождает язык $\{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$.

6. Какой язык порождается грамматикой:

а) $\{S \to \mathtt{a}S\mathtt{a} \mid \mathtt{b}S\mathtt{b} \mid \mathtt{c}\}$;

б) $\{S \to S\mathtt{ab}, S\mathtt{a} \to \mathtt{a}D, D\mathtt{b} \to \mathtt{b}\}$;

в) $\{S \to \mathtt{a}SS \mid \mathtt{a}\}$;

г) $\{S \to \mathtt{a}SD\mathtt{b}, SD \to DS, S\mathtt{b} \to \varepsilon, DD \to \varepsilon\}$;

д) $\{S \to \mathtt{b}SS \mid \mathtt{a}\}$;

е) $\{S \to \mathtt{ab}S\mathtt{b} \mid \mathtt{a}A, A \to \mathtt{b}A \mid \mathtt{a}\}$;

ж) $\{S \to SS \mid \mathtt{1}A\mathtt{0}, A \to \mathtt{1}A\mathtt{0} \mid \varepsilon\}$;

з)* $\{S \to CD, C \to \mathtt{a}CA \mid \mathtt{b}CB \mid \varepsilon, D \to \varepsilon, AD \to \mathtt{a}D, BD \to \mathtt{b}D,$
$A\mathtt{a} \to \mathtt{a}A, A\mathtt{b} \to \mathtt{b}A, B\mathtt{a} \to \mathtt{a}B, B\mathtt{b} \to \mathtt{b}B\}$?

7. Какие из цепочек aab, aaabb, aabb, aabbb, aabbbab можно вывести в грамматике $\{S \to \mathtt{a}A \mid B\mathtt{b}, A \to \mathtt{a}A \mid C, B \to B\mathtt{b} \mid C, C \to \mathtt{a}C\mathtt{b} \mid \varepsilon\}$? Если цепочка выводима, построить вывод и дерево вывода.

8. Какие из цепочек aacaaacbca, caab, aaacabbc можно вывести в грамматике $\{S \to \mathtt{a}A\mathtt{c}B \mid B\mathtt{d}S, A \to BaB \mid \mathtt{a}B\mathtt{c} \mid \mathtt{a}, B \to \mathtt{a}S\mathtt{c}A \mid \mathtt{c}AB \mid \mathtt{b}\}$? Если цепочка выводима, построить вывод и дерево вывода.

9. Выводимая в КС-грамматике G цепочка bbaa имеет дерево вывода, изображенное на рис. 3.8. Найти левый вывод, соответствующий этому дереву. Сколько всего выводов ему соответствует?

10. Пусть G – КС-грамматика, не содержащая правил вида $A \to \varepsilon$.

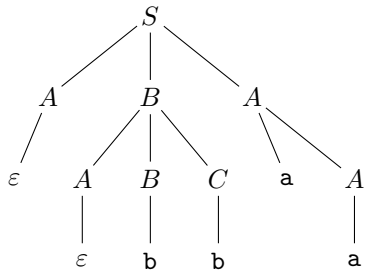


Рис. 3.8. *Дерево вывода (к задаче 9)*

Доказать, что вывод длины m цепочки длины n в грамматике G представляется деревом с $(n+m)$ узлами.

11. Узел дерева вывода назовем *аннулирующим*, если все листья – потомки данного узла – имеют метку ε. Пусть G – произвольная КС-грамматика, $w \in L(G)$ и $|w| = n$. Если длина вывода цепочки w равна m, то дерево этого вывода содержит не более $(n+2m-1)$ неаннулирующих узлов.

12. Показать, что грамматика является неоднозначной:

а) $G = \{S \to \mathtt{a}S \,|\, \mathtt{a}S\mathtt{b}S \,|\, \varepsilon\}$;

б) $G = \{S \to A\mathtt{1}B, A \to \mathtt{0}A \,|\, \varepsilon, B \to \mathtt{0}B \,|\, \mathtt{1}B \,|\, \varepsilon\}$.

Найти однозначную грамматику, порождающую язык $L(G)$.

13. Доказать, что грамматика $G = \{E \to +EE \,|\, *EE \,|\, \mathtt{x}\}$ однозначна.

14. Построить КС-грамматику, порождающую множество регулярных выражений с учетом установленных приоритетов операций.

15. Построить КС-грамматику для условных выражений языка Паскаль.

16. Будут ли следующие языки контекстно-свободными:

а) $\{\mathtt{a}^n\mathtt{b}^m\mathtt{c}^k \,|\, 0 < n < m < k\}$;
б) $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^k \,|\, 0 < k < n\}$;
в) $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^k \,|\, 0 \le n < k < 2n\}$;
г) $\{\mathtt{a}^p \,|\, p$ – простое число$\}$;
д) $\{\mathtt{a}^n\mathtt{b}^{n^2} \,|\, n \in \mathbb{N}\}$;
е) $\{\mathtt{a}^n\mathtt{b}^m\mathtt{a}^{2n} \,|\, m, n \in \mathbb{N}\}$;
ж) $\{\mathtt{a}^n\mathtt{b}^m\mathtt{a}^m\mathtt{b}^n \,|\, m, n \in \mathbb{N}\}$;
з) $\{\mathtt{a}^n\mathtt{b}^m\mathtt{a}^n \,|\, 0 < n \le m\}$?

17. Доказать, что язык $\mathtt{a}^* \cup \mathtt{b}^*$ не порождается КС-грамматикой, у которой множество нетерминалов состоит только из аксиомы.

18*. Доказать, что класс КС-языков замкнут относительно операций $\mathtt{a}\backslash L$, $L\backslash\mathtt{a}$, $init(L)$ и не замкнут относительно операции $min(L)$. (Определение этих операций приведено в § 2.)

19*. Пусть $perm(w)$ – множество цепочек, полученных всевозможными перестановками букв в цепочке w, и $perm(L) = \{perm(w) \mid w \in L\}$. Привести пример регулярного языка L в алфавите $\{\mathtt{a}, \mathtt{b}\}$ такого, что $perm(L)$ – нерегулярный язык. Привести пример регулярного языка L в алфавите $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ такого, что язык $perm(L)$ не является контекстно-свободным. Доказать, что для любого регулярного языка L в алфавите $\{\mathtt{a}, \mathtt{b}\}$ язык $perm(L)$ является КС-языком.

20. В табл. 3.1, *а* приведена управляющая таблица МП-автомата $\mathcal{M}$ с одним состоянием. Начальное содержимое стека равно A. Построить КС-грамматику, порождающую язык $L(\mathcal{M})$.

21. В табл. 3.1, *б* приведена управляющая таблица МП-автомата $\mathcal{M}$ с одним состоянием. Начальное содержимое стека равно B. Построить КС-грамматику, порождающую язык $L(\mathcal{M})$.

Таблица 3.1. *Управляющие таблицы МП-автоматов для задач 20, 21*

	a	b	⊣
A	$AA, \rightarrow$	$\varepsilon \rightarrow$	
∇			✓

а

	a	b	⊣
B	$\varepsilon \rightarrow$	$BB, \rightarrow$	
∇	$B, \rightarrow$		✓

б

Глава 4. Преобразования грамматик

Контекстно-свободный язык может порождаться многими КС-грамматиками. Естественно поэтому ставить вопрос о том, существует ли грамматика, порождающая данный язык и удовлетворяющая определенным ограничениям. Так как язык, как правило, задается некоторой исходной грамматикой, этот вопрос уточняется следующим образом: можно ли преобразовать исходную грамматику в грамматику с этими ограничениями и если можно, то как это сделать. В данной главе изучаются некоторые преобразования, имеющие практическое значение.

§ 20. Приведенные грамматики

Мы рассматриваем грамматики как способ задания языка, т. е. задания множества цепочек над основным алфавитом, выводимых

из аксиомы. Естественно, мы интересуемся эффективностью такого задания, в частности стараемся уменьшить размер грамматики (с сохранением желаемых свойств). Если грамматика содержит нетерминальный символ, из которого не выводима ни одна цепочка над основным алфавитом, то этот символ является бесполезным в плане порождения языка грамматикой. Так же бесполезен символ, который не встречается ни в одной цепочке, выводимой из аксиомы. Целью данного параграфа является алгоритм устранения бесполезных символов из грамматики. Дадим соответствующие определения.

Определение. Нетерминальный символ называется *производящим*, если из него выводима цепочка над основным алфавитом.

Например, для грамматики $G_{4,1} = \{S \to \mathtt{a}S\mathtt{Ac} \mid B\mathtt{a}C, A \to \mathtt{abc}, B \to A\mathtt{d}, C \to BD\mathtt{e} \mid \mathtt{c}, D \to EC\mathtt{a}, E \to AE\}$ символ S является производящим, так как $S \Rightarrow B\mathtt{a}C \Rightarrow A\mathtt{da}C \Rightarrow \mathtt{abcda}C \Rightarrow \mathtt{abcdac}$. Символы D и E производящими не являются.

Алгоритм 4.1. *Нахождение производящих символов грамматики.*
Вход. Контекстно-свободная грамматика $G = (\Sigma, \Gamma, P, S)$.
Выход. Множество Γ^p производящих символов грамматики G.

1. $i \leftarrow 1$; $\Gamma_0 \leftarrow \emptyset$; $\Gamma_1 \leftarrow \{A \in \Gamma \mid \exists \alpha \in \Sigma^* : (A \to \alpha) \in P\}$
2. **пока** $(\Gamma_{i-1} \subset \Gamma_i)$ **выполнять**
3. $\quad \Gamma_{i+1} \leftarrow \{A \in \Gamma \mid \exists \alpha \in (\Sigma \cup \Gamma_i)^* : (A \to \alpha) \in P\}$
4. $\quad i \leftarrow i + 1$
5. $\Gamma^p \leftarrow \Gamma_i$

Каждое из множеств Γ_i содержит только производящие символы грамматики, а каждый производящий символ по определению должен содержаться во множестве Γ_n для некоторого натурального n. Поскольку равенство $\Gamma_i = \Gamma_{i+1}$ влечет $\Gamma_i = \Gamma_n$ для всех $n > i$, алгоритм 4.1 работает корректно.

Применив алгоритм 4.1 к грамматике $G_{4,1}$, получим $\Gamma_1 = \{A, C\}$, $\Gamma_2 = \{A, C, B\}$, $\Gamma_3 = \{A, C, B, S\}$, $\Gamma_4 = \Gamma_3$. Следовательно, множество производящих символов грамматики $G_{4,1}$ равно $\{A, C, B, S\}$.

Определение. Нетерминальный символ называется *достижимым*, если он является аксиомой или содержится в некоторой цепочке, выводимой из аксиомы.

К примеру, в грамматике $G_{4,2} = \{S \to b\mathtt{A}\mathtt{c} \mid \mathtt{A}\mathtt{c}B, A \to \mathtt{abc}, B \to E\mathtt{a}, C \to BD\mathtt{e}, D \to BC\mathtt{a}, E \to F\mathtt{bb}, F \to \mathtt{a}\}$ символ F явля-

ется достижимым, так как $S \Rightarrow AcB \Rightarrow AcEa \Rightarrow AcFbba$. Символы C и D достижимыми не являются.

Алгоритм 4.2. *Нахождение достижимых символов грамматики.*
Вход. Контекстно-свободная грамматика $G = (\Sigma, \Gamma, P, S)$.
Выход. Множество Γ^r достижимых символов грамматики G.

1. $i \leftarrow 1$; $\Gamma_0 \leftarrow \emptyset$; $\Gamma_1 \leftarrow \{S\}$
2. **пока** $(\Gamma_{i-1} \subset \Gamma_i)$ **выполнять**
3. $\quad \Gamma_{i+1} \leftarrow \Gamma_i \cup \{A \in \Gamma \mid \exists B \in \Gamma_i, \alpha, \beta \in (\Sigma \cup \Gamma)^* : (B \to \alpha A \beta) \in P\}$
4. $\quad i \leftarrow i + 1$
5. $\Gamma^r \leftarrow \Gamma_i$

Доказательство корректности алгоритма аналогично доказательству для алгоритма 4.1.

Для грамматики $G_{4,2}$ имеем $\Gamma_2 = \{S, A, B\}$, $\Gamma_3 = \{S, A, B, E\}$, $\Gamma_4 = \{S, A, B, E, F\}$, $\Gamma_5 = \Gamma_4$. Следовательно, $\Gamma^r = \{S, A, B, E, F\}$.

Перейдем к основному понятию параграфа.

Определение. Контекстно-свободная грамматика называется *приведенной*, если любой вспомогательный символ этой грамматики является достижимым и производящим.

Замечание 4.1. Грамматика является приведенной тогда и только тогда, когда *каждое ее правило участвует в выводе из аксиомы некоторой терминальной цепочки.* Действительно, если правило $A \to X_1 \dots X_n$ не участвует в выводе никакой терминальной цепочки, то либо символ A является недостижимым, либо некоторый символ X_i – непроизводящим.

Алгоритм 4.3. *Построение приведенной грамматики.*
Вход. КС-грамматика $G = (\Sigma, \Gamma, P, S)$.
Выход. Приведенная КС-грамматика G', эквивалентная G, в случае $L(G) \neq \emptyset$; «пустая» грамматика в случае $L(G) = \emptyset$.

1. **вычислить** Γ^p %% алгоритм 4.1
2. **если** $(S \notin \Gamma^p)$
3. $\quad G' \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset)$ %% $L(G) = \emptyset$
4. **иначе** $\widehat{P} \leftarrow \{(A \to \alpha) \in P \mid A, \alpha \in (\Gamma^p \cup \Sigma)^*\}$
5. $\quad \widehat{G} \leftarrow (\Gamma^p, \Sigma, \widehat{P}, S)$
6. $\quad$ **вычислить** $(\Gamma^p)^r$ в $\widehat{G}$ %% алгоритм 4.2
7. $\quad P' \leftarrow \{(A \to \alpha) \in \widehat{P} \mid A, \alpha \in ((\Gamma^p)^r \cup \Sigma)^*\}$
8. $\quad G' \leftarrow (\Sigma, (\Gamma^p)^r, P', S)$

Предложение 4.1. *Алгоритм 4.3 работает корректно.*

Доказательство. Во-первых, отметим, что из определения производящего символа немедленно следует, что $L(G) = \emptyset$ тогда и только тогда, когда аксиома не является производящим символом. Эквивалентность перехода от грамматики G к грамматике G' в случае $L(G) \neq \emptyset$ очевидна, так как устраняемые непроизводящие и недостижимые символы не влияют на множество выводимых в грамматике цепочек. Осталось показать, что все нетерминалы грамматики G' – производящие. В самом деле, любой нетерминал $A \in (\Gamma^p)^r$ является производящим в грамматиках G и $\widehat{G}$ и достижимым в грамматиках $\widehat{G}$ и G'. В частности, найдется цепочка $w_A \in \Sigma^*$ такая, что $S \Rightarrow^*_{\widehat{G}} A \Rightarrow^*_{\widehat{G}} w_A$. Но тогда в выводе $A \Rightarrow^*_{\widehat{G}} w_A$ недостижимые символы грамматики $\widehat{G}$ не могут встретиться по определению. Следовательно, $A \Rightarrow^*_{G'} w_A$, т. е. A – производящий символ в G'. □

Замечание 4.2. Порядок применения алгоритмов 4.1 и 4.2 в алгоритме 4.3 является существенным, как показывает пример 4.1.

Пример 4.1. Возьмем грамматику $G_{4,3} = \{S \to \mathtt{ab} \mid \mathtt{b}A\mathtt{c}, A \to \mathtt{c}B, B \to \mathtt{a}SA, C \to \mathtt{b}C \mid \mathtt{d}\}$. Ее производящими символами будут S и C. Следовательно, $\widehat{G} = \{S \to \mathtt{ab}, C \to \mathtt{b}C \mid \mathtt{d}\}$. Грамматика $\widehat{G}$ имеет только один достижимый символ S. Это означает, что G'=$\{S \to \mathtt{ab}\}$.

Если же из грамматики $G_{4,3}$ вначале удалить недостижимые символы, а потом – непроизводящие, то на первом шаге грамматика не изменится, так как все ее символы достижимы, а на втором шаге мы получим грамматику $\widehat{G}$, которая приведенной не является.

§ 21. ε-свободные грамматики

Определением контекстно-свободной грамматики разрешаются правила вывода вида $A \to \varepsilon$. Более того, без таких правил не обойтись, если порождаемый язык содержит пустую цепочку. Такие правила удобно использовать при описании различных конструкций формальных языков. Однако для некоторых алгоритмов синтаксического анализа наличие подобных правил приводит к значительным неудобствам. Поэтому необходимо уметь преобразовывать заданную КС-грамматику в эквивалентную КС-грамматику «почти» без таких правил.

Введем необходимые понятия.

Определение. Правило вывода вида $A \to \varepsilon$ называется *аннулирующим*, или *ε-правилом*. Нетерминальный символ B называется *аннулирующим*, если $B \Rightarrow^* \varepsilon$. Множество аннулирующих символов грамматики G обозначается через $Ann(G)$.

В грамматике $G_{4,4}$=$\{S \to \mathtt{a}BC \,|AE, A \to \mathtt{b}C \,|\, \varepsilon, B \to ACA, C \to \varepsilon,$ $E \to CA, D \to \mathtt{b}E \,|\, \mathtt{c}\}$ символы A, C, S являются аннулирующими. Для A и C это очевидно, для S приведем соответствующий вывод: $S \Rightarrow AE \Rightarrow E \Rightarrow CA \Rightarrow A \Rightarrow \varepsilon$. Символ D аннулирующим не является.

Алгоритм построения множества $Ann(G)$, необходимый для доказательства основного результата параграфа, идейно похож на алгоритмы 4.1 и 4.2. Проверку его корректности мы оставляем читателю.

Алгоритм 4.4. *Нахождение аннулирующих символов грамматики.*
Вход. Контекстно-свободная грамматика $G = (\Sigma, \Gamma, P, S)$.
Выход. Множество $Ann(G)$ аннулирующих символов грамматики G.

1. $i \leftarrow 1$; $Ann_0 \leftarrow \emptyset$; $Ann_1 \leftarrow \{A \in \Gamma \,|\, (A \to \varepsilon) \in P\}$
2. **пока** $(Ann_{i-1} \subset Ann_i)$ **выполнять**
3. $\quad Ann_{i+1} \leftarrow \{A \in \Gamma \,|\, \exists \alpha \in (Ann_i)^* : (A \to \alpha) \in P\}$
4. $\quad i \leftarrow i + 1$
5. $Ann(G) \leftarrow Ann_i$

Для грамматики $G_{4,4}$ имеем $Ann_1 = \{A, C\}$, $Ann_2 = \{A, C, B, E\}$, $Ann_3 = \{A, C, B, E, S\}$, $Ann_4 = Ann_3$. Следовательно, $Ann(G_{4,4}) =$ $= \{A, C, B, E, S\}$.

Определение. КС-грамматика называется ε-свободной, если она либо содержит единственное ε-правило $S \to \varepsilon$, причем аксиома S не появляется в правых частях правил вывода, либо не содержит ε-правил вообще.

Теорема 4.1. *Для любой КС-грамматики G существует эквивалентная ей ε-свободная КС-грамматика.*

Доказательство. Пусть $G = (\Sigma, \Gamma, P, S)$. Через $\beta \preceq \gamma$, где β и γ – цепочки над $(\Gamma \cup \Sigma)$, будем записывать тот факт, что цепочка β является подпоследовательностью цепочки γ такой, что все символы из γ, не принадлежащие β, являются аннулирующими. Отметим, что условие $\beta \preceq \gamma$ влечет $\gamma \Rightarrow^*_G \beta$.

Докажем, что искомая ε-свободная грамматика корректно строится следующим алгоритмом.

Алгоритм 4.5. *Построение ε-свободной грамматики.*
Вход. КС-грамматика $G = (\Sigma, \Gamma, P, S)$.
Выход. ε-свободная КС-грамматика $\widetilde{G}$, эквивалентная G.

1. **вычислить** $Ann(G)$ %% алгоритм 4.4
2. $\widetilde{P} \leftarrow \{A \to \beta \,|\, \exists\, \gamma \in (\Gamma \cup \Sigma)^* : (A \to \gamma) \in P,\ \varepsilon \neq \beta \preceq \gamma\}$
   %% для вычисления отношения $\preceq$ используется $Ann(G)$
3. **если** $(S \notin Ann(G))$
4. $\quad \widetilde{G} \leftarrow (\Gamma, \Sigma, \widetilde{P}, S)$
5. **иначе** $\widetilde{P} \leftarrow \widetilde{P} \cup (S' \to S) \cup (S' \to \varepsilon)$
6. $\quad \widetilde{G} \leftarrow (\Sigma, \Gamma \cup \{S'\}, \widetilde{P}, S')$

Грамматика $\widetilde{G}$, которая строится алгоритмом 4.5, является по построению ε-свободной, причем пустая цепочка выводима в ней тогда и только тогда, когда аксиома S является аннулирующим символом грамматики G, т. е. $S \Rightarrow_G^* \varepsilon$. Докажем, что множества непустых цепочек, выводимых в этих грамматиках, совпадают. Поскольку преобразования в строках 5–6 алгоритма 4.5 никак не влияют на множество выводимых в грамматике непустых цепочек, то можно считать, что грамматика $\widetilde{G}$ определена в строке 4 этого алгоритма.

Пусть $(A \to \beta) \in \widetilde{P}$. По определению $\widetilde{P}$, найдется цепочка γ такая, что $(A \to \gamma) \in P$ и $\beta \preceq \gamma$. Как отмечалось выше, тогда $\gamma \Rightarrow_G^* \beta$, откуда $A \Rightarrow_G^* \beta$. Используя транзитивность, заключаем, что для любых цепочек γ_1, γ_2 выводимость $\gamma_1 \Rightarrow_{\widetilde{G}}^* \gamma_2$ влечет выводимость $\gamma_1 \Rightarrow_G^* \gamma_2$. Таким образом, если $w \in L(\widetilde{G})$, то $w \in L(G)$.

Обратно, пусть w – непустая цепочка из $L(G)$. Вывод цепочки w в грамматике G представлен некоторым деревом T. «Обрежем» это дерево, удалив все поддеревья, представляющие вывод цепочки ε. Поскольку $w \neq \varepsilon$, в результате останется непустое поддерево $\widetilde{T}$, на листьях которого написана цепочка w. Покажем, что это дерево является деревом вывода в грамматике $\widetilde{G}$. Дерево $\widetilde{T}$ подходит под определение дерева вывода; пусть x – какой-нибудь его внутренний узел. Тогда сыновья этого узла в $\widetilde{T}$ образуют подпоследовательность последовательности его сыновей в T, причем все отсутствующие в $\widetilde{T}$ сыновья x помечены в T аннулирующими символами по построению. Таким образом, если метки сыновей узла x в $\widetilde{T}$ образуют цепочку β, а метки его сыновей в T – цепочку γ, то $\beta \preceq \gamma$ (и $\beta \neq \varepsilon$, так как

узел x – внутренний). Итак, мы показали, что в любом внутреннем узле дерева $\widetilde{T}$ реализуется правило вывода грамматики $\widetilde{G}$. Осталось заметить, что корень $\widetilde{T}$ совпадает с корнем T. Таким образом, нами построено дерево вывода цепочки w в грамматике $\widetilde{G}$. Теорема доказана. □

Пример 4.2. Рассмотрим грамматику

$$G_{4,5} = \{S \to BC \mid \mathtt{c}B\mathtt{d}, B \to BC \mid \mathtt{ab} \mid \varepsilon, C \to \mathtt{ac} \mid \varepsilon\}.$$

Применив алгоритмы 4.4 и 4.5, получим $Ann(G) = \{S, B, C\}$ и

$$\widetilde{G} = \{S \to BC \mid C \mid B \mid \mathtt{cBd} \mid \mathtt{cd} \mid \varepsilon, B \to BC \mid C \mid \mathtt{ab}, C \to \mathtt{ac}\}.$$

(Мы немного оптимизировали результат работы алгоритма 4.5: не стали добавлять новую аксиому, поскольку аксиома S и так не встречается в правых частях правил, ограничившись добавлением правила $S \to \varepsilon$. Кроме того, мы не стали записывать правило $B \to B$.)

Замечание 4.3. В результате применения алгоритма 4.5 размер грамматики (т. е. число правил вывода) может вырасти экспоненциально.

§ 22. Некоторые преобразования специального вида

В этом параграфе будут рассмотрены эквивалентные преобразования КС-грамматик, целью которых является более эффективное порождение и распознавание языков. В частности, рассматриваются алгоритмы, приводящие грамматику к виду, пригодному для нисходящего синтаксического анализа (см. ч. 2, гл. 2). Все рассматриваемые грамматики будем без ограничения общности считать приведенными.

Определение. Вывод $A \Rightarrow^+ A$ называется *циклическим*. КС-грамматика называется *циклической*, если в ней существует циклический вывод, и *ациклической* в противном случае.

Очевидно, цикличность является «плохим» свойством грамматики, поскольку одна и та же цепочка может иметь бесконечно много выводов (и деревьев вывода) в циклической грамматике, т. е. процесс вывода фиксированной цепочки никак не ограничен по числу шагов. Поэтому первое из рассматриваемых преобразований – это устранение цикличности.

Циклы в грамматиках могут возникать из-за правил вида $A \to B$, называемых *цепными* (допустим, грамматика содержит правила $A \to B$, $B \to C$ и $C \to A$). Другой причиной появления циклов могут стать ε-правила. Например, наличие в грамматике правил $A \to BC$, $B \to CA$, $C \to \varepsilon$ делает возможным цикл $A \Rightarrow BC \Rightarrow CAC \Rightarrow AC \Rightarrow A$.

В силу теоремы 4.1 можно считать рассматриваемую грамматику ε-свободной. Тогда все циклы в ней получаются применением только цепных правил. Переход к эквивалентной ацикличной грамматике основан на следующей лемме.

Лемма 4.1. *Если грамматика G содержит цепные правила $A_1 \to A_2$, $\ldots, A_{n-1} \to A_n, A_n \to A_1$, то грамматика G', полученная из G заменой нетерминалов $A_1, \ldots, A_n$ на один нетерминал A и удалением получившегося правила вида $A \to A$, эквивалентна G.*

Доказательство. Если цепочка w выводима в грамматике G, то ее вывод в G' получается из вывода в G «стиранием индексов» у символов A_i, $1 \le i \le n$. Обратно, пусть $S \Rightarrow^*_{G'} w$. Сконструируем соответствующий вывод в G, последовательно анализируя шаги вывода в G'. Те шаги вывода, на которых применялись правила, не содержащие символ A, переносятся в вывод в G без изменений. Вывод по правилу с символом A в правой части заменяется на такой же вывод с некоторым символом A_i в правой части (такой символ существует по определению грамматики G'). Наконец, если применяется вывод по правилу $A \to \alpha$, то данный символ A заменен на некоторый символ A_j в уже построенной части вывода в G, а кроме того, в G существует некоторое правило вида $A_k \to \alpha'$, где цепочка α' получена из α заменой всех вхождений символа A на подходящие символы вида A_i. Тогда шаг с применением правила $A \to \alpha$ можно смоделировать в G последовательностью шагов, производящих вывод $A_j \Rightarrow \ldots \Rightarrow A_k \Rightarrow \alpha'$. Итак, мы показали, что $S \Rightarrow^*_G w$ и, значит, $L(G) = L(G')$. □

Алгоритм построения эквивалентной ацикличной грамматики основан на последовательном применении данной леммы до удаления всех циклов. Данный процесс конечен, так как при каждом применении леммы количество цепных правил в грамматике уменьшается, а при их отсутствии циклы невозможны (при преобразовании свойство грамматики быть ε-свободной сохраняется). Детали реализации алгоритма мы оставляем читателю. Мы доказали

Предложение 4.2. *Любая КС-грамматика эквивалентна ациклической КС-грамматике.*

В оставшейся части параграфа мы рассматриваем только ациклические грамматики. Это означает, что в следующем определении можно считать, что $\gamma \neq \varepsilon$.

Определение. Нетерминальный символ B называется *леворекурсивным*, если $B \Rightarrow^+ B\gamma$, и *непосредственно леворекурсивным*, если $B \to B\gamma$ – правило вывода (такие правила также называются леворекурсивными). Грамматика называется леворекурсивной, если леворекурсивным является хотя бы один из ее нетерминалов.

Как мы увидим позже, леворекурсивные грамматики непригодны для синтаксического анализа некоторыми простыми и эффективными методами. Поэтому мы рассмотрим преобразование, устраняющее леворекурсивные символы. Начнем с частного случая – устранения непосредственной левой рекурсии для одного нетерминала грамматики.

Лемма 4.2. *Пусть A – непосредственно леворекурсивный нетерминал грамматики G,*

$$A \to A\alpha_1 \mid \ldots \mid A\alpha_k \mid \beta_1 \mid \ldots \mid \beta_m, \tag{4.1}$$

где ни одна из цепочек β_i не начинается с A, – все альтернативы этого нетерминала. Тогда грамматика G', полученная из G добавлением нетерминала A' и заменой вышеприведенных правил на правила

$$A \to \beta_1 A' \mid \ldots \mid \beta_m A', \quad A' \to \alpha_1 A' \mid \ldots \mid \alpha_k A' \mid \varepsilon, \tag{4.2}$$

эквивалентна G.

Доказательство. Применением как правил (4.1), так и правил (4.2) из нетерминала A можно вывести в точности все цепочки вида $\beta_i\alpha_{j_1}\ldots\alpha_{j_r}$, где $i \in \{1,\ldots,m\}$, $r \geq 0$, $j_1,\ldots,j_r \in \{1,\ldots,k\}$. Все остальные правила грамматик G и G' и их аксиомы совпадают. □

Эквивалентное преобразование множества правил P, состоящее в замене правил (4.1) на правила (4.2), обозначим через $nl(P, A)$. Если данное преобразование применить для обоих леворекурсивных символов грамматики GA_2 (см. с. 70), то получим грамматику

$$GA_3 = \{E \to TE', E' \to +TE' \mid \varepsilon, T \to FT', T' \to *FT' \mid \varepsilon, F \to (E) \mid \mathtt{x}\}.$$

Теперь опишем общий алгоритм устранения левой рекурсии (методом динамического программирования).

Алгоритм 4.6. *Устранение левой рекурсии в КС-грамматике.*
Вход. ε-свободная КС-грамматика $G = (\Sigma, \Gamma = \{A_1, \ldots, A_n\}, P, A_1)$.
Выход. КС-грамматика $\bar{G}$ без левой рекурсии, эквивалентная G.

1. $\bar{P} \leftarrow P$; $\bar{\Gamma} \leftarrow \Gamma$; $i \leftarrow 1$
2. **пока** $(i \le n)$ **повторять**
3. $\quad j \leftarrow 1$
4. $\quad$ **пока** $(j < i)$ **повторять**
5. $\quad\quad K_j \leftarrow \{\beta \mid (A_j \to \beta) \in \bar{P}\}$
6. $\quad\quad$ **для каждого** $(A_i \to A_j\alpha) \in \bar{P}$
7. $\quad\quad\quad \bar{P} \leftarrow \bar{P} \backslash (A_i \to A_j\alpha) \cup \bigcup\limits_{\beta \in K_j} (A_i \to \beta\alpha)$
8. $\quad\quad j \leftarrow j + 1$
9. $\quad$ **если** $(\exists \gamma : (A_i \to A_i\gamma) \in \bar{P})$
10. $\quad\quad \bar{\Gamma} \leftarrow \bar{\Gamma} \cup A_i'$
11. $\quad\quad \bar{P} \leftarrow nl(\bar{P}, A_i)$
12. $\quad i \leftarrow i + 1$
13. $\bar{G} \leftarrow (\Sigma, \bar{\Gamma}, \bar{P}, A_1)$

Предложение 4.3. *Алгоритм 4.6 работает корректно.*

Доказательство. Преобразование множества правил, описанное в строке 7, является эквивалентным. В самом деле, если при выводе терминальной цепочки использовалось правило $A_i \to A_j\alpha$, то впоследствии должно быть применено одно из правил $A_j \to \beta$, т. е. из нетерминала A_i обязательно выводится цепочка $\beta\alpha$. После преобразования множества правил этот вывод производится за один шаг. Преобразование, производимое в строке 11, является эквивалентным по лемме 4.2. Таким образом, $L(\bar{G}) = L(G)$.

Докажем, что грамматика $\bar{G}$ нелеворекурсивна. Для этого достаточно показать по индукции, что во всех правилах $A_i \to A_m\alpha$ выполнено $i < m$. База индукции справедлива, так как при $i = 1$ цикл по j не выполняется, а правила вида $A_1 \to A_1\alpha$ устранены в строке 11. Предположим, что требуемое условие выполнено для всех $k < i$ и рассмотрим i-ую итерацию внешнего цикла. После k-й итерации цикла по j для правил вида $A_i \to A_m\alpha$ выполняется $m > k$, поскольку в любом правиле вида $A_k \to A_m\beta$ выполнено $k < m$ по предположению индукции. Следовательно, после завершения цикла

по j останутся только правила $A_i \to A_m\alpha$ с условием $m \geq i$, а после преобразования в строке 11 – с условием $m > i$. Предложение доказано. □

Из доказанного предложения с учетом теоремы 4.1 и предложения 4.2 вытекает

Теорема 4.2. *Для любой КС-грамматики существует эквивалентная ей КС-грамматика без левой рекурсии.*

Другим преобразованием, продиктованным нуждами синтаксического анализа, является устранение из грамматики альтернатив с непустым общим префиксом, называемое *левой факторизацией*. Рассмотрим два правила из грамматики языка Паскаль:

<оператор>→ if <условие> then <оператор>
<оператор>→ if <условие> then <оператор> else <оператор>

Когда при просмотре цепочки (т. е. программы на Паскале) синтаксический анализатор доходит до ключевого слова if, он понимает, что в выводе нужно применить одно из двух данных правил. Но, чтобы понять, какое именно из них применимо, может возникнуть необходимость заглянуть очень далеко вперед (фрагменты программы, выводимые из нетерминалов <условие> и <оператор> в правой части, могут быть сколь угодно длинными). Далее, фрагмент программы, выводимый из нетерминала <оператор> в правой части, может, в свою очередь, содержать условные операторы, так что проблема выбора правила становится рекурсивной и не сводится к заглядыванию вперед. Проблема исчезнет, если заменить рассмотренную пару правил тройкой

<оператор>→ if <условие> then <оператор><иначе>
<иначе>→ else <оператор> | ε

Выбор правила в этом случае отодвигается до момента, когда он станет очевидным. Опишем реализацию данного преобразования в общем случае. Результатом преобразования является *левофакторизованная* грамматика, т. е. не содержащая альтернатив с непустым общим префиксом.

Алгоритм 4.7. *Левая факторизация КС-грамматики.*
Вход. КС-грамматика $G = (\Sigma, \Gamma, P, S)$.
Выход. Левофакторизованная КС-грамматика $\widehat{G}$, эквивалентная G.

1. $\widehat{\Gamma} \leftarrow \Gamma$; $\widehat{P} \leftarrow P$

2. **для каждого** $A \in \Gamma$
3. $\quad M \leftarrow \{\beta \in (\widehat{\Gamma} \cup \Sigma)^+ \mid \exists\, \alpha_1, \alpha_2 \in (\widehat{\Gamma} \cup \Sigma)^* : (A \to \beta\alpha_1 \mid \beta\alpha_2) \in \widehat{P}\}$
4. $\quad$ **если** $(M \neq \emptyset)$
5. $\quad\quad$ **выбрать** $\beta' \in M : |\beta'| = \max_M |\beta|$
6. $\quad\quad$ **выбрать** $A' \notin \widehat{\Gamma}$; $\widehat{\Gamma} \leftarrow \widehat{\Gamma} \cup \{A'\}$
7. $\quad\quad$ **для каждого** $(A \to \beta'\alpha) \in P$
8. $\quad\quad\quad \widehat{P} \leftarrow (\widehat{P} \backslash (A \to \beta'\alpha)\,) \cup (A' \to \alpha)$
9. $\quad\quad \widehat{P} \leftarrow \widehat{P} \cup (A \to \beta' A')$
10. $\quad\quad$ **перейти на** 3
11. $\widehat{G} \leftarrow (\Sigma, \widehat{\Gamma}, \widehat{P}, S)$

Докажем корректность алгоритма 4.7. Преобразование множества правил, описанное в строках 6–9, не изменяет множества выводимых цепочек: применение правила $A \to \beta\alpha$ в выводе взаимно однозначно заменяется последовательным применением правил $A \to \beta A'$ и $A' \to \alpha$. Далее, это преобразование никак не затрагивает правила вывода из прочих нетерминалов грамматики. Наконец, в результате преобразования множество M уменьшается, т. е. через конечное число шагов мы получим $M = \emptyset$ (а значит, у A не осталось альтернатив с непустым общим префиксом) и выйдем из внутреннего цикла. Осталось заметить, что внешний цикл достаточно выполнить для каждого нетерминала *исходной* грамматики, поскольку вновь вводимые нетерминалы не имеют альтернатив с непустым общим префиксом ввиду максимальности длины цепочки β'. Тем самым алгоритм корректен и справедливо

Предложение 4.4. *Для любой КС-грамматики существует эквивалентная ей левофакторизованная КС-грамматика.*

Замечание 4.4. Некоторые из рассмотренных преобразований «противоречат» друг другу. Так, устранение левой рекурсии и левая факторизация могут приводить к возникновению ε-правил, нарушая ε-свободу исходной грамматики. В свою очередь, переход к ε-свободной грамматике часто вызывает появление альтернатив с общим префиксом. Однако для методов анализа, чувствительных к наличию ε-правил, некритичны левая рекурсия и общие префиксы альтернатив. Верно и обратное: методы, не допускающие левую рекурсию и общие префиксы, не имеют затруднений с обработкой ε-правил.

Задачи

1. Найти приведенную грамматику, эквивалентную грамматике:
а) $G = \{S \to \mathtt{a}SA\mathtt{b} \mid \mathtt{a}DF\mathtt{b}, A \to \mathtt{c}E, B \to \mathtt{ba}, C \to \mathtt{a}D\mathtt{c} \mid \mathtt{ac},$
$D \to \mathtt{a}D\mathtt{b} \mid \mathtt{c}, E \to \mathtt{a}BE, F \to \mathtt{b}B\}$;
б) $G = \{S \to \mathtt{b}AE\mathtt{c}, A \to \mathtt{a}A\mathtt{c} \mid \mathtt{a}D\mathtt{b}, B \to \mathtt{b}AC \mid \mathtt{ab}, C \to \mathtt{b}E \mid \mathtt{a},$
$D \to BF \mid \mathtt{c}F, E \to \mathtt{b}A \mid \varepsilon\}$.

2. Для данной грамматики найти эквивалентную ей приведенную ε-свободную грамматику:
а) $G = \{S \to \mathtt{a}A\mathtt{b}, A \to BC \mid \mathtt{ab}, B \to D\mathtt{c} \mid D, C \to \varepsilon, D \to \varepsilon\}$;
б) $G = \{S \to \mathtt{b}ABC, A \to \mathtt{a}B \mid C, B \to \mathtt{b}D \mid \varepsilon, C \to \varepsilon, D \to \mathtt{a}\}$.

3. Применить алгоритм построения ε-свободной грамматики к грамматике GA_3.

4. Устранить циклы в грамматике $G = \{S \to SS \mid (S) \mid \varepsilon\}$.

5. Устранить левую рекурсию в грамматике

$$G = \{S \to A\mathtt{a} \mid A\mathtt{b} \mid \mathtt{c}, A \to SS \mid \mathtt{d}\} \ .$$

6. Произвести левую факторизацию грамматики

$$G = \{S \to A\mathtt{bc} \mid A\mathtt{c}, A \to BS\mathtt{a} \mid BS\mathtt{c} \mid \mathtt{b}, B \to A\mathtt{c} \mid \mathtt{a}\} \ .$$

7*. КС-грамматика имеет *нормальную форму Хомского* (ХНФ), если она ε-свободна и ее неаннулирующие правила вывода имеют вид $A \to BC$ или $A \to \mathtt{a}$. Доказать, что каждая КС-грамматика эквивалентна КС-грамматике в ХНФ. *Указание.* Воспользоваться теоремой 4.1.

8*. КС-грамматика имеет *нормальную форму Грейбах* (ГНФ), если она ε-свободна и ее неаннулирующие правила вывода имеют вид $A \to \mathtt{a}A_1 \ldots A_n$, где $n \geq 0$. Доказать, что каждая КС-грамматика эквивалентна КС-грамматике в ГНФ. *Указание.* Воспользоваться предыдущим упражнением.

9*. КС-грамматика называется *операторной*, если она ε-свободна и ее неаннулирующие правила вывода имеют вид $A \to \gamma_0 \mathtt{a}_1 \ldots \gamma_{n-1} \mathtt{a}_n \gamma_n$, где $n \geq 0$ и $\gamma_i \in \Gamma \cup \{\varepsilon\}$ для всех i. Доказать, что любая КС-грамматика эквивалентна операторной КС-грамматике. *Указание.* Воспользоваться предыдущим упражнением.

Часть 2.

Лексический и синтаксический анализ в формализованных языках

Данная часть посвящена обсуждению фактически одной задачи: *по данной КС-грамматике G и цепочке w над основным алфавитом этой грамматики определить, принадлежит ли цепочка w языку* $L(G)$. Если $w \in L(G)$, то необходимо выдать информацию о выводе этой цепочки, например привести дерево вывода. Если же $w \notin L(G)$, то желательно указать предполагаемую ошибку (ошибки), приводящую к невыразимости w средствами грамматики G. Эта задача называется задачей *синтаксического анализа*[†]. Необходимость решения такой задачи возникает в различных приложениях, в частности при проектировании компиляторов.

Как уже обсуждалось во введении, компилятор – это программа, которая переводит цепочку, принадлежащую одному языку (исходному) в семантически эквивалентную цепочку из другого языка (целевого). Чаще всего исходным языком является язык программирования высокого уровня (понятный человеку), а целевым – язык машинных команд (понятный компьютеру). При этом компилятору необходимо решить задачу синтаксического анализа входной цепочки. Наиболее распространенная технология создания компиляторов такова, что решение задачи синтаксического анализа делится на две

[†]В англоязычной литературе чаще всего используется термин parsing (от parse – анализировать), а программа-анализатор называется parser. Кальки с этих терминов встречаются и в книгах на русском языке.

части, первая выполняется в блоке лексического анализа, вторая – в блоке синтаксического анализа. Блок лексического анализа преобразует исходную программу, написанную на языке высокого уровня, в цепочку так называемых *токенов* либо выдает сообщение об ошибках. Блок синтаксического анализа анализирует поступившую на вход цепочку токенов и выдает информацию о структуре цепочки (например, дерево вывода) или сообщение об ошибках. Этот блок является, как правило, основой компилятора. Принципиальные решения, принятые при проектировании синтаксического блока, в значительной степени определяют структуру и функциональные возможности других блоков.

Стадия анализа исходной программы завершается в блоке семантического анализа. В этом блоке происходит вычисление некоторых семантических характеристик объектов компилируемой программы (например, типов идентификаторов) и проверка на наличие ряда семантических ошибок. После прохождения этого блока компилятор обладает всей необходимой информацией об исходной программе и переходит в стадию реализации, где осуществляется генерация и оптимизация кода. На рис. 0.1 приведена упрощенная модель компилятора, в которой вся вторая стадия изображена в виде одного блока. Более подробно о структуре и функциях компилятора можно прочитать в [АСУ].

В данной части будут рассмотрены методы организации работы лексического блока (гл. 1) и синтаксического блока (гл. 2–4). Обсуждению вопросов работы семантического блока будет посвящена третья часть. В основе работы блока генерации кода лежат математические принципы, изложение которых выходит за рамки этой книги, поэтому данный блок мы не рассматриваем. Читатель может ознакомиться с вопросами генерации кода по книге [АСУ] или по более современному, но, увы, менее доступному учебнику [СТ].

Необходимо отметить, что построение лексического и синтаксического анализаторов основано на нетривиальных, но ясных и достаточно легко программируемых алгоритмах. Существует класс программ, называемых «компиляторами компиляторов», предназначенных для генерации анализаторов по грамматике. В качестве метода синтаксического анализа обычно выбирается LR-анализ (см. гл. 4). Пожалуй, самым известным в настоящее время генератором лексических анализаторов является Flex, а генератором синтаксических анализаторов – Bison (а также их предшественники Lex и Yacc соот-

Рис.0.1. *Упрощенная модель компилятора*

ветственно)[†]. Однако даже при использовании подобных программ без «ручной» доводки компилятора все равно не обойтись, а ее проведение требует хорошего знания того, *как* и *почему* работают различные методы анализа.

Глава 1. Лексический анализ

Задача лексического анализа состоит в выделении во входной цепочке (т. е. тексте программы) минимальных смысловых единиц, называемых *лексемами*, и определении соответствующих лексемам синтаксических категорий языка – *токенов*. Попутно из текста исходной программы удаляются комментарии и пробелы, не имеющие смысловой нагрузки, а также обнаруживаются ошибки некоторых

[†]Подробнее о компиляторах компиляторов можно прочесть, например, на сайте www.compilertools.net

видов. Преобразованная лексическим анализатором программа является цепочкой уже не в исходном алфавите, а в алфавите токенов.

§ 1. Основные понятия лексического анализа

Предположим, что на вход компилятора подана цепочка

```
if a13>2300 then ab12:=1400
```

На стадии синтаксического анализа несущественно, что слева от знака неравенства стоит переменная, имя которой есть последовательность из буквы и двух цифр, а существенно только то, что это *идентификатор*. Точно так же неважно, что справа от знака неравенства стоит число 2300, а не 2301, а важно только то, что это *числовая константа*. В этом примере блок лексического анализа выделит восемь лексем: ключевое слово `if`, идентификатор `a13`, знак неравенства $>$, целое число 2300, ключевое слово `then`, идентификатор `ab12`, оператор присваивания $:=$ и целое число 1400 (предполагается, что пробелы не имеют смысловой нагрузки). Каждая лексема принадлежит определенному классу лексем. Лексемы одного класса неразличимы с точки зрения синтаксического анализатора (как, например, идентификаторы `a13` и `ab12`). Класс лексем называется *токеном*. Таким образом, приведенная выше цепочка из 27 символов будет преобразована в цепочку из 8 токенов. Отметим, что каждое ключевое слово образует свой класс лексем, состоящий из одного этого слова.

Токены описываются *шаблонами*, в качестве которых чаще всего используются регулярные выражения или аналогичные конструкции. В табл. 1.1 приведены примеры токенов, лексем и шаблонов (регулярные выражения `<имя>` и `<константа>` приведены в ч.1, § 8).

В шаблонах для удобства обычно используют расширенный синтаксис регулярных выражений[†]. Поскольку алфавит линейно упорядочен, перечисляемые через знак альтернативы | символы часто образуют интервалы. Для записи интервала используют дефис: `a-z|0-9` вместо `a|b|...|z|0|...|9`. Кроме того, для регулярного выражения r вместо rr^* обычно пишут r^+, а вместо $r|\varepsilon$ – $(r)?$. С учетом этого, шаблоны токенов `id` и `num` могут быть записаны в виде

[†]Расширенные регулярные выражения являются одним из основных элементов синтаксиса языка Perl. Однако название «регулярные выражения» там весьма условно, так как далеко не все разрешенные к использованию операции могут быть выражены через объединение, умножение и итерацию. Как следствие, класс задаваемых такими выражениями языков значительно шире класса регулярных языков.

Таблица 1.1. *Токены, лексемы, шаблоны*

Токен	Примеры лексем	Шаблон
if	if	if
id	a13, ab12, pi	<имя>
rel	<, <=, >	<|<=|=|<>|>|>=
aop	+, -	+|-
mop	*, /	*|/
num	2300,3.14,3.1E12	<константа>
then	then	then
assign	:=	:=
comma	,	,

<имя> = a-z(a-z|0-9)*

<константа> = (-)?(1-9(0-9)*|0)(.(0-9)$^+$)?(E(+|-)?(0-9)$^+$)?

Как уже отмечалось, идентификаторы a13 и ab12 в блоке синтаксического анализа можно не различать. Однако в следующих блоках компилятора их различать придется. Следовательно, если токен – это класс, состоящий из более чем одной лексемы, то лексический анализатор должен обеспечить возможность получения соответствующей информации. Обычно это достигается путем приписывания токену так называемых атрибутов. *Атрибут* – это переменная, принимающая значение в некотором множестве. На практике токены обычно имеют один атрибут – указатель на запись в таблице символов, в которой хранится информация о токене. Результат обработки лексическим анализатором цепочки, приведенной в начале параграфа, можно записать в виде последовательности

if	id	rel	num	then	id	assign	num
	↓	↓	↓		↓		↓
	ptr_1	ptr_2	ptr_3		ptr_4		ptr_5

Процесс занесения данных о токене в таблицу символов с последующим выводом пары (токен, атрибут) на выход анализатора называют *регистрацией токена*.

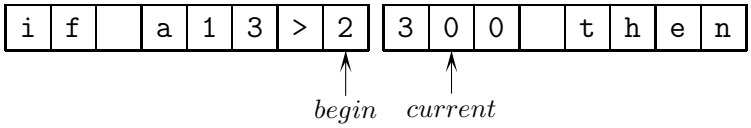
Содержимое записи в таблице символов зависит от токена, и идентификатор, очевидно, требует наибольшего количества сведений. В запись для идентификатора лексический анализатор вносит имя лексемы и (для диагностических целей) место в программе, где впервые появился идентификатор. На последующих стадиях ком-

пиляции будут добавлены сведения о типе, адресе для хранения значения, а также различная служебная информация.

§ 2. Методы работы лексического анализатора

Рассмотрим некоторые из проблем, возникающих при организации блока лексического анализа. Хотя это, пожалуй, самый простой блок компилятора, ему следует уделить должное внимание хотя бы потому, что это единственный блок, читающий *всю исходную программу*, символ за символом. Следовательно, скорость работы лексического анализатора существенно сказывается на эффективности компилятора в целом. В частности, естественно потребовать, чтобы лексический анализатор выполнял свою работу (разбиение текста программы на лексемы и регистрацию соответствующих токенов) за один проход по входной цепочке.

Вначале кратко обсудим организацию ввода и обработки данных. Если программа достаточно длинная, то ее посимвольное чтение из памяти с последующей обработкой – идея вряд ли удачная. Гораздо эффективнее считывать в буфер целый блок (при помощи одной системной команды чтения), а затем обрабатывать. При этом удобно пользоваться не одним буфером, а двумя. В самом деле, при единственном буфере возникает сложность с обработкой ситуации, когда блок закончился на середине лексемы. Имея второй буфер, можно считать в него следующий блок и продолжить работу. Когда будет заканчиваться блок во втором буфере, блок в первом будет уже не нужен (размер блока можно сделать заведомо больше максимального размера лексемы) и на его месте можно разместить следующий блок, и т. д. В процессе работы с буферами поддерживаются два указателя. Указатель *begin* показывает на первый символ текущей лексемы, а указатель *current* – на обрабатываемый символ:



Когда обнаружено соответствие шаблону некоторого токена, его лексема находится между указателями. Тем самым доступна вся информация о найденном токене для внесения в таблицу символов. После регистрации токена оба указателя перемещаются на следующую позицию после найденной лексемы.

Неоднозначности, возникающие при распознавании лексем, разрешаются при помощи *принципа наидлиннейшей лексемы: среди всех лексем, начинающихся в данной позиции, выбирается самая длинная*. Так, в приведенном примере между указателями находится лексема `230`, но данный принцип требует продолжить чтение и выбрать лексему `2300` (что, очевидно, правильно).

Теперь перейдем к описанию процесса распознавания лексем. Основная идея – использовать конечный автомат, распознающий шаблоны всех токенов, и подавать ему на вход фрагмент входной цепочки, начинающийся в позиции указателя *begin*. При этом каждому заключительному состоянию автомата должен однозначно соответствовать токен, и с этим состоянием должна быть связана подпрограмма регистрации токена. Для разрешения неоднозначности при выборе токена во многих языках программирования ключевые слова зарезервированы (т. е. `if` – это всегда начало условного оператора и никогда – идентификатор).

Удобно поступать следующим образом: построить автоматы для каждого шаблона в отдельности, затем соединить их в один ε-НКА, добавив новую начальную вершину, после чего построить эквивалентный минимальный ДКА[†] (см. ч. 1, § 4–7).

Диаграммы автоматов для некоторых шаблонов из табл. 1.1 приведены на рис. 1.1–1.3. Обратите внимание на рис. 1.3, на котором совмещены диаграммы для зарезервированных ключевых слов с диаграммой для идентификаторов.

Рядом с каждым заключительным состоянием указана процедура регистрации токена в одном из двух вариантов, REG или REG-. Процедура REG- применяется, если последний прочитанный символ не принадлежит найденной лексеме (но его необходимо было прочитать, чтобы обнаружить эту лексему!). В этом случае для инициализации поиска следующей лексемы нужно присвоить указателю *begin* значение указателя *current*. Вариант REG используется, если последний символ принадлежит лексеме. В этом случае надо оба указателя разместить на символ правее текущего положения указателя *current*. Если переход из состояния по данному символу на диаграмме отсутствует, то он осуществляется по дуге с меткой other (иногда его называют «переходом по умолчанию»).

[†]Поскольку нужно один раз построить анализатор, а затем многократно обрабатывать им программы, то выгодно один раз применить вычислительно сложные алгоритмы детерминирования и минимизации автомата, получая взамен неплохой выигрыш в скорости обработки цепочек.

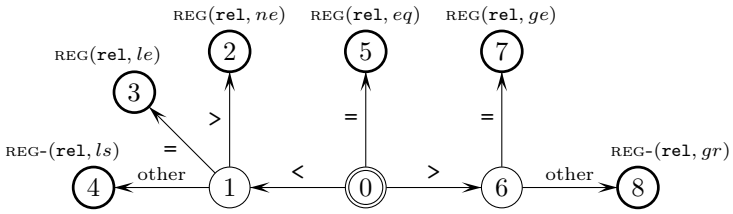


Рис. 1.1. *Автомат для шаблона* `rel`

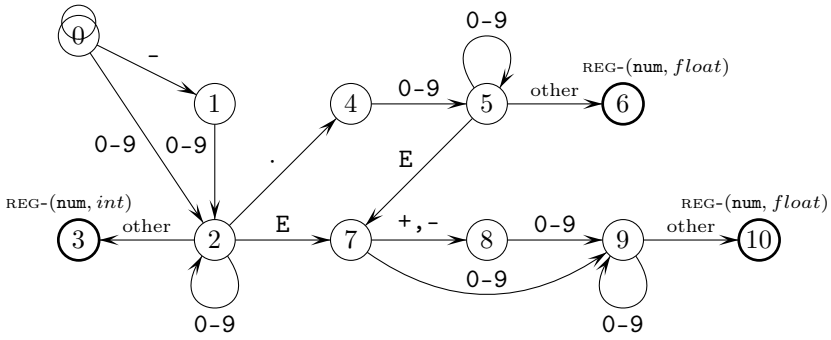


Рис. 1.2. *Автомат для шаблона* `num`

Замечание 1.1. В реальном анализаторе кроме переходов, указанных на диаграммах, будут выделены еще переходы, соответствующие лексическим ошибкам в программе. Например, при анализе фрагмента `1.1E3E4` входной цепочки нужно выдать сообщение о лексической ошибке. Однако при использовании переходов по умолчанию на рис. 1.2 будет зарегистрирован токен `num`, а оставшиеся символы `E4` будут приняты за идентификатор.

В этом параграфе не рассмотрены некоторые проблемы, возникающие при проектировании лексического анализатора, в частности обработка лексических ошибок и эффективная организация таблицы переходов ДКА. Более подробную информацию читатель может почерпнуть в [АСУ] или [ЛРС].

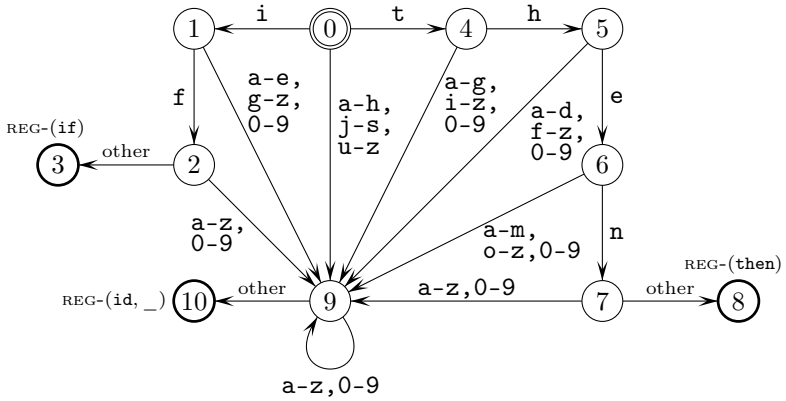


Рис. 1.3. *Автомат для шаблонов* `if`, `then` *и* `id`. *Не изображены дуги из вершин 1, 4, 5, 6 в вершину 10 с меткой other*

Задачи

1. В следующем фрагменте программы на Паскале найти лексемы, преобразовать фрагмент в цепочку токенов, дать токенам обоснованные атрибуты:

```
function frac_sum(n: integer): real;
{вычисление суммы 1 + 1/2 + 1/3 + ... + 1/n}
begin
frac_sum:=0;
while n>0 do
  begin
  frac_sum:=frac_sum+1/n; n:=n-1;
  end;
end;
```

2. Рассмотрим язык, имеющий четыре ключевых слова: `do, of, or, to`. Имена в этом языке состоят из одной или двух букв латиницы, числовые константы – только неотрицательные целые числа. В языке имеются и другие символы, кроме букв и цифр. Начертить диаграмму конечного автомата, распознающего ключевые слова, имена и числовые константы.

Глава 2. Нисходящие синтаксические анализаторы

Задача синтаксического анализа для грамматики G состоит, как уже отмечалось, в том, чтобы по входной цепочке w определить, принадлежит ли она $L(G)$. Если $w \in L(G)$, то необходимо найти вывод цепочки w (обычно путем построения дерева вывода). Построение дерева вывода может идти двумя путями – от корня к листьям или от листьев к корню. Методы синтаксического анализа в связи с этим делятся на два больших класса: нисходящие («top-down») и восходящие («bottom-up»)[†]. Эта глава посвящена изучению нисходящих методов. В главе по умолчанию предполагается, что *все выводы являются левосторонними.*

§ 3. Разделенные грамматики

Начнем с конкретного примера нисходящего анализа.

Пример 2.1. Пусть даны цепочка $w = \mathtt{aaccbccc}$ и грамматика

$$G_{2,1} = \{S \to \mathtt{a}DS, S \to \mathtt{b}D\mathtt{c}, D \to \mathtt{a}D, D \to \mathtt{cc}\}\,,$$

требуется проверить принадлежность w языку $L(G_{2,1})$ и в случае положительного ответа найти левосторонний вывод этой цепочки. Предположим, что $w \in L(G_{2,1})$, т. е. $S \Rightarrow^* w$. Поскольку w начинается символом $\mathtt{a}$, первым в левостороннем выводе могло применяться только первое правило, т. е. $S \Rightarrow \mathtt{a}DS \Rightarrow^* w$. Следовательно, $DS \Rightarrow^* \mathtt{accbccc}$. Это означает, что на втором шаге вывода применялось правило $D \Rightarrow \mathtt{a}D$, откуда $DS \Rightarrow^* \mathtt{ccbccc}$, и т. д. Продолжая рассуждение, мы выясняем, что $w \in L(G_{2,1})$, и получаем вывод

$$S \Rightarrow \mathtt{a}DS \Rightarrow \mathtt{aa}DS \Rightarrow \mathtt{aacc}S \Rightarrow \mathtt{aaccb}D\mathtt{c} \Rightarrow \mathtt{aaccbccc}\,.$$

Проведя подобный анализ для цепочки $w' = \mathtt{baabcc}$, из предположения $S \Rightarrow^* w'$ мы последовательно получим $D \Rightarrow^* \mathtt{aabc}$ (после применения второго правила), $D \Rightarrow^* \mathtt{abc}$ и $D \Rightarrow^* \mathtt{bc}$ (после двух применений третьего правила). Но альтернативы нетерминала D начинаются с $\mathtt{a}$ или с $\mathtt{c}$, т. е. вывод $D \Rightarrow^* \mathtt{bc}$ невозможен. Следовательно, $w' \notin L(G_{2,1})$.

[†]Корень дерева на диаграммах изображается вверху!

Определим по возможности наиболее широкий класс грамматик, для которого приведенный в примере 2.1 способ дает корректный алгоритм синтаксического анализа. Внимательное изучение примера показывает, что существенны две особенности грамматики $G_{2,1}$. Во-первых, правая часть каждого правила вывода начинается терминалом. Это позволяет среди всех правил вывода определять «претендентов» на применение на очередном шаге по начальному символу суффикса анализируемой цепочки. (Для цепочки w из примера 2.1 правила выбирались по суффиксам `aaccbccc`, `accbccc`, `ccbccc`, `bccc`, `ccc`.) Во-вторых, все альтернативы любого нетерминала начинаются разными терминалами. Следовательно, если есть «претендент», то он один. Эти наблюдения приводят нас к следующему определению.

Определение. КС-грамматика называется *разделенной*, если:

1) правая часть любого правила вывода начинается терминалом;

2) все альтернативы любого нетерминала начинаются разными терминалами.

Для разделенных грамматик можно предложить несложный алгоритм синтаксического анализа, реализованный в МП-автомате. Вначале сделаем важное

Замечание 2.1. Все МП-автоматы в этой главе будут иметь единственное состояние и допускать цепочки при помощи команды допуска. Такой МП-автомат задается не семеркой объектов, а четверкой: основной и вспомогательный алфавиты, множество команд и начальное содержимое стека. *Обозначение МП-автомата как четверки объектов будет указывать на автомат именно такого типа.*

Алгоритм 2.1. *Построение распознающего МП-автомата для разделенной грамматики.*
Вход: разделенная грамматика $G = (\Sigma, \Gamma, P, S)$.
Выход: МП-автомат $\mathcal{M} = (\widehat{\Sigma}, \widehat{\Gamma}, \delta, \gamma_0)$.

1. $\widehat{\Sigma} \leftarrow \Sigma \cup \{\dashv\}$
2. $\widehat{\Gamma} \leftarrow \Sigma \cup \Gamma \cup \{\nabla\}$
3. $\gamma_0 \leftarrow S$
4. $\delta \leftarrow \{(\dashv, \nabla) \rightarrow \checkmark\}$
5. **для каждого** $(B \rightarrow \mathtt{a}\gamma) \in P$ **выполнить**
6. $\qquad \delta \leftarrow \delta \cup \{(\mathtt{a}, B) \rightarrow (\mathtt{a}\gamma, \_)\}$
7. **для каждого** $\mathtt{a} \in \Sigma$ **выполнить**
8. $\qquad \delta \leftarrow \delta \cup \{(\mathtt{a}, \mathtt{a}) \rightarrow (\varepsilon, \rightarrow)\}$

Мы не будем сейчас доказывать корректность алгоритма 2.1, поскольку в § 5 будет доказана корректность более общего алгоритма.

Внимательный читатель заметит сходство автомата, построенного алгоритмом 2.1, с недетерминированным МПА из доказательства теоремы 3.6 (ч.1, § 18). Оба автомата моделируют применение правил вывода в левостороннем выводе цепочки. Отличие состоит в том, что в НМПА в качестве «претендентов» предлагались все альтернативы текущего нетерминала (благодаря недетерминированности из них всегда можно выбрать подходящего). Для разделенной грамматики оказывается возможным выбор единственного претендента на стадии построения автомата, и мы получаем ДМПА.

Пример 2.2. Для рассмотренной в предыдущем примере грамматики $G_{2,1}$ управляющей таблицей МП-автомата будет табл. 2.1.

Таблица 2.1. *Управляющая таблица МП-автомата для грамматики* $G_{2,1}$

	a	b	c	⊣
S	aDS	bDc		
D	aD		cc	
a	$\varepsilon,\rightarrow$			
b		$\varepsilon,\rightarrow$		
c			$\varepsilon,\rightarrow$	
▽				✓

Построенный МП-автомат можно упростить. В самом деле, если автомат на некотором шаге заменяет нетерминал на верхушке стека на правую часть соответствующего правила вывода, то на следующем шаге будут произведены снятие терминала со стека и сдвиг по входной цепочке. Можно «объединить» эти два шага в один. При этом символы a и b, которые могли появиться только на верхушке стека, никогда не будут встречаться в стеке, так что их можно удалить из стекового алфавита. В результате получится эквивалентный автомат $\mathcal{A}_{2,1}$ (табл. 2.2).

На этом примере продемонстрируем одно важное свойство МП-автоматов, построенных алгоритмом 2.1. Для этого посмотрим на анализ цепочки $w =$ aaccbccc автоматом $\mathcal{A}_{2,1}$ (табл. 2.3).

Рассмотрим последовательность цепочек, которые являются произведениями обработанной части входной цепочки на содержимое стека (без маркера дна):

Таблица 2.2. *Управляющая таблица МП-автомата* $\mathcal{A}_{2,1}$

	a	b	c	⊣
S	$DS,^{\rightarrow}$	$D\mathtt{c},^{\rightarrow}$		
D	$D,^{\rightarrow}$		$\mathtt{c},^{\rightarrow}$	
c			$\varepsilon,^{\rightarrow}$	
∇				✓

Таблица 2.3. *Протокол обработки цепочки* $w =$ aaccbccc *автоматом* $\mathcal{A}_{2,1}$

Такт	Позиция указателя	Содержимое стека
1	◇aaccbccc ⊣	$S\nabla$
2	a◇accbccc ⊣	$DS\nabla$
3	aa◇ccbccc ⊣	$DS\nabla$
4	aac◇cbccc ⊣	$\mathtt{c}S\nabla$
5	aacc◇bccc ⊣	$S\nabla$
6	aaccb◇ccc ⊣	$D\mathtt{c}\nabla$
7	aaccbc◇cc ⊣	$\mathtt{cc}\nabla$
8	aaccbcc◇c ⊣	$\mathtt{c}\nabla$
9	aaccbccc◇ ⊣	∇

S, aDS, aaDS, aacc$S$, aacc$S$, aaccbDc, aaccbccc, aaccbccc, aaccbccc .

Если убрать повторы соседних цепочек, то данная последовательность дает левосторонний вывод цепочки w в грамматике $G_{2,1}$. Этим свойством обладает не только приведенный выше МП-автомат, но и МП-автомат, построенный алгоритмом 2.1 для любой разделенной грамматики. Мы докажем последний факт позднее – для аналогичного алгоритма и более широкого класса грамматик.

§ 4. LL(1)-грамматики: определение и примеры

Разделенные грамматики на практике возникают редко. Из грамматик, порождающих фрагменты языков программирования (см. ч. 1, § 15), ни одна грамматика не является разделенной. Посмотрим, как можно расширить класс разделенных грамматик, сохранив возможность построения МП-автомата, моделирующего левосторонний вывод.

Вернемся к описанию МП-автомата, распознающего принадлежность языку, порожденному разделенной грамматикой. Мы видели, что если на верхушке стека находится нетерминальный символ B, а на входной цепочке обозревается символ $\mathtt{a}$, то автомат моделирует применение правила вида $B \to \mathtt{a}\gamma$. Если такого правила нет, то анализатор выдает сообщение об ошибке, а если есть, то оно единственно по определению разделенной грамматики. На самом деле, чтобы в данной ситуации найти правило, которое необходимо моделировать, вовсе не обязательно требовать, чтобы правая часть каждого правила вывода начиналась терминалом. Достаточно иметь информацию о том, можно ли из правой части правила $B \to \beta$, т. е. из цепочки β, вывести цепочку, начинающуюся символом $\mathtt{a}$. Если такое правило вывода только одно, то нужно смоделировать применение этого правила. Это наблюдение приводит нас к следующему определению.

Определение. Пусть $G = (\Sigma, \Gamma, P, S)$ – КС-грамматика, α – цепочка грамматических символов. Тогда $\textsc{first}(\alpha)$ – это подмножество из $\Sigma \cup \{\varepsilon\}$ такое, что
$\mathtt{a} \in \textsc{first}(\alpha)$, если и только если $\alpha \Rightarrow^* \mathtt{a}\beta$ для некоторой цепочки β;
$\varepsilon \in \textsc{first}(\alpha)$, если и только если $\alpha \Rightarrow^* \varepsilon$.

Другими словами, $\textsc{first}(\alpha)$ – это множество терминалов, с которых начинаются цепочки, выводимые из α, плюс пустое слово в случае, если α можно аннулировать. В качестве примера рассмотрим грамматику $G_{2,2} = \{S \to AC, A \to \mathtt{ab}C \mid \mathtt{b}B, B \to \mathtt{b}, C \to \mathtt{c} \mid \varepsilon\}$. Для этой грамматики $\textsc{first}(AC) = \{\mathtt{a}, \mathtt{b}\}$, $\textsc{first}(CA) = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$.

Для того чтобы моделировать применение аннулирующего правила вывода, множества $\textsc{first}(\alpha)$ недостаточно. Предположим, что в левостороннем выводе терминальной цепочки w было применено аннулирующее правило $A \to \varepsilon$. Это означает, что в процессе вывода был сделан переход $uA\alpha \Rightarrow u\alpha$, где u – префикс цепочки w, а затем из цепочки α была выведена оставшаяся часть v цепочки w. Пусть $\mathtt{b}$ – первая буква цепочки v, т. е. $v = \mathtt{b}v'$. Имеем $S \Rightarrow^* uA\alpha$ и $\alpha \Rightarrow^* \mathtt{b}v'$, а значит, в грамматике выводима (не обязательно в ходе левостороннего вывода) цепочка, в которой за символом A идет символ $\mathtt{b}$. Оказывается, что во многих случаях для определения необходимости применения аннулирующего правила $A \to \varepsilon$ достаточно информации о том, какие основные символы могут следовать за символом A в цепочках, выводимых из аксиомы. Дадим соответствующее

Определение. Пусть $G = (\Sigma, \Gamma, P, S)$ – КС-грамматика, A – нетер-

минал. Тогда $\text{FOLLOW}(A)$ – это подмножество из $\Sigma \cup \{\dashv\}$ такое, что $\mathtt{a} \in \text{FOLLOW}(\alpha)$, если и только если $S \Rightarrow^* \alpha A \mathtt{a} \beta$ для некоторых α, β; $\dashv \in \text{FOLLOW}(\alpha)$, если и только если $S \Rightarrow^* \alpha A$ для некоторой α.

Подчеркнем, что выводимость здесь не обязательно левосторонняя. Нетрудно понять, что в любой *приведенной* грамматике $\text{FOLLOW}(A) \neq \emptyset$ для каждого нетерминала A. Для грамматики $G_{2,2}$ выполняются равенства $\text{FOLLOW}(A) = \text{FOLLOW}(B) = \{\mathtt{c}, \dashv\}$.

Приведем основные определения данного параграфа.

Определение. Пусть $G = (\Sigma, \Gamma, P, S)$ – КС-грамматика, $A \to \alpha$ – ее правило вывода. Тогда *множество выбора* правила $A \to \alpha$ – это подмножество из $\Sigma \cup \{\dashv\}$ такое, что

$$\text{SELECT}(A \to \alpha) = \begin{cases} \text{FIRST}(\alpha), & \text{если } \varepsilon \notin \text{FIRST}(\alpha), \\ (\text{FIRST}(\alpha) \backslash \{\varepsilon\}) \cup \text{FOLLOW}(A), & \text{если } \varepsilon \in \text{FIRST}(\alpha). \end{cases}$$

Определение. КС-грамматика G называется *LL(1)-грамматикой*[†], если множества выбора правил с одинаковой левой частью не имеют общих элементов, т. е. для любого терминала A и неравных цепочек α и β выполняется $\text{SELECT}(A \to \alpha) \cap \text{SELECT}(A \to \beta) = \emptyset$.

Пример 2.3. В табл. 2.4 записаны множества выбора правил грамматики арифметических выражений

$$GA_3 = \{E \to TE', E' \to +TE' \,|\varepsilon, T \to FT', T' \to *FT' \,|\varepsilon, F \to (E) \,|\mathtt{x}\}.$$

Мы видим, что грамматика GA_3 является LL(1)-грамматикой.

Грамматика GA_2 (см. с. 70), порождающая тот же язык, что и GA_3, LL(1)-грамматикой не является. В этом можно убедиться непосредственно, проверив, что

$$\text{SELECT}(E \to E{+}T) = \text{SELECT}(E \to T) = \{(, \mathtt{x}\}.$$

Тот же факт очевидно вытекает из следующего утверждения.

Предложение 2.1. *Если приведенная грамматика содержит леворекурсивное правило, то она не является LL(1)-грамматикой.*

[†]Происхождение термина «LL(1)-грамматика» следующее. Как мы увидим в следующем параграфе, МП-автомат для анализа такой грамматики читает входную цепочку с*Л*ева направо, производит *Л*евосторонний вывод, а правило вывода для применения на очередном шаге выбирает, просматривая *один* символ входной цепочки. В литературе рассматриваются также «LL(k)-грамматики» при $k > 1$ (см. § 8).

Таблица 2.4. *Множества выбора для правил грамматики* GA_3

	Правило	Множество выбора
1	$E \to TE'$	$\{(, \mathtt{x}\}$
2	$E' \to +TE'$	$\{+\}$
3	$E' \to \varepsilon$	$\{), \dashv\}$
4	$T \to FT'$	$\{(, \mathtt{x}\}$
5	$T' \to *FT'$	$\{*\}$
6	$T' \to \varepsilon$	$\{+,), \dashv\}$
7	$F \to (E)$	$\{(\}$
8	$F \to \mathtt{x}$	$\{\mathtt{x}\}$

Доказательство. Предположим, что $A \to A\alpha$ – леворекурсивное правило приведенной грамматики G, $\alpha \neq \varepsilon$. Так как G – приведенная грамматика, символ A является производящим. Это означает, что существует правило вывода $A \to \beta$, правая часть которого не начинается с символа A, и, кроме того, $\textsc{first}(\beta) \neq \emptyset$. Покажем, что

$$\textsc{select}(A \to A\alpha) \cap \textsc{select}(A \to \beta) \neq \emptyset, \tag{2.1}$$

что противоречит определению LL(1)-грамматики.

В случае когда $\textsc{first}(\beta) \neq \{\varepsilon\}$, для терминала $\mathtt{a} \in \textsc{first}(\beta)$ по определению выполняется $\mathtt{a} \in \textsc{select}(A \to \beta)$. Но в то же время $\mathtt{a} \in \textsc{first}(A) \subseteq \textsc{first}(A\alpha)$, откуда $\mathtt{a} \in \textsc{select}(A \to A\alpha)$, т. е. условие (2.1) выполнено.

Пусть теперь $\textsc{first}(\beta) = \{\varepsilon\}$. Тогда $\beta \Rightarrow^* \varepsilon$, откуда $A \Rightarrow^* \varepsilon$. Рассмотрим множество $\textsc{first}(\alpha)$. Если найдется терминал $\mathtt{b} \in \textsc{first}(\alpha)$, то, с одной стороны, $\mathtt{b} \in \textsc{first}(A\alpha)$, а значит $\mathtt{b} \in \textsc{select}(A \to A\alpha)$. С другой стороны, $\mathtt{b} \in \textsc{follow}(A) \subseteq \textsc{select}(A \to \beta)$, и (2.1) снова выполнено.

В оставшемся случае $\textsc{first}(\alpha) = \textsc{first}(\beta) = \{\varepsilon\}$ получаем

$$\textsc{follow}(A) \subseteq \textsc{select}(A \to A\alpha) \cap \textsc{select}(A \to \beta).$$

Поскольку множество для $\textsc{follow}(A)$ непусто для любого нетерминала приведенной грамматики, условие (2.1) опять выполняется. Предложение доказано. □

Справедлив и гораздо более общий результат: *никакая приведенная леворекурсивная грамматика не является LL(1)-грамматикой.*

Его доказательство основано на той же идее, что и доказательство предложения 2.1, но существенно сложнее, см. напр., [АУ1].

Из приведенных в ч. 1, § 15 грамматик, порождающих язык списков LL, первые две содержат леворекурсивные правила вывода и поэтому не являются LL(1)-грамматиками. Грамматика $GL_3 = \{S \to L; S \mid L,\ L \to \mathtt{a} \mid [S]\}$ также не является LL(1)-грамматикой, поскольку множества выбора первого и второго правила совпадают. Эти два правила имеют общее начало. В общем случае справедливо

Замечание 2.2. Если приведенная грамматика G содержит правила $A \to \alpha\beta_1 \mid \alpha\beta_2$ и существует терминал $\mathtt{a} \in \textsc{first}(\alpha)$, то G не является LL(1)-грамматикой, так как $\mathtt{a} \in \textsc{select}(A{\to}\alpha\beta_1) \cap \textsc{select}(A{\to}\alpha\beta_2)$.

Применив к грамматике GL_3 алгоритм левой факторизации (см. алгоритм 4.7, ч. 1, § 22), получим грамматику

$$GL'_3 = \{S \to LS',\ S' \to ;\, S \mid \varepsilon,\ L \to \mathtt{a} \mid [S]\},$$

порождающую тот же язык, что и GL_3. Таблица 2.5 показывает, что GL'_3 является LL(1)-грамматикой.

Таблица 2.5. *Множества выбора для правил грамматики GL'_3*

	Правило	Множество выбора
1	$S \to LS'$	$\{[, \mathtt{a}\}$
2	$S' \to ;\, S$	$\{;\}$
3	$S' \to \varepsilon$	$\{\dashv\}$
4	$L \to \mathtt{a}$	$\{\mathtt{a}\}$
5	$L \to [S]$	$\{[\}$

Применяя левую факторизацию или устранение левой рекурсии, можно доказать, что любой из языков, обсуждаемых в ч. 1, § 15, порождается LL(1)-грамматикой. Вообще, класс LL(1)-грамматик является довольно широким классом. На основе этого класса реализован ряд практически используемых компиляторов.

§ 5. Алгоритм анализа LL(1)-грамматик

Пусть G – КС-грамматика, из аксиомы которой выводима цепочка $wA\alpha$ (как обычно, A – нетерминал, а w – цепочка терминалов),

и $\mathtt{b}$ – терминал. Предположим, что из цепочки $wA\alpha$ можно вывести цепочку с префиксом $w\mathtt{b}$. Тогда *символ A в этом выводе заменяется по правилу $A \to \beta$ такому, что* $\mathtt{b} \in \textsc{select}(A \to \beta)$.

В самом деле, пусть

$$S \Rightarrow^* wA\alpha \Rightarrow w\beta\alpha \Rightarrow^* w\mathtt{b}\gamma. \tag{2.2}$$

Если символ A в этом выводе не аннулируется, то $\mathtt{b} \in \textsc{first}(\beta)$ и поэтому $\mathtt{b} \in \textsc{select}(A \to \beta)$. Если же A аннулируется, то $\varepsilon \in \textsc{first}(A)$ и $\mathtt{b} \in \textsc{follow}(A)$, откуда снова $\mathtt{b} \in \textsc{select}(A \to \beta)$.

Поскольку множества выбора различных правил LL(1)-грамматики не пересекаются, справедливо следующее замечание.

Замечание 2.3. Для LL(1)-грамматики цепочка β в выводе (2.2) определяется по нетерминалу A единственным образом.

На проведенном рассуждении основан алгоритм синтаксического анализа LL(1)-грамматик, реализованный в виде МП-автомата. Алгоритм анализа использует множества выбора правил грамматики, способ вычисления которых будет приведен в следующем параграфе.

Алгоритм 2.2. *Построение нисходящего синтаксического анализатора для LL(1)-грамматики.*
Вход: LL(1)-грамматика $G = (\Sigma, \Gamma, P, S)$.
Выход: МП-автомат $\mathcal{M} = (\widehat{\Sigma}, \widehat{\Gamma}, \delta, \gamma_0)$.

1. $\widehat{\Sigma} \leftarrow \Sigma \cup \{\dashv\}$
2. $\widehat{\Gamma} \leftarrow \Sigma \cup \Gamma \cup \{\nabla\}$
3. $\gamma_0 \leftarrow S$
4. $\delta \leftarrow \{(\dashv, \nabla) \to \checkmark\}$
5. **для каждого** $(B \to \gamma) \in P$
6. **вычислить** $\textsc{select}(B \to \gamma)$
7. **для каждого** $\mathtt{a} \in \textsc{select}(B \to \gamma)$ **выполнить**
8. $\delta \leftarrow \delta \cup \{(\mathtt{a}, B) \to (\gamma, \_)\}$
9. **для каждого** $\mathtt{a} \in \Sigma$ **выполнить**
10. $\delta \leftarrow \delta \cup \{(\mathtt{a}, \mathtt{a}) \to (\varepsilon, \rightarrow)\}$

Замечание 2.4. Алгоритм 2.2 по любой LL(1)-грамматике строит *детерминированный* МПА (т. е. пригодный для практического использования). В самом деле, если $\mathcal{M}$ содержит две команды с одинаковой левой частью, то они, очевидно, имеют вид $(\mathtt{a}, B) \to (\gamma_1, \_)$ и $(\mathtt{a}, B) \to (\gamma_2, \_)$. Но тогда $\mathtt{a} \in \textsc{select}(B \to \gamma_1) \cap \textsc{select}(B \to \gamma_2)$, что противоречит определению LL(1)-грамматики.

Покажем, что $\mathcal{M}$ действительно является синтаксическим анализатором для грамматики G, т. е. справедлива следующая теорема:

Теорема 2.1. *Если МП-автомат $\mathcal{M}$ построен алгоритмом 2.2 по LL(1)-грамматике G, то $L(G) = L(\mathcal{M})$.*

Доказательство. Пусть автомат $\mathcal{M}$ обрабатывает некоторую цепочку w в течение n тактов, после чего останавливается (отвергая или допуская цепочку)[†]. Для каждого k, $0 \le k \le n$, через w_k обозначим префикс w, обработанный за первые k тактов работы автомата, а через β_k – содержимое стека после k-го такта. Таким образом, $w_0 = \varepsilon$, $\beta_0 = S$. Пусть $\alpha_k = w_k\beta_k$. Рассмотрим последовательность $(\alpha_0, \alpha_1, \ldots, \alpha_n)$. Вычеркнем из нее все элементы α_k такие, что $\alpha_k = \alpha_{k-1}$. После этой процедуры останется подпоследовательность $\chi = (\alpha_0{=}\alpha_{i_0}, \alpha_{i_1}, \ldots, \alpha_{i_s}{=}\alpha_n)$. Докажем, что

(∗) χ – левосторонний вывод цепочки α_n в грамматике G.

Имеем $\alpha_0 = S$. Рассмотрим k-й такт работы автомата. Если на нем применялась команда вида $(\mathtt{a}, \mathtt{a}) \to (\varepsilon, {}^{\rightarrow})$, то $w_k = w_{k-1}\mathtt{a}$ и $\beta_{k-1} = \mathtt{a}\beta_k$, откуда $\alpha_k = \alpha_{k-1}$. Если же применялась команда вида $(\mathtt{a}, B) \to (\gamma, \_)$, то $w_k = w_{k-1}$, $\beta_{k-1} = B\beta'$, $\beta_k = \gamma\beta'$ и $(B \to \gamma) \in P$. Отсюда $\alpha_{k-1} = w_kB\beta' \Rightarrow w_k\gamma\beta' = \alpha_k$, причем в цепочке α_{k-1} правило вывода применено к самому левому нетерминалу, т. е. данный переход является шагом левостороннего вывода. Итак, последовательность χ начинается с аксиомы, заканчивается цепочкой α_n, а любые два соседних ее элемента связаны шагом левостороннего вывода в G. Условие (∗) доказано.

Если автомат допустил цепочку w, то $w_n = w$, $\beta_n = \varepsilon$, $\alpha_n = w$. Значит, из (∗) получаем $w \in L(G)$. Достаточность доказана.

Докажем необходимость. Пусть $w \in L(G)$ и левосторонний вывод имеет вид $(S{=}\gamma_0, \gamma_1, \ldots, \gamma_s{=}w)$. По индукции покажем, что

(∗∗) для любого $i \le s$ найдется k такое, что $\gamma_i = \alpha_k$ (определение α_k дано выше).

Поскольку $\alpha_0 = S = \gamma_0$, база индукции справедлива. Перейдем к шагу индукции. Пусть для некоторого $i < s$ выполняются равенства $\gamma_i = \alpha_k = w_k\beta_k$. Поскольку $i < s$, форма γ_i содержит нетерминал. Пусть A – самый левый нетерминал в γ_i. Поскольку w_k – цепочка

[†]Таким образом, либо не существует команды, которую автомат мог бы выполнить на $(n{+}1)$-м такте, либо эта команда является командой допуска.

терминалов, для некоторого $m \geq 0$ имеем $\gamma_i = w_k \mathtt{a}_1 \ldots \mathtt{a}_m A\beta'$ и соответственно $\beta_k = \mathtt{a}_1 \ldots \mathtt{a}_m A\beta'$. При этом цепочка $w_k \mathtt{a}_1 \ldots \mathtt{a}_m$ является префиксом w, так что необработанная часть входного потока после k-го шага начинается цепочкой $\mathtt{a}_1 \ldots \mathtt{a}_m$. Это значит, что на шагах с $(k{+}1)$-го по $(k{+}m)$-й автомат последовательно «сократит» символы $\mathtt{a}_1, \ldots, \mathtt{a}_m$ в стеке и входном потоке, при этом $\alpha_{k+m} = \alpha_k$ (см. доказательство достаточности).

Мы получили $\gamma_i = \alpha_{k+m}$ и $\beta_{k+m} = A\beta'$. Пусть w' – необработанная к этому моменту часть цепочки w и $\mathtt{c}$ – ее первый символ (если $w' = \varepsilon$, то $\mathtt{c} = \dashv$). Форма γ_{i+1} получена из формы γ_i заменой самого левого нетерминала, т. е. применением некоторого правила $A \to \gamma$. Покажем, что $\mathtt{c} \in \textsc{select}(A \to \gamma)$.

Поскольку $\gamma_{i+1} \Rightarrow^* w$ и эти цепочки имеют общий префикс $w_k \mathtt{a}_1 \ldots \mathtt{a}_m$, то из оставшейся части γ_{i+1} выводима оставшаяся часть w, т. е. $\gamma\beta' \Rightarrow^* w'$. Если в последнем выводе цепочка γ не аннулируется, то $\mathtt{c} \in \textsc{first}(\gamma) \subseteq \textsc{select}(A \to \gamma)$. Если же $\gamma \Rightarrow^* \varepsilon$, то $\beta' \Rightarrow^* w'$. В цепочке γ_i цепочка β' следует непосредственно за нетерминалом A. По определению получаем $\mathtt{c} \in \textsc{follow}(A) \subseteq \textsc{select}(A \to \gamma)$.

Итак, $\mathtt{c} \in \textsc{select}(A \to \gamma)$, тогда в автомате имеется команда $(\mathtt{c}, A) \to (\gamma, \_)$ (строки 7, 8 алгоритма 2.2). Эту команду автомат выполнит на $(k{+}m{+}1)$-м шаге, и мы получим

$$\alpha_{k+m+1} = w_{k+m+1}\beta_{k+m+1} = w_{k+m}\gamma\beta' = w_k \mathtt{a}_1 \ldots \mathtt{a}_m \gamma\beta' = \gamma_{i+1}.$$

Шаг индукции доказан, т. е. условие $(**)$ верно для любого $i \leq s$. В частности, найдется $\bar{k}$ такое, что $\alpha_{\bar{k}} = \gamma_s = w$. Пусть w' – необработанная к этому моменту часть w. Тогда $w_{\bar{k}}\beta_{\bar{k}} = \alpha_{\bar{k}} = w = w_{\bar{k}}w'$, т. е. $\beta_{\bar{k}} = w'$. Следовательно, автомат сократит эти равные цепочки в стеке и входном потоке при помощи команд вида $(\mathtt{a}, \mathtt{a}) \to (\varepsilon, {\to})$, приходя в конфигурацию $[\varepsilon, \varepsilon]$, и применит в ней команду допуска. Итак, $w \in L(\mathcal{M})$. Доказательство теоремы завершено. □

Пример 2.4. Используя алгоритм 2.2 и значения множеств выбора из табл. 2.4, построим управляющую таблицу МП-автомата для грамматики арифметических выражений GA_3 (табл. 2.6). Достаточно указать команды для обработки нетерминалов в стеке, поскольку обработка терминалов в стеке не зависит от конкретной грамматики, а именно, доступна единственная команда вида $(\mathtt{a}, \mathtt{a}) \to (\varepsilon, {\to})$. Для обработки пустого стека (символа ∇) тоже доступна единственная команда $(\dashv, \nabla) \to \checkmark$. Кроме того, заметим, что любая команда для нетерминала в стеке не содержит сдвига по входной цепочке.

Поэтому в клетках таблицы указаны только цепочки, помещаемые в стек, т. е. правые части соответствующих правил вывода грамматики. Отметим также, что если записать все строки таблицы, а затем выполнить «оптимизацию» автомата, как в примере 2.2, то получится табл. 2.12 из ч. 1, § 10.

Таблица 2.6. *Управляющая таблица МПА для грамматики* GA_3

	x	+	*	(	)	⊣
E	TE'			TE'		
E'		$+TE'$			ε	ε
T	FT'			FT'		
T'		ε	$*FT'$		ε	ε
F	x			(E)		

§ 6. Вычисление множеств выбора правил

Как видно из определения, множество выбора правила $A \to \beta$ получается из множеств FIRST(β) и FOLLOW(A) в зависимости от того, содержит ли FIRST(β) пустую цепочку. Следовательно, для вычисления множества SELECT($A \to \beta$) достаточно уметь вычислять множества FIRST(β) и FOLLOW(A).

Вначале приведем итеративный алгоритм вычисления множества FIRST для символов грамматики. Он основан на следующем замечании, вытекающем непосредственно из определения множества FIRST.

Замечание 2.5. Если a – терминал, то FIRST(a) = {a}. Если A – нетерминал, то FIRST(A) есть объединение множеств FIRST(β) для всех цепочек β таких, что $A \to \beta$ является правилом вывода. Предположим, что $\beta = X_1 \dots X_n$. Тогда терминалы из FIRST(X_1) входят в FIRST(β), терминалы из FIRST(X_2) входят в FIRST(β) при условии $\varepsilon \in$ FIRST(X_1), терминалы из FIRST(X_3) входят в FIRST(β) при условии $\varepsilon \in$ FIRST(X_1) $\cap$ FIRST(X_2) и т. д. Наконец, если все символы X_i – аннулирующие, то $\varepsilon \in$ FIRST(β).

Алгоритм 2.3. *Вычисление множеств* FIRST *для нетерминалов.*
Вход: LL(1)-грамматика $G = (\Sigma, \Gamma, P, S)$.
Выход: массив множеств FIRST для символов грамматики.

1. **для каждого** $\mathtt{a} \in \Sigma$ **выполнить**
2. $\quad$ FIRST($\mathtt{a}$) $\leftarrow \{\mathtt{a}\}$
3. **для каждого** $A \in \Gamma$ **выполнить**
4. $\quad$ **если** $(\,(A \to \varepsilon) \in P)$
5. $\quad\quad$ FIRST(A) $\leftarrow \{\varepsilon\}$
6. $\quad$ **иначе** FIRST(A) $\leftarrow \emptyset$
7. **пока** все множества FIRST(A) не стабилизировались, **повторять**
8. $\quad$ **для каждого** $(A \to X_1 \dots X_n) \in P$, где $n > 0$, **выполнить**
9. $\quad\quad$ $i \leftarrow 1$
10. $\quad\quad$ FIRST(A) $\leftarrow$ FIRST(A) $\cup$ (FIRST(X_i) $\cap \Sigma$)
11. $\quad\quad$ **если** $(\varepsilon \in$ FIRST(X_i)$\,)$
12. $\quad\quad\quad$ **если** $(i < n)$
13. $\quad\quad\quad\quad$ $i \leftarrow i{+}1$; перейти на 10
14. $\quad\quad\quad$ **иначе** FIRST(A) $\leftarrow$ FIRST(A) $\cup \{\varepsilon\}$

Имея в распоряжении массив значений FIRST для грамматических символов, нетрудно вычислить FIRST(β) для любой цепочки $\beta = X_1 \dots X_n$. Достаточно инициализировать FIRST(β) $= \emptyset$ и применить процедуру в строках 9–14 алгоритма 2.3, заменив A на β. Корректность такого вычисления следует из замечания 2.5.

Теперь перейдем к вычислению множеств FOLLOW. Для этого, в частности, потребуются значения FIRST для некоторых цепочек. Приводимый ниже алгоритм основан на следующем замечании.

Замечание 2.6. По определению множества FOLLOW, выполняется $\dashv \in$ FOLLOW(S). Для того чтобы определить, какие символы из $\Sigma \cup \{\dashv\}$ могут следовать за произвольным нетерминалом B, нужно рассмотреть все вхождения этого нетерминала в правые части правил вывода. Пусть такое вхождение имеет вид $A \to \alpha B \beta$. Терминал $\mathtt{a}$ может следовать непосредственно за B в некоторой выводимой из аксиомы цепочке в одном из двух случаев. Во-первых, $\mathtt{a}$ может начинать некоторую цепочку, выводимую из β; тогда $\mathtt{a} \in$ FIRST(β). Во-вторых, когда $\beta \Rightarrow^* \varepsilon$, символ $\mathtt{a}$ может находиться правее β (или выводиться из нетерминала, находящегося правее β)[†]. Тогда, «откатив» применение правила $A \to \alpha B \beta$, мы видим, что $\mathtt{a}$ может следовать непосредственно за нетерминалом A, т. е. $\mathtt{a} \in$ FOLLOW(A).

Алгоритм 2.4. *Вычисление множеств* FOLLOW *для нетерминалов.*
Вход: LL(1)-грамматика $G = (\Sigma, \Gamma, P, S)$.
Выход: массив множеств FOLLOW для нетерминалов грамматики.

[†]Сюда же относится случай $\mathtt{a} = \dashv$.

1. **для каждого** $A \in \Gamma$ **выполнить**
2. FOLLOW$(A) \leftarrow \emptyset$
3. FOLLOW$(S) \leftarrow \{\dashv\}$
4. **пока** `все множества` FOLLOW `не стабилизировались`, **повторять**
5. **для каждого** $(A \to X_1 \dots X_n) \in P$, `где` $n > 0$, **выполнить**
6. $i \leftarrow n$; $ann \leftarrow$ `true`
7. **если** $(X_i \in \Gamma)$
8. FOLLOW$(X_i) \leftarrow$FOLLOW$(X_i) \cup$(FIRST$(X_{i+1} \dots X_n) \cap \Sigma$)
9. **если** (ann)
10. FOLLOW$(X_i) \leftarrow$ FOLLOW$(X_i) \cup$ FOLLOW(A)
11. **если** $i > 1$
12. **если** $(\varepsilon \notin$ FIRST$(X_i))$
13. $ann \leftarrow$ `false`
14. $i \leftarrow i-1$; `перейти на 7`

Флаг *ann* используется для хранения информации о том, является ли «хвост» правой части рассматриваемого правила аннулирующим. Отметим, что просмотр правых частей правил «справа налево» позволяет существенно сократить вычисления.

В заключение параграфа рассмотрим подробно пример построения множеств FIRST и FOLLOW.

Пример 2.5. Применим алгоритмы 2.3 и 2.4 к грамматике арифметических выражений GA_3 и рассмотрим, как шаг за шагом будут «пополняться» множества FIRST и FOLLOW для нетерминалов. Для удобства занумеруем неаннулирующие правила грамматики:

$$(1)\ E \to TE' \quad (2)\ E' \to +TE' \quad (3)\ T \to FT'$$
$$(4)\ T' \to *FT' \quad (5)\ F \to (E) \quad (6)\ F \to \mathtt{x}$$

Вначале рассмотрим построение множеств FIRST (алгоритм 2.3). Договоримся добавление символа `a` к множеству FIRST(A) записывать в виде $\mathtt{a} \mapsto A$, указывая рядом, при обработке какого из правил (1)–(6) это добавление произведено. Будем перебирать правила «снизу вверх» – от правил, описывающих более частные синтаксические категории, к правилам, описывающим более общие. Тогда «протокол» построения массива множеств FIRST будет выглядеть как в табл. 2.7 слева. Таким образом, правильные значения FIRST будут получены уже на первой итерации внешнего цикла алгоритма. Читателю предлагается проверить, что при обходе правил «сверху вниз» значение FIRST(E) будет получено лишь на третьей итерации этого цикла.

Таблица 2.7. *Построение* FIRST *и* FOLLOW *для грамматики* GA_3

Построение FIRST	Построение FOLLOW
$\varepsilon \mapsto E'$	$\dashv \mapsto E$
$\varepsilon \mapsto T'$	(1) $\dashv \mapsto E'$
(6) $\mathtt{x} \mapsto F$	(1) $+ \mapsto T$
(5) $(\mapsto F$	(1) $\dashv \mapsto T$
(4) $* \mapsto T'$	(3) $+, \dashv \mapsto T'$
(3) $\mathtt{x}, (\mapsto T$	(3) $* \mapsto F$
(2) $+ \mapsto E'$	(3) $+, \dashv \mapsto F$
(1) $\mathtt{x}, (\mapsto E$	(5) $) \mapsto E$
далее без изменений	(1) $) \mapsto E'$
	(1) $) \mapsto T$
	(3) $) \mapsto T'$
	(3) $) \mapsto F$
	далее без изменений

Аналогичный протокол построения множеств FOLLOW (алгоритм 2.4) приведен в той же таблице справа. Независимо от того, «сверху вниз» или «снизу вверх» просматриваются правила данной грамматики, правильные значения множеств будут получены на второй итерации внешнего цикла. Итоговые значения множеств FIRST и FOLLOW приведены в табл. 2.8. Из них по определению получаем множества выбора правил грамматики, приведенные в табл. 2.4, и управляющую таблицу синтаксического анализатора из табл. 2.6.

Таблица 2.8. *Множества* FIRST *и* FOLLOW *для грамматики* GA_3

Символ	FIRST	FOLLOW
E	$(, \mathtt{x}$	$), \dashv$
E'	$+, \varepsilon$	$), \dashv$
T	$(, \mathtt{x}$	$+,), \dashv$
T'	$*, \varepsilon$	$+,), \dashv$
F	$(, \mathtt{x}$	$+, *,), \dashv$

§ 7. Обработка синтаксических ошибок

Хороший синтаксический анализатор должен сообщать о наличии ошибки с указанием места в программе, где эта ошибка была

обнаружена, и предполагаемого типа ошибки. Но этого недостаточно. Анализатор должен уметь «исправить»[†] ошибку так, чтобы стало возможным продолжение анализа входной цепочки для обнаружения последующих ошибок (или установления их отсутствия). Как мы увидим, построенный нисходящий анализатор для LL(1)-грамматик обладает весьма ограниченными возможностями исправления ошибок.

Рассматриваемый анализатор обнаруживает ошибку в тот момент, когда не может продолжить работу из-за отсутствия в МП-автомате команды, левая часть которой есть текущая пара (входной символ, верхний символ стека). В соответствующую пустую клетку управляющей таблицы можно поместить ссылку на подпрограмму обработки возникшей ошибки. Для примера разберем типы синтаксических ошибок для грамматики GA_3. Их четыре:

(1) отсутствует операнд (например, два оператора подряд);
(2) отсутствует оператор (два операнда подряд);
(3) преждевременная правая скобка;
(4) незакрытая левая скобка.

Цепочка `)(x+x)(*x+x` содержит все эти ошибки[‡]. В табл. 2.9 приведены ключевые шаги разбора этой цепочки, на которых обнаруживаются и обрабатываются ошибки. Весь ход разбора читателю предлагается восстановить самостоятельно. (Управляющая таблица МП-автомата приведена в табл. 2.6).

Таблица 2.9. *Протокол обработки ошибок в «арифметической» цепочке*

Такт	Позиция указателя	Содержимое стека	Исправление
1	◇`)(x+x)(*x+x`⊣	$E\nabla$	пропустить)
19	`)(x+x)`◇`(*x+x`⊣	$T'E'\nabla$	вставить *
23	`)(x+x)(`◇`*x+x`⊣	$E)T'E'\nabla$	пропустить *
36	`)(x+x)(*x+x`◇⊣	$)T'E'\nabla$	вставить)

В приведенном примере анализатор может дойти до конца входной цепочки и обнаружить все ошибки в ней. Однако посмотрим, что произойдет, если обработать цепочку `(x+x))(*x+x` , полученную из

[†] В случае МП-автомата – подходящим образом скорректировав содержимое стека и входного потока.

[‡] Напомним, что в данной грамматике `x` надо рассматривать как токен, в который превратились все переменные и константы исходного выражения.

предыдущей переносом одной правой скобки (табл. 2.10). Лишняя правая скобка в ней будет обнаружена слишком поздно, когда ничего исправить уже нельзя – стек пуст и продолжение анализа после исправления ошибки невозможно.

Таблица 2.10. *Неудачная обработка ошибки в «арифметической» цепочке*

Такт	Позиция указателя	Содержимое стека	Исправление
18	(x+x)◇)(*x+x⊣	$T'E'\nabla$	
19	(x+x)◇)(*x+x⊣	$E'\nabla$	
20	(x+x)◇)(*x+x⊣	∇	невозможно

Рассмотрим кратко общую стратегию обработки ошибок при нисходящем анализе. Основной метод использует *режим паники*: пропуск символов из входной цепочки до тех пор, пока не будет обнаружен терминал, позволяющий продолжить анализ. Что это может быть за терминал? Если ошибка обнаружена в момент, когда наверху стека находится нетерминал A, то естественно предположить, что она локализована во фрагменте, выводимом из A. По окончании этого фрагмента можно снять A со стека и продолжить анализ. Терминал, следующий за фрагментом, выводимым из A, по определению должен принадлежать множеству $\textsc{follow}(A)$. Итак, базовая идея состоит в пропуске нуля или более символов во входной цепочке до первого символа из $\textsc{follow}(A)$, после чего A снимается со стека. Возможны различные усовершенствования этой идеи. Например, если мы раньше встретим символ из множества $\textsc{first}(A)$, то можем попытаться продолжить анализ, оставляя A в стеке.

В случае когда при обнаружении ошибки наверху стека находится терминал, обработка очевидна: этот терминал надо снять со стека (или, эквивалентно, вставить его во входную цепочку) и продолжить анализ.

Как видно из разобранных примеров, в некоторых случаях никакая обработка ошибки не позволит продолжить анализ, так как ошибочный символ во входной цепочке может непоправимо повредить стек еще до того, как ошибку удастся обнаружить. Этот недостаток нисходящего анализа связан, в частности, с тем, что преобразования, устраняющие левую рекурсию, «размывают» синтаксические категории языка, представляемые нетерминалами порождающей грамматики. Нетерминалы E, T и F грамматики арифметиче-

ских выражений GA_2 (см. с. 70) представляли три основные категории соответствующего языка – выражение, слагаемое и множитель. При устранении левой рекурсии появились «хвост выражения» E' и «хвост слагаемого» T', которые и привели к неудачной обработке ошибки в табл. 2.10.

§ 8. LL(k)-грамматики и языки

Класс LL(1)-грамматик допускает различные обобщения. В литературе обычно рассматривается иерархия классов LL(k)-грамматик при различных k (см. напр., [АУ1]). Дадим основные определения.

Определение. Пусть G – КС-грамматика, α – цепочка грамматических символов, k – натуральное число. Тогда $\textsc{first}_k(\alpha)$ – это подмножество из Σ^*, состоящее из всех цепочек w таких, что
– либо $|w| = k$ и $\alpha \Rightarrow^* w\beta$ для некоторой цепочки β,
– либо $|w| < k$ и $\alpha \Rightarrow^* w$.

Ясно, что определенное в § 4 множество $\textsc{first}(\alpha)$ совпадает с $\textsc{first}_1(\alpha)$. Обозначением $\textsc{first}_k(\alpha)$ можно пользоваться и в случае, когда $\alpha \in \Sigma^*$, используя то, что $\Rightarrow^*$ – это рефлексивно-транзитивное замыкание. Например, $\textsc{first}_2(\mathtt{aba}) = \mathtt{ab}$, $\textsc{first}_4(\mathtt{aba}) = \mathtt{aba}$.

Определение. КС-грамматика G называется *LL(k)-грамматикой*, если из существования выводов

$$S \Rightarrow^* wA\alpha \Rightarrow w\beta\alpha \Rightarrow^* wu,$$
$$S \Rightarrow^* wA\alpha \Rightarrow w\gamma\alpha \Rightarrow^* wv,$$

для которых $\textsc{first}_k(u) = \textsc{first}_k(v)$, следует, что $\beta = \gamma$. Грамматика G называется *LL-грамматикой*, если она является LL(k)-грамматикой для некоторого k.

Замечание 2.3 показывает, что приведенное выше определение LL(k)-грамматики при $k = 1$ согласовано с определением LL(1)-грамматики, приведенным в § 4.

Пусть $LL(k)$ обозначает класс всех LL(k)-грамматик. Включения $LL(1) \subseteq LL(2) \subseteq \ldots \subseteq LL(k) \subseteq \ldots$ легко следуют из определения. Нетрудно убедиться, что эти включения являются строгими. Действительно, грамматика $G = \{S \to \mathtt{a}^k\mathtt{b} \mid \mathtt{a}^k\mathtt{c}\}$ принадлежит классу $LL(k{+}1)$ и не принадлежит классу $LL(k)$.

Существуют достаточно простые грамматики, которые не являются LL-грамматиками. Следующий пример взят из [АУ1]:

$$G_{2,3} = \{S \to A \,|\, B,\ A \to \mathtt{a}A\mathtt{b} \,|\, 0,\ B \to \mathtt{a}B\mathtt{bb} \,|\, 1\}.$$

Мы видим, что

$$S \Rightarrow^* S \Rightarrow A \Rightarrow^* u = \mathtt{a}^k 0 \mathtt{b}^k,$$
$$S \Rightarrow^* S \Rightarrow A \Rightarrow^* v = \mathtt{a}^k 1 \mathtt{b}^{2k},$$

при этом $\textsc{first}_k(u) = \textsc{first}_k(v)$, но $A \neq B$. Так как число k может быть любым, грамматика $G_{2,3}$ не является LL-грамматикой.

Способы построения анализаторов для LL(k)-грамматик известны, но при $k > 1$ управляющие таблицы становятся громоздкими и редко используются на практике.

Перейдем к LL-языкам.

Определение. Язык называется *LL(k)-языком*, если существует LL(k)-грамматика, порождающая этот язык. Язык называется *LL-языком*, если он является LL(k)-языком для некоторого числа k.

Оказывается, что, как и в случае LL(k)-грамматик, LL(k)-языки образуют собственное подмножество в классе всех LL(k+1)-языков. Существуют довольно простые языки, которые не являются LL-языками. В качестве такого языка можно взять язык, порожденный приведенной выше грамматикой $G_{2,3}$. С теорией LL-языков также можно ознакомиться по книге [АУ1].

§ 9. Метод рекурсивного спуска

В заключительном параграфе главы мы кратко рассмотрим общий метод нисходящего анализа, позволяющий, в принципе, использовать любые нелеворекурсивные грамматики. Основная идея *метода рекурсивного спуска* состоит в следующем. Для каждого нетерминала A пишется своя процедура A, которая напрямую работает с входной цепочкой, распознавая в ней фрагмент, выводимый из A. Эта процедура осуществляет выбор правила вывода (из правил вида $A \to \alpha$) и обработку этого правила, при которой, в частности, рекурсивно вызываются процедуры для нетерминалов в правой части правила[†]. Если обработка завершена успешно (получен вывод фрагмента входной цепочки), то процедура завершается, передавая управление вызвавшей ее процедуре. Если обработка не была успешной, производится «откат» – возврат в точку выбора правила для нового выбора. Если никакой выбор правила невозможен, происходит

[†]Таким образом, наличие леворекурсивного правила в грамматике может вызвать бесконечную рекурсию.

более далекий откат: управление передается вызывающей процедуре для нового выбора правила в ней. Если все возможности для отката исчерпаны, выдается сообщение об ошибке во входной цепочке. «Запуск» распознавателя осуществляется вызовом процедуры S для аксиомы.

Остановимся подробнее на выборе и обработке правила. В доказательстве теоремы 2.1 отмечалось, что если из нетерминала A выводится цепочка, начинающаяся с терминала $\mathtt{c}$ и этот вывод начинается применением правила $A \to \alpha$, то $\mathtt{c} \in \textsc{select}(A \to \alpha)$. Таким образом, для выбора правила в процедуре A будем использовать таблицу МП-автомата, построенную на основе множеств SELECT для правил в соответствии с алгоритмом 2.2. Этот автомат для LL(1)-грамматики является детерминированным (замечание 2.4), а в общем случае – недетерминированным, т. е. в одной клетке могут быть несколько записей. Откаты возможны как раз при выборе одной из таких записей. Тем самым для LL(1)-грамматик алгоритм рекурсивного спуска работает без откатов.

Обработка правила $A \to X_1 \dots X_n$ состоит в последовательной обработке символов $X_1, \dots, X_n$. Если X_i – терминал, то он сравнивается с текущим входным символом $\mathtt{c}$. При совпадении выполняется сдвиг по входной цепочке и переход к обработке X_{i+1}, а при несовпадении производится откат. Если X_i – нетерминал, то вызывается соответствующая процедура. После обработки символа X_n работа процедуры A завершается.

В третьей части книги будет построен транслятор, работающий данным методом и реализующий, помимо синтаксического анализа цепочки, ряд дополнительных функций.

Задачи

1. Для каждого из следующих языков напишите разделенную грамматику, порождающую этот язык, и управляющую таблицу соответствующего МП-распознавателя:

а) $\{\mathtt{a}^n\mathtt{c}\mathtt{b}^n \mid n \in \mathbb{N}_0\}$;
б) $\{wc\overleftarrow{w} \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$;
в) $\{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$;
г) $\{\mathtt{a}^n\mathtt{c}\mathtt{b}^n \mid n \in \mathbb{N}\} \cup \{\mathtt{c}\mathtt{a}^n\mathtt{c}\mathtt{b}^{2n} \mid n \in \mathbb{N}\}$;
д) $\{\mathtt{a}^n\mathtt{c}\mathtt{b}^n\mathtt{a}^m\mathtt{c}\mathtt{b}^m \mid n \in \mathbb{N}, m \in \mathbb{N}_0\}$.

2. Грамматика G называется *квазиразделенной*, если она ε-свободна и для любых двух (различных) правил вывода $A \to \beta$ и $A \to \gamma$ таких, что $\beta \neq \varepsilon$ и $\gamma \neq \varepsilon$, выполняется условие $\text{FIRST}(\beta) \cap \text{FIRST}(\gamma) = \emptyset$.

Для каждого из следующих языков постройте квазиразделенную грамматику, порождающую этот язык, и соответствующий МП-распознаватель. Докажите, что для этих языков не существует порождающей разделенной грамматики:

а) $\{\mathtt{a}^n \mid n \in \mathbb{N}\}$;

б) $\{\mathtt{a}^m\mathtt{b}^n \mid m, n \in \mathbb{N}_0, m \leq n\}$;

в) $\{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}_0\}$.

3. Для следующих грамматик найдите множество выбора каждого правила. Будут ли эти грамматики квазиразделенными (см. задачу 2), будут ли они LL(1)-грамматиками?

а) $S \to \mathtt{a}AB \mid \mathtt{b}BS \mid \varepsilon, A \to \mathtt{c}BS \mid \varepsilon, B \to \mathtt{d}B \mid \varepsilon$;

б) $S \to \mathtt{a}A\mathtt{b}B\mathtt{b}S \mid \varepsilon, A \to \mathtt{a}BC \mid \mathtt{b}A, B \to \mathtt{a}B \mid \varepsilon, C \to \mathtt{c}C \mid \varepsilon$;

в) $S \to \mathtt{a}AB \mid \mathtt{b}A \mid \varepsilon, A \to \mathtt{a}A\mathtt{b} \mid \varepsilon, B \to \mathtt{b}B \mid \mathtt{c}$;

г) $S \to \mathtt{a}AB\mathtt{b}CD \mid \varepsilon, A \to AS\mathtt{d} \mid \varepsilon, B \to SA\mathtt{c} \mid \mathtt{e}C \mid \varepsilon,$
$C \to S\mathtt{f} \mid C\mathtt{g} \mid \varepsilon, D \to \mathtt{a}AD \mid \varepsilon$.

4. Доказать, что каждый регулярный язык порождается подходящей квазиразделенной грамматикой (см. задачу 2).

5. Пусть $L \in \Sigma$ – регулярный язык. Доказать, что язык $L\mathtt{a}$, где $\mathtt{a} \notin \Sigma$, порождается разделенной грамматикой.

6*. Доказать, что язык $\{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\} \cup \{\mathtt{a}^n\mathtt{c}^n \mid n \in \mathbb{N}\}$ не порождается LL(1)-грамматикой.

7. Среди грамматик, порождающих языки LB, LL, LN (см. ч. 1, § 15), найти LL(1)-грамматики.

8. Построить LL(1)-грамматики для следующих конструкций языка Паскаль, считая терминалами `<логическое_выражение>`, `<оператор>` и `<арифметическое_выражение>`:

а) описания массивов;

б) описания переключателей;

в) операторы цикла.

9. Грамматика $G = \{S \to D{\uparrow}S \mid D, D \to \mathtt{a} \mid (S)\}$ порождает выражения с операцией возведения в степень. Возведение в степень в отсутствие скобок выполняется справа налево. Показать, что G – LL(1)-грамматика, составить управляющую таблицу МП-распознавателя.

10. Грамматика G задана списком правил:

`<оператор>` → `begin <описание>, <список_операторов> end`

`<описание>` → `d; <описание>` | ε

`<список_операторов>` → `s; <список_операторов>` | ε

Показать, что G – LL(1)-грамматика, составить управляющую таблицу МП-распознавателя.

11. Доказать, что любая LL(1)-грамматика однозначна.

12. Доказать, что МП-анализатор из табл. 2.6 не может обнаружить синтаксическую ошибку при сравнении *терминала* на верхушке стека и терминала во входной цепочке.

13. Доказать, что грамматики

$G_1 = \{S \to BA, A \to BS \mid \mathtt{d}, B \to \mathtt{a}A \mid \mathtt{b}S \mid \mathtt{c}\}$,

$G_2 = \{S \to \mathtt{a}AA \mid \mathtt{b}SA \mid \mathtt{c}A, A \to \mathtt{a}AS \mid \mathtt{b}SS \mid \mathtt{c}S \mid \mathtt{d}\}$

порождают один и тот же язык. Убедиться в том, что G_2 – разделенная грамматика, а G_1 – LL(1)-грамматика, построить управляющие таблицы соответствующих МП-автоматов, сравнить работу этих автоматов при распознавании цепочки `abcdbcdccdd`.

14. Написать LL(1)-грамматику, порождающую множество регулярных выражений над алфавитом $\{\mathtt{a}, \mathtt{b}\}$. По грамматике составить управляющую таблицу соответствующего МП-распознавателя.

15. Написать LL(1)-грамматику, порождающую множество формул логики высказываний, построенных из букв `X` и `Y`, логических связок $\neg$, &, $\vee$, $\to$, $\leftrightarrow$ и скобок. Считать, что связка $\neg$ имеет наивысший приоритет, далее идут & и $\vee$ (эти связки имеют одинаковый приоритет), связки $\to$ и $\leftrightarrow$ имеют самый низкий приоритет. По грамматике составить управляющую таблицу МП-распознавателя.

Глава 3. Восходящий анализ на основе отношений предшествования

При восходящем анализе дерево вывода строится от листьев к корню, т. е. цепочка терминалов «сворачивается» в аксиому грамматики, при этом строится (от конца к началу) *правосторонний* вывод этой цепочки. Примем ряд соглашений. В этой главе по умолчанию КС-грамматики являются (1) *приведенными*, (2) *ε-свободными*, (3) *однозначными*, а выводы являются правосторонними.

Первое условие на грамматику не является обременительным, так как алгоритм 4.3 построения эквивалентной приведенной грамматики (см. ч. 1, § 20) вычислительно прост и может разве что уменьшить размер исходной грамматики. Второе и третье условия более существенны. Алгоритм 4.5 построения эквивалентной ε-свободной

грамматики (см. ч. 1, § 21) в практических ситуациях часто оказывается бесполезным, поскольку в результате его применения число правил вывода может сильно возрасти. Как отмечалось в первой части книги, для проверки третьего условия вообще не существует универсального алгоритма, при этом однозначность очевидно необходима для любого практически используемого формализованного языка.

Условие, касающееся выводов, не ограничивает общности рассуждений, так как для выводимой в грамматике терминальной цепочки всегда существует правосторонний вывод. Из определений очевидно, что среди всех *форм* в этом случае нужно рассматривать только *r-формы*.

§ 10. Общая схема восходящего анализа

Определим основное понятие восходящего анализа.

Определение. Пусть $S{=}\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n{=}w$ – правосторонний вывод в грамматике G. Для любого $k \geq 1$ *основа* r-формы α_k – это цепочка β такая, что $\alpha_{k-1} = \gamma_1 A \gamma_2$, $\alpha_k = \gamma_1 \beta \gamma_2$ для некоторых цепочек γ_1, γ_2 и нетерминала A [†].

Ввиду однозначности грамматики основа r-формы определяется единственным образом. В качестве примера рассмотрим грамматику

$$G_{3,1} = \{S \to \mathtt{a}FS\mathtt{d} \,|\, \mathtt{c}, F \to F\mathtt{b} \,|\, \mathtt{b}\}\,.$$

Цепочка $\mathtt{a}F\mathtt{bcd}$ является r-формой в этой грамматике; ее основой, как нетрудно видеть, является цепочка $F\mathtt{b}$.

Для удобства обсуждения понятий, связанных с выводом цепочек при восходящем анализе, дадим следующее

Определение. Поддерево K дерева T называется *кустом*, если оно имеет высоту 1 и для любого листа x этого поддерева выполнены условия

1) x и все его братья в T являются листьями T;

2) все братья x в T являются листьями K.

[†]Цепочка β может входить в α_k несколько раз. Строго говоря, основа – это цепочка β *вместе с ее местоположением в форме* α_k, т. е. основу надо задавать парой вида (β, m), где $m = |\gamma_1|$. Но поскольку местоположение основы в форме обычно ясно из контекста, нам удобнее говорить об основе как о цепочке.

В частности, любое нетривиальное дерево имеет хотя бы один куст. На рис. 3.1 изображено дерево и выделены все его кусты. Мы будем рассматривать кусты в деревьях вывода и их стандартных поддеревьях. Поскольку все узлы такого дерева линейно упорядочены (напомним, что мы называем этот порядок «порядком слева направо» и обозначаем знаком $\lhd$), можно считать, что все кусты дерева также линейно упорядочены слева направо порядком на корнях кустов.

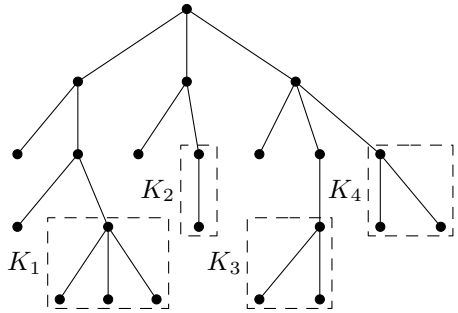


Рис.3.1. *Все кусты дерева.* $K_1 \lhd K_2 \lhd K_3 \lhd K_4$

Замечание 3.1. Пусть правосторонний вывод $S \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n = w$ в грамматике G представлен деревом вывода T, а T_i и T_{i+1} – стандартные поддеревья в T, представляющие формы α_i и α_{i+1}. Тогда дерево T_i получается из T_{i+1} удалением всех листьев *самого левого куста*. Это означает, что *на листьях самого левого куста стандартного поддерева слева направо написана основа r-формы, представленной этим поддеревом.*

Восходящий анализ работает по следующей схеме. Предполагается, что терминальная цепочка $w = \gamma_0$ выводима из аксиомы. Находится предполагаемая основа этой цепочки – некоторая цепочка β. Если грамматика не содержит правила $A \to \beta$ ни для какого нетерминала A, то делается вывод о том, что цепочка γ_0 невыводима. Если правило указанного вида единственно, то производится *свертка по этому правилу*, т. е. основа цепочки γ_0 заменяется символом A. Если правил вида $A \to \beta$ несколько, то из них выбирается одно, после чего производится свертка. (Способ выбора правила предписывается конкретным методом анализа и, в частности, может предполагать возможность возврата в данную точку выбора.) Весь процесс повто-

ряется для цепочки γ_1, полученной из γ_0 указанной сверткой, и т. д. Если некоторая цепочка γ_n равна аксиоме грамматики, то цепочка w выводима в грамматике и $S{=}\gamma_n \Rightarrow \gamma_{n-1} \Rightarrow \ldots \Rightarrow \gamma_0{=}w$ – ее правосторонний вывод.

Пример 3.1. Возьмем грамматику $G_{3,1}$ и цепочку $w = \texttt{abbbcd}$. Рисунок 3.2, *а* иллюстрирует процесс нахождения основы и последующей свертки. Мы видим, что вначале выполнялась свертка по правилу $F \to \texttt{b}$, затем дважды по правилу $F \to F\texttt{b}$, после этого свертка выполнялась по правилам $S \to \texttt{c}$ и $S \to \texttt{a}FS\texttt{d}$. Дерево вывода цепочки w приведено на рис. 3.2, *б*. Процесс «сворачивания» цепочки в аксиому соответствует последовательной «обрезке» кустов $K_1 - K_5$ (поддерево K_2 становится кустом после «обрезки» K_1 и т. д.).

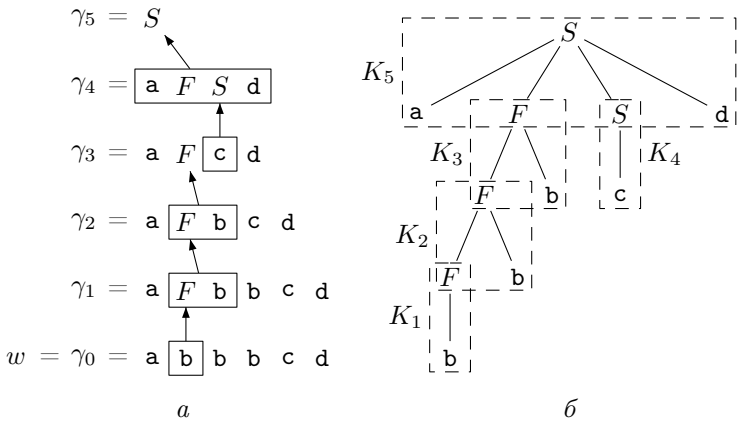


Рис. 3.2. *Последовательность сверток (а), дерево вывода (б) цепочки w в грамматике $G_{3,1}$*

Восходящие методы анализа обычно реализуют при помощи стека. Идея алгоритма такова: вначале стек пуст; цепочка, записанная на входной ленте, посимвольно переносится в стек до тех пор, пока наверху стека не окажется основа цепочки (способ поиска основы определяется используемым методом анализа). Затем в стеке производится свертка по соответствующему правилу[†]. Процедура повторяется, пока вся цепочка не перенесена в стек (и не произведены возможные свертки по завершении такого переноса). Если в итоге

[†]Такие анализаторы часто называют анализаторами типа *перенос–свертка*.

стек содержит только аксиому грамматики, то цепочка допускается. Корректность такого алгоритма опирается на тот факт, что каждая основа обязательно окажется наверху стека (см. предложение 3.1 ниже). Алгоритм может быть представлен МП-автоматом, но не всегда оказывается удобным выписывать управляющую таблицу явно.

В табл. 3.1 представлен вывод цепочки $w = \texttt{abbbcd}$ в грамматике $G_{3,1}$ при помощи стека.

Таблица 3.1. *Протокол восходящего анализа цепочки в стеке*

Такт	Содержимое стека	Позиция указателя
1	∇	$\diamond$abbbcd $\dashv$
2	∇a	a$\diamond$bbbcd $\dashv$
3	∇ab	ab$\diamond$bbcd $\dashv$
4	∇aF	ab$\diamond$bbcd $\dashv$
5	∇aFb	abb$\diamond$bcd $\dashv$
6	∇aF	abb$\diamond$bcd $\dashv$
7	∇aFb	abbb$\diamond$cd $\dashv$
8	∇aF	abbb$\diamond$cd $\dashv$
9	∇aFc	abbbc$\diamond$d $\dashv$
10	∇aFS	abbbc$\diamond$d $\dashv$
11	∇aFSd	abbbcd$\diamond$ $\dashv$
12	∇S	abbbcd$\diamond$ $\dashv$

Замечание 3.2. В течение этой и следующей глав при записи содержимого стека мы пишем символ дна слева, а верхушку – справа, считая ее последним символом стековой цепочки, а не первым, как до сих пор. Тогда произведение содержимого стека на необработанную часть входной цепочки w является r-формой в выводе w (ср. табл. 3.1 и рис. 3.2, *а*).

Предложение 3.1. *Если восходящий анализ цепочки производится при помощи стека, то верхний символ стека либо принадлежит основе текущей r-формы, либо находится слева от этой основы.*

Доказательство. Докажем этот факт индукцией по порядку сверток. База индукции выполняется: к моменту первой свертки основа полностью перенесена в стек и на вершине находится ее последний символ. Для доказательства шага индукции рассмотрим два последовательных шага в правом выводе терминальной цепочки. Нетер-

минал B, разворачиваемый на втором из этих шагов, либо был выведен из нетерминала A, развернутого на первом шаге, либо находился слева от него:

$$S \Rightarrow^* \alpha A w \Rightarrow \alpha\beta B v w \Rightarrow \alpha\beta\gamma v w, \tag{3.1}$$

$$S \Rightarrow^* \alpha B u A w \Rightarrow \alpha B u v w \Rightarrow \alpha\beta u v w. \tag{3.2}$$

Рассмотрим вывод (3.1). Основой последней r-формы является цепочка γ. По предположению индукции верхний символ стека принадлежит основе. По замечанию 3.2 при помощи переноса можно добиться того, чтобы в стеке была цепочка $\alpha\beta\gamma$, а необработанная часть входной цепочки была равна vw. После свертки в стеке будет цепочка $\alpha\beta B$, и ее верхний символ принадлежит основе $\beta B v$ очередной r-формы. Шаг индукции выполнен.

В случае вывода (3.2) основа последней r-формы равна β, и при помощи переноса можно получить цепочку $\alpha\beta$ в стеке и цепочку uvw на входе. После свертки наверху стека окажется символ B, который находится левее цепочки v, являющейся основой очередной r-формы. Шаг индукции снова выполнен. Предложение доказано. □

В книге будут рассмотрены два типа алгоритмов восходящего анализа. Алгоритм первого типа анализирует пары соседних символов в текущей цепочке и на основании этого делает вывод о местоположении левой и правой границ основы. После определения границ производится свертка по соответствующему правилу вывода. Эти алгоритмы основаны на так называемых *отношениях предшествования*. В алгоритмах второго типа для выделения основы используется вся информация о символах, расположенных до предполагаемой правой границы основы (а не только информация о соседних символах), плюс информация о k символах, идущих после правой границы. Этот подход более сложен, но позволяет использовать значительно более широкий класс грамматик – так называемых *LR(k)-грамматик*. На втором подходе базируется основная промышленная технология синтаксического анализа. Изучению первого типа алгоритмов анализа посвящена данная глава. Алгоритмы анализа LR(k)-грамматик будут рассмотрены в следующей главе.

§ 11. Отношения простого предшествования

В предыдущем параграфе отмечалось, что определение границ основы возможно в результате анализа пар соседних символов в це-

почке. Будем считать, что информация, необходимая для такого анализа, представлена в виде трех бинарных отношений $\doteq$, $\lessdot$, $\gtrdot$, определенных на множестве всех грамматических символов.

Определение. Пусть X и Y – грамматические символы. Тогда

$X \doteq Y$, если XY содержится в основе некоторой r-формы;

$X \lessdot Y$, если основа некоторой r-формы начинается с символа Y, перед которым стоит символ X;

$X \gtrdot Y$, если основа некоторой r-формы заканчивается символом X, после которого стоит символ Y.

Рисунок 3.3 иллюстрирует эти отношения.

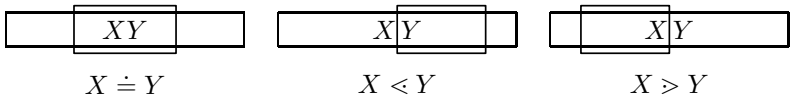


Рис.3.3. *Расположение основы в цепочке и отношения предшествования*

Замечание 3.3. Поскольку мы рассматриваем только r-формы, в случае отношения $\gtrdot$ можно считать, что Y – терминал.

Введенные отношения называются *отношениями простого предшествования*. Существуют и другие, например, отношения *операторного* предшествования, рассматриваемые в § 14.

Если исходить только из приведенного определения отношений предшествования, то неясно, как узнать, какие из отношений предшествования выполняются для данных символов X и Y, поскольку множество r-форм, как правило, бесконечно. При вычислении отношений простого предшествования для данной грамматики будем пользоваться следующим утверждением.

Теорема 3.1. *Пусть $G = (\Sigma, \Gamma, P, S)$ – КС-грамматика, $X, Y \in \Sigma \cup \Gamma$. Тогда*

1) $X \doteq Y$, если и только если существует правило вывода вида $A \to \alpha XY\beta$;

2) $X \lessdot Y$, если и только если существует правило вывода вида $A \to \alpha XZ\beta$ такое, что $Z \Rightarrow^+ Y\gamma$;

3) $X \gtrdot Y$, если и только если Y – терминал и существует правило вывода вида $A \to \alpha Z_1 Z_2 \beta$ такое, что $Z_1 \Rightarrow^+ \gamma_1 X$ и $Z_2 \Rightarrow^* Y\gamma_2$.

Доказательство. Фрагменты деревьев вывода, иллюстрирующие все три пункта теоремы, схематично изображены на рис. 3.4.

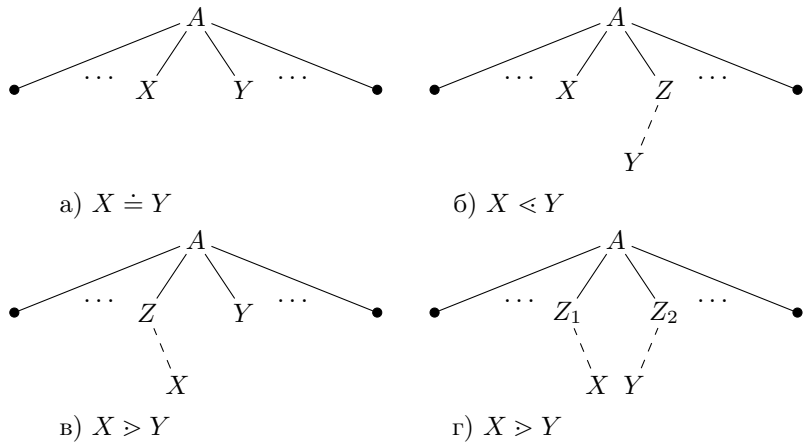


Рис.3.4. *Отношения предшествования и фрагменты деревьев вывода*

Докажем утверждение 1. Если $X \doteq Y$, то по определению отношения $\doteq$ основа некоторой r-формы имеет вид $\alpha XY\beta$ для подходящих цепочек α и β, откуда по определению основы найдется правило вида $A \to \alpha XY\beta$. Обратно, пусть существует правило вывода $A \to \alpha XY\beta$. Так как исходная грамматика является приведенной, это правило участвует в выводе некоторой терминальной цепочки w из аксиомы (см. замечание 4.1). По соответствующему дереву вывода можно восстановить правый вывод

$$S \Rightarrow^* \xi Au \Rightarrow \gamma\xi XY\beta u \Rightarrow^* w,$$

где u – цепочка терминалов. По определению основой r-формы $\xi\alpha XY\beta u$ является цепочка $\alpha XY\beta$, откуда $X \doteq Y$.

Перейдем к утверждению 2. Предположим, что $X \lessdot Y$. Тогда существует r-форма вида $\mu XY\nu$, основа которой начинается с символа Y (рис. 3.3). Рассмотрим дерево T, представляющее вывод этой r-формы. Пусть x и y – листья этого дерева с метками X и Y соответственно. Лист y входит в самый левый куст T (замечание 3.1). Лист x, расположенный левее y, не входит в тот же самый куст (листья куста помечены символами основы), а значит и ни в какой другой.

Тогда, во-первых, x и y не братья, а во-вторых, среди братьев x есть внутренний узел дерева T. Далее, все левые братья x – листья, так как в противном случае в T найдется куст, который расположен левее x, а следовательно левее основы. Итак, у x есть хотя бы один правый брат. Пусть z – ближайший правый брат x. Если z – лист, то $z = y$, что невозможно, так как y не брат x. Следовательно, узел z является внутренним и его метка – нетерминал, скажем Z. Тогда узлы x и z появились в дереве в результате применения правила вида $A \to \alpha X Z \beta$ в их родительском узле. Поскольку грамматика G является ε-свободной, при дальнейшем выводе формы $\mu X Y \nu$ из нетерминала Z была получена непустая цепочка. Значит, эта цепочка начинается символом Y. Необходимость доказана.

Предположим, что выполняется правая часть утверждения 2, т. е. существует правило вывода $A \to \alpha X Z \beta$ такое, что $Z \Rightarrow^+ Y\gamma$. Поскольку грамматика является приведенной, $\beta \Rightarrow^* v$ для некоторой терминальной цепочки v. Далее, возьмем правый вывод $Z \Rightarrow^+ Y\gamma$. Пусть $Y\gamma'$ – первая цепочка в этом выводе, начинающаяся символом Y. Значит, на последнем шаге правого вывода $Z \Rightarrow^+ Y\gamma'$ правило применялось к самому левому символу формы и, так как грамматика ε-свободна, правая часть этого правила начинается с Y. Аналогично доказательству утверждения 1, построим правый вывод

$$S \Rightarrow^* \xi A u \Rightarrow \xi \alpha X Z \beta u \Rightarrow^* \xi \alpha X Z v u \Rightarrow^+ \xi \alpha X Y \gamma' v u \Rightarrow^* w.$$

Согласно сделанному нами заключению о правом выводе $Z \Rightarrow^+ Y\gamma'$ основа r-формы $\xi \alpha X Y \gamma' v u$ начинается с Y, откуда $X \lessdot Y$.

Осталось доказать утверждение 3. Пусть $X \gtrdot Y$. Тогда Y – терминал по замечанию 3.3. Далее, существует r-форма вида $\mu X Y \nu$, основа которой заканчивается символом X (рис. 3.3). Аналогично утверждению 2, рассмотрим дерево T, представляющее вывод этой r-формы, и пусть x, y – листья T с метками X и Y соответственно. Через z обозначим узел, являющийся «ближайшим» общим предком x и y (т. е. узел z имеет сыновей $z_1 \lhd z_2$ таких, что x – потомок z_1, а y – потомок z_2). Пусть A, Z_1 и Z_2 – метки узлов z, z_1 и z_2 соответственно. Поскольку грамматика ε-свободна, z_2 – ближайший правый брат узла z_1, а следовательно, в узле z применялось правило вида $A \to \alpha Z_1 Z_2 \beta$. Самым правым листом в T, входящим в поддерево с корнем z_1, является узел x. Значит, это поддерево представляет вывод вида $Z_1 \Rightarrow^* \gamma_1 X$. Аналогично, поддерево с корнем z_2 представляет вывод вида $Z_2 \Rightarrow^* Y\gamma_2$. Так как x входит в самый левый

куст T (замечание 3.1), то все его братья являются листьями в T (и среди них нет y). Значит, отец x не является предком y, т. е. $z_1 \neq x$ и в записи $Z_1 \Rightarrow^* \gamma_1 X$ можно заменить «звездочку» на «плюс», завершая доказательство необходимости.

Докажем достаточность. Итак, Y – терминал и существует правило вывода вида $A \to \alpha Z_1 Z_2 \beta$ такое, что $Z_1 \Rightarrow^+ \gamma_1 X$ и $Z_2 \Rightarrow^* Y\gamma_2$. Рассуждая как в доказательствах утверждений 1 и 2, выводим, что $\beta \Rightarrow^* v$, $\gamma_2 \Rightarrow^* v'$ для некоторых терминальных цепочек v и v', а в правом выводе $Z_1 \Rightarrow^+ \gamma_1 X$ можно указать первую цепочку, которая заканчивается на X (обозначим ее через $\gamma_1' X$). Рассмотрим правый вывод

$$S \Rightarrow^* \xi A u \Rightarrow \xi\alpha Z_1 Z_2 \beta u \Rightarrow^* \xi\alpha Z_1 Z_2 v u \Rightarrow^* \xi\alpha Z_1 Y \gamma_2 v u \Rightarrow^* \\ \Rightarrow^* \xi\alpha Z_1 Y v' v u \Rightarrow^* \xi\alpha\gamma_1' X Y v' v u \Rightarrow^* w$$

Аналогично доказательству утверждения 2 получаем, что основа r-формы $\xi\alpha\gamma_1' X Y v' v u$ заканчивается на X, откуда $X \gtrdot Y$. Теорема полностью доказана. □

Используя теорему (и ее графическое представление на рис. 3.4), легко построить алгоритм вычисления отношений предшествования. Для нетерминала X определим множество $\textsc{first}'(X)$ как множество символов, с которых начинаются цепочки, выводимые из X. В отличие от множества $\textsc{first}(X)$, в $\textsc{first}'(X)$ могут входить нетерминалы, а пустую цепочку учитывать не нужно, так как используется ε-свободная грамматика. Ввиду этого свойства грамматики, алгоритм построения множества $\textsc{first}'(X)$ значительно проще алгоритма 2.3 построения множества $\textsc{first}(X)$, и мы оставляем его реализацию читателю. Через $\textsc{last}'(X)$ обозначим двойственное множество, т. е. множество последних символов в выводимых из X цепочках.

Отношение $\doteq$ вычисляется непосредственно по п. 1 теоремы (или рис. 3.4, *а*). Далее, $X \lessdot Y$, если и только если найдется нетерминал Z такой, что $X \doteq Z$ и $Y \in \textsc{first}'(Z)$ (п. 2 теоремы или рис. 3.4, *б*). Наконец, $X \gtrdot Y$, если и только если найдутся символы Z_1 и Z_2 такие, что $Z_1 \doteq Z_2$, $X \in \textsc{last}'(Z_1)$ и либо $Z_2 = Y$, либо $Y \in \textsc{first}'(Z_2)$ (п. 3 теоремы, рис. 3.4, *в*, *г*).

Пример 3.2. Вычислим отношения предшествования для грамматики $G_{3,1}$ (см. с. 130). Рассмотрение пар соседних символов в правилах вывода дает $\mathtt{a} \doteq F$, $F \doteq S$, $S \doteq \mathtt{b}$, $F \doteq \mathtt{b}$.

Для вычисления отношения $\lessdot$ заметим, что $\textsc{first}'(X) = \{F, \mathtt{b}\}$ и $\textsc{first}'(S) = \{\mathtt{a}, \mathtt{c}\}$. Тогда пара $\mathtt{a} \doteq F$ дает $\mathtt{a} \lessdot F$ и $\mathtt{a} \lessdot \mathtt{b}$, а пара $F \doteq S$

дает $F \lessdot \mathtt{a}$ и $F \lessdot \mathtt{c}$. Других пар с нетерминалом справа отношение $\doteq$ не содержит.

Чтобы вычислить отношение $\gtrdot$, дополнительно заметим, что $\textsc{last}'(F) = \{\mathtt{b}\}$ и $\textsc{last}'(S) = \{\mathtt{b}, \mathtt{c}\}$. Пара $F \doteq S$ дает $\mathtt{b} \gtrdot \mathtt{a}$ и $\mathtt{b} \gtrdot \mathtt{c}$. Из пары $S \doteq \mathtt{b}$ получаем $\mathtt{b} \gtrdot \mathtt{b}$ и $\mathtt{c} \gtrdot \mathtt{b}$. Наконец, из пары $F \doteq \mathtt{b}$ получается только уже известное нам отношение $\mathtt{b} \gtrdot \mathtt{b}$. Все результаты собраны в табл. 3.2.

Таблица 3.2. *Отношения предшествования для грамматики* $G_{3,1}$

	S	F	a	b	c
S				$\doteq$	
F	$\doteq$		$\lessdot$	$\doteq$	$\lessdot$
a		$\doteq$,$\lessdot$		$\lessdot$	
b			$\gtrdot$	$\gtrdot$	$\gtrdot$
c				$\gtrdot$	

Мы видим, что между некоторыми парами символов не выполнено ни одного отношения предшествования, между другими – в точности одно отношение, а между символами a и F – два отношения. Нетрудно привести пример грамматики такой, что между некоторыми ее символами выполняются все три отношения предшествования.

Предложение 3.2. *Пусть γ – r-форма и $\gamma = \alpha X Y \beta$ для некоторых символов X, Y и цепочек α, β. Тогда между X и Y выполнено хотя бы одно из отношений $\doteq, \lessdot, \gtrdot$.*

Доказательство. Вывод цепочки γ из аксиомы представлен некоторым деревом T, в котором два соседних листа (скажем, x и y) имеют метки X и Y соответственно. Пусть узел z – ближайший общий предок x и y (ср. доказательство утверждения 3 теоремы 3.1), A – метка этого узла. Если и x, и y – сыновья z, то получаем ситуацию, изображенную на рис. 3.4, *а*. Если только один из этих узлов является сыном z, то получаем ситуацию с рис. 3.4, *б* или *в*, а если ни один из них – ситуацию с рис. 3.4, *г*. Во всех случаях, согласно теореме 3.1, между X и Y выполняется одно из отношений. □

§ 12. Грамматики простого предшествования

В этом параграфе будет рассмотрен класс грамматик, в которых для нахождения основ r-форм достаточно только отношений просто-

го предшествования. Говорят, что грамматика *обратима*, если любые два различных правила вывода имеют различные правые части.

Определение. КС-грамматика G называется *грамматикой простого предшествования* (*ПП-грамматикой*), если она ациклична, обратима и между любыми ее символами выполнено не более одного отношения предшествования.

Как видно из табл. 3.2, грамматика $G_{3,1}$ не является ПП-грамматикой. В примере 3.3 приведены отношения простого предшествования для грамматики $G_{3,2} = \{S \to \mathtt{a}SS\mathtt{b} \mid \mathtt{c}\}$ (табл. 3.3). Таблица демонстрирует, что $G_{3,2}$ – ПП-грамматика.

В случае нисходящих анализаторов, как мы видели в предыдущей главе, анализируемую цепочку удобно заканчивать специальным символом – маркером конца цепочки. Для восходящего анализа на основе отношений предшествования будем использовать два маркера: $\vdash$ в начале и $\dashv$ в конце цепочки. Будем считать, что $\vdash \lessdot Y$ для любого символа Y, с которого может начинаться r-форма, и $X \gtrdot \dashv$ для любого символа X, которым r-форма может заканчиваться. Отметим, что X и Y могут совпадать с S.

Предложение 3.3. *Пусть G – ПП-грамматика, γ – ее r-форма и $\vdash\gamma\dashv = X_0X_1\ldots X_nX_{n+1}$. Тогда основой формы γ является цепочка $X_k\ldots X_l$ такая, что l – минимальный номер, для которого выполнено $X_l \gtrdot X_{l+1}$, $k \le l$ и*

$$X_{k-1} \lessdot X_k, X_k \doteq X_{k+1}, \ldots, X_{l-1} \doteq X_l, X_l \gtrdot X_{l+1}. \qquad (3.3)$$

Доказательство. Поскольку существует правый вывод цепочки γ из аксиомы (представленный некоторым деревом T), имеем $X_0 \lessdot X_1$ и $X_n \gtrdot X_{n+1}$, т. е. указанная в формулировке пара индексов (k, l) существует. Дерево T имеет листья $x_1 \lhd \ldots \lhd x_n$, помеченные символами $X_1\ldots X_n$ соответственно.

Пусть основа цепочки γ имеет вид $X_s\ldots X_t$. Тогда $X_t \gtrdot X_{t+1}$ по определению, откуда $t \ge l$. Пусть $t > l$. Символы X_l и X_{l+1} не могут принадлежать основе одновременно (иначе по определению $X_l \doteq X_{l+1}$, что противоречит определению ПП-грамматики), следовательно, $s > l$. В дереве T самый левый куст имеет листья $x_s, \ldots, x_t$, т. е. лист с меткой x_l не принадлежит никакому кусту. Этот лист не имеет правых братьев (рис. 3.4, *в*, *г*), значит, среди его левых братьев есть внутренний узел дерева. Но тогда в дереве есть куст левее x_l, т. е. «левее самого левого куста». Данное противоречие показывает,

что $t = l$. По определению $X_{s-1} \lessdot X_s$, а между любыми соседними символами фрагмента $X_s \dots X_t$ стоит знак $\doteq$. Отсюда однозначно следует $s = k$. □

Из данного предложения легко получается алгоритм анализа для ПП-грамматик. Мы будем писать $X \circ Y$, если X и Y не связаны ни одним из отношений $\doteq, \lessdot, \gtrdot$.

Алгоритм 3.1. *Восходящий анализ для ПП-грамматик.*
Вход: ПП-грамматика $G = (\Sigma, \Gamma, P, S)$, цепочка $w \in \Sigma^+$.
Выход: «ДА», если $w \in L(G)$, и «НЕТ» в противном случае.

```
1. γ ← ⊢w⊣  %% далее считаем, что γ = X_0 ... X_{n+1}
2. пока (γ ≠ ⊢S⊣), повторять
3.         i ← 0; k ← 0; l ← 0
4.         пока (i ≤ n и l = 0), повторять
5.                 если (X_i ∘ X_{i+1})
6.                         ответить «НЕТ»; выход
7.                 иначе если (X_i ⋖ X_{i+1})
8.                                 k ← i+1
9.                         иначе если (X_i ⋗ X_{i+1})
10.                                        l ← i
11.                 i ← i+1
12.        если (l = 0)
13.                 ответить «НЕТ»; выход
14.        иначе если (∄A : (A → X_k ... X_l) ∈ P)
15.                         ответить «НЕТ»; выход
16.        иначе γ ← X_0 ... X_{k-1} A X_{l+1} ... X_{n+1}; перейти на 2
17. ответить «ДА»; выход
```

Предложение 3.4. *Алгоритм 3.1 работает корректно.*

Доказательство. Мы докажем, что алгоритм 3.1 останавливается для любой цепочки $w \in \Sigma^+$, и если он отвечает «ДА», то $w \in L(G)$, а если «НЕТ», то $w \notin L(G)$.

Итерация внешнего цикла алгоритма заканчивается либо ответом «НЕТ», либо сверткой. При свертке длина цепочки γ не увеличивается, так как грамматика ε-свободна. Далее, ввиду ацикличности грамматики, т. е. невозможности выводов вида $A \Rightarrow^+ A$, длина цепочки γ может оставаться неизменной в течение не более чем $|\Gamma|$ сверток. Значит, через конечное число сверток γ приобретет вид $\vdash A \dashv$. Если

$A = S$, то алгоритм закончит работу немедленно, а если $A \neq S$, то – на следующем шаге.

Алгоритм отвечает «ДА», если цепочка w свернута в аксиому грамматики. Алгоритм отвечает «НЕТ» в трех случаях, в каждом из которых цепочка γ не является r-формой. В самом деле, если ответ «НЕТ» получен в строке 6, то этот факт следует из предложения 3.2. Если же ответ получен в строке 13 или 15, то в цепочке γ отсутствует основа по предложению 3.3. Поскольку все свертки определяются однозначно (это следует из обратимости грамматики), то из того, что γ не r-форма, следует, что цепочка w невыводима. □

Пример 3.3. Рассмотрим работу алгоритма для грамматики $G_{3,2} = \{S \to \mathtt{a}S\mathtt{b} \mid \mathtt{c}\}$ и цепочки $\gamma = \mathtt{acaccbb}$. Таблица отношений простого предшествования этой грамматики (с учетом маркеров) приведена в табл. 3.3.

Таблица 3.3. *Отношения предшествования для грамматики* $G_{3,2}$

	S	a	b	c	⊣
S	$\doteq$	$\lessdot$	$\doteq$	$\lessdot$	$\gtrdot$
a	$\doteq$	$\lessdot$		$\lessdot$	
b		$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
c		$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
⊢	$\lessdot$	$\lessdot$		$\lessdot$	

Укажем отношения предшествования и свертки при обработке цепочки $\gamma = \mathtt{acaccbb}$.

$$
\begin{array}{l}
\vdash \lessdot \mathtt{a} \lessdot \mathtt{c} \gtrdot \mathtt{a} \lessdot \mathtt{c} \gtrdot \mathtt{c} \gtrdot \mathtt{b} \gtrdot \mathtt{b} \gtrdot \dashv \\
\vdash \lessdot \mathtt{a} \doteq S \lessdot \mathtt{a} \lessdot \mathtt{c} \gtrdot \mathtt{c} \gtrdot \mathtt{b} \gtrdot \mathtt{b} \gtrdot \dashv \\
\vdash \lessdot \mathtt{a} \doteq S \lessdot \mathtt{a} \doteq S \lessdot \mathtt{c} \gtrdot \mathtt{b} \gtrdot \mathtt{b} \gtrdot \dashv \\
\vdash \lessdot \mathtt{a} \doteq S \lessdot \mathtt{a} \doteq S \doteq S \doteq \mathtt{b} \gtrdot \mathtt{b} \gtrdot \dashv \\
\vdash \lessdot \mathtt{a} \doteq S \doteq S \qquad\qquad\qquad \doteq \mathtt{b} \gtrdot \dashv \\
\vdash \quad S \qquad\qquad\qquad\qquad\qquad\qquad\quad \dashv
\end{array}
$$

§ 13. Грамматики слабого предшествования

Большинство грамматик, возникающих при описании языков программирования и их фрагментов, не являются ПП-грамматиками. В частности, ни одна из грамматик, введенных в ч. 1, §15, не является ПП-грамматикой. Попытки найти для естественно возникающих

языков порождающие их ПП-грамматики приводят к довольно громоздким конструкциям. Поэтому возникает задача: расширить класс ПП-грамматик так, чтобы новый класс грамматик допускал алгоритм анализа, близкий к рассмотренному в предыдущем параграфе.

Одна из идей расширения состоит в том, чтобы снять ограничение, что отношения $\lessdot$ и $\doteq$ не имеют общих элементов. Отношение $\gtrdot$ по-прежнему будет использоваться для нахождения правого конца основы. Тогда для определения левого конца основы надо найти правило вывода, правая часть которого расположена левее найденного правого конца. Трудность возникает в том случае, когда анализируемая цепочка имеет вид $\alpha\beta'\beta w$, правый конец основы расположен между β и w и имеется два правила вывода: $A \to \beta'\beta$ и $B \to \beta$. В этой ситуации неясно, какое из них применять для свертки. Мы будем применять *наиболее длинное из возможных правил*. Класс грамматик, для которых такое решение всегда является корректным, и будет искомым расширением класса ПП-грамматик. Мы будем использовать обозначение $X \leq Y$, если $X \lessdot Y$ или $X \doteq Y$.

Определение. КС-грамматика G называется *грамматикой слабого предшествования* (*СП-грамматикой*), если она ациклична, обратима и выполняются следующие условия:

1) не существует символов X и Y таких, что $X \leq Y$ и $X \gtrdot Y$;

2) не существует правил $A \to \alpha X \beta$ и $B \to \beta$ таких, что $X \leq B$.

Предложение 3.5. *Любая ПП-грамматика является СП-грамматикой.*

Доказательство. Достаточно проверить последнее условие из определения СП-грамматики. Пусть ПП-грамматика G имеет правила вывода $A \to \alpha X \beta$ и $B \to \beta$. Так как G ε-свободна, $\beta \neq \varepsilon$, поэтому $\beta = Y\gamma$ для некоторого символа Y. Тогда $X \doteq Y$, поскольку правило вывода $A \to \alpha X \beta$ можно записать как $A \to \alpha X Y \gamma$.

Предположим, что $X \doteq B$. Тогда существует правило вывода $C \to \mu X B \nu$. Поскольку $B \Rightarrow^+ Y\gamma$, получаем $X \lessdot Y$. Это противоречит определению ПП-грамматики, так как $X \doteq Y$.

Пусть $X \lessdot B$. Тогда найдется правило вывода $C \to \mu X Z \nu$ такое, что $Z \Rightarrow^+ B\xi$. Но $B \Rightarrow^+ Y\gamma$, поэтому $Z \Rightarrow^+ Y\gamma\xi$. Снова получаем $X \lessdot Y$ и противоречие с определением ПП-грамматики. □

Итак, класс ПП-грамматик содержится в классе СП-грамматик. Это включение строгое, например, грамматика $G_{3,1}$ (табл. 3.2) явля-

ется СП-грамматикой, но не ПП-грамматикой. Отметим, что эффективно проверить грамматику на принадлежность к СП-грамматикам можно прямо по определению.

Пример 3.4. Покажем, что грамматика арифметических выражений

$$GA_2 = \{E \to E + T \,|\, T,\ T \to T * F \,|\, F,\ F \to (E) \,|\, \mathtt{x}\}$$

является СП-грамматикой. Отношения простого предшествования для нее собраны в табл. 3.4. Выполнение условия 1 следует из этой таблицы, обратимость и ацикличность проверяются непосредственно. Существуют две пары правил, которые удовлетворяют посылке условия 2, это пары $E \to E + T$ и $E \to T$, $T \to T * F$ и $T \to F$. Но поскольку не верно ни $+ \trianglelefteq E$, ни $* \trianglelefteq T$, то условие 2 также выполняется. Следовательно, GA_2 – СП-грамматика.

Таблица 3.4. *Отношения предшествования для грамматики* GA_2

	E	T	F	(	)	x	+	*	⊣
E					$\doteq$		$\doteq$		$\gtrdot$
T					$\gtrdot$		$\gtrdot$	$\doteq$	$\gtrdot$
F					$\gtrdot$		$\gtrdot$	$\gtrdot$	$\gtrdot$
(	$\doteq, \lessdot$	$\lessdot$	$\lessdot$	$\lessdot$		$\lessdot$	$\lessdot$		
)					$\gtrdot$				$\gtrdot$
x					$\gtrdot$		$\gtrdot$	$\gtrdot$	$\gtrdot$
+		$\doteq, \lessdot$	$\lessdot$	$\lessdot$		$\lessdot$			
*			$\doteq, \lessdot$	$\lessdot$		$\lessdot$			
⊢	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$		$\lessdot$			

Следующее предложение показывает, что при восстановлении правого вывода в СП-грамматиках свертку на каждом шаге действительно нужно производить по самому длинному правилу вывода.

Предложение 3.6. *Пусть* $S \Rightarrow^+ \mu C w \Rightarrow \nu X \beta w$ *– фрагмент правого вывода в СП-грамматике* G *(здесь* S *– аксиома,* X *– символ,* C *– нетерминал,* w *– цепочка терминалов,* μ*,* ν *и* β *– цепочки). Тогда правило вида* $B \to \beta$ *применялось последним в данном выводе в том и только в том случае, когда в* G *нет правила вида* $A \to \alpha X \beta$*.*

Доказательство. Пусть последним в указанном выводе применялось правило $B \to \beta$. Тогда цепочка $\nu X B w$ является r-формой и

между символами X и B по предложению 3.2 выполнено хотя бы одно из отношений предшествования. Отношение $X \gtrdot B$ выполняться не может, так как B – нетерминал (замечание 3.3). Значит, $X \leq B$, и правил вида $A \to \alpha X \beta$ не существует по определению СП-грамматики.

Предположим теперь, что в G нет правила вида $A \to \alpha X \beta$. Правый конец основы r-формы $\nu X \beta w$ находится непосредственно слева от цепочки w (иначе предыдущая цепочка в правом выводе не могла бы заканчиваться на Cw). Если эта основа является собственным суффиксом β, то, рассуждая как в предыдущем абзаце, мы получим противоречие с тем, что в грамматике есть правило $B \to \beta$. Эта основа не может иметь суффикс $X\beta$, так как в грамматике нет правил вида $A \to \alpha X \beta$. Следовательно, эта основа равна β. Предложение доказано. □

Теперь приведем алгоритм анализа для СП-грамматик. Единственное существенное его отличие от алгоритма 3.1 состоит в способе поиска основы. Поэтому мы не будем выписывать новый алгоритм явно, а ограничимся обсуждением этого отличия.

В СП-грамматике между двумя соседними символами r-формы выполнено отношение $\doteq$, $\lessdot$, или $\gtrdot$ (возможно, первые два одновременно). При этом правый конец основы находится однозначно (он помечен самым «левым» отношением $\gtrdot$ в цепочке), а о левом конце основы известно, что он помечен отношением $\lessdot$ так, что между соседними символами внутри основы выполняется $\doteq$. Таким образом, символ X_l – правый конец основы – определяется однозначно, а на роль левого ее конца (символа X_k) может быть несколько кандидатов $X_{k_1}, \ldots, X_{k_s}$ $(k_1 < \ldots < k_s)$, при этом отношения предшествования на интересующем нас фрагменте цепочки устроены следующим образом:

$$\ldots \lessdot X_{k_1} \doteq \ldots \doteq X_{k_2-1} \mathrel{\substack{\lessdot \\ \doteq}} X_{k_2} \doteq \ldots \doteq X_{k_s-1} \mathrel{\substack{\lessdot \\ \doteq}} X_{k_s} \doteq \ldots \doteq X_l \gtrdot \ldots$$

Цикл в строках 4–11 алгоритма 3.1 несложно перестроить так, чтобы он составлял список всех возможных левых концов основы. Мы оставляем это читателю. Согласно предложению 3.6, кандидатов в основы нужно перебирать по убыванию длины. А именно, если есть правило с правой частью $X_{k_1} \ldots X_l$, то основа найдена, если такого правила нет – проверяем цепочку $X_{k_2} \ldots X_l$, и т. д., пока не найдем подходящее правило или не переберем все s возможных левых кон-

цов основы. Если правило найдено – по нему производим свертку, если не найдено – алгоритм заканчивает работу с ответом «НЕТ».

Пример 3.5. Укажем отношения предшествования и свертки при обработке цепочки $\gamma = \mathtt{x} + \mathtt{x} * \mathtt{x}$ в соответствии с табл. 3.4.

$$
\begin{array}{ll}
\vdash & \lessdot\ \mathtt{x}\ \gtrdot + \lessdot\ \mathtt{x}\ \gtrdot * \lessdot\ \mathtt{x}\ \gtrdot \dashv \\
\vdash & \lessdot\ F\ \gtrdot + \lessdot\ \mathtt{x}\ \gtrdot * \lessdot\ \mathtt{x}\ \gtrdot \dashv \\
\vdash & \lessdot\ T\ \gtrdot + \lessdot\ \mathtt{x}\ \gtrdot * \lessdot\ \mathtt{x}\ \gtrdot \dashv \\
\vdash & \lessdot\ E\ \doteq + \lessdot\ \mathtt{x}\ \gtrdot * \lessdot\ \mathtt{x}\ \gtrdot \dashv \\
\vdash & \lessdot\ E\ \doteq + \lessdot\ F\ \gtrdot * \lessdot\ \mathtt{x}\ \gtrdot \dashv \\
\vdash & \lessdot\ E\ \doteq + \mathrel{\dot{\leqslant}} T\ \doteq * \lessdot\ \mathtt{x}\ \gtrdot \dashv \\
\vdash & \lessdot\ E\ \doteq + \mathrel{\dot{\leqslant}} T\ \doteq * \mathrel{\dot{\leqslant}} F\ \gtrdot \dashv \\
\vdash & \lessdot\ E\ \doteq + \mathrel{\dot{\leqslant}} T\ \gtrdot \dashv \\
\vdash & S\ \dashv
\end{array}
$$

§ 14. Отношения операторного предшествования

В данном параграфе рассматриваются *отношения операторного предшествования*, называемые также *отношениями приоритета (операторов)*. Метод разбора выражений, основанный на этих отношениях, является одним из первых методов синтаксического анализа. Он возник еще до явного использования КС-грамматик, использовался, например, в языке FORTRAN[†] и находит применение до сих пор. В частности, с его помощью удобно анализировать формулы в табличных редакторах типа Excel.

Опишем идею метода. *Выражение* состоит из операндов, операторов и скобок. Семантическое значение выражения определяется значениями «атомарных» операндов и порядком выполнения операций, который естественно описывать при помощи дерева (ср. пример 3.1 из ч. 1, § 14). Порядок выполнения, там где он не указан явно при помощи скобок, основан на соглашениях о приоритете операторов и их *ассоциативности*, т. е. порядке выполнения одноименных операторов. К примеру, приоритет возведения в степень ($\uparrow$) выше приоритета умножения ($*$), который, в свою очередь, выше приоритета сложения ($+$). Оператор $\uparrow$ *правоассоциативен*, а $*$ и $+$ – *левоассоциативны*, т. е. $\mathtt{x} \uparrow \mathtt{y} \uparrow \mathtt{z} = \mathtt{x} \uparrow (\mathtt{y} \uparrow \mathtt{z})$ и $\mathtt{x} * \mathtt{y} * \mathtt{z} = (\mathtt{x} * \mathtt{y}) * \mathtt{z}$. Если приоритеты и ассоциативность известны, можно «расставить порядок действий» и в соответствии с ним построить дерево снизу

[†]FORTRAN = FORmula TRANslator.

вверх. Фактически, выполнение оператора – это свертка: оператор вместе со своими операндами группируется в единое целое, являющееся операндом для применения одного из следующих операторов. А если относиться к выполнению оператора как к свертке, то нужно указать правила выбора основы. Для этого на основании приоритетов и ассоциативности вводятся отношения приоритета $\doteq$, $\lessdot$ и $\gtrdot$ и используется тот же принцип, что и для отношений простого предшествования: основа цепочки слева ограничена отношением $\lessdot$, справа – самым левым в цепочке отношением $\gtrdot$, а внутри нее допускается только $\doteq$. Основное отличие отношений приоритета в том, что они определяются *только для терминалов* и нахождение основы имеет свою специфику.

Из сказанного выше видно, что *язык выражений можно задать (неоднозначной!) грамматикой выражений с единственным нетерминалом E («операнд»), а для определения порядка сверток использовать отношения приоритета, построенные на основе знания приоритета и ассоциативности операторов.* Таким образом можно избежать построения сложной грамматики, правильно отражающей приоритеты и ассоциативность, а также значительно сократить время анализа выражения (не тратится время на многочисленные и семантически бессмысленные свертки по цепным правилам, таким как правила $E \to T$ и $T \to F$ в грамматике LA_2).

Заметим, что в синтаксически правильном выражении два операнда не могут стоять рядом – они обязательно разделены либо знаком оператора, либо разделителем (в том случае, когда оператор, применяемый к этим операндам, записан в виде функции). Следовательно, любая форма грамматики выражений не содержит двух рядом стоящих нетерминалов. Два терминала такой формы назовем *соседними*, если между ними нет других терминалов. Итак, между соседними терминалами в форме находится либо одиночный нетерминал, либо не находится ничего.

Замечание 3.4. Если форма грамматики выражений содержит цепочку $\mathtt{a}E\mathtt{b}$ и ровно один из терминалов $\mathtt{a}, \mathtt{b}$ принадлежит основе этой формы, то E также принадлежит основе (иначе форма, получающаяся после свертки, будет иметь два соседних нетерминала).

Сделанное замечание показывает, что для определения основы формы достаточно рассматривать только терминалы в ней. Для пар соседних терминалов определим отношения приоритета следующим образом:

- $\mathtt{a} \doteq \mathtt{b}$, если a и b должны быть свернуты на одном шаге;
- $\mathtt{a} \lessdot \mathtt{b}$, если b должен быть свернут раньше a;
- $\mathtt{a} \gtrdot \mathtt{b}$, если a должен быть свернут раньше b;
- $\vdash \lessdot \mathtt{b}$, если b может быть первым терминалом выражения;
- $\mathtt{a} \lessdot \dashv$, если a может быть последним терминалом выражения.

Если a и b – операторы, то первым из них должен быть свернут оператор более высокого приоритета, а в случае равенства приоритетов (в частности, когда $\mathtt{a} = \mathtt{b}$) раньше сворачивается левый оператор при левоассоциативности и правый – при правоассоциативности.

Расставить отношения приоритета для прочих пар терминалов помогают следующие замечания. Во-первых, «атомарные» операнды сворачиваются в первую очередь. Далее, содержимое скобок сворачивается раньше того, что за скобками; при вложенных скобках вначале сворачивается содержимое внутренних скобок. Парные скобки сворачиваются на одном шаге, так же как и функциональные конструкции типа $\min(E;E)$ из приводимого ниже примера 3.6.

Пример 3.6. Расширим язык арифметических выражений LA за счет двух дополнительных операций: унарного минуса и операции взятия минимума из двух чисел. Полученный язык задается неоднозначной грамматикой

$$G_{3,3} = \{E \to E{*}E \mid E{+}E \mid {-}E \mid \min(E;E) \mid (E) \mid \mathtt{x}\}\ ^{\dagger}$$

(ср. с грамматикой GA_1, ч. 1, § 14). Выстроив операции по убыванию приоритета, получаем: min, $-$, $*$, $+$. Унарный минус правоассоциативен, умножение и сложение, как уже отмечалось, левоассоциативны, а ассоциативность взятия минимума не имеет значения, так как два символа этой операции не могут оказаться рядом в корректном выражении: их разделяет по крайней мере один терминал – левая скобка. Применив изложенные выше правила, получаем таблицу отношений приоритета (табл. 3.5).

Пустые клетки таблицы означают, что данные нетерминалы не могут соседствовать в r-форме. Например, последним нетерминалом не может быть левая скобка, так как после нее обязательно должна присутствовать правая; после правой скобки не может идти x, так как и фрагмент, заканчивающийся правой скобкой, и x сворачиваются в нетерминал; и т. д. Проанализируем выражение $\mathtt{x}*-\min(\mathtt{x},\mathtt{x})$ при помощи табл. 3.5. Для удобства записи отношений приоритета между терминалами полученные в ходе сверток символы E будем

[†] min – это токен, а не последовательность букв m, i и n!

Таблица 3.5. *Отношения приоритета операторов для грамматики* $G_{3,3}$

	$+$	$*$	$-$	min	x	(	)	;	$\dashv$
$+$	$\gtrdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$*$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$-$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
min						$\doteq$			
x	$\gtrdot$	$\gtrdot$	$\gtrdot$				$\gtrdot$	$\gtrdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\doteq$	
)	$\gtrdot$	$\gtrdot$	$\gtrdot$				$\gtrdot$	$\gtrdot$	$\gtrdot$
;	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$		
$\vdash$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$			

писать не в строке, а в качестве индекса предшествующего терминала. При выборе основы мы не должны забывать про замечание 3.4.

$$
\begin{array}{lllllllllllll}
\vdash & \lessdot \mathtt{x} \gtrdot & * & \lessdot - & \lessdot \min & \doteq (& \lessdot \mathtt{x} \gtrdot & ; & \lessdot \mathtt{x} \gtrdot &) & \gtrdot & \dashv \\
\vdash_E & & \lessdot * & \lessdot - & \lessdot \min & \doteq (& \lessdot \mathtt{x} \gtrdot & ; & \lessdot \mathtt{x} \gtrdot &) & \gtrdot & \dashv \\
\vdash_E & & \lessdot * & \lessdot - & \lessdot \min & \doteq (_E & \doteq & ; & \lessdot \mathtt{x} \gtrdot &) & \gtrdot & \dashv \\
\vdash_E & & \lessdot * & \lessdot - & \lessdot \min & \doteq (_E & \doteq & ;_E & \doteq &) & \gtrdot & \dashv \\
\vdash_E & & \lessdot * & \lessdot -_E & & & & & & & \gtrdot & \dashv \\
\vdash_E & & \lessdot *_E & & & & & & & & \gtrdot & \dashv \\
\vdash_E & & & & & & & & & & & \dashv
\end{array}
$$

Немного изменим исходное выражение: $\mathtt{x} - \min(\mathtt{x}, \mathtt{x})$. Все шаги будут такими же, как приведенные выше, до получения строки

$$\vdash_E \lessdot -_E \gtrdot \dashv,$$

в которой согласно замечанию 3.4 будет найдена основа $E{-}E$. Поскольку $E{-}E$ не является правой частью правила вывода, обнаружена синтаксическая ошибка.

Можно подойти к построению отношения приоритетов и с другой стороны. Для этого нужно взять грамматику G, правые части правил которой не содержат подряд идущих нетерминалов (это так называемые *операторные* грамматики). Используя несложный алгоритм, похожий на алгоритм построения отношений простого предшествования, построить отношения приоритетов, затем отождествить все нетерминалы в G, получая неоднозначную грамматику выражений, – и анализатор готов. Однако вряд ли такой способ построения можно рекомендовать. Причин тому несколько.

1. Несмотря на то, что любая грамматика эквивалентна операторной грамматике (задача 9 к ч. 1, гл. 4), при таком преобразовании размер грамматики может вырасти очень сильно.
2. Если какие-либо два отношения приоритетов имеют непустое пересечение, то у некоторых форм найти основу будет невозможно. Таким образом, построенная алгоритмом таблица может оказаться непригодной для анализа.
3. Без отождествления нетерминалов не обойтись (при анализе на основе приоритета операторов нет возможности произвести свертку по цепному правилу типа правила $T \to F$ грамматики GA_2), а такое отождествление может расширить язык, порождаемый грамматикой (см. задачу 10).
4. Если же никакие из вышеприведенных проблем нам не помешали, то, по-видимому, грамматика G порождает некоторый язык выражений и данный анализатор было бы легче построить вручную, как описано в этом параграфе.

§ 15. Реализация методов анализа в МП-автомате

Как уже отмечалось, анализатор перенос–свертка работает следующим образом. Символы входной цепочки по одному переносятся в стек, пока наверху его не окажется основа, основа заменяется в стеке на соответствующий нетерминал, и вся процедура повторяется. Анализ заканчивается успешно, если просмотрена вся входная цепочка и стек содержит только аксиому грамматики. Синтаксическая ошибка обнаруживается одним из двух способов: либо отсутствие отношения между соседними символами в цепочке, либо отсутствие в грамматике правила, правая часть которого равна найденной основе. Мы не будем в этой главе отдельно обсуждать обработку синтаксических ошибок, отметим только, что пустые клетки таблицы отношений предшествования можно использовать для указания процедур коррекции ошибок (типа вставки/удаления символа).

Такой анализатор, конечно, можно представлять себе как МП-автомат, но обычно явное выписывание управляющих таблиц излишне. Тем не менее опишем для примера переход от алгоритма анализа для ПП-грамматик к МП-автомату.

Результатом перехода будет ДМПА с несколькими состояниями, одно из которых предназначено для реализации переноса, а все остальные – для сверток. Стековый алфавит состоит в точности из всех символов грамматики (левый конец цепочки соответствует дну

стека, а правый никогда не будет перенесен в стек).

В состоянии «перенос» (оно является начальным) автомат выполняет перенос, если между верхним символом X стека и текущим входным символом $\mathtt{a}$ выполнено одно из отношений $\doteq$, $<$. Если выполнено отношение $\gtrdot$, то символ X снимается со стека и происходит переход в состояние «снятие X». В этом состоянии верхний символ Y стека сравнивается с X. Если $Y \doteq X$, то автомат снимает Y со стека и переходит в состояние «снятие YX». Так продолжается до тех пор, пока не будет найден левый конец основы, т. е. в состоянии «снятие α» верхний символ Z стека находится в отношении $<$ с первым символом цепочки α. Когда левый конец основы найден, автомат завершает свертку: добавляет в стек поверх символа Z нетерминал A такой, что $A \to \alpha$ – правило грамматики, и возвращается в состояние «перенос». Единственная команда допуска используется в состоянии «снятие S», где S – аксиома, для пары $(\dashv, \nabla)$.

Как легко убедиться, в состоянии «снятие α» цепочка α – это непустой суффикс основы обрабатываемой r-формы. Поэтому для корректной работы автомата нужны состояния «снятие α» для всех суффиксов α правых частей правил грамматики (плюс состояние «снятие S» для допуска цепочки). Если верхний символ стека Z находится в отношении $\doteq$ с первым символом α, то происходит переход в состояние «снятие $Z\alpha$», если такое существует, и выдача сообщения об ошибке в противном случае.

В качестве упражнения предлагаем читателю самостоятельно построить таблицу переходов МП-автомата для восходящего анализа грамматики $G_{3,2}$ (пример 3.3).

Замечание 3.5. Если снабдить МП-автомат возможностью выталкивать на шаге любое количество верхних символов из стека (а не только один) и существенно расширить стековый алфавит, то можно сконструировать распознаватель с единственным состоянием. Однако соответствующие идеи скорее принадлежат следующей главе и в ней же будут подробно изложены.

Задачи

1. Вычислить отношения предшествования для грамматик

а) $G_1 = \{S \to S\mathtt{a}S\mathtt{b} \mid \mathtt{c}\}$;

б) $G_2 = \{S \to \mathtt{a}S\mathtt{bb} \mid \mathtt{abb}\}$;

в) $G_3 = \{S \to \mathtt{a}SA\mathtt{b} \mid \mathtt{a}S\mathtt{b},\ A \to \mathtt{b}\}$;

г) $G_4 = \{S \to \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \mid \texttt{a},\ E \to E\ \texttt{or b} \mid \texttt{b}\}$;
д) $G_5 = \{S \to SA \mid A,\ A \to (S) \mid ()\}$;
е) $G_6 = \{S \to AS \mid A,\ A \to (S) \mid ()\}$.

Какие из них являются ПП-грамматиками, СП-грамматиками?

2. Вычислить отношения предшествования для грамматики

$$G = \{S \to \texttt{b}M\texttt{b},\ M \to (L \mid \texttt{a},\ L \to M\texttt{a})\}.$$

Провести восходящий анализ цепочек bab, b(aa)b, b(a)b, b(((aa)a)a)b.

3. Привести примеры грамматик, показывающих, что для свойств рефлексивности, симметричности и транзитивности отношения $\doteq$ выполняются все восемь логически возможных вариантов.

4. Доказать, что любой регулярный язык без пустой цепочки порождается ПП-грамматикой.

5. Найти ПП-грамматику для языка
а) $L_1 = \{\texttt{a}^n\texttt{cb}^n \mid n \in \mathbb{N}\}$,
б) $L_2 = \{\texttt{a}^n\texttt{cb}^{2n} \mid n \in \mathbb{N}\}$,
в) $L_3 = \{\texttt{a}^n\texttt{cb}^n \mid n \in \mathbb{N}\} \cup \{\texttt{a}^n\texttt{db}^{2n} \mid n \in \mathbb{N}\}$.

6. Доказать, что язык $L = \{\texttt{ca}^n\texttt{b}^n \mid n \in \mathbb{N}\} \cup \{\texttt{da}^n\texttt{b}^{2n} \mid n \in \mathbb{N}\}$ не порождается ПП-грамматикой.

7. Для СП-грамматик из задачи 1 построить эквивалентные им ПП-грамматики.

8. Доказать, что грамматика

$$G = \{S \to \texttt{a} \mid [L],\ L \to L; S \mid S\},$$

порождающая собственное подмножество языка списков LL, является ПП-грамматикой.

9. Построить таблицу приоритета операторов для языка
а) регулярных выражений;
б) булевых выражений;
в) формул логики высказываний;
г) условных операторов и операторов цикла языка Паскаль.

10. Построить таблицу приоритета операторов по грамматике

$$G = \{S \to \texttt{a}A \mid \texttt{b}B,\ A \to \texttt{0}A\texttt{1} \mid \texttt{a},\ B \to \texttt{0}B\texttt{1} \mid \texttt{b}\}.$$

Отождествив нетерминалы, проиллюстрировать работу алгоритма анализа на примере цепочек a0a1, a01a, a0b1. Убедиться, что цепочка a0b1 невыводима в грамматике, но допускается алгоритмом.

Глава 4. LR-анализ

Одним из наиболее известных классов контекстно-свободных грамматик, допускающих эффективный восходящий анализ, является класс $LR(k)$-грамматик. Буква L в названии этого класса означает, что анализируемая цепочка просматривается слева направо, а буква R – о том, что в результате анализа восстанавливается правосторонний вывод. Символ k – количество символов анализируемой цепочки, которые просматриваются вперед после обнаружения предполагаемой правой границы основы для принятия решения о свертке. В настоящее время класс $LR(k)$-грамматик является основным классом грамматик, для которого производятся коммерческие компиляторы. Если другое не оговорено явно, мы считаем, что $k = 1$. Это наиболее важный случай: при бóльших значениях k никаких принципиально новых идей и приемов в LR-анализе не появляется.

В этой главе будем по умолчанию предполагать, что все грамматики являются *приведенными* и *однозначными*, а все выводы – *правосторонними*. По сравнению с предыдущей главой снимается запрет на использование аннулирующих правил вывода.

§ 16. Общая схема LR-анализа

LR-анализатор является анализатором типа перенос–свертка, т. е. он переносит символы входной цепочки по одному в стек и производит свертку, когда наверху стека образовалась основа текущей r-формы. Отличительной характеристикой LR-анализа является то, что в стек записываются не грамматические символы, а так называемые *состояния*. Состояние однозначно определяет символ, а также несет информацию о символах, лежащих ниже в стеке. Одному символу X может соответствовать несколько состояний, и мы будем использовать для таких состояний обозначения вида X^i. Таким образом, если $X_1^{i_1} \dots X_n^{i_n}$ – содержимое стека[†] LR-анализатора, а v – необработанная часть входной цепочки, то $X_1 \dots X_n v$ является r-формой.

Решение об очередном действии принимается анализатором, исходя из верхнего состояния в стеке (о нем часто говорят как о *текущем состоянии* анализатора) и k очередных символов входного

[†]В предыдущей главе мы договорились, что записываем содержимое стека «вершиной вправо».

потока. Требуемые действия указаны в *таблице анализа*, разделенной на две части – ACTION и GOTO. Строки таблицы проиндексированы состояниями, а столбцы – терминалами в части ACTION[†] и нетерминалами в части GOTO (см. ниже пример 4.1). В клетке таблицы ACTION указывается, перенос или свертку необходимо произвести на текущем шаге. Если производится перенос, то в клетке указано состояние, которое нужно добавить в стек, если свертка – то дается ссылка на правило, по которому ее нужно произвести. В этом случае анализатор извлекает правую часть правила из стека и обращается к таблице GOTO для определения состояния, которое нужно добавить в стек. Одна из клеток таблицы ACTION содержит команду «допуск». Пустые клетки таблиц указывают на ошибки. Разберем работу LR-анализатора на примере.

Пример 4.1. Возьмем грамматику арифметических выражений GA_2 и занумеруем ее правила:

$$(1)\ E \to E{+}T \quad (2)\ E \to T \quad (3)\ T \to T{*}F$$
$$(4)\ T \to F \quad (5)\ F \to (E) \quad (6)\ F \to \mathtt{x}$$

Таблица LR-анализа для этой грамматики приведена в табл. 4.1. Переносы помечены стрелкой ($\leftarrow$) с указанием нового состояния. Свертки помечены знаком $\otimes$ с указанием номера правила. Если символу соответствует единственное состояние, то оно обозначено так же, как этот символ.

Поясним работу анализатора на примере цепочки $\mathtt{x} + \mathtt{x} * (\mathtt{x} + \mathtt{x})$ (табл. 4.2). Вначале стек пуст (анализатор находится в состоянии, которое соответствует символу ∇). На первом шаге в стек заносится $\mathtt{x}$ (значение ACTION$[\nabla, \mathtt{x}]$). Затем производится свертка по правилу $F \to \mathtt{x}$ (значение ACTION$[\mathtt{x}, +]$). Чтобы произвести эту свертку, надо извлечь из стека один символ (правая часть правила имеет длину 1) и обратиться к значению GOTO от новой вершины стека и нетерминала в левой части правила. В данном случае имеем GOTO$[\nabla, F] = F^1$, и данное состояние записывается в стек. На следующем шаге производится свертка по правилу $T \to F$ и т. д. В качестве упражнения предлагаем читателю восстановить пропущенную часть разбора.

Построение LR-анализатора по грамматике – это конструирование таблиц ACTION и GOTO. Различают несколько вариантов LR-анализа в зависимости от метода построения таблиц. Однако все методы имеют много общего, и мы начнем с обсуждения этого общего.

[†]Если $k > 1$, то цепочками терминалов.

Таблица 4.1. *Таблица LR-анализа для грамматики* GA_2

	ACTION						GOTO		
	$+$	$*$	x	(	)	$\dashv$	E	T	F
E^1	$\leftarrow+$					✓			
T^1	$\otimes 2$	$\leftarrow*$			$\otimes 2$	$\otimes 2$			
F^1	$\otimes 4$	$\otimes 4$			$\otimes 4$	$\otimes 4$			
(			$\leftarrow$x	$\leftarrow($			E^2	T^1	F^1
x	$\otimes 6$	$\otimes 6$			$\otimes 6$	$\otimes 6$			
$+$			$\leftarrow$x	$\leftarrow($				T^2	F^1
$*$			$\leftarrow$x	$\leftarrow($					F^2
E^2	$\leftarrow+$				$\leftarrow)$				
T^2	$\otimes 1$	$\leftarrow*$			$\otimes 1$	$\otimes 1$			
F^2	$\otimes 3$	$\otimes 3$			$\otimes 3$	$\otimes 3$			
)	$\otimes 5$	$\otimes 5$			$\otimes 5$	$\otimes 5$			
∇			$\leftarrow$x	$\leftarrow($			E^1	T^1	F^1

Таблица 4.2. *Протокол разбора цепочки* x+x∗(x+x) *LR-анализатором*

Такт	Содержимое стека	Позиция указателя	Действие
1	∇	$\diamond$x+x∗(x+x)$\dashv$	Перенос
2	∇x	x$\diamond$+x∗(x+x)$\dashv$	Свертка по $F\rightarrow$x
3	∇F^1	x$\diamond$+x∗(x+x)$\dashv$	Свертка по $T\rightarrow F$
...			
15	∇E^1+T^2∗(E^2+	x+x∗(x+$\diamond$x)$\dashv$	Перенос
16	∇E^1+T^2∗(E^2+x	x+x∗(x+x$\diamond$)$\dashv$	Свертка по $F\rightarrow$x
17	∇E^1+T^2∗(E^2+F^1	x+x∗(x+x$\diamond$)$\dashv$	Свертка по $T\rightarrow F$
18	∇E^1+T^2∗(E^2+T^2	x+x∗(x+x$\diamond$)$\dashv$	Свертка по $E\rightarrow E+T$
19	∇E^1+T^2∗(E^2	x+x∗(x+x$\diamond$)$\dashv$	Перенос
20	∇E^1+T^2∗(E^2)	x+x∗(x+x)$\diamond\dashv$	Свертка по $F\rightarrow(E)$
21	∇E^1+T^2∗F^2	x+x∗(x+x)$\diamond\dashv$	Свертка по $T\rightarrow T*F$
22	∇E^1+T^2	x+x∗(x+x)$\diamond\dashv$	Свертка по $E\rightarrow E+T$
23	∇E^1	x+x∗(x+x)$\diamond\dashv$	Допуск

Во-первых, для построения анализатора удобно расширить исходную грамматику, добавив в нее внешнюю аксиому.

Определение. Пусть $G = (\Sigma, \Gamma, P, S)$ – контекстно-свободная грамматика. Ее *расширенной грамматикой* называется грамматика $G' = (\Sigma, \Gamma \cup \{S'\}, P \cup \{S' \to S\}, S')$, где S' – новый нетерминал.

Очевидно, что грамматики G и G' эквивалентны, а восходящий анализ любой корректной цепочки в G' заканчивается сверткой по правилу $S' \to S$. Таким образом, применение этой свертки сигнализирует анализатору о завершении разбора и допуске цепочки. Далее везде предполагается, что перед построением LR-анализатора производится расширение грамматики.

Во-вторых, ключевым шагом построения любого LR-анализатора является построение ДКА, распознающего язык всех цепочек, которые могут оказаться в стеке при разборе корректных терминальных цепочек. Этому шагу посвящен следующий параграф.

§ 17. Активные префиксы и LR(k)-пункты

Определение. Префикс r-формы грамматики, не выходящий за правый конец основы этой формы, называется *активным префиксом.*

Как видно из определения, при разборе корректной цепочки стек анализатора на любом шаге содержит активный префикс. Язык активных префиксов грамматики, как правило, бесконечен, но всегда является регулярным (мы докажем это, построив распознающий автомат). Состояния построенного автомата и будут состояниями анализатора.

Определение. *LR(k)-пунктом*[†] (грамматики G) называется четверка (A, β_1, β_2, v), где A – нетерминал, β_1, β_2 – цепочки такие, что $A \to \beta_1\beta_2$ – правило вывода, а v – цепочка терминалов, которая либо имеет длину ровно k, либо имеет длину, меньшую k и заканчивается символом $\dashv$. Для LR(k)-пункта принято обозначение $[A \to \beta_1 \cdot \beta_2, v]$, причем пустые цепочки не пишутся.

Смысл «правила с точкой» таков: если $A \to \beta_1\beta_2$ – правило, по которому нужно произвести одну из следующих сверток, то β_1 находится наверху стека, а β_2 совпадает с началом необработанной части входной цепочки (или это начало может быть свернуто к β_2). Таким образом, точка в пункте соответствует границе между стеком и остатком входной цепочки в текущей r-форме, т. е. месту, в

[†] В оригинале – LR(k)-item, этим объясняется выбор буквы I для обозначений.

котором находится в данный момент анализатор. Цепочка v служит для более точной «настройки» анализатора и используется не всегда. Это та цепочка, которая должна следовать во входном потоке сразу после β_2 для того, чтобы свертка по правилу $A \to \beta_1\beta_2$ была корректной.

Определение. LR(k)-пункт $[A \to \beta_1{\cdot}\beta_2, v]$ называется *допустимым* для активного префикса $\alpha\beta_1$, если существует (правый) вывод

$$S' \Rightarrow^* \alpha A w \Rightarrow \alpha\beta_1\beta_2 w \Rightarrow^* uw \tag{4.1}$$

и цепочка v является префиксом цепочки $w{\dashv}$.

Допустимость пункта $[A \to \beta_1{\cdot}\beta_2, v]$ для активного префикса $\alpha\beta_1$ определяет предполагаемые действия распознавателя в случае, когда содержимое стека равно $\alpha\beta_1$. Если при этом $\beta_2 \neq \varepsilon$, то выполняется перенос, если же $\beta_2 = \varepsilon$, то надо выполнить свертку по правилу $A \to \beta_1$. Конечно, для данного активного префикса могут быть допустимыми несколько пунктов, и они могут указать на разные действия. Возникающий конфликт может быть разрешен сравнением цепочки v с очередными k символами анализируемой цепочки или некоторыми другими средствами.

В этом параграфе мы рассматриваем только LR(0)-пункты, которые для удобства называем просто пунктами. Допустимость LR(0)-пункта, очевидно, определяется только наличием вывода вида (4.1). Каждая грамматика имеет лишь конечное множество пунктов, а значит, это множество может являться множеством состояний конечного автомата. Определим такой автомат (это будет ε-НКА, см. ч. 1, § 5).

Определение. Автоматом пунктов (расширенной) грамматики $G = (\Sigma, \Gamma, P, S')$ называется ε-НКА $\mathcal{I}_G{=}(I, \Sigma{\cup}\Gamma, \delta, \mathtt{i}_0, I)$, где Q – множество всех пунктов грамматики G, $\mathtt{i}_0 = [S' \to {\cdot}S]$, а множество переходов δ состоит из всех переходов вида $([A \to \beta_1{\cdot}X\beta_2], X, [A \to \beta_1 X{\cdot}\beta_2])$ и всех переходов вида $([A \to \beta_1{\cdot}B\beta_2], \varepsilon, [B \to {\cdot}\beta])$. (Как обычно, X – произвольный символ грамматики, A и B – нетерминалы, β_1, β_2, β – произвольные цепочки.)

Пусть грамматика G зафиксирована. Следующая теорема, иногда называемая *основной теоремой LR-анализа*, связывает автомат $\mathcal{I}_G$ с введенными ранее понятиями.

Теорема 4.1. *Пункт* $\mathtt{i} = [A \to \beta_1{\cdot}\beta_2]$ *грамматики* G *является допустимым для активного префикса* γ *этой грамматики тогда и только*

тогда, когда в автомате $\mathcal{I}_G$ существует путь из $\mathtt{i}_0$ *в* $\mathtt{i}$, *помеченный цепочкой* γ.

Договоримся называть *базисными пунктами* те пункты, точка в которых стоит не в начале правой части правила, и пункт $[S'\to\cdot S]$. *Базисными переходами* назовем переходы автомата $\mathcal{I}_G$, метка которых отлична от ε. Как нетрудно заметить, в базисные пункты ведут базисные переходы и только они. Доказательство теоремы 4.1 начнем с двух лемм.

Лемма 4.1. *Пусть γ – активный префикс. Тогда найдется базисный пункт, допустимый для γ.*

Доказательство. Пусть γ – префикс некоторой r-формы, принадлежащей выводу $S' \Rightarrow^* w$, не выходящий за правый конец основы этой формы. Предположим, что $\gamma\alpha$ – первая (т. е. ближайшая к S) r-форма в данном выводе, для которой γ – такой префикс. Пусть эта r-форма возникла не на первом шаге вывода. Тогда основа формы $\gamma\alpha$ не может целиком лежать в α. В самом деле, в этом случае после перенесения основы в стек и свертки цепочка γ останется в стеке, т. е. будет являться активным префиксом предыдущей r-формы по предложению 3.1, что противоречит выбору формы $\gamma\alpha$. Поскольку γ не выходит за правый конец основы, эта основа имеет вид $\beta_1\beta_2$, где $\gamma = \gamma'\beta_1$, $\alpha = \beta_2 u$ и $\beta_1 \neq \varepsilon$. (Цепочка u состоит из терминалов, так как находится справа от основы.) Но тогда есть правило $A \to \beta_1\beta_2$ и рассматриваемый вывод $S' \Rightarrow^* w$ можно конкретизировать:

$$S' \Rightarrow^* \gamma' A u \Rightarrow \gamma'\beta_1\beta_2 u = \gamma\alpha \Rightarrow^* w.$$

Мы показали, что пункт $A \to \beta_1\cdot\beta_2$ является базисным и допусти́м для γ по определению.

Осталось рассмотреть случай, когда r-форма $\gamma\alpha$ возникла на первом шаге вывода. Но первым в выводе применялось правило $S' \to S$, т. е. либо $\gamma = \varepsilon$ и допустимым для него базисным пунктом является $[S' \to \cdot S]$, либо $\gamma = S$, а допустимый пункт – это $[S' \to S\cdot]$. □

Замечание 4.1. Из доказательства леммы 4.1 следует, что активный префикс γ впервые появляется в выводе сразу после применения правила, один из базисных пунктов которого допусти́м для γ.

Лемма 4.2. *Пункт $B \to \cdot\beta$ допусти́м для активного префикса γ тогда и только тогда, когда он достижим по ε-переходам автомата $\mathcal{I}_G$ из некоторого базисного пункта, допустимого для γ.*

Доказательство. Докажем достаточность. Пусть пункт $[A{\to}\beta_1{\cdot}B\beta_2]$ (не обязательно базисный) допустим для $\gamma = \gamma'\beta_1$ и $B \to \beta$ – правило. Тогда возможен вывод

$$S' \Rightarrow^* \gamma' Au \Rightarrow \gamma'\beta_1 B\beta_2 u = \gamma B\beta_2 u \Rightarrow^* \gamma Bvu \Rightarrow \gamma\beta vu \Rightarrow^* w,$$

откуда по определению пункт $[B \to {\cdot}\beta]$ допустим для γ. Поскольку $([A \to \beta_1{\cdot}B\beta_2], \varepsilon, [B \to {\cdot}\beta])$ – ε-переход автомата $\mathcal{I}_G$, то допустимость «наследуется» по ε-переходу, а значит и по любому количеству ε-переходов. Достаточность доказана.

Перейдем к доказательству необходимости. По определению, существует вывод $S' \Rightarrow^* \gamma Bv \Rightarrow \gamma\beta v \Rightarrow^* w$. Рассмотрим первое появление активного префикса γ в данном выводе. Ему соответствует некоторый базисный пункт $[A \to \beta_1{\cdot}\beta_2]$ (лемма 4.1, замечание 4.1), что позволяет записать рассматриваемый вывод подробнее:

$$S' \Rightarrow^* \gamma' Au \Rightarrow \gamma'\beta_1\beta_2 u = \gamma\beta_2 u \Rightarrow^* \gamma Bv \Rightarrow \gamma\beta v \Rightarrow^* w$$

(цепочки $\gamma\beta_2 u$ и γBv могут и совпадать). Так как u – цепочка терминалов, из β_2 выводима (правым выводом!) цепочка, начинающаяся с B. Поскольку вывод – правый, β_2 начинается с нетерминала (обозначим его C), из которого и выводится B[†]. Если $\gamma\beta_2 u = \gamma Bv$, то $C = B$, а если эти цепочки различны, то существует правый вывод

$$C \Rightarrow B_1\alpha_1 \Rightarrow^* B_1 u_1 \Rightarrow B_2\alpha_2 u_1 \Rightarrow^* B_2 u_2 \Rightarrow \ldots \Rightarrow^* B_k u_k = Bu_k.$$

В этом выводе использовались, в частности, правила

$$C \to B_1\alpha_1, B_1 \to B_2\alpha_2, \ldots, B_{k-1} \to B\alpha_k.$$

Тогда по определению автомата $\mathcal{I}_G$ в нем имеются ε-переходы

$$\begin{array}{lll} ([A \to \beta_1{\cdot}\beta_2], & \varepsilon, & [C \to {\cdot}B_1\alpha_1]), \\ ([C \to {\cdot}B_1\alpha_1], & \varepsilon, & [B_1 \to {\cdot}B_2\alpha_2]), \\ & \ldots, & \\ ([B_{k-1} \to {\cdot}B\alpha_k], & \varepsilon, & [B \to {\cdot}\beta]). \end{array}$$

Так как пункт $[A \to \beta_1{\cdot}\beta_2]$ – базисный, доказательство необходимости завершено. □

[†]Существенно, что вывод именно правый. Без этого ограничения β_2 может начинаться, к примеру, с пары нетерминалов DC таких, что D впоследствии аннулируется, а из C выводится цепочка, начинающаяся с B. Но при правом выводе D не может быть аннулирован раньше, чем будет развернут B, и цепочка, начинающаяся с B, из β_2 не выведется.

Доказательство теоремы 4.1. Доказательство достаточности проведем индукцией по длине цепочки γ. Для базы индукции возьмем $\gamma = \varepsilon$. Пункт $\mathtt{i} = \mathtt{i}_0$ допустим для активного префикса ε по определению. Тогда по лемме 4.2 любой пункт, достижимый из $\mathtt{i}_0$ по ε-переходам, допустим для ε. База индукции доказана.

Перейдем к шагу индукции. Пусть $\gamma = \bar{\gamma}X$, тогда последний базисный переход в пути, помеченном γ, может быть записан как $([A \to \beta_1 \cdot X\beta_2], X, [A \to \beta_1 X \cdot \beta_2])$, а за ним следует ноль или более ε-переходов. Так как отрезок пути до пункта $[A \to \beta_1 \cdot X\beta_2]$ помечен цепочкой $\bar{\gamma}$, то по предположению индукции этот пункт допустим для $\bar{\gamma}$. Следовательно, цепочка $\bar{\gamma}$ представима в виде $\gamma'\beta_1$, и существует вывод

$$S' \Rightarrow^* \gamma' A u \Rightarrow \gamma'\beta_1 X \beta_2 u \Rightarrow^* w. \tag{4.2}$$

Но тогда базисный пункт $[A \to \beta_1 X \cdot \beta_2]$ по определению допустим для цепочки $\gamma = \gamma'\beta_1 X$. Последний пункт в рассматриваемом пути достижим из этого базисного пункта по ε-переходам, т. е. допустим для γ по лемме 4.2. Шаг индукции завершен.

Теперь докажем необходимость, также используя индукцию по длине γ. В базе индукции возьмем произвольный пункт $[A \to \beta_1 \cdot \beta_2]$, допустимый для $\gamma = \varepsilon$. Поскольку β_1 – суффикс γ, то $\beta_1 = \varepsilon$. Значит, единственный базисный пункт, допустимый для γ, – это начальный пункт $\mathtt{i}_0$. Требуемое утверждение следует отсюда по лемме 4.2.

Для доказательства шага индукции возьмем активный префикс $\gamma = \bar{\gamma}X$. По лемме 4.1 для него найдется допустимый базисный пункт, скажем $\mathtt{i} = [A \to \beta_1 X \cdot \beta_2]$. Тогда $\gamma = \gamma'\beta_1 X$ и существует вывод (4.2). Следовательно, пункт $[A \to \beta_1 \cdot X\beta_2]$ по определению допустим для цепочки $\bar{\gamma} = \gamma'\beta_1$. По предположению индукции, существует путь из пункта $\mathtt{i}_0$ в $[A \to \beta_1 \cdot X\beta_2]$, помеченный цепочкой $\bar{\gamma}$. Добавив к этому пути переход $([A \to \beta_1 \cdot X\beta_2], X, [A \to \beta_1 X \cdot \beta_2])$, получим путь из $\mathtt{i}_0$ в $\mathtt{i}$, помеченный цепочкой γ. Проведенное рассуждение справедливо для всех базисных пунктов, допустимых для γ. Теперь можно применить лемму 4.2, получая утверждение шага индукции для всех пунктов, допустимых для γ. Теорема доказана. □

Следствие 4.1. *Язык, распознаваемый автоматом $\mathcal{I}_G$, совпадает с множеством всех активных префиксов грамматики G.*

Доказательство. Напомним, что все состояния автомата $\mathcal{I}_G$ являются заключительными. Если γ – активный префикс, то по теореме 4.1 в автомате найдется путь из пункта $\mathtt{i}_0$, помеченный цепочкой

γ, т. е. автомат распознает γ. Обратно, если в $\mathcal{I}_G$ есть путь из $\mathtt{i}_0$ в $\mathtt{i}$, помеченный цепочкой γ, то γ – активный префикс, так как по теореме 4.1 пункт $\mathtt{i}$ является для него допустимым пунктом. □

Итак, в нашем распоряжении имеется конечный автомат, распознающий язык активных префиксов грамматики. Поскольку ε-НКА не очень удобен для работы, следующий шаг очевиден: нужно перейти к эквивалентному ДКА. Для этого имеется алгоритм 2.3 построения подмножеств (см. ч. 1, § 5). Полученный автомат, состояниями которого являются множества пунктов, мы будем называть *LR(0)-автоматом* грамматики и обозначать через $\mathcal{A}_G$. Множество состояний $\mathcal{A}_G$ часто называют *канонической системой пунктов* грамматики. Согласно алгоритму построения подмножеств в одно состояние ДКА «собираются» все состояния НКА, достижимые из начального состояния по некоторому пути, помеченному фиксированной цепочкой. Отсюда получаем

Следствие 4.2. *Состояние I автомата $\mathcal{A}_G$, достижимое из начального состояния I_0 по пути, помеченному цепочкой γ, совпадает с множеством допустимых пунктов активного префикса γ.*

Обычно $\mathcal{A}_G$ рассматривают как неполный ДКА, в этом случае все его состояния являются заключительными. (Если считать, что $\mathcal{A}_G$ – полный ДКА, то в нем будет одно незаключительное состояние – «мусорная корзина» для всех не определенных в автомате $\mathcal{I}_G$ переходов.)

Напомним, что в автомате $\mathcal{I}_G$ в базисный пункт $[A \to \alpha_1 X \cdot \alpha_2]$ ведет только переход с меткой X. Значит, если базисные пункты $[A \to \alpha_1 X \cdot \alpha_2]$ и $[B \to \beta_1 Y \cdot \beta_2]$ попали в одно состояние LR(0)-автомата, то $X = Y$. Следовательно, справедливо следующее замечание.

Замечание 4.2. Состояние, в которое пришел LR(0)-автомат в результате обработки некоторой цепочки, однозначно определяет последний символ этой цепочки (если состояние совпадает с начальным, то однозначно определяется, что эта цепочка – пустая). Этот символ удобно использовать для обозначения состояния (если символу соответствуют несколько состояний, то будем добавлять верхний индекс, см. также пример 4.1). Для обозначения начального состояния будем использовать символ дна стека.

Разберем несложный пример построения LR(0)-автомата.

Пример 4.2. Рассмотрим расширенную грамматику

$$G_{4,1} = \{S' \to S, S \to \mathtt{a}D\mathtt{c}, D \to D\mathtt{b}, D \to \mathtt{b}\}.$$

Начальное состояние автомата ∇ – это замыкание пункта $[S' \to \cdot S]$. Из этого пункта есть один ε-переход в пункт $[S \to \cdot\mathtt{a}D\mathtt{c}]$, а из последнего нет ε-переходов. Следовательно, $\nabla = \{[S' \to \cdot S], [S \to \cdot\mathtt{a}D\mathtt{c}]\}$.

Из ∇ возможны базисные переходы по символам S и $\mathtt{a}$. Переход по S ведет в пункт $[S' \to S\cdot]$, к которому не добавится пунктов в результате замыкания. Тем самым, состояние S – это множество из одного пункта. Переход по $\mathtt{a}$ происходит в пункт $[S \to \mathtt{a}\cdot D\mathtt{c}]$, откуда ε-переходы ведут в пункты $[D \to \cdot D\mathtt{b}]$, $[D \to \cdot\mathtt{b}]$; построено состояние $\mathtt{a}$ из трех пунктов. Теперь нужно рассмотреть базисные переходы из $\mathtt{a}$ (из состояния S нет переходов), и т. д. Полностью диаграмма переходов автомата $\mathcal{A}_{G_{4,1}}$ приведена на рис. 4.1.

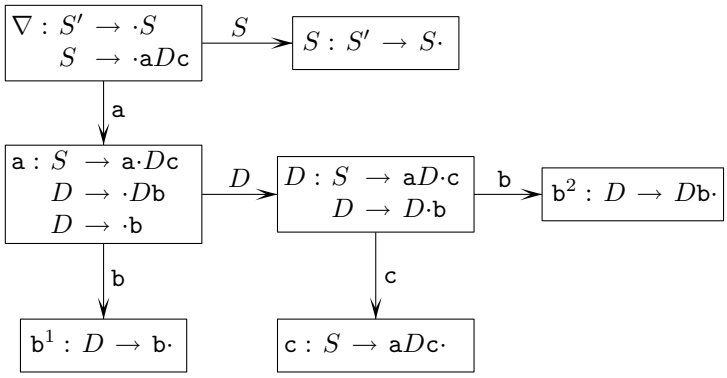


Рис. 4.1. *LR(0)-автомат грамматики $G_{4,1}$. Начальное состояние – ∇*

Приведенный пример иллюстрирует еще одну идею: для построения LR(0)-автомата $\mathcal{A}_G$ совсем не нужно строить ε-НКА $\mathcal{I}_G$ – этот ε-НКА неявно задан самой грамматикой. Обычно алгоритм 2.3 адаптируют для построения LR(0)-автомата. Для этой цели определим две вспомогательные функции.

Определение. Пусть M – произвольное множество пунктов грамматики G. Тогда CLOSURE(M) – это минимальное по включению множество пунктов, содержащее M, и такое, что вместе с пунктом вида $[A \to \alpha\cdot B\beta]$ в нем содержатся все пункты $[B \to \cdot\gamma]$, где $B \to \gamma$ – правило вывода.

Легко проверить, что множество $\textsc{closure}(M)$ – это и есть *замыкание* в терминах ε-НКА. Алгоритм построения $\textsc{closure}(M)$ очень прост. В качестве первого приближения берем само множество M и все его пункты упорядочиваем в виде очереди. В основном цикле обрабатываем первый элемент очереди, добавляя пункты вида $A \to \cdot\gamma$ к множеству $\textsc{closure}(M)$ и к концу очереди (сам первый элемент из очереди удаляем). Когда очередь становится пустой, построение завершено.

Определение. Пусть M – произвольное множество пунктов грамматики G, X – ее произвольный символ. Тогда

$$\textsc{goto}(M, X) = \textsc{closure}(\{[A \to \alpha X \cdot \beta] \mid [A \to \alpha \cdot X\beta] \in M\}) .$$

Функция GOTO и есть функция переходов LR(0)-автомата. Алгоритм построения множества $\textsc{goto}(M, X)$ достаточно очевиден. Алгоритм построения LR(0)-автомата, использующий эти две функции, приведен ниже.

Алгоритм 4.1. *Построение LR(0)-автомата по грамматике.*
Вход. Расширенная грамматика $G = (\Sigma, \Gamma, P, S')$.
Выход. ДКА $\mathcal{A}_G = (Q, \Sigma \cup \Gamma, \delta, I_0, Q)$.

1. $I_0 \leftarrow \textsc{closure}([S' \to \cdot S])$; $label(I_0) \leftarrow 0$
2. $Q \leftarrow \{I_0\}$
3. **пока** $(\exists I \in Q : label(I) = 0)$, **повторять**
4. **для каждого** $X \in \Sigma \cup \Gamma$
5. $\delta(I, X) \leftarrow \textsc{goto}(I, X)$
6. **если** $(\delta(I, X) \notin Q)$
7. $Q \leftarrow Q \cup \{\delta(I, X)\}$
8. $label(\delta(I, X)) \leftarrow 0$
9. $label(I) \leftarrow 1$

§ 18. Анализ на основе LR(0)-автомата

Напомним, что наша основная цель – построение таблиц ACTION и GOTO LR-анализатора. Однако будет удобно начать с неформального описания роли LR(0)-автомата в анализе «перенос–свертка». Итак, возьмем цепочку w, тестируемую на выводимость в грамматике G, а подадим ее на вход автомата $\mathcal{A}_G$. После каждого такта ра-

боты автомата будем складывать новое состояние автомата в стек[†] и анализировать это новое состояние, возможно, производя некоторые манипуляции перед тем, как разрешить автомату следующий такт.

В чем состоит анализ состояния? Наличие в состоянии пункта $A \to \beta_1 \cdot \beta_2$ ($\beta_2 \neq \varepsilon$) говорит о необходимости переноса следующего символа, и мы просто переходим к следующему такту работы автомата с цепочкой w.

Если в состоянии есть пункт вида $A \to \alpha\cdot$, то в стек перенесена основа, и нужно выполнить свертку по указанному правилу. При этом активный префикс $\gamma\alpha$ в стеке должен замениться на активный префикс γA. Вытолкнем из стека верхние $|\alpha|$ состояний автомата. После этого на вершине стека будет состояние автомата после прочтения цепочки γ. Вернем автомат в это состояние и подадим ему на вход на очередном такте символ A. *Так как автомат распознает все активные префиксы, в том числе цепочку γA, он сможет выполнить этот такт.*

Пока автомат работает, в стеке всегда будет находиться активный префикс грамматики. Но поскольку LR(0)-автомат является неполным (некоторые переходы отсутствуют), то иногда он не сможет прочитать очередной символ. Это означает, что цепочка невыводима. В самом деле, в предыдущем абзаце мы показали, что автомат обязательно прочтет поданный на вход нетерминал, т. е. *не прочесть* он может только очередной терминал из w. Но это означает, что перед данным тактом наверху стека не было основы (невозможна свертка), а при переносе очередного символа не получится активного префикса (т. е. перенос также невозможен).

Если же автомат дочитает цепочку w до конца и после этого придет в состояние, содержащее пункт $[S' \to S\cdot]$, то выводимость цепочки установлена.

Трудности при анализе возникают, если пункты внутри одного состояния «конфликтуют» – один пункт указывает на перенос, а другой – на свертку (*конфликт перенос–свертка*) или два пункта указывают на свертку по разным правилам (*конфликт свертка–свертка*). Ниже будет предложен способ разрешения таких конфликтов на основе анализа текущего символа цепочки. Этот способ неплох, но не универсален. Если он не помогает, то нужно либо видоизменить грамматику, либо вместо LR(0)-автомата использовать LR(1)-автомат (такие автоматы вводятся в следующем параграфе).

[†]Тем самым в стек помещается и прочитанный символ, см. замечание 4.2.

Начнем с «бесконфликтных» грамматик.

Определение. Грамматика G называется $LR(0)$-*грамматикой*, если каждое состояние ее LR(0)-автомата, содержащее пункт вида $A \to \alpha\cdot$, состоит из единственного пункта.

Грамматика $G_{4,1}$ подходит под это определение (см. рис. 4.1). Для этих грамматик таблицы ACTION и GOTO строятся следующим алгоритмом.

Алгоритм 4.2. *Построение LR(0)-анализатора.*
Вход. Расширенная LR(0)-грамматика $G = (\Sigma, \Gamma, P, S')$.
Выход. Таблицы ACTION и GOTO для LR-анализа.

1. `построить` LR(0)-автомат $\mathcal{A}_G$ %% алгоритм 4.1
2. `индексировать` строки таблиц ACTION и GOTO состояниями $\mathcal{A}_G$
3. **для каждого** `состояния` I
4. **если** $(I = \{[S' \to S\cdot]\})$ %% допуск
5. ACTION$(I, \dashv) \leftarrow \checkmark$
6. **иначе если** $(I = \{[A \to \alpha\cdot]\})$ %% свертка
7. **для каждого** $\mathtt{a} \in \Sigma \cup \{\dashv\}$
8. ACTION$(I, \mathtt{a}) \leftarrow (\otimes number(A \to \alpha))$
9. **иначе** %% перенос
10. **для каждого** $\mathtt{a} \in \Sigma$
11. **если** $(\delta(I, \mathtt{a}) \neq \emptyset)$
12. ACTION$(I, \mathtt{a}) \leftarrow (_{\leftarrow}\delta(I, \mathtt{a}))$
13. **для каждого** $A \in \Gamma$
14. **если** $(\delta(I, A) \neq \emptyset)$
15. GOTO$(I, A) \leftarrow \delta(I, A)$

Из определения LR(0)-грамматики и детерминированности автомата $\mathcal{A}_G$ следует, что в клетке каждой из таблиц может оказаться не более одной записи (проверку этого простого факта мы оставляем читателю). В табл. 4.3 приведен результат работы алгоритма 4.2 для грамматики $G_{4,1}$ (правила занумерованы по порядку их записи в определении грамматики, «расширяющее» правило $S' \to S$ имеет номер 0). Как уже говорилось в § 16, пустые клетки таблицы указывают на ошибки[†].

[†]Вообще говоря, к некоторым пустым клеткам корректно работающий анализатор не может обратиться независимо от цепочки, поданной ему на вход. В частности, он не может обратиться к пустым клеткам таблицы GOTO – см. замечание курсивом на с. 164.

Таблица 4.3. *Таблица LR(0)-анализатора для грамматики* $G_{4,1}$

	ACTION				GOTO	
	a	b	c	⊣	S	D
S				✓		
D		$\leftarrow$b^2	$\leftarrow$c			
a		$\leftarrow$b^1				D
b^1	$\otimes 3$	$\otimes 3$	$\otimes 3$	$\otimes 3$		
b^2	$\otimes 2$	$\otimes 2$	$\otimes 2$	$\otimes 2$		
c	$\otimes 1$	$\otimes 1$	$\otimes 1$	$\otimes 1$		
∇	$\leftarrow$a				S	

Предложение 4.1. *LR-анализатор с таблицами* ACTION *и* GOTO*, построенными алгоритмом 4.2 по грамматике G, допускает цепочку w если и только если $w \in L(G)$.*

Доказательство. Из алгоритма построения таблиц следует, что данный анализатор работает как LR(0)-автомат с «перезапуском» в случае свертки, описанный в начале данного параграфа. Анализатор выполняет свертку в том и только в том случае, когда он находится в состоянии, содержащем пункт вида $A \to \alpha\cdot$ (и не содержащем других пунктов по определению LR(0)-грамматики). Если автомат пришел в такое состояние по прочтении цепочки $\gamma = \gamma'\alpha$, то по теореме 4.1 и определению допустимого пункта γ является активным префиксом, а α – основой некоторой r-формы, начинающейся с γ. Поскольку других пунктов в рассматриваемом состоянии нет, α является основой *любой* r-формы, начинающейся с γ, и свертка всегда производится по правилу $A \to \alpha$. Итак, анализатор выполняет свертку тогда и только тогда, когда наверху стека находится основа текущей r-формы. В противном случае анализатор пытается осуществить перенос. В случае корректной цепочки перенос заведомо возможен, так как LR(0)-автомат распознает все активные префиксы грамматики, а значит соответствующее множество $\delta(I, \mathtt{a})$ непусто. Следовательно, анализатор правильно восстанавливает правый вывод цепочки в грамматике G. Осталось заметить, что допуск происходит тогда и только тогда, когда цепочка свернута в аксиому грамматики. □

Итак, мы построили самый простой вариант LR-анализатора. К сожалению, класс LR(0)-грамматик достаточно узок, а для грамматик за пределами этого класса некоторые состояния LR(0)-автомата

содержат конфликты. Алгоритм 4.2 просто скопирует эти конфликты в таблицу ACTION, что сделает полученный анализатор непригодным для практических целей. Рассмотрим два примера не LR(0)-грамматик, похожих на LR(0)-грамматику $G_{4,1}$.

Пример 4.3. LR(0)-автомат грамматики

$$G_{4,2} = \{S' \to S,\ S \to \mathtt{a}D\mathtt{b},\ D \to D\mathtt{b} \mid \mathtt{b}\}$$

приведен на рис. 4.2. В состоянии $\mathtt{b}^2 = \{[S \to \mathtt{a}D\mathtt{b}\cdot], [D \to D\mathtt{b}\cdot]\}$ присутствует конфликт «свертка-свертка». Этот конфликт можно разрешить при помощи очередного входного символа: если это символ конца цепочки, то надо выполнять свертку к S, а если это символ $\mathtt{b}$, то корректной будет свертка к D.

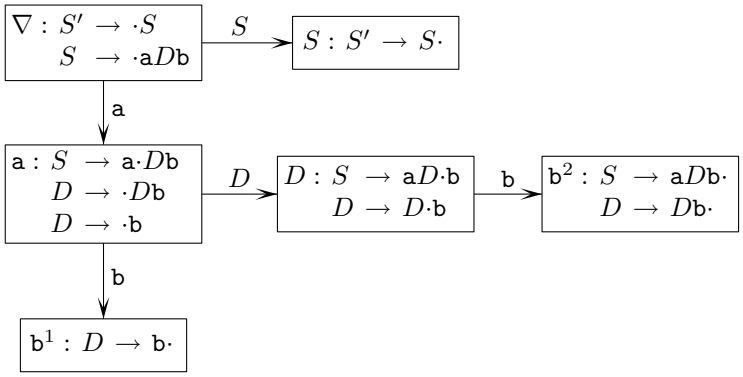


Рис.4.2. *LR(0)-автомат грамматики $G_{4,2}$. Начальное состояние* – ∇

Пример 4.4. LR(0)-автомат грамматики

$$G_{4,3} = \{S' \to S,\ S \to \mathtt{ac} \mid \mathtt{a}D\mathtt{bc},\ D \to D\mathtt{b} \mid \varepsilon\}$$

содержит два состояния, в которых имеется конфликт «перенос–свертка» (построение всего автомата мы предоставляем читателю). Это состояния $\mathtt{a} = \{[S \to \mathtt{a}\cdot\mathtt{c}], [S \to \mathtt{a}\cdot D\mathtt{bc}], [D \to \cdot D\mathtt{b}], [D \to \cdot]\}$ и $\mathtt{b} = \{[S \to \mathtt{a}D\mathtt{b}\cdot\mathtt{c}], [D \to D\mathtt{b}\cdot]\}$. Снова конфликты могут быть разрешены просмотром очередного входного символа. В обоих случаях нужно выполнять свертку при входном символе $\mathtt{b}$ и перенос при символе $\mathtt{c}$ (проверьте это!).

Второй пример, в частности, иллюстрирует следующую простую идею: если очередной символ входной цепочки «ожидается» для переноса каким-либо пунктом текущего состояния LR(0)-автомата, то производится перенос (в обоих «конфликтных» состояниях последнего примера таким символом был символ `c`), а иначе – свертка. Однако непонятно, всегда ли правилен ли такой подход. В самом деле, в принципе может найтись такая r-форма, в которой нужно выполнять свертку, несмотря на то, что возможен перенос.

Заметим, что в некоторых случаях свертка заведомо не приведет к успешному разбору цепочки! Необходимым условием успешного разбора является то, что результатом свертки является некоторая грамматическая *форма*. А если результатом свертки к нетерминалу A является форма, то в ней сразу после A находится символ из множества $\textsc{follow}(A)$ (либо A является последним символом формы и $\dashv \in \textsc{follow}(A)$). Тем самым мы показали справедливость следующего важного замечания.

Замечание 4.3. Свертка по правилу $A \to \alpha$ будет правильным действием LR-анализатора только в том случае, когда очередной символ входной цепочки принадлежит множеству $\textsc{follow}(A)$. В частности, допуск (т. е. свертка по правилу $S' \to S$) может производиться только по окончании входной цепочки, так как $\textsc{follow}(S') = \{\dashv\}$.

Таким образом, при заполнении таблицы ACTION для состояния, содержащего пункт вида $A \to \alpha\cdot$, *свертку достаточно записать в клетках, проиндексированных символами из* $\textsc{follow}(A)$. Эта простая идея значительно усиливает метод LR(0)-анализа. Изменив построение таблицы ACTION указанным способом, мы получаем *простой LR(1)-анализатор* (*SLR(1)-анализатор*). Грамматики, для которых построенная таким способом таблица ACTION не содержит множественных записей (т. е. конфликтов), называются *SLR(1)-грамматиками*. Ввиду практической значимости этого класса грамматик, выпишем алгоритм построения SLR(1)-анализатора явно, хотя он лишь незначительно отличается от алгоритма 4.2.

Алгоритм 4.3. *Построение SLR(1)-анализатора.*
Вход. Расширенная грамматика $G = (\Sigma, \Gamma, P, S')$.
Выход. Таблицы ACTION и GOTO для LR-анализа.

1. `построить` LR(0)-автомат $\mathcal{A}_G$ %% алгоритм 4.1
2. `индексировать` строки таблиц ACTION и GOTO состояниями $\mathcal{A}_G$
3. **для каждого** $A \in \Gamma$

4. `построить` множество FOLLOW(A) %% алгоритм 2.4
5. **для каждого** `состояния` I
6. **для каждого** `пункта` $\mathtt{i} \in I$
7. **если** $(\mathtt{i} = [S' \to S\cdot])$ %% допуск
8. ACTION$(I, \dashv) \leftarrow \checkmark$
9. **иначе если** $(\mathtt{i} = [A \to \alpha\cdot])$ %% свертка
10. **для каждого** $\mathtt{a} \in$ FOLLOW(A)
11. ACTION$(I, \mathtt{a}) \leftarrow (\otimes number(A \to \alpha))$
12. **иначе если** $(\mathtt{i} = [A \to \beta_1\cdot\mathtt{a}\beta_2])$ %% перенос
13. ACTION$(I, \mathtt{a}) \leftarrow ({}_{\leftarrow}\delta(I, \mathtt{a}))$
14. **иначе** %% $\mathtt{i} = [A \to \beta_1\cdot B\beta_2]$
15. GOTO$(I, B) \leftarrow \delta(I, B)$

Замечание 4.4. Предложение 4.1 остается верным при замене в нем алгоритма 4.2 алгоритмом 4.3. Действительно, в результате модернизации алгоритма в таблицу ACTION не попадут некоторые заведомо некорректные свертки. Результат некорректной свертки – это цепочка, не являющаяся r-формой, а значит такая цепочка не может быть свернута в аксиому грамматики. Таким образом, наличие или отсутствие таких сверток не влияет на распознаваемость цепочек (но позволяет расширить класс анализируемых грамматик).

Проверим, что грамматика $G_{4,2}$ является SLR(1)-грамматикой. Так как FOLLOW$(S) = \{\dashv\}$, FOLLOW$(D) = \{\mathtt{b}\}$, алгоритм 4.3 построит бесконфликтную таблицу LR-анализа (табл. 4.4).

Таблица 4.4. *Таблица SLR(1)-анализатора для грамматики* $G_{4,2}$

	ACTION			GOTO	
	a	b	$\dashv$	S	D
S			$\checkmark$		
D		${}_{\leftarrow}\mathtt{b}^2$			
a		${}_{\leftarrow}\mathtt{b}^1$			D
$\mathtt{b}^1$		$\otimes 3$			
$\mathtt{b}^2$		$\otimes 2$	$\otimes 1$		
∇	${}_{\leftarrow}\mathtt{a}$			S	

Грамматика $G_{4,3}$ также является SLR(1)-грамматикой (см. задачу 2а). К классу SLR(1)-грамматик принадлежат и существенно более сложные грамматики (в частности, грамматики выражений, как показывает следующий пример).

Пример 4.5. Построим LR(0)-автомат для грамматики арифметических выражений GA_2 (расширив ее правилом $E' \to E$). Детали построения мы оставляем читателю (см. пример 4.2), а результат приведен на рис. 4.3. Заметим, что этот автомат содержит циклы, в отличие от приведенных на рис. 4.1, 4.2 LR(0)-автоматов для грамматик $G_{4,1}$ и $G_{4,2}$. Наличие циклов (и, следовательно, бесконечное множество активных префиксов грамматики) связано с потенциально неограниченной глубиной вложенности скобок.

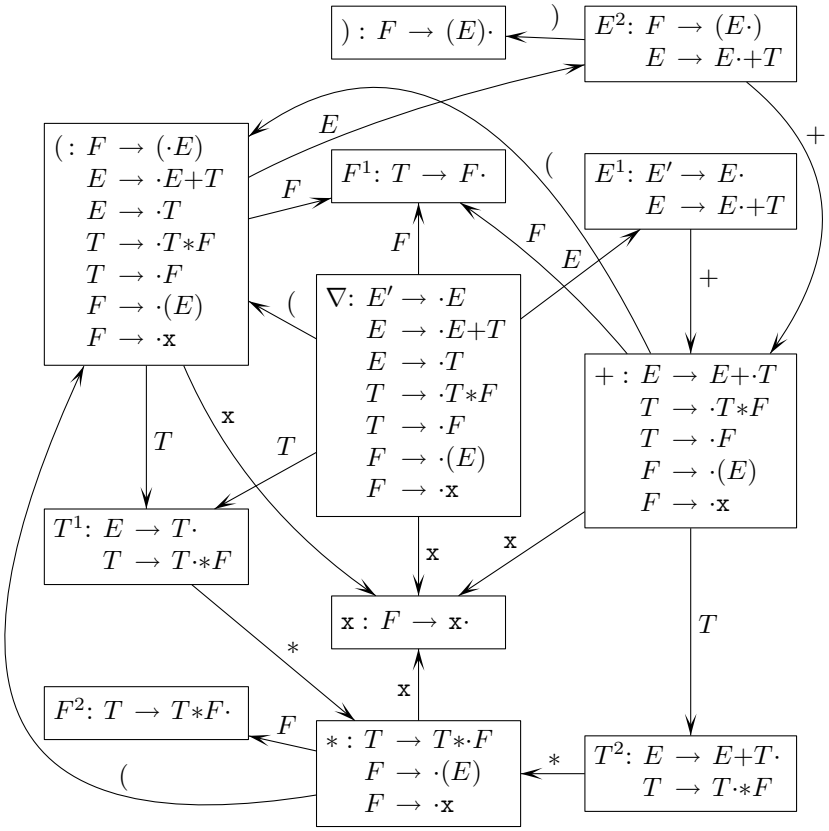


Рис. 4.3. *LR(0)-автомат грамматики GA_2. Начальное состояние – ∇*

Мы видим, что грамматика GA_2 не является LR(0)-грамматикой: состояния E^1, T^1 и T^2 LR(0)-автомата содержат как пункт, указывающий на свертку, так и пункт, указывающий на перенос. Если же применить алгоритм 4.3, то результатом будет таблица анализа, приведенная в табл. 4.2 (применяя алгоритм 2.4 построения множеств FOLLOW, получим $\text{FOLLOW}(E') = \{\dashv\}$, $\text{FOLLOW}(E) = \{+,), \dashv\}$, $\text{FOLLOW}(T) = \text{FOLLOW}(F) = \{+, *,), \dashv\}$).

§ 19. Анализ на основе LR(1)-автомата

Рассмотрим грамматику

$$G_{4,4} = \{S' \to S,\ S \to \mathtt{ac} \mid \mathtt{b}D\mathtt{c} \mid D\mathtt{a},\ D \to \mathtt{a}\}.$$

Ее LR(0)-автомат приведен на рис. 4.4. Состояние $\mathtt{a}^1$ этого автомата содержит конфликт «перенос–свертка». Нетрудно убедиться, что SLR(1)-анализатор не сможет разрешить этот конфликт, так как $\text{FOLLOW}(D) = \{\mathtt{a}, \mathtt{c}\}$.

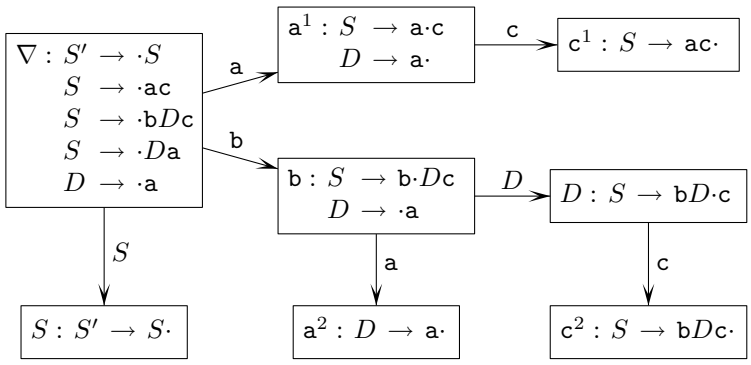


Рис.4.4. *LR(0)-автомат грамматики $G_{4,4}$. Начальное состояние* – ∇

Проведем небольшой анализ. Проследим «историю» появления пункта $[D \to \mathtt{a}\cdot]$ в состоянии $\mathtt{a}^1$. Этот пункт появился из пункта $[D \to \cdot\mathtt{a}]$ состояния ∇ в результате переноса входного символа a. В свою очередь, пункт $[D \to \cdot\mathtt{a}]$ появился в состоянии ∇ в результате замыкания пункта $[S \to \cdot D\mathtt{a}]$. После свертки к D в состоянии $\mathtt{a}^1$ автомат вернется в состояние ∇ и выполнит там переход по D, получая пункт $[S \to D\cdot\mathtt{a}]$, в котором ожидается появление на входе символа

a. Значит, свертка к D в рассматриваемой ситуации будет корректной только при входном символе a и конфликт для символа c может быть разрешен в пользу переноса.

SLR(1)-анализатору «не хватает мощности» для выведения подобных заключений. Поэтому сейчас мы рассмотрим наиболее общую технологию построения таблиц LR(1)-анализа. Для этого нужно перейти от LR(0)-пунктов к LR(1)-пунктам. Напомним, что LR(1)-пункт имеет вид $[A \to \beta_1 \cdot \beta_2,\ \mathtt{a}]$, где $A \to \beta_1 \cdot \beta_2$ – LR(0)-пункт (называемый *ядром* LR(1)-пункта), a – терминал либо символ $\dashv$. LR(1)-пункт допусти́м для активного префикса $\gamma = \gamma'\beta_1$, если существует правый вывод $S' \Rightarrow^* \gamma' A v \Rightarrow \gamma'\beta_1\beta_2 v \Rightarrow^* uv$ и цепочка $v\dashv$ начинается символом a. Иначе говоря, при разборе r-формы $\gamma'\beta_1\beta_2 v$ LR-анализатор, перенеся β_2 в стек, «увидит» на входе символ a.

На множестве LR(1)-пунктов грамматики можно задать структуру ε-НКА, определив множество базисных переходов, состоящее из всех переходов вида

$$([A \to \beta_1 \cdot X \beta_2,\ \mathtt{a}], X, [A \to \beta_1 X \cdot \beta_2,\ \mathtt{a}]),$$

и множество ε-переходов, состоящее из всех переходов вида

$$([A \to \beta_1 \cdot B \beta_2,\ \mathtt{a}], \varepsilon, [B \to \cdot\beta,\ \mathtt{b}]),$$

где $\mathtt{b} \in \textsc{first}(\beta_2\mathtt{a})$ [†]. Последнее требование объяснимо: после переноса B в стек текущий входной символ должен позволить продолжить формирование основы $\beta_1 B \beta_2$, за которой следует символ a. Начальным состоянием автомата будет пункт $([S' \to \cdot S,\ \dashv]$, а заключительными – все пункты. Нетрудно заметить, что распознаваемость цепочки полученным автоматом зависит только от ядер пунктов, а значит, для него справедливы основная теорема LR-анализа и ее следствия. В частности, детерминируя данный ε-НКА, мы получим автомат $\mathcal{B}_G$, эквивалентный LR(0)-автомату грамматики и называемый ее *LR(1)-автоматом*. При помощи этого автомата можно справиться с более широким спектром конфликтов при построении таблицы ACTION LR-анализатора. Тем самым становится возможной работа в более широком классе грамматик, называемых *LR(1)-грамматиками*.

Как и в случае LR(0)-автомата, LR(1)-автомат строится напрямую по грамматике при помощи двух вспомогательных функций.

[†]Напомним, что построение множеств FIRST обсуждается в гл. 2. В нашем случае удобно при построении считать символ $\dashv$ терминалом.

Определение. Пусть M – произвольное множество LR(1)-пунктов грамматики G, X – ее произвольный символ, $\mathtt{a}$ – терминал или символ конца цепочки. Тогда $\textsc{closure}_1(M)$ – это минимальное по включению множество LR(1)-пунктов, содержащее M и такое, что вместе с LR(1)-пунктом вида $[A \to \beta_1 \cdot B\beta_2, \mathtt{a}]$ в нем содержатся все LR(1)-пункты $[B \to \cdot\gamma, \mathtt{b}]$, где $B \to \gamma$ – правило грамматики и $\mathtt{b} \in \textsc{first}(\beta_2\mathtt{a})$. Множество $\textsc{goto}_1(M, X)$ определяется равенством

$$\textsc{goto}_1(M, X) = \textsc{closure}_1(\{[A \to \beta_1 X \cdot \beta_2, \mathtt{a}] \mid [A \to \beta_1 \cdot X\beta_2, \mathtt{a}] \in M\}).$$

Построение множества $\textsc{closure}_1(M)$ отличается от построения множества $\textsc{closure}(M)$ только привлечением алгоритма 2.3 для построения множеств FIRST, а алгоритм построения $\textsc{goto}_1(M, X)$ вполне очевиден. Алгоритм построения LR(1)-автомата по этим двум функциям почти совпадает с алгоритмом 4.1: в строке 1 нужно положить $I_0 \leftarrow \textsc{closure}_1([S' \to \cdot S, \dashv])$, а в строке 5 заменить GOTO на $\textsc{goto}_1$.

LR(1)-автомат для грамматики $G_{4,4}$ будет иметь восемь состояний, которые отличаются от состояний LR(0)-автомата на рис. 4.4 только вторыми компонентами входящих в них пунктов. Рассмотрим более подробно построение «конфликтного» состояния $\mathtt{a}^1$. В состояние ∇ при замыкании базисного пункта $[S' \to \cdot S, \dashv]$ будут добавлены пункты $[S \to \cdot\mathtt{ac}, \dashv]$, $[S \to \cdot\mathtt{b}D\mathtt{c}, \dashv]$, $[S \to \cdot D\mathtt{a}, \dashv]$ и $[D \to \cdot\mathtt{a}, \mathtt{a}]$. Состояние $\mathtt{a}^1$ строится переходом из ∇ по символу $\mathtt{a}$; при этом получаются пункты $[S \to \mathtt{a}\cdot\mathtt{c}, \dashv]$ и $[D \to \mathtt{a}\cdot, \mathtt{a}]$. Таким образом, в состоянии $\mathtt{a}^1$ после свертки к D ожидается входной символ $\mathtt{a}$. Таким образом, при входном символе $\mathtt{c}$ всегда должен осуществляться перенос, и конфликта не возникает.

Определение. Грамматика G называется *LR(1)-грамматикой*, если приводимый ниже алгоритм 4.4 строит таблицу ACTION, не содержащую конфликтов.

Алгоритм 4.4. *Построение LR(1)-анализатора.*
Вход. Расширенная грамматика $G = (\Sigma, \Gamma, P, S')$.
Выход. Таблицы ACTION и GOTO для LR-анализа.

1. `построить` LR(1)-автомат $\mathcal{B}_G$
2. `индексировать` строки таблиц ACTION и GOTO состояниями $\mathcal{B}_G$
3. **для каждого** `состояния` I
4. **для каждого** `пункта` $\mathtt{i} \in I$
5. **если** $(\mathtt{i} = [S' \to S\cdot, \dashv])$ %% допуск

6. ACTION$(I, \dashv) \leftarrow \checkmark$
7. **иначе если** $(\mathtt{i} = [A \to \alpha\cdot,\ \mathtt{a}])$ %% свертка
8. ACTION$(I, \mathtt{a}) \leftarrow (\otimes number(A \to \alpha))$
9. **иначе если** $(\mathtt{i} = [A \to \beta_1\cdot\mathtt{a}\beta_2])$ %% перенос
10. ACTION$(I, \mathtt{a}) \leftarrow ({}_{\leftarrow}\delta(I, \mathtt{a}))$
11. **иначе** %% $\mathtt{i} = [A \to \beta_1\cdot B\beta_2]$
12. GOTO$(I, B) \leftarrow \delta(I, B)$

Замечание 4.5. Как и алгоритм 4.3, алгоритм 4.4 не допускает попадания в таблицу ACTION только заведомо некорректных сверток. А значит, предложение 4.1 остается в силе и при использовании алгоритма 4.4 вместо алгоритма 4.2.

Класс LR(1)-грамматик включает класс SLR(1)-грамматик, и это включение строгое, как показывает, например, рассмотренная выше грамматика $G_{4,4}$. Использование LR(1)-автоматов для нужд синтаксического анализа имеет, однако, и одно неудобство. Такой автомат обычно намного больше LR(0)-автомата (для языка программирования размером с Паскаль первый автомат содержит несколько тысяч состояний, а второй – несколько сотен). Поэтому на практике часто используют «гибридный» метод построения LR-автомата. Этот метод носит название *LALR(1)-метода* (от LookAhead LR). Мы изложим его очень кратко, более подробно см., напр., в [АСУ].

LALR-автомат грамматики определяется так: берется LR(1)-автомат этой грамматики, и *каждая группа его состояний, имеющих одно и то же множество ядер LR(1)-пунктов, сливается в одно состояние*. Такое определение корректно, так как переходы между состояниями определяются только ядрами пунктов. Можно доказать (см. [АСУ]), что ни одно состояние LALR-автомата не содержит конфликта «перенос–свертка», если таких конфликтов не было в LR(1)-автомате. Однако в некоторых, достаточно редких, случаях при переходе к LALR-автомату может возникнуть конфликт «свертка–свертка» (см. пример 4.6 ниже). Нетрудно заметить, что LALR-автомат имеет такое же количество состояний, как и LR(0)-автомат. Более того, каждое состояние LALR-автомата получается из состояния LR(0)-автомата добавлением к каждому входящему в него пункту подходящего набора символов в качестве вторых компонентов LR(1)-пунктов. На практике LALR-автомат строится как раз по LR(0)-автомату вычислением таких наборов символов по всем пунктам всех его состояний.

Грамматика называется *LALR(1)-грамматикой*, если таблица ACTION, построенная по ее LALR-автомату, не содержит конфликтов. Грамматика $G_{4,4}$ является LALR(1)-грамматикой: как уже отмечалось выше, ее LR(1)- и LR(0)-автоматы имеют одинаковое число состояний, а значит LALR-автомат совпадает с LR(1)-автоматом. Как показывает следующий пример, существуют LR(1)-грамматики, не являющиеся LALR(1)-грамматиками.

Пример 4.6. Рассмотрим грамматику

$$G_{4,5} = \{S' \to S,\ S \to \mathtt{a}A\mathtt{a} \mid \mathtt{a}B\mathtt{b} \mid \mathtt{b}A\mathtt{b} \mid \mathtt{b}B\mathtt{a},\ A \to \mathtt{c},\ B \to \mathtt{c}\}\ .$$

Предлагаем читателю самостоятельно убедиться, что $G_{4,5}$ – LR(1)-грамматика. На рис. 4.5 изображена часть ее LR(1)-автомата. Ни одно из состояний $\mathtt{c}^1$, $\mathtt{c}^2$ не содержит конфликта. Но поскольку ядра этих состояний совпадают, при их объединении возникает конфликт сверток по правилам $A \to \mathtt{c}$ и $B \to \mathtt{c}$ на входных символах a и b. Следовательно, $G_{4,5}$ не является LALR(1)-грамматикой.

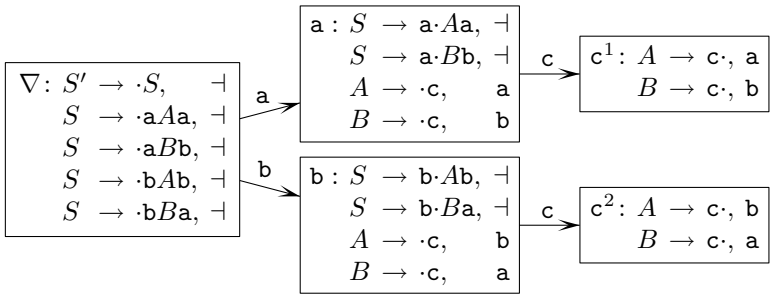


Рис. 4.5. *Фрагмент LR(1)-автомата грамматики* $G_{4,5}$

§ 20. Использование неоднозначных грамматик

Неоднозначные грамматики при умелом применении могут принести большую пользу в синтаксическом анализе. Разумеется, *полное описание языка должно быть однозначным, т. е. каждая принадлежащая языку цепочка должна представляться единственным деревом вывода.* Но для такого описания можно использовать и неоднозначную грамматику, снабженную набором дополнительных «внеграмматических» правил устранения неоднозначности. Такой подход

может, в частности, уменьшить размер LR-анализатора и сократить время его работы. Рассмотрим два примера, иллюстрирующих эту идею.

Пример 4.7. Для построения LR-анализатора арифметических выражений возьмем неоднозначную грамматику

$$GA_1 = \{E \to E{+}E \mid E{*}E \mid (E) \mid \mathtt{x}\},$$

используя для устранения неоднозначности знание приоритетов и ассоциативности операторов. LR(0)-автомат этой грамматики (мы оставляем его построение читателю) содержит 10 состояний – начальное, по одному для каждого терминала, и четыре для символа E. В двух из этих четырех состояний имеется конфликт «перенос–свертка»:

$$\begin{array}{lll} E^3: & E \to E{+}E\cdot & \qquad E^4: \; E \to E{*}E\cdot \\ & E \to E\cdot{+}E & \qquad\quad\; E \to E\cdot{+}E \\ & E \to E\cdot{*}E & \qquad\quad\; E \to E\cdot{*}E \end{array}$$

Всего конфликтов четыре: в каждом из этих состояний при входном символе $+$ и при входном символе $*$. Любой метод построения LR-анализатора просто скопирует эти конфликты в таблицу ACTION, поскольку необходимой для их разрешения информации нет в грамматике. Однако эти конфликты легко разрешить вручную при помощи приоритетов и ассоциативности. Так, если в состоянии E^3 на вход подан символ $*$, то его нужно перенести в стек, так как его приоритет выше, чем приоритет уже перенесенного в стек «плюса». Если же на вход в этом состоянии подан символ $+$, то нужно выполнить свертку, так как из двух последовательных «плюсов» первым выполняется левый, т. е. уже находящийся в стеке. Аналогично показывается, что в состоянии E^4 оба конфликта должны быть решены в пользу свертки, в результате чего получается бесконфликтная таблица ACTION (табл. 4.5).

Полученный LR-анализатор не только меньше LR-анализатора, построенного по однозначной грамматике GA_2 (см. табл. 4.1), но и работает эффективнее. В самом деле, на цепочку `x+x*(x+x)`, протокол разбора которой приведен в табл. 4.2, он затратит 18 тактов вместо 23. Такое увеличение скорости связано с тем, что новый анализатор не тратит времени на свертки по цепным правилам $E \to T$ и $T \to F$, которые появились в грамматике арифметических выражений для обеспечения ее однозначности и отражения приоритета операторов.

Таблица 4.5. *Таблица LR-анализа для грамматики* GA_1

	ACTION						GOTO
	$+$	$*$	x	(	)	$\dashv$	E
E^1	$\leftarrow +$	$\leftarrow *$				✓	
(			$\leftarrow$x	$\leftarrow$(			E^2
x	$\otimes 4$	$\otimes 4$			$\otimes 4$	$\otimes 4$	
$+$			$\leftarrow$x	$\leftarrow$(			E^3
$*$			$\leftarrow$x	$\leftarrow$(			E^4
E^2	$\leftarrow +$	$\leftarrow *$			$\leftarrow$)		
E^3	$\otimes 1$	$\leftarrow *$			$\otimes 1$	$\otimes 1$	
E^4	$\otimes 2$	$\otimes 2$			$\otimes 2$	$\otimes 2$	
)	$\otimes 3$	$\otimes 3$			$\otimes 3$	$\otimes 3$	
∇			$\leftarrow$x	$\leftarrow$(			E^1

Пример 4.8. В большинстве языков программирования условные операторы имеют необязательную часть (`else`). Соответствующий фрагмент грамматики естественно записывать в виде

`<оператор>` → `if <условие> then <оператор>`
`<оператор>` → `if <условие> then <оператор> else <оператор>`

Однако это очевидно приводит к неоднозначности разбора цепочек со вложенными условными операторами, таких как

`if <условие> then if <условие> then <оператор> else <оператор>`

Общий принцип во всех языках одинаков: `else` всегда относится к ближайшему `if`. Этот принцип можно встроить в грамматику, это несложно, но существенно утяжеляет фрагмент грамматики, отвечающий за условные операторы – обычная плата за однозначность грамматики. Альтернативой является встраивание данного принципа непосредственно в LR-анализатор, при этом грамматику можно оставить неоднозначной. В самом деле, в состоянии LR(0)-автомата, содержащем пункты

[`<оператор>` → `if <условие> then <оператор>` ·]
[`<оператор>` → `if <условие> then <оператор>` ·`else <оператор>`]

будет конфликт «перенос–свертка». В соответствии с принципом использования `else` этот конфликт разрешается в пользу переноса.

§ 21. Обработка синтаксических ошибок

LR-анализатор обнаруживает ошибку во входной цепочке при обращении к пустой клетке таблицы ACTION (как уже отмечалось ранее, корректно работающий LR-анализатор в принципе не может обратиться к пустой клетке таблицы GOTO). Таким образом, в пустые клетки таблицы ACTION можно поместить ссылки на процедуры обработки соответствующих ошибок. Главным достоинством LR-анализатора в области обработки ошибок является быстрота реагирования: ошибка обнаруживается, как только очередной символ не может корректно продолжить уже прочитанную и перенесенную в стек часть входной цепочки. В частности, этот ошибочный символ не будет перенесен в стек.

Оснастим обработчиком ошибок LR-анализатор арифметических выражений, приведенный в табл. 4.5, и сравним эффективность обработки ошибок с результатами § 7 для нисходящего анализатора.

Как уже отмечалось в § 7, в выражениях из рассматриваемого нами языка LA возможны четыре вида синтаксических ошибок. Определим соответствующие процедуры.

Имя	Сообщение об ошибке	Действие
ERR1	«Отсутствует операнд»	добавить `x` в стек
ERR2	«Отсутствует оператор»	добавить + в стек
ERR3	«Преждевременная правая скобка»	удалить) из цепочки
ERR4	«Незакрытая левая скобка»	добавить) в стек

Отметим, что добавление состояния в стек позволяет сэкономить такт работы анализатора по сравнению со вставкой символа непосредственно во входной поток. Однако, чтобы добавить исправляющее ошибку состояние на вершину стека, нужно вначале произвести все свертки в стеке (иначе содержимое стека после исправления не будет активным префиксом). Поэтому в пустые клетки в тех строках таблицы, в которых имеются свертки, нужно поместить не процедуры обработки ошибок, а свертки, которые подготовят стек для исправления ошибки. Например, если в состоянии E^3 на входе анализатора оказывается символ `x`, то во входной цепочке, вероятно, пропущен оператор. Исправление этой ошибки имеет эффект вставки оператора + в текущую позицию входной цепочки. Так как действие анализатора на символе + в данном состоянии – это свертка по правилу 1, то нужно положить $action(E^3, \mathtt{x}) = \otimes 1$. Пропущенный оператор неизбежно будет обнаружен при попытке перенести `x`

в стек. В итоге, с процедурами обработки ошибок и дополнительными свертками, рассматриваемый LR-анализатор будет выглядеть как в табл. 4.6.

Таблица 4.6. *Таблица LR-анализа арифметических выражений с процедурами обработки ошибок*

	ACTION						GOTO
	$+$	$*$	`x`	(	)	$\dashv$	E
E^1	$\leftarrow+$	$\leftarrow*$	ERR2	ERR2	ERR3	✓	
(	ERR1	ERR1	$\leftarrow$`x`	$\leftarrow$(	ERR3	ERR1	E^2
`x`	$\otimes4$	$\otimes4$	$\otimes4$	$\otimes4$	$\otimes4$	$\otimes4$	
$+$	ERR1	ERR1	$\leftarrow$`x`	$\leftarrow$(	ERR3	ERR1	E^3
$*$	ERR1	ERR1	$\leftarrow$`x`	$\leftarrow$(	ERR3	ERR1	E^4
E^2	$\leftarrow+$	$\leftarrow*$	ERR2	ERR2	$\leftarrow$)	ERR4	
E^3	$\otimes1$	$\leftarrow*$	$\otimes1$	$\otimes1$	$\otimes1$	$\otimes1$	
E^4	$\otimes2$	$\otimes2$	$\otimes2$	$\otimes2$	$\otimes2$	$\otimes2$	
)	$\otimes3$	$\otimes3$	$\otimes3$	$\otimes3$	$\otimes3$	$\otimes3$	
∇	ERR1	ERR1	$\leftarrow$`x`	$\leftarrow$(	ERR3	ERR1	E^1

Рассмотрим цепочку `(x+x))(*x+x` , при обработке которой LL-анализатор не смог продвинуться дальше первой ошибки (см. § 7). LR-анализатор не встретит никаких затруднений – ключевые шаги разбора приведены в табл. 4.7, а восстановить весь разбор полностью мы предлагаем читателю.

Таблица 4.7. *Протокол обработки ошибок в «арифметической» цепочке*

Такт	Содержимое стека	Позиция указателя	Ошибка
9	$\nabla(E)$	`(x+x)`$\diamond$`)(*x+x`$\dashv$	
10	∇E	`(x+x)`$\diamond$`)(*x+x`$\dashv$	
11	∇E	`(x+x))`$\diamond$`(*x+x`$\dashv$	ERR3
12	$\nabla E+$	`(x+x))`$\diamond$`(*x+x`$\dashv$	ERR2
13	$\nabla E+($	`(x+x))(`$\diamond$`*x+x`$\dashv$	
14	$\nabla E+($`x`	`(x+x))(`$\diamond$`*x+x`$\dashv$	ERR1
...			
23	$\nabla E+(E$	`(x+x))(*x+x`$\diamond\dashv$	
24	$\nabla E+(E)$	`(x+x))(*x+x`$\diamond\dashv$	ERR4

§ 22. LR(k)-грамматики и языки

В этом, заключительном, параграфе второй части книги мы кратко обсудим некоторые общие вопросы, связанные с очень важным с практической точки зрения классом LR-грамматик (и соответствующим классом LR-языков). Начнем с основного определения.

Определение. Грамматика G называется *LR(k)-грамматикой*, если из того, что при правом выводе в ней некоторой r-формы $\alpha\beta u$ последним применялось правило $A \to \beta$, следует, что это же правило применялось последним при выводе *любой* r-формы $\alpha\beta v$ такой, что $\mathrm{FIRST}_k(v) = \mathrm{FIRST}_k(u)$. (Как обычно, здесь u и v – цепочки терминалов, α и β – произвольные цепочки.) Грамматика G называется *LR-грамматикой*, если она является *LR(k)-грамматикой* для некоторого k.

Другими словами, это определение означает, что анализатор типа перенос–свертка может однозначно определить основу и правило для свертки, зная содержимое стека и k первых символов входного потока (либо весь остаток входной цепочки, если он короче k символов). Тем самым данное определение в случае $k = 0$ эквивалентно определению LR(0)-грамматики из § 18, а в случае $k = 1$ – определению LR(1)-грамматики из § 19.

Для LR-анализа таких грамматик строится LR(k)-автомат, состояниями которого являются множества LR(k)-пунктов. Этот автомат распознает множество активных префиксов грамматики и может быть использован для построения бесконфликтной таблицы ACTION LR-анализатора. Такие анализаторы являются весьма громоздкими; заинтересованного читателя мы отсылаем к книге [АУ1].

Пусть $LR(k)$ обозначает класс всех LR(k)-грамматик, LR – класс всех LR-грамматик. Из определений легко следуют включения

$$LR(0) \subseteq LR(1) \subseteq \ldots \subseteq LR(k) \subseteq \ldots \subseteq LR.$$

Убедимся в том, что все включения являются строгими. Рассмотрим грамматику $G_{4,6} = \{S \to B\mathtt{a}^k\mathtt{b} \mid C\mathtt{a}^k\mathtt{c},\ B \to \mathtt{d},\ C \to \mathtt{d}\}$. При правом выводе цепочки $\mathtt{da}^k\mathtt{b}$ последним применялось правило $B \to \mathtt{d}$. Значит, в определении LR(k)-грамматики можно взять $\alpha = \varepsilon$, $\beta = \mathtt{d}$, $u = \mathtt{a}^k\mathtt{b}$. Заменив u на $v = \mathtt{a}^k\mathtt{c}$, получим $\mathrm{FIRST}_k(v) = \mathrm{FIRST}_k(u)$, но в правом выводе цепочки $\mathtt{da}^k\mathtt{c}$ последним применялось правило $C \to \mathtt{d}$. Таким образом, $G_{4,6} \notin LR(k)$. В то же время непосредственно проверяется, что $G_{4,6}$ является LR(k+1)-грамматикой.

Класс LR-грамматик достаточно широк и содержит в качестве собственного подкласса класс LL-грамматик (см. задачу 6). Тем не менее нетрудно привести пример однозначной грамматики, которая не является LR-грамматикой. Легко проверить, что любая цепочка, выводимая в грамматике $G_{4,7} = \{S \to B\mathtt{b} \mid C\mathtt{c},\ B \to B\mathtt{a} \mid \varepsilon,\ C \to C\mathtt{a} \mid \varepsilon\}$, представлена единственным деревом вывода. В то же время для любого k данная грамматика не принадлежит классу $LR(k)$. Действительно, при правом выводе цепочки $\mathtt{a}^k\mathtt{b}$ последним применялось правило $B \to \varepsilon$. Положим в определении LR(k)-грамматики $\alpha = \beta = \varepsilon$, $u = \mathtt{a}^k\mathtt{b}$ и заменим u на $v = \mathtt{a}^k\mathtt{c}$. Имеем $\textsc{first}_k(v) = \textsc{first}_k(u)$, но в правом выводе цепочки v последним применялось правило $C \to \varepsilon$, противоречие с рассматриваемым определением.

Перейдем к LR-языкам.

Определение. Язык называется *LR(k)-языком*, если существует LR(k)-грамматика, его порождающая.

Как отмечалось выше, класс LR(k+1)-грамматик строго содержит класс LR(k)-грамматик. В отличие от грамматик, в «иерархии» LR(k)-языков есть только две ступени: классы LR(0)- и LR(1)-языков, т. е. любая LR-грамматика порождает язык, который можно породить и при помощи LR(1)-грамматики. Справедливо даже более сильное и весьма неожиданное утверждение.

Теорема 4.2. *Любой язык, порождаемый LR-грамматикой, порождается некоторой SLR(1)-грамматикой.*

Доказательство этой нетривиальной теоремы можно найти в книге [АУ2]. Из теоремы следует, что SLR(1)-, LALR(1)-, LR(1)- и LR(k)-грамматики порождают один и тот же класс языков – класс *LR-языков*. Возникает естественный вопрос: как можно охарактеризовать класс LR-языков внутри класса всех контекстно-свободных языков?

Фактически LR-анализатор представляет собой детерминированный МП-автомат с одним состоянием, обладающий дополнительной возможностью: вытолкнуть из стека несколько символов за один такт и тут же выполнить команду, соответствующую символу на новой вершине стека. Такая возможность без труда моделируется в «обычном» ДМПА при помощи нескольких дополнительных состояний. В случае LR-анализатора это сделать даже проще, чем для анализатора ПП-грамматик, для которого подобная процедура описана в § 15. Отсюда можно сделать вывод, что все LR-языки распознаются

детерминированными МП-автоматами. На самом деле, это свойство является критерием, т. е. справедлива следующая теорема, с доказательством которой также можно ознакомиться по книге [АУ2].

Теорема 4.3. *Класс LR-языков совпадает с классом языков, распознаваемых детерминированными МП-автоматами.*

Этим фундаментальным результатом мы и заканчиваем вторую часть книги.

Задачи

1. Расширив грамматику G, построить ее LR(0)-автомат:
а) $G = \{S \to [L] \mid \mathtt{a},\ L \to L;S \mid S\}$;
б) $G = \{D \to L : T,\ L \to L;\mathtt{a} \mid \mathtt{a},\ T \to \mathtt{i}\}$;
в) $G = \{S \to SD \mid D,\ D \to [S] \mid [\ ]\}$.
Будет ли G LR(0)-грамматикой?

2. Доказать, что грамматика G является SLR(1)-грамматикой:
а) $G = G_{4,3}$ (см. пример 4.4 в § 18);
б) $G = \{S \to \mathtt{a}EF\mathtt{b},\ E \to \mathtt{c},\ F \to F\mathtt{b}E \mid \mathtt{d}\}$.

3. Построить LR(1)-автомат расширенной грамматики

$$G = \{S' \to S,\ S \to B\mathtt{d},\ B \to BC \mid \mathtt{e},\ C \to \mathtt{d}\}.$$

Как изменится этот автомат, если добавить правило вывода $C \to \mathtt{e}$?

4. Доказать, что грамматика

$$G = \{S \to A\mathtt{a} \mid \mathtt{b}A\mathtt{c} \mid \mathtt{dc} \mid \mathtt{bda},\ A \to \mathtt{d}\}$$

является LALR(1)-грамматикой, но не является SLR(1)-грамматикой.

5. Построить LR-распознаватель с обработчиком ошибок
а) для языка списков LL;
б) для языка двоичных чисел LN';
в) для языка формул логики высказываний.

6. Доказать, что любая LL-грамматика является LR-грамматикой.

Часть 3.

Семантический анализ в формализованных языках

В этой части в роли цепочек (строк) выступают – практически без исключений – тексты компьютерных программ, понимаемые в самом широком смысле слова. И сообщение в интернет-форуме с двумя HTML-тэгами, и формула =A1-(B3*2), введенная в ячейку Excel-таблицы, являются программами, требующими трансляции и последующего исполнения. Однако информации о программе, собранной на стадиях лексического и синтаксического анализа, заведомо недостаточно компилятору для преобразования исходного текста в работоспособный исполняемый код, и даже для заключения о том, что программа написана корректно с точки зрения правил языка.

Целей у семантического (или *контекстного*) анализа две. Во-первых, это проверка корректности программы на более глубоком уровне, чем это возможно при помощи контекстно-свободной грамматики, и, во-вторых, сбор и систематизация недостающей информации о программе. Вторая цель реализуется в *промежуточном представлении* программы, используемом компилятором для оптимизации ее работы и генерации исполняемого кода. Вопросы, ответы на которые могут потребоваться компилятору для дальнейшей работы, не исчерпываются следующим списком (который, впрочем, достаточно представителен).

- Является ли переменная `x` скаляром, массивом, записью, функцией? Объявлена ли она до использования? Если да, то внутри какой процедуры (функции, класса)? Какой у нее тип? Присвоено ли ей значение при объявлении?

- Для ссылки на элемент массива `y[i, j, k]`, действительно ли `y` объявлена как массив, причем с тем же числом индексов? Имеют ли переменные `i, j, k` допустимый для индексирования тип? Если им присвоены значения, то находится ли `y[i, j, k]` в объявленных границах?
- Где можно хранить значения `x` и `y`? В течение какого времени это нужно делать?
- Для ссылки `*z`, объявлена ли переменная `z` как указатель? Имеет ли объект `*z` соответствующий тип? Существуют ли другие указатели на значение `*z`?
- Сколько аргументов у функции `f`? Всегда ли она вызывается с корректным числом аргументов? Правильные ли типы имеют аргументы при вызове? Является ли `f` функцией *только* своих аргументов или она зависит, скажем, от значения какой-либо глобальной переменной?
- Какова вычислительная сложность данного блока программы (например, в тактах)[†]?

В первой главе части вводятся основные инструменты семантического анализа и описывается построение графов, часто используемых в промежуточном представлении программы. Во второй главе подробно – от более простых к более сложным ситуациям – рассматривается совместное проведение синтаксического и семантического анализа («однопроходная» трансляция). В конце главы описаны на примерах некоторые приемы «многопроходной» трансляции. Третья глава почти полностью посвящена наиболее важной и сложной группе семантических проверок, выполняемых компилятором, – проверке типов.

Глава 1. Атрибутные грамматики

В данной главе мы расширим понятие контекстно-свободной грамматики до понятия *атрибутной грамматики*, наделив грам-

[†]Это отнюдь не праздный вопрос: на стадии оптимизации кода часто бывает необходимо сравнить «стоимость», к примеру, постоянного хранения значения некоторого выражения и его повторного вычисления.

матические символы и правила вывода определенной смысловой нагрузкой. В результате каждому грамматическому выводу можно будет сопоставить конкретный вычислительный процесс над конкретными данными, тем самым связав синтаксическую и семантическую категории. Одну и ту же грамматику можно преобразовать в атрибутную грамматику многими способами в зависимости от того, какую именно семантическую информацию, содержащуюся в программе, мы хотим отследить.

§ 1. Атрибуты грамматических символов

Возьмем произвольную контекстно-свободную грамматику G и поставим в соответствие каждому грамматическому символу конечное (возможно пустое) множество параметров. Эти параметры будем называть *атрибутами*. Каждый из атрибутов имеет свою область изменения (*домен*). Каждому правилу вывода грамматики G сопоставим множество *семантических правил*, или *действий* – функциональных зависимостей вида $b := f(c_1, \ldots, c_k)$, где $b, c_1, \ldots, c_k$ – атрибуты грамматических символов, входящих в правило вывода. Полученный объект называется *атрибутной грамматикой*. Процесс *вывода* цепочки в атрибутной грамматике происходит так же, как и в контекстно-свободной грамматике, только при применении каждого правила вывода выполняются и все связанные с ним семантические правила. В результате вывода в атрибутной грамматике каждому узлу дерева вывода сопоставляется список значений атрибутов грамматического символа, находящегося в этом узле. Полученное дерево называют *аннотированным* деревом вывода (см. пример 1.1).

Атрибут a грамматического символа A мы будем всегда обозначать через $A.a$. Атрибуты терминальных символов не вычисляются; обычно в роли атрибута терминала выступает его *лексическое значение*, хранимое в таблице символов (см. ч. 2, гл. 1). Нетерминальный символ может иметь несколько атрибутов различных видов. В качестве значений атрибутов могут быть типы данных, числа, наборы битов, строки символов, поля строк таблиц, адреса памяти и т. д.

Пример 1.1. Возьмем грамматику GA_2, порождающую язык арифметических выражений LA. Напомним, что

$$GA_2 = \{E \to E{+}T,\ E \to T,\ T \to T{*}F,\ T \to F,\ F \to (E),\ F \to \mathtt{x}\}.$$

Введем атрибут *val* для символов E, T, F и используем его для

вычисления значения арифметического выражения. Значение суммы арифметических выражений равно сумме значений слагаемых; следовательно, правилу вывода $E \to E_1{+}T$ соответствует семантическое правило $E.val := E_1.val + T.val$. (Индексы введены, чтобы различать одинаковые символы в левой и правой части правила вывода.) Обратим внимание на то, что знак «+» в правиле вывода является просто символом, а тот же знак в семантическом правиле – операция сложения. Аналогичным образом правилу вывода $T \to T_1{*}F$ соответствует семантическое правило $T.val := T_1.val * F.val$.

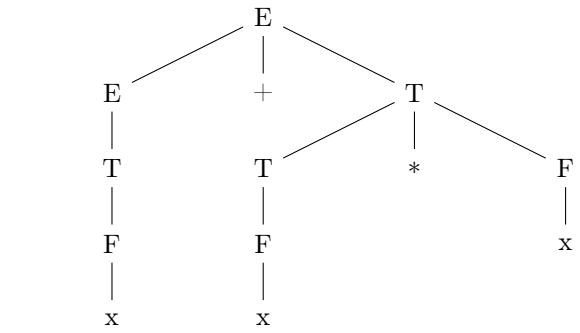
В таблице символов для терминала `x` (для простоты будем считать, что `x` – константа) указано лексическое значение $\texttt{x}.lexval$ (это и есть числовое значение константы). Полностью атрибутная грамматика (назовем ее AGA_2) содержится в табл. 1.1.

Таблица 1.1. *Атрибутная грамматика* AGA_2

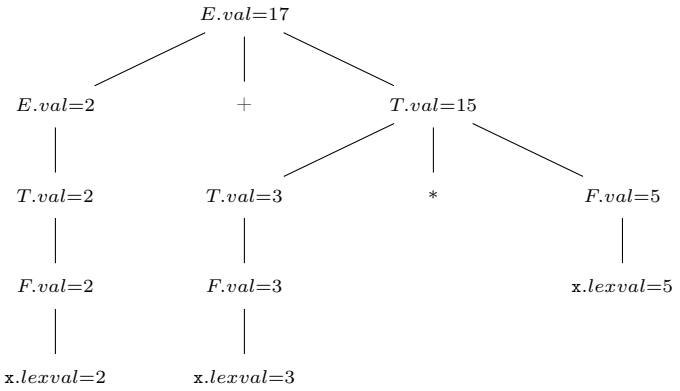
	Правило вывода	Семантические правила
1	$E \to E_1{+}T$	$E.val := E_1.val + T.val$
2	$E \to T$	$E.val := T.val$
3	$T \to T_1{*}F$	$T.val := T_1.val * F.val$
4	$T \to F$	$T.val := F.val$
5	$F \to (E)$	$F.val := E.val$
6	$F \to \texttt{x}$	$F.val := \texttt{x}.lexval$

Вычислим при помощи AGA_2 значение выражения $\texttt{2}+\texttt{5}*\texttt{3}$. Для этого построим дерево вывода цепочки $\texttt{x}+\texttt{x}*\texttt{x}$ (рис. 1.1, *а*) и соответствующее аннотированное дерево вывода (рис. 1.1, *б*).

Выделяют два основных класса атрибутов в зависимости от способа их вычисления. *Синтезируемый атрибут* $A.s$ символа A вычисляется при применении правила вывода вида $A \to \alpha$. Таким образом, значения синтезируемых атрибутов узла зависят от значений атрибутов сыновей данного узла в дереве вывода. *Наследуемый атрибут* $A.i$ символа A вычисляется при выполнении правила вывода вида $B \to \alpha A \beta$. Тем самым значения наследуемых атрибутов узла зависят от значений атрибутов родителя и/или братьев данного узла в дереве вывода. Использование обоих способов вычисления применительно к одному атрибуту ведет к проблемам при вычислении атрибутов, часто неразрешимым, и поэтому на практике не



а



б

Рис. 1.1. *Дерево вывода цепочки* `x + x * x` *(а); аннотированное дерево вывода той же цепочки (б)*

используется. Отметим, что в примере 1.1 использованы только синтезируемые атрибуты. Легко видеть, что они вычисляются по дереву вывода «снизу вверх». В то же время вычисление наследуемых атрибутов происходит при обходе дерева сверху вниз или при более сложном, комбинированном обходе.

Атрибутная грамматика в примере 1.2 использует наследуемый атрибут i символа L. Кроме того, этот пример иллюстрирует расширенную трактовку семантических правил, удобную для практическо-

го использования. А именно, *семантическое правило может быть произвольным фрагментом кода*, который следует выполнять при применении данного правила вывода. (Такой фрагмент кода обычно связан с атрибутами символов, входящих в правило вывода, например: «напечатать значение атрибута», «вычислить функцию атрибута и присвоить значение глобальной переменной», «внести данные в таблицу символов».) При этом можно считать, что такое семантическое правило вычисляет *фиктивный* синтезируемый атрибут нетерминала в левой части правила вывода[†].

Пример 1.2. Возьмем грамматику

$$GD = \{D \to TL,\ T \to \texttt{int},\ T \to \texttt{real},\ L \to L;\texttt{a},\ L \to \texttt{a}\},$$

порождающую язык описаний типов LD. В табл. 1.2 приведена атрибутная грамматика AGD для внесения типов переменных в таблицу символов. Атрибут *entry* является указателем на запись в таблице символов. На рис. 1.2 показано аннотированное дерево вывода цепочки `real a1; a2; a3` в этой грамматике. Отметим, что в трех L-узлах этого дерева вызывается процедура ADDTYPE, записывающая в таблицу символов тип $L.i$ для правого сына этого узла.

Таблица 1.2. *Атрибутная грамматика AGD*

	Правило вывода	Семантические правила
1	$D \to TL$	$L.i := T.type$
2	$T \to \texttt{int}$	$T.type := int$
3	$T \to \texttt{real}$	$T.type := real$
4	$L \to L_1;\texttt{a}$	$L_1.i := L.i$ ADDTYPE($\texttt{a}.entry, L.i$)
5	$L \to \texttt{a}$	ADDTYPE($\texttt{a}.entry, L.i$)

В следующем примере мы рассмотрим два существенно различных, но «семантически эквивалентных» способа преобразования грамматики двоичных чисел GN_1 в атрибутную грамматику, вычисляющую десятичное значение записанного двоичного числа.

[†]Атрибутные грамматики с такой расширенной трактовкой действий часто называют *синтаксически управляемыми определениями*.

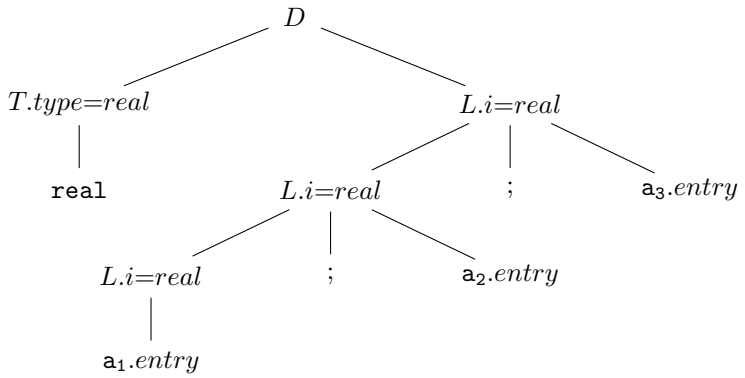


Рис. 1.2. *Аннотированное дерево вывода цепочки* `real a1; a2; a3`

Пример 1.3. Напомним, что множество всех строк над трехэлементным алфавитом $\{\texttt{0}, \texttt{1}, \texttt{.}\}$, допускающих интерпретацию в виде двоичных чисел (с возможными нулями в начале и в конце), порождается грамматикой $GN_1 = \{S \to L|.L|L.L,\ L \to LB|B,\ B \to \texttt{0}|\texttt{1}\}$. Для обозначения величины числа будем использовать синтезируемый атрибут v. Если бы мы ограничивались только целыми двоичными числами, то этого атрибута было бы достаточно (см. задачу 3 в конце главы). Если допускаются дроби, то одного атрибута недостаточно. Первый способ определения числового значения строки состоит во введении синтезируемого атрибута $L.l$ (длина строки). Тогда величина двоичного числа, получаемого по правилу $S \to L_1.L_2$, может быть вычислена как $S.v := L_1.v + L_2.v \cdot 2^{-L_2.l}$. В итоге грамматика GN_1 преобразуется в атрибутную грамматику AGN_1 так, как это сделано в табл. 1.3. Обратите внимание на то, что точка использована в двух смыслах – как разделитель целой и дробной части числа и как разделитель в имени атрибута.

Второй способ вычисления значения двоичного числа по его выводу в грамматике GN_1 состоит во введении наследуемого атрибута p, содержащего разряд самой правой цифры, выводимой из данного нетерминала. Для того чтобы одинаково обрабатывать и целую, и дробную часть, будем использовать функцию SN – один из вариантов функции «знак числа»: значение $\mathrm{SN}(x)$ равно -1, если x отрицательно, и 0 в противном случае. Полученная атрибутная грамматика AGN_2 приведена в табл. 1.4, а на рис. 1.3 изображено аннотированное дерево вывода цепочки 101.11.

Таблица 1.3. *Атрибутная грамматика* AGN_1

	Правило вывода	Семантические правила
1	$S \to L$	$S.v := L.v$
2	$S \to .L$	$S.v := L.v \cdot 2^{-L.l}$
3	$S \to L_1.L_2$	$S.v := L_1.v + L_2.v \cdot 2^{-L_2.l}$
4	$L \to L_1B$	$L.v := 2 \cdot L_1.v + B.v$ $L.l := L_1.l + 1$
5	$L \to B$	$L.v := B.v$ $L.l := 1$
6	$B \to 0$	$B.v := 0$
7	$B \to 1$	$B.v := 1$

Таблица 1.4. *Атрибутная грамматика* AGN_2

	Правило вывода	Семантические правила
1	$S \to L$	$S.v := L.v$ $L.p := 0$
2	$S \to .L$	$S.v := L.v$ $L_2.p := -1$
3	$S \to L_1.L_2$	$S.v := L_1.v + L_2.v$ $L_1.p := 0$ $L_2.p := -1$
4	$L \to L_1B$	$L.v := L_1.v + B.v$ $B.p := L.p + \text{SN}(L.p)$ $L_1.p := L.p + \text{SN}(L.p) + 1$
5	$L \to B$	$L.v := B.v$ $B.p := L.p$
6	$B \to 0$	$B.v := 0$
7	$B \to 1$	$B.v := 2^{B.p}$

Заключительный пример наглядно демонстрирует основное преимущество расширенной трактовки семантических действий: возможность беспроблемно отслеживать глобальные, а не только локальные зависимости атрибутов в дереве вывода.

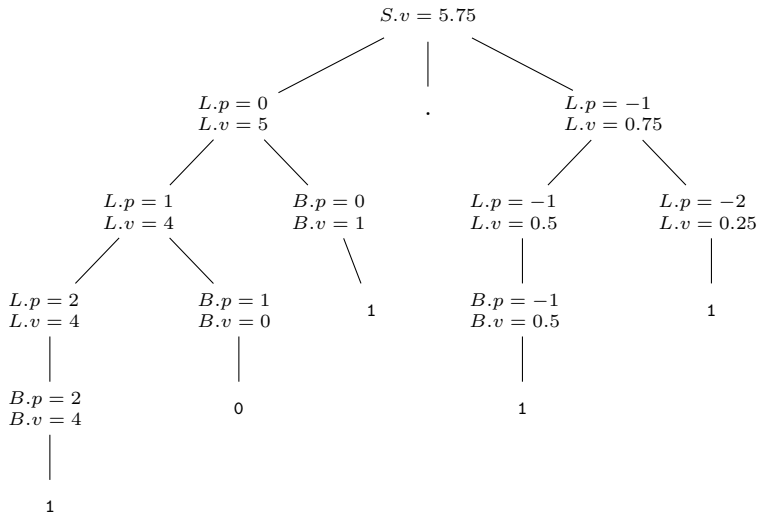


Рис. 1.3. *Аннотированное дерево вывода в грамматике* AGN_2

Пример 1.4. На основе грамматики арифметических выражений GA_2 построим атрибутную грамматику для определения вычислительной сложности выражения (считаем, что все символы `x` в выражении обозначают переменные). Каждая элементарная операция (чтение значения переменной из памяти в регистр и арифметическое действие) имеет свою стоимость (функция COST). Сложность вычисления всего выражения определяется суммой стоимостей элементарных операций. Поэтому можно ограничиться единственным синтезируемым атрибутом (обозначим его *.cost*), вычисление которого определяется простыми правилами (табл. 1.5).

Но тут имеются подводные камни. Вычисленное по такой атрибутной грамматике значение сложности часто будет неверным, поскольку не учитывает, что для вычисления выражения $a+a$ совсем не нужно читать значение a из памяти дважды. Следовательно, атрибуту *F.cost*, вычисляемому в правиле 6, нужно присваивать значение COST(*load*), только если значение переменной с именем `x`.*lexval* не перенесено в регистр ранее, иначе нужно присвоить $F.cost := 0$.

Эту проблему можно решить введением дополнительного атрибута у каждого нетерминала, значением которого является множество переменных, уже находящихся в регистрах. (На самом деле, потре-

Таблица 1.5. *«Наивное» вычисление сложности выражения*

	Правило вывода	Семантические правила
1	$E \to E_1 + T$	$E.cost := E_1.cost + T.cost + \text{COST}(add)$
2	$E \to T$	$E.cost := T.cost$
3	$T \to T_1 * F$	$T.cost := T_1.cost + F.cost + \text{COST}(mult)$
4	$T \to F$	$T.cost := F.cost$
5	$F \to (E)$	$F.cost := E.cost$
6	$F \to \texttt{x}$	$F.cost := \text{COST}(load)$

буются даже два атрибута – наследуемый и синтезируемый, см. задачу 4 к этой главе.) Такой подход имеет очевидные недостатки. Объем памяти, занимаемой аннотированным деревом, многократно возрастает; существенно усложняется и сама атрибутная грамматика, а ее реализация содержит огромное количество операций копирования нового атрибута, необходимых для донесения его значения в нужный узел дерева при помощи только локальных связей. Всех перечисленных недостатков можно избежать, если использовать для хранения информации о прочитанных переменных один внешний массив. Фактически можно использовать таблицу символов, заготовленную лексическим анализатором (ч. 2, гл. 1), добавив в нее дополнительное битовое поле *loaded*. Тогда для правильного вычисления значения сложности достаточно заменить последнюю строку табл. 1.5 строкой

6	$F \to \texttt{x}$	$F.cost := \text{COST}(load) \cdot \texttt{x}.entry.loaded$ $\texttt{x}.entry.loaded := 1$

(напомним, что атрибут $\texttt{x}.entry$ указывает на запись в таблице символов).

§ 2. Граф зависимости

Если значение атрибута $A.a$ в некотором узле дерева вывода определяется семантическим правилом $A.a := f(B.b, ...)$, то значение атрибута $B.b$ должно быть вычислено ранее. В этой ситуации мы будем говорить, что атрибут $A.a$ *зависит* от атрибута $B.b$. Зависимость между атрибутами грамматических символов в вершинах дерева вывода цепочки может быть представлена помеченным орграфом, называемым *графом зависимости цепочки*.

Граф зависимости цепочки w определяется следующим образом. *Каждому* атрибуту грамматического символа в *каждом* узле дерева вывода цепочки w сопоставляется узел графа зависимости. (Не следует забывать о фиктивных атрибутах символов, см. выше.) Из узла, помеченного экземпляром атрибута $B.b$, проводится дуга к узлу, помеченному экземпляром $A.a$, если соответствующие узлы с метками A и B в дереве вывода цепочки w связаны правилом вывода (вида $A \to \alpha B\beta$, $B \to \alpha A\beta$, $C \to \alpha A\beta B\gamma$ или $C \to \alpha B\beta A\gamma$), которому соответствует семантическое правило вида $A.a := f(B.b, ...)$.

Пример 1.5. На рис. 1.4 приведен граф зависимости для дерева вывода из примера 1.2 (ребра самого дерева изображены пунктиром). Узлы графа зависимости пронумерованы, рядом с номером указан атрибут, по которому построен узел. Дуга (4,5) отражает наследование атрибутом $L.i$ значения атрибута $T.type$ при применении правила $D \to TL$. Дуги (7,9) и (5,7) отражают применение семантического правила $L_1.i := L.i$ для правила вывода $L \to L_1;\mathtt{a}$. Узлы 6, 8 и 10 построены для фиктивных атрибутов, связанных с выполнением процедуры ADDTYPE($\mathtt{a}.entry, L.i$).

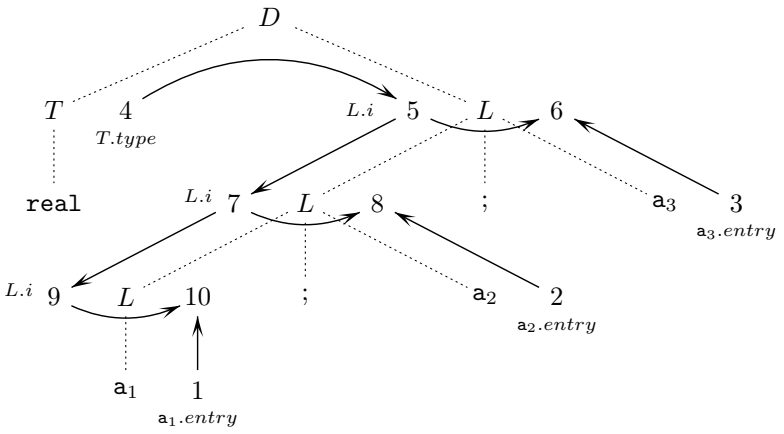


Рис. 1.4. *Граф зависимости для дерева вывода цепочки* `real a1; a2; a3`

Предложение 1.1. *Для того чтобы все атрибуты в аннотированном дереве вывода цепочки w могли быть вычислены, необходимо и достаточно, чтобы граф зависимости этой цепочки не содержал циклов.*

Доказательство. Необходимость очевидна: ни один из атрибутов, находящихся в цикле, нельзя вычислить. Докажем достаточность. Напомним, что узел v орграфа *достижим* из узла u, если существует путь из u в v. Отношение достижимости на ациклическом орграфе является отношением порядка. Известная теорема теории множеств гласит, что всякое отношение порядка можно дополнить до отношения линейного порядка. Этот линейный порядок на узлах графа зависимости (т. е. на экземплярах атрибутов) и задает последовательность вычисления атрибутов. □

Чтобы построить требуемое отношение линейного порядка, фактически нужно пронумеровать узлы ациклического орграфа так, чтобы любой путь в орграфе вел из вершины с меньшим номером в вершину с бóльшим номером. Такая нумерация называется *топологической сортировкой* орграфа. Граф зависимости на рис. 1.4 топологически отсортирован. Заметим, что топологическая сортировка обычно не единственна.

Атрибутная грамматика называется *циклической*, если граф зависимости дерева вывода некоторой цепочки в этой грамматике имеет цикл. Циклические грамматики нельзя применять на практике, поскольку некоторые синтаксически корректные входные строки являются семантически бессмысленными и не могут быть обработаны. Задача проверки атрибутной грамматики на цикличность алгоритмически разрешима, однако не для любой грамматики такая проверка может быть проведена за полиномиальное время (см. [АСУ]).

Есть три основных способа определения порядка выполнения семантических правил.

1. Анализ дерева вывода. Порядок выполнения семантических правил определяется в процессе компиляции конкретной строки с помощью топологической сортировки ее графа зависимости.

2. Анализ семантических правил при создании компилятора. Порядок вычисления атрибутов, связанных с каждым правилом вывода, фиксируется в процессе разработки компилятора.

3. Игнорирование природы конкретных семантических правил. Семантический анализ совмещается с синтаксическим и порядок выполнения семантических правил определяется методом синтаксического анализа.

Первый способ применим всегда, когда данный граф зависимости ацикличен (предложение 1.1). В то же время он наиболее трудоемок, так как требует, помимо прохода входной строки для построения дерева вывода, еще и два обхода этого дерева – для сортировки и для вычисления атрибутов. Второй способ занимает промежуточное положение между первым и последним как по трудоемкости, так и по рамкам применимости. Последний способ наиболее быстр (требуется один-единственный проход входной строки), но существенно ограничивает класс реализуемых грамматик, который, впрочем, остается достаточно широк. Этот способ очень часто применяется в компиляторах, ему посвящена почти вся следующая глава.

§ 3. Синтаксическое дерево и синтаксический даг

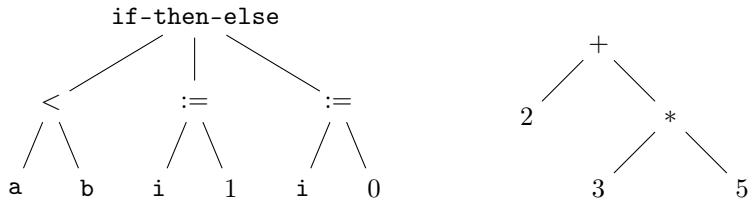
В качестве промежуточного представления программы удобно использовать «сжатое» дерево вывода, называемое *синтаксическим деревом*. Синтаксические деревья используются, в частности, для представления объявлений (см. гл. 3, § 10).

Для построения синтаксического дерева требуется, чтобы исходная грамматика была операторной. Это вполне разумное ограничение, так как каждую грамматику можно преобразовать в эквивалентную операторную (см. задачу 9 к ч. 1, гл. 4). Синтаксическое дерево получается из дерева вывода применением (до тех пор, пока это возможно) следующих двух преобразований. Первое: листья, соответствующие терминалам-операторам (и ключевым словам), удаляются, а данные терминалы связываются с родительским узлом. Второе: ветви дерева, соответствующие цепным правилам, сворачиваются в одну вершину. В результате все узлы преобразованного дерева помечены терминалами (листья – операндами, а внутренние узлы – операторами и другими аналогичными по смыслу языковыми конструкциями). Атрибуты при семантическом анализе присоединяются к узлам дерева, как и в случае дерева вывода.

Пример 1.6. Строка «`if a<b then i:=1 else i:=0`» языка Паскаль представляется синтаксическим деревом на рис. 1.5, *а*.

Пример 1.7. Дерево вывода из примера 1.1 превращается в синтаксическое дерево на рис. 1.5, *б*.

Построение синтаксических деревьев для данной грамматики удобно производить, дополнив ее до атрибутной грамматики такой,



а *б*

Рис. 1.5. *Синтаксическое дерево для строки Паскаля (а) и арифметического выражения (б)*

что атрибутами грамматических символов являются указатели на узлы синтаксического дерева. Приведем пример построения синтаксического дерева. Будем считать, что функция MKLEAF(a, $entry$) создает лист, помеченный операндом a и содержащий значение атрибута $entry$ – указателя на запись в таблице символов. Функция MKNODE($op, left, right$) создает узел операции, помеченный оператором op и содержащий указатели $left$ и $right$ на узлы операндов.

Пример 1.8. Рассмотрим еще раз грамматику

$$GA_2 = \{E \to E{+}T,\ E \to T,\ T \to T{*}F,\ T \to F,\ F \to (E),\ F \to \mathtt{x}\}.$$

Приводимая в табл. 1.6 атрибутная грамматика $STGA_2$ строит синтаксические деревья для арифметических выражений. Синтезируемый атрибут ptr у нетерминалов используется для хранения указателя на узел синтаксического дерева, соответствующий данному нетерминалу.

Таблица 1.6. *Атрибутная грамматика $STGA_2$*

	Правило вывода	Семантические правила
1	$E \to E_1{+}T$	$E.ptr :=$ MKNODE$(+, E_1.ptr, T.ptr)$
2	$E \to T$	$E.ptr := T.ptr$
3	$T \to T_1{*}F$	$T.ptr :=$ MKNODE$(*, T_1.ptr, F.ptr)$
4	$T \to F$	$T.ptr := F.ptr$
5	$F \to (E)$	$F.ptr := E.ptr$
6	$F \to \mathtt{x}$	$F.ptr :=$ MKLEAF$(\mathtt{x}, \mathtt{x}.entry)$

Синтаксический даг (от англ. dag – Directed Acyclic Graph) – это граф, получаемый из синтаксического дерева отождествлением его равных (изоморфных) поддеревьев[†]. Даг в той же мере, как и синтаксическое дерево, представляет синтаксическую структуру анализируемой строки. При этом даг гораздо компактнее. Например, если переменная используется в программе 10 раз, то в синтаксическом дереве ей соответствует 10 листьев, а в даге – только один. Кроме того, даг демонстрирует некоторые возможности оптимизации кода программы, которые могут быть использованы компилятором (многократное использование один раз вычисленного значения выражения).

Пример 1.9. Арифметическое выражение $(\mathtt{c} + \mathtt{a} * \mathtt{b}) * (\mathtt{a} * \mathtt{b} + \mathtt{a})$ представляется дагом на рис. 1.6.

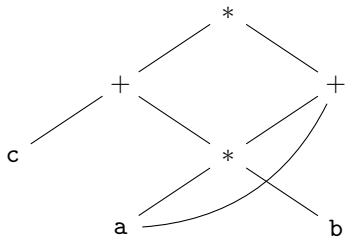


Рис.1.6. *Даг для арифметического выражения*

Очевидно, что за пространственную эффективность дага нужно чем-то платить. Вполне естественно, что платить приходится временно́й сложностью построения. Атрибутная грамматика для построения синтаксического дерева будет строить даг, если внести соответствующие изменения в функции MKLEAF() и MKNODE(). А именно, достаточно потребовать, чтобы такая функция проверяла существование идентичного узла; если такой узел есть, функция возвращает указатель на него, если идентичного узла нет – строит новый. Таким образом, скорость построения синтаксического дага анализируемой строки зависит от эффективности реализации процедуры проверки существования узла с заданными параметрами.

[†]Даг наследует от дерева вывода и синтаксического дерева свойство упорядоченности: любое множество братьев линейно упорядочено «слева направо».

Данная процедура может быть эффективно реализована при хранении построенных узлов в виде *хэш-таблицы* (см. подробно [Кнут]). Идея состоит в том, что все построенные узлы хранятся не в виде одного списка, а в виде «массива списков», т. е. набора списков и массива *заголовков*-указателей на начала этих списков. Эффективный доступ к этой структуре данных обеспечивает *хэш-функция*, т. е. функция, которая по произвольной тройке $(op, left, right)$ однозначно (и быстро) вычисляет элемент массива заголовков. Далее проверка существования узла состоит в просмотре списка с данным заголовком на предмет наличия этого узла. Если узел существует, то возвращается указатель на него, если не существует – то такой узел создается, возвращается указатель на него, а сам созданный узел добавляется в конец просмотренного списка.

Задачи

1. Построить аннотированное дерево вывода цепочки $(2 + 3) * 5 + 1$ в атрибутной грамматике AGA_2.

2*. Написать атрибутную грамматику, которая описывает построение нормального ε-НКА по регулярному выражению (см. ч. 1, § 6).

3. Написать атрибутную грамматику, которая вычисляет значения целых двоичных чисел без знака, используя только один атрибут.

4*. Написать атрибутную грамматику, которая оценивает вычислительную сложность выражения (см. пример 1.4), не используя внешних структур данных. *Указание: для каждого нетерминала ввести два атрибута, содержащих списки переменных, прочитанных к моментам «до» и «после» обработки подвыражения, представимого этим нетерминалом.*

5. Построить граф зависимости и выполнить топологическую сортировку для строки

а) `10.1101` в атрибутной грамматике AGN_2 (табл. 1.4);

б) `x * y * (x + y)` в атрибутной грамматике для вычислительной сложности (пример 1.4, «улучшенный» вариант).

6. Построить синтаксическое дерево и даг для фрагмента программы на Паскале.

```
while i<n do
      if 2*i>n then begin a[i]:=2*i; i:=i+1 end
               else begin a[i]:=i; i:=2*i end;
```

Глава 2. Семантический анализ в процессе синтаксического разбора

В этой главе подробно разбираются методы вычисления атрибутов в процессе разбора строки LL- и LR-анализаторами (см. ч. 2, гл. 2, 4). В последнем параграфе рассматриваются некоторые приемы «многопроходной» трансляции, когда не все атрибуты могут быть вычислены одновременно с проведением синтаксического разбора строки.

§ 4. S-атрибутные грамматики. Восходящий анализ

S-атрибутной называется атрибутная грамматика, имеющая только синтезируемые атрибуты. Естественный порядок вычисления таких атрибутов – снизу вверх по дереву вывода – совпадает с порядком построения дерева вывода при восходящем анализе. Мы покажем, как LR-анализатор может хранить значения атрибутов и производить над ними вычисления непосредственно в стеке.

Для начала напомним, что в стеке LR-анализатора хранятся не грамматические символы, а *состояния*, содержащие информацию об уже разобранных поддеревьях дерева вывода. При этом символ однозначно определяется состоянием, поэтому в примерах мы часто подменяем состояние символом. Для семантического анализа элемент стека делают записью, имеющей синтаксическое поле *state* (состояние) и семантическое поле *val* (значение синтезируемого атрибута соответствующего символа; если атрибутов несколько, то *val* рассматривается как вектор). Через $state[top]$ ($val[top]$) мы будем обозначать соответствующие значения для верхней записи в стеке; через $state[top{-}i]$ ($val[top{-}i]$) – значения для $(i{+}1)$-й сверху записи.

Шаг работы анализатора начинается с вычисления значения указателя на новую вершину стека (*ntop*). Опуская детали реализации, мы можем написать, что при переносе выполняется присваивание $ntop \leftarrow top{+}1$, а при свертке по правилу вывода с r символами в правой части – присваивание $ntop \leftarrow top{-}r{+}1$. Заканчивается шаг присваиванием $top \leftarrow ntop$. Если выполняется перенос терминала `b`, то анализатор присвоит $state[top] \leftarrow$ `b` и $val[top] \leftarrow$ `b`.*lexval*. Пусть перед данным шагом стек выглядит следующим образом:

$state$	val	
Z	$Z.z$	$\leftarrow top$
Y	$Y.y$	
X	$X.x$	
$\ldots$	$\ldots$	

и на шаге выполняется свертка по правилу $A \to XYZ$ с вычислением синтезируемого атрибута $A.a = f(X.x, Y.y, Z.z)$. Тогда анализатор присвоит

$$state[ntop] \leftarrow A, \quad val[ntop] \leftarrow f(val[top{-}2], val[top{-}1], val[top])$$

и выполнит какие-то обусловленные конкретной реализацией действия (например, очистит освободившуюся память).

Пример 2.1. Возьмем атрибутную грамматику AGA_2 для языка арифметических выражений LA (пример 1.1, табл. 1.1) и добавим к ней внешнюю аксиому E', чтобы можно было применить LR-анализ (ср. ч. 2, гл. 4). Дополнительному правилу вывода $E' \to E$ сопоставим семантическое действие PRINT$(E.val)$. В приводимой ниже табл. 2.1 семантические действия из табл. 1.1 приведены в виде фрагментов кода, выполняемых LR-анализатором *перед* соответствующей сверткой. Пустые клетки в таблице означают, что никаких действий производить не требуется, т. е. $ntop = top$ и значение $val[ntop]$ уже находится в поле $val[top]$.

Таблица 2.1. *Реализация «калькулятора» с помощью LR-анализатора*

	Правило вывода	Фрагмент кода
1	$E' \to ET$	PRINT$(val[top])$
2	$E \to E_1 + T$	$val[ntop] \leftarrow val[top{-}2] + val[top]$
3	$E \to T$	
4	$T \to T_1 * F$	$val[ntop] \leftarrow val[top{-}2] * val[top]$
5	$T \to F$	
6	$F \to (E)$	$val[ntop] \leftarrow val[top{-}1]$
7	$F \to$ `x`	

Как будет показано в следующем параграфе, атрибуты в S-атрибутных грамматиках могут быть вычислены и при нисходящем анализе (с использованием обхода дерева вывода в глубину).

§ 5. L-атрибутные грамматики. Нисходящий анализ. Схемы трансляции

Атрибутная грамматика называется *L-атрибутной*, если для любого правила вывода $A \to X_1 \dots X_n$ и любого $i = 1, \dots, n$ каждый наследуемый атрибут символа X_i из правой части зависит только от атрибутов символов $X_1, \dots, X_{i-1}$ и наследуемых атрибутов символа A. В частности, любая S-атрибутная грамматика является L-атрибутной.

Обход в глубину дерева вывода назовем *стандартным*, если сыновья любого узла просматриваются слева направо (т. е. если в данном узле применялось правило $A \to X_1 X_2 \dots X_n$, то его сыновья, помеченные $X_1, X_2, \dots, X_n$, просматриваются именно в этом порядке). Отметим, что такой порядок просмотра является и порядком создания узлов дерева при нисходящем анализе (см. ч. 2, гл. 2).

Предложение 2.1. *Приведенная атрибутная грамматика является L-атрибутной тогда и только тогда, когда для любой синтаксически корректной цепочки все атрибуты грамматических символов могут быть вычислены стандартным обходом в глубину дерева вывода.*

Доказательство. Вычисление атрибутов по дереву вывода строки в L-атрибутной грамматике может быть произведено применением к корню дерева рекурсивной процедуры DFVISIT, реализующей стандартный обход в глубину:

DFVISIT (узел n)
1. **для каждого** `сына` m `узла` n
2. `вычислить наследуемые атрибуты узла` m
3. DFVISIT(m)
4. `вычислить синтезируемые атрибуты узла` n

Чтобы доказать обратное утверждение, напомним, что каждый узел дерева при обходе в глубину посещается дважды: при первом посещении он заносится в стек, а при втором – по завершении обхода всех своих потомков – удаляется. Отсюда следует, что синтезируемые атрибуты узла можно вычислить только при втором посещении (к этому моменту должны быть известны значения атрибутов его сыновей). Кроме того, узлы-братья не находятся в стеке одновременно. При стандартном обходе к моменту первого посещения узла известны атрибуты всех его «левых» братьев, а «правые» братья впервые будут посещены уже после второго посещения узла. Таким

образом, атрибуты узла не могут зависеть от атрибутов правых братьев. Нетрудно заметить, что если наследуемые атрибуты вычислять при втором посещении узла, то они не могут зависеть от атрибутов родителя вообще. А если это вычисление проводить при первом посещении, то наследуемые атрибуты могут зависеть от наследуемых атрибутов родителя. Таким образом, в каждом узле должны реализовываться семантические правила L-атрибутной грамматики. Поскольку каждое правило вывода приведенной грамматики реализуется в некотором узле некоторого дерева вывода, анализируемая грамматика должна быть L-атрибутной. □

Если семантический анализ строки производится одновременно с синтаксическим, то удобно использовать вариант атрибутной грамматики, называемый *схемой трансляции*. В схеме трансляции семантические действия, сопоставленные данному правилу вывода, «встроены» в правую часть правила, которое приобретает вид

$$\begin{aligned} A \to\ & [\texttt{действие}_1]X_1[\texttt{действие}_2]X_2 \ldots \\ & \ldots [\texttt{действие}_n]X_n[\texttt{действие}_{n+1}] \end{aligned} \tag{2.1}$$

(некоторые из действий могут быть пустыми). Таким образом, схема трансляции явно задает порядок выполнения семантических действий компилятором при разборе строки. Так, нисходящий анализатор при развертке правила (2.1) в узле дерева вывода будет обрабатывать правую часть посимвольно слева направо: действия – выполнять, а узлы – создавать и рекурсивно вызывать для них процедуру развертки. Следовательно, схема трансляции должна быть построена так, чтобы значения атрибутов, участвующих в действии, были доступны к моменту выполнения действия. Три приводимых ниже ограничения гарантируют доступность значений атрибутов; эти ограничения естественно вытекают из определения L-атрибутной грамматики.

1. Наследуемые атрибуты X_i вычисляются в i-м действии.
2. i-ое действие не должно обращаться к синтезируемым атрибутам символа X_i и к атрибутам символов X_j, где $j > i$.
3. Синтезируемые атрибуты A вычисляются в $(n{+}1)$-м действии.

В следующем примере описан в упрощенной форме фрагмент атрибутной грамматики TEX′a, вычисляющий «физическую» высоту формулы, содержащей нижние индексы (возможно, вложенные).

Пример 2.2. TEX рассматривает формулу (либо строку текста) как последовательность блоков. Блоком в данном примере будет либо символ, либо последовательность блоков, заключенная в фигурные скобки. Блоки размещаются либо друг за другом, либо в качестве нижних индексов других блоков (символ подчеркивания является «оператором» нижнего индекса). К примеру, строка `z_zz` транслируется в $\mathtt{z}_{\mathtt{z}}\mathtt{z}$, строка `z_z_z` – в $\mathtt{z}_{\mathtt{zz}}$[†], а строка `z_{z_z}` – в $\mathtt{z}_{\mathtt{z}_{\mathtt{z}}}$. Блоки ($B$) и последовательности блоков (A) имеют синтезируемый атрибут *.ht* (height) – высоту занимаемого на странице пространства, и наследуемый атрибут *.ps* (pointsize) – размер шрифта, или *кегль*. Атрибут $\mathtt{sym}.h$ означает относительную высоту конкретного символа (определяется по таблице символов). В табл. 2.2 приведена соответствующая атрибутная грамматика *AGT*.

Таблица 2.2. *Атрибутная грамматика AGT*

	Правило вывода	Семантические правила
1	$S \to A$	$A.ps := 10$ $S.ht := A.ht$
2	$A \to A_1 B$	$A_1.ps := A.ps$ $B.ps := A.ps$ $A.ht := \text{MAX}(A_1.ht, B.ht)$
3	$A \to B$	$B.ps := A.ps$ $A.ht := B.ht$
4	$B \to B_1 \_ B_2$	$B_1.ps := B.ps$ $B_2.ps := \text{SHRINK}(B.ps)$ $B.ht := \text{DISPOSE}(B_1.ht, B_2.ht)$
5	$B \to \{A\}$	$A.ps := B.ps$ $B.ht := A.ht$
6	$B \to \mathtt{sym}$	$B.ht := \mathtt{sym}.h * B.ps$

Базовый кегль в примере выбран равным 10. Функция SHRINK вычисляет по кеглю текста кегль его нижнего индекса (индексирование может быть многоступенчатым). Функция DISPOSE вычисляет высоту текста с индексом по высотам текста и индекса. Данная грам-

[†]В современных версиях TEX строка `z_z_z` считается ошибочной. В этом есть смысл, так как пользователь, который ввел подобную строку, вряд ли имел в виду именно такой результат верстки, для достижения которого гораздо естественнее ввести `z_{zz}`.

матика является L-атрибутной, так как единственный наследуемый атрибут *.ps* зависит только от наследуемого атрибута родителя. Ниже приводится соответствующая схема трансляции TGT:

S	$\to$		$[A.ps := 10]$	A	$[S.ht := A.ht]$
A	$\to$		$[A_1.ps := A.ps]$	A_1	$[B.ps := A.ps]$
				B	$[A.ht := \text{MAX}(A_1.ht, B.ht)]$
A	$\to$		$[B.ps := A.ps]$	B	$[A.ht := B.ht]$
B	$\to$		$[B_1.ps := B.ps]$	B_1 _	$[B_2.ps := \text{SHRINK}(B.ps)]$
				B_2	$[B.ht := \text{DISPOSE}(B_1.ht, B_2.ht)]$
B	$\to$	{	$[A.ps := B.ps]$	A }	$[B.ht := A.ht]$
B	$\to$			`sym`	$[B.ht := \texttt{sym}.h * B.ps]$

§ 6. Трансляция при нисходящем анализе

Схему трансляции из примера 2.2 нельзя реализовать при нисходящем анализе, так как она использует леворекурсивную (и не левофакторизованную) грамматику. Данный раздел мы начнем с рассмотрения проблемы устранения левой рекурсии из схемы трансляции. Заметим, что в общем случае левую рекурсию из схемы трансляции с наследуемыми атрибутами устранить нельзя. Однако мы сейчас покажем, как устранить непосредственную левую рекурсию из любой S-атрибутной схемы трансляции, получая в результате L-атрибутную схему.

В основе данного преобразования лежит процедура устранения непосредственной левой рекурсии из грамматики (см. ч. 1, § 22). Напомним, что если набор альтернатив нетерминала A выглядит как

$$A \to A\alpha_1 \mid \ldots \mid A\alpha_k \mid \beta_1 \mid \ldots \mid \beta_m,$$

где строки $\beta_1, \ldots, \beta_m$ не начинаются с A, то при устранении непосредственной левой рекурсии мы получаем фрагмент грамматики

$$\begin{aligned} A &\to \beta_1 R \mid \ldots \mid \beta_m R \\ R &\to \alpha_1 R \mid \ldots \mid \alpha_k R \mid \varepsilon \end{aligned} \tag{2.2}$$

Поскольку схема трансляции является S-атрибутной, действия выполняются только в конце правил (см. предыдущий параграф). Поэтому для упрощения записи мы без ограничения общности заменим строки α_i и β_j нетерминалами Y^i и X^j с синтезируемыми атрибутами $Y^i.y$ и $X^j.x$ соответственно. Тогда рассматриваемый фрагмент

леворекурсивной схемы трансляции будет иметь вид

$$\begin{array}{lll} A \to A_1 Y^1 & [A.a = g_1(A_1.a, Y^1.y)] \\ \quad \ldots \\ A \to A_1 Y^k & [A.a = g_k(A_1.a, Y^k.y)] \\ A \to X^1 & [A.a = f_1(X^1.x)] \\ \quad \ldots \\ A \to X^m & [A.a = f_m(X^m.x)] \end{array} \tag{2.3}$$

Введем дополнительный нетерминал R с наследуемым атрибутом $R.i$ и синтезируемым атрибутом $R.s$ и заменим фрагмент (2.3) следующим фрагментом:

$$\begin{array}{llll} A \to X^1 & [R.i = f_1(X^1.x)] & R & [A.a = R.s] \\ \quad \ldots \\ A \to X^m & [R.i = f_m(X^m.x)] & R & [A.a = R.s] \\ R \to Y^1 & [R_1.i = g_1(R.i, Y^1.y)] & R_1 & [R.s = R_1.s] \\ \quad \ldots \\ R \to Y^k & [R_1.i = g_k(R.i, Y^k.y)] & R_1 & [R.s = R_1.s] \\ R \to \varepsilon & [R.s = R.i] \end{array} \tag{2.4}$$

Фрагмент грамматики, лежащий в основе (2.4), является частным случаем (2.2). Таким образом, из результатов ч. 1, гл. 4 следует, что в (2.3) и (2.4) выводится одно и то же множество строк. Таким образом, чтобы обосновать предложенный способ устранения левой рекурсии, достаточно доказать, что эти два фрагмента совпадают семантически; это сделано в следующем предложении.

Предложение 2.2. *Для любой строки, выводимой из нетерминала A, значение атрибута $A.a$, вычисленное по схеме (2.3), равно значению $A.a$, вычисленному по схеме (2.4).*

Доказательство. Строка, выводимая в (2.3) и (2.4), представляет собой последовательность «игреков», которой предшествует ровно один «икс». На рис. 2.1 приведены деревья вывода одной и той же строки в (2.3) и (2.4) без указания значений атрибутов.

Рассмотрим вначале дерево вывода в (2.3). Атрибуты символов A в нем вычисляются снизу вверх, и мы получаем последовательно

$$\begin{array}{rl} A_t.a = & f_j(X^j.x), \\ A_{t-1}.a = & g_{i_1}(f_j(X^j.x), Y^{i_1}.y), \\ \ldots \quad & \ldots \\ A.a = & g_{i_t}(g_{i_{t-1}}(\ldots g_{i_1}(f_j(X^j.x), Y^{i_1}.y)\ldots), Y^{i_t}.y). \end{array}$$

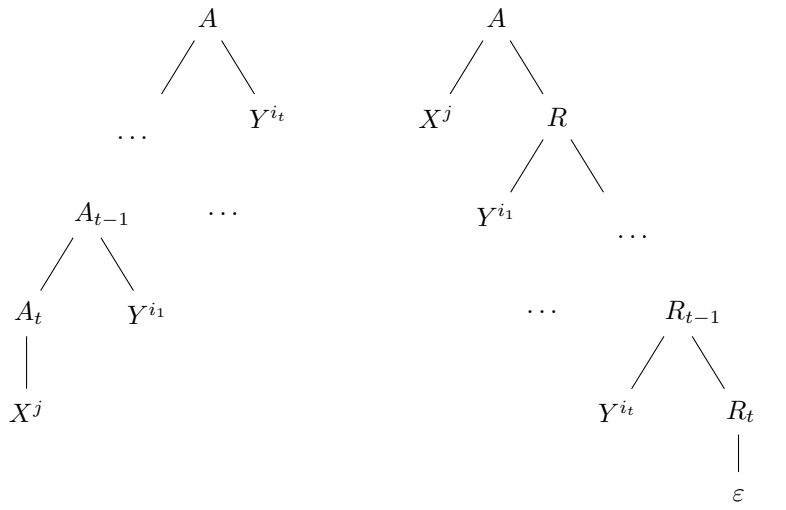


Рис. 2.1. *Деревья вывода строки $X^j Y^{i_1} \dots Y^{i_t}$ из нетерминала A в схемах (2.3) (слева) и (2.4) (справа)*

Теперь изучим дерево вывода в (2.4). Наследуемые атрибуты символов R вычисляются в нем сверху вниз:

$$\begin{aligned} R.i &= f_j(X^j.x), \\ R_1.i &= g_{i_1}(f_j(X^j.x), Y^{i_1}.y), \\ \dots &\quad \dots \\ R_t.i &= g_{i_t}(g_{i_{t-1}}(\dots g_{i_1}(f_j(X^j.x), Y^{i_1}.y)\dots), Y^{i_t}.y). \end{aligned}$$

При развертке правила $R_t \to \varepsilon$ выполняется присвоение $R_t.s = R_t.i$, после чего синтезируемый атрибут символа R копируется вверх по дереву без изменений. В итоге имеем $A.a = R_t.i$; предложение доказано. □

Пример 2.3. Рассмотрим схему трансляции TGA_2 на основе леворекурсивной грамматики GA_2 (см. пример 1.1):

$$\begin{aligned} E &\to E_1 + T \ [E.val := E_1.val + T.val] \\ E &\to T \ [E.val := T.val] \\ T &\to T_1 * F \ [T.val := T_1.val * F.val] \\ T &\to F \ [T.val := F.val] \\ F &\to (E) \ [F.val := E.val] \\ F &\to \mathtt{x} \ [F.val := \mathtt{x}.lexval] \end{aligned}$$

Устраним левую рекурсию, получая схему трансляции TGA_3:

$$
\begin{array}{lcrl}
E & \to & T & [R.i := T.val]\ E'\ [E.val := E'.s] \\
E' & \to & +T & [E'_1.i := E'.i + T.val]\ E'_1\ [E'.s := E'_1.s] \\
E' & \to & \varepsilon & [E'.s := E'.i] \\
T & \to & F & [Q.i := F.val]\ T'\ [T.val := T'.s] \\
T' & \to & *F & [T'_1.i := T'.i * F.val]\ T'_1\ [T'.s := T'_1.s] \\
T' & \to & \varepsilon & [T'.s := T'.i] \\
F & \to & (E) & [F.val := E.val] \\
F & \to & \mathtt{x} & [F.val := \mathtt{x}.lexval]
\end{array}
$$

Алгоритм устранения глобальной левой рекурсии из S-атрибутной схемы трансляции можно построить на основе алгоритма 4.6 из ч. 1, гл. 4, используя приведенное выше преобразование для устранения непосредственной левой рекурсии. Отметим, что рассмотренное преобразование несложно модернизируется и на случай некоторых L-атрибутных схем трансляции, а именно тех, в которых наследование атрибутов происходит только от родителя. Мы проделаем такое преобразование для схемы трансляции TGT из примера 2.2.

Пример 2.4. Устраним левую рекурсию из схемы трансляции TGT, введя дополнительные нетерминалы R и Q, каждый из которых имеет два наследуемых атрибута *.ps* и *.i*, а также синтезируемый атрибут *.s*.

$$
\begin{array}{llll}
S \to & [A.ps := 10] & A & [S.ht := A.ht] \\
A \to & [B.ps := A.ps] & B & [R.ps := A.ps,\ R.i := B.ht] \\
 & & R & [A.ht := R.s] \\
R \to & [B.ps := R.ps] & B & [R_1.i := \textsc{max}(R.i, B.ht), \\
 & & & \ R_1.ps := R.ps] \\
 & & R_1 & [R.s := R_1.s] \\
R \to \varepsilon & [R.s := R.i] & & \\
B \to \{ & [A.ps := B.ps] & A\,\} & [Q.i := A.ht,\ Q.ps := B.ps] \\
 & & Q & [B.ht := Q.s] \\
B \to & & \mathtt{sym} & [Q.i := \mathtt{sym}.h * B.ps, \\
 & & & \ Q.ps := B.ps] \\
 & & Q & [B.ht := Q.s] \\
Q \to \_ & [B.ps := \textsc{shrink}(Q.ps)] & B & [Q_1.i := \textsc{dispose}(Q.i, B.ht), \\
 & & & \ Q_1.ps := Q.ps] \\
 & & Q_1 & [Q.s := Q_1.s] \\
Q \to \varepsilon & [Q.s := Q.i] & &
\end{array}
$$

Если построенная схема трансляции пригодна для нисходящего анализа, то можно сконструировать нисходящий *транслятор* – программу, осуществляющую нисходящий синтаксический анализ параллельно с вычислением атрибутов, в соответствии с правилами данной схемы трансляции. Нисходящий транслятор представляет собой набор функций, взаимно однозначно соответствующих нетерминалам порождающей грамматики. Эти функции рекурсивно вызывают друг друга и имеют общую глобальную переменную *lookahead*, значение которой указывает на текущий символ входной строки в процессе ее разбора. Обработка входной строки транслятором производится путем установки указателя *lookahead* на первый символ строки и вызова функции S, соответствующей аксиоме грамматики.

Приведем алгоритм построения рекурсивной функции, соответствующей нетерминалу схемы трансляции. Для простоты будем полагать, что каждый нетерминал A имеет единственный синтезируемый атрибут $A.s$, а аксиома S не имеет наследуемых атрибутов (т. е., как мы увидим далее, упомянутый выше вызов S производится без параметров).

Алгоритм 2.1. *Построение функции для нетерминала.*

Вход. Схема трансляции, основанная на LL(1)-грамматике G, нетерминал A.

Выход. Код функции для A.

Функция $\textsc{type}(x)$ возвращает тип переменной (или константы) x, а функция $\textsc{choose}(A, \mathtt{a})$ возвращает «атрибутный» вариант правила вывода, содержащегося в клетке $[A, \mathtt{a}]$ таблицы нисходящего анализатора грамматики G (предполагается, что таблица построена предварительно, см. ч. 2, алгоритм 2.2). Построенная для нетерминала функция имеет вид

$$\text{A}\,(i_1 : \textsc{type}(A.i_1),\ \ldots\ , i_k : \textsc{type}(A.i_k)) : \textsc{type}(A.s)$$

1. **для каждого** `символа из множества` $\{X \mid \exists \alpha, \beta : (A \to \alpha X \beta) \in G\}$
2. **для каждого** `атрибута` $X.i_j$ $(X.s)$
3. `объявить локальную переменную` x_{i_j} `(соотв.` x_s`)`
4. `объявить локальный массив` $A\_develop[\,]$
5. %% тело функции
6. $A\_develop[\,] \leftarrow \textsc{choose}(A, lookahead)$
7. **если** $A\_develop =$ `error`
8. A $\leftarrow$ `error`; **выход**
9. $i \leftarrow 1$

```
10. пока i ≤ |A_develop| повторять
11.       Y ← A_develop[i]
12.       если Y – терминал
13.             если Y = lookahead
14.                   y_s ← Y.s
15.                   lookahead ← следующий символ
16.             иначе A ← error; выход
17.       иначе если Y – нетерминал
18.                   y_s ← Y(y_{i_1}, ... y_{i_m})
19.             иначе %% Y – действие
20.                   выполнить действие,
                      заменив атрибуты переменными для них
                      (A.s в последнем действии
                      заменяется на A)
21.       i ← i + 1
```

Заметим, что к моменту вызова функции Y в строке 18 уже выполнено действие, в котором вычисляются значения переменных $y_{i_1}, \ldots y_{i_m}$, т. е. наследуемых атрибутов Y.

Завершая параграф, рассмотрим конкретный пример построения рекурсивной функции для нетерминала.

Пример 2.5. Рассмотрим функцию для нетерминала E' схемы трансляции TGA_3 из примера 2.3. Наследуемый и синтезируемый атрибуты этого символа являются числами. Значения функции CHOOSE нам известны из таблицы МП-автомата для соответствующей грамматики (ч. 2, пример 2.6): нетерминал E' развертывается как $+TE'_1$, если очередной символ входной строки равен $'+'$ и аннулируется, если этот символ равен $')'$ или концу строки; в остальных случаях выдается сообщение об ошибке. Мы соответственно адаптируем конструкцию, изложенную выше. Заметим также, что вызываемая в строке 4 процедура T не имеет параметров, так как нетерминал T не имеет наследуемых атрибутов.

```
E' (i : int) : int

1. объявить t, i1, s1: int
2. %% тело функции
3. если lookahead = '+'
4.       t ← T
5.       i1 ← i + t
```

```
6.        s1 ← E'(i1)
7.        E' ← s1
8. иначе если lookahead ∈ { ), $}
9.                E' ← i
10.       иначе E' ← error
```

§ 7. Восходящий анализ L-атрибутных грамматик

В данном параграфе нам предстоит установить совершенно неочевидный факт: наследуемые атрибуты можно вычислять и при восходящем анализе! Более того, класс атрибутных грамматик, для которых такое вычисление возможно, шире класса атрибутных грамматик, обрабатываемых нисходящим анализом.

Рассмотрим две основные проблемы, возникающие при попытке реализовать вычисление атрибутов в L-атрибутной грамматике при восходящем анализе. Во-первых, действия с атрибутами можно выполнять в конце правила вывода (при свертке), но невозможно «внутри» правила, как это делалось при нисходящем анализе (см. § 6). В самом деле, в момент появления символа в стеке может быть неизвестно, в правой части какого правила вывода он окажется, что приводит к невозможности вычисления наследуемых атрибутов до появления в стеке всей основы. Во-вторых, наследуемый атрибут символа может зависеть от наследуемого атрибута родителя; родитель же данного символа появится в стеке только в момент удаления самого́ символа, т. е. вычисление такого наследуемого атрибута вообще не представляется возможным.

В некоторых случаях удается снять все проблемы заменой L-атрибутной грамматики на S-атрибутную, как в приводимом ниже примере 2.6 (о вычислении синтезируемых атрибутов в стеке при восходящем анализе см. § 4).

Пример 2.6. Рассмотрим атрибутную грамматику *AGD* из примера 1.2. Ее можно перестроить, избавившись от наследования типа идентификаторов нетерминалом L. Для этого включим наименование типа первым элементом в список идентификаторов. В итоге получим S-атрибутную грамматику AGD_2, приводимую в табл. 2.3.

В общем случае, разумеется, нельзя избавиться от наследуемых атрибутов. Однако обе изложенные выше проблемы могут быть изящно решены при помощи введения дополнительных нетерминалов-маркеров, помещаемых в правую часть правила – там, где в

Таблица 2.3. *S-атрибутная грамматика для объявлений*

	Правило вывода	Семантические правила
1	$D \to L\ \mathtt{a}$	$addtype(\mathtt{a}.entry, L.type)$
2	$L \to L_1\ \mathtt{a}\ ;$	$L.type := L_1.type$ $addtype(\mathtt{a}.entry, L.type)$
3	$L \to \mathtt{int}$	$L.type := \mathtt{int}$
4	$L \to \mathtt{real}$	$L.type := \mathtt{real}$

схеме трансляции предусмотрено вычисление наследуемых атрибутов. Использование маркеров вначале поясним на примере.

Пример 2.7. Рассмотрим схему трансляции TGT из примера 2.2, описывающую вычисление высоты текста в TEX. Попробуем проанализировать какую-нибудь строку (например, из одного символа x) восходящим методом. Мы перенесем x в стек и свернем к B, однако выполнить соответствующее семантическое действие не сможем: необходимому значению атрибута $B.ps$ просто неоткуда взяться в стеке, который был пуст до переноса x. Тем не менее очевидно, что для первого символа кегль известен заранее и равен 10; таким образом, необходимая для выполнения семантического действия информация доступна. Добьемся того, чтобы эта информация была доступна в стеке.

Наследование базового кегля происходит в правиле $S \to A$. Вставим перед символом A *маркер* L и добавим аннулирующее правило $L \to \varepsilon$. Чего можно добиться этим (очевидно, эквивалентным) преобразованием грамматики с точки зрения вычислимости атрибутов? Оказывается, многого. Теперь LR-анализатор при разборе любой строки обнаружит, что первой выполняется свертка по правилу $L \to \varepsilon$ (мы оставляем проверку этого факта читателю); с символом L можно положить в стек значение атрибута, а именно, базового кегля. Тем самым это значение будет доступно при всех последующих свертках.

Аналогичные проблемы с наследованием кегля возникают и в других правилах грамматики; в каждом случае для разрешения проблемы вводится уникальный маркер, функции которого аналогичны функциям маркера L. В итоге мы получаем следующую атрибутную грамматику (табл. 2.4). Для удобства в этой же таблице приведены инструкции, реализующие вычисление атрибутов в стеке.

Таблица 2.4. *L-атрибутная грамматика с маркерами для определения высоты текста и соответствующие вычисления в стеке*

	Правило вывода	Семантические правила	Фрагмент кода
1	$S \to LA$	$A.ps := L.s$ $S.ht := A.ht$	$val[ntop] \leftarrow val[top]$
2	$L \to \varepsilon$	$L.s := 10$	$val[ntop] \leftarrow 10$
3	$A \to A_1 M B$	$A_1.ps := A.ps$ $M.i := A.ps$ $B.ps := M.s$ $A.ht$:=MAX$(A_1.ht, B.ht)$	$val[ntop] \leftarrow$ MAX$(val[top{-}2], val[top])$
4	$M \to \varepsilon$	$M.s := M.i$	$val[ntop] \leftarrow val[top{-}1]$
5	$A \to B$	$B.ps := A.ps$ $A.ht := B.ht$	
6	$B \to B_1 \_ N B_2$	$B_1.ps := B.ps$ $N.i = B.ps$ $B_2.ps := N.s$ $B.ht :=$ DISPOSE$(B_1.ht, B_2.ht)$	$val[ntop] \leftarrow$ DISPOSE$(val[top{-}3], val[top])$
7	$N \to \varepsilon$	$N.s :=$ SHRINK$(N.i)$	$val[ntop] \leftarrow$ SHRINK$(val[top{-}2])$
8	$B \to \{PA\}$	$P.i = B.ps$ $A.ps := P.s$ $B.ht := A.ht$	$val[ntop] \leftarrow val[top{-}1]$
9	$P \to \varepsilon$	$P.s := P.i$	$val[ntop] \leftarrow val[top{-}1]$
10	$B \to$ `sym`	$B.ht :=$ `sym`$.h * B.ps$	$val[ntop] \leftarrow$ $val[top]{*}val[top{-}1]$

Данная таблица, безусловно, требует пояснений. Мы видим, что при каждой свертке требуется только одно «явное» вычисление (цепное правило $A \to B$ не требует вычислений, аналогичная ситуация встречалась в примере 2.1). Часть вычислений, связанных с наследуемыми атрибутами, реализуется неявно. Происходит это благодаря тому, что

(∗) после любого шага работы анализатора в стеке под каждым «основным» нетерминалом находится маркер, значение атри-

бута которого есть значение наследуемого атрибута данного нетерминала.

В самом деле, если нетерминал A или B является не первым символом в основе, то прямо под ним в стеке лежит маркер, хранящий значение наследуемого атрибута этого нетерминала (правила 1, 3, 6 и 8). Если же A или B – первый символ в основе, то его наследуемый атрибут копирует значение наследуемого атрибута родителя (т. е. может храниться там же). Так как родитель – это снова символ A или B, то для него справедливы те же рассуждения. Поскольку нижним символом в стеке является маркер L, рано или поздно рассматриваемый нетерминал будет свернут вместе с символом, лежащим под ним, который обязан являться маркером. Тем самым условие $(*)$ выполняется во всех случаях.

Как видно из примера, использование маркеров позволяет, во-первых, производить семантические действия только в конце правил вывода, а во-вторых, хранить наследуемые атрибуты «основных» нетерминалов в стеке в форме атрибутов маркеров; при этом значение наследуемого атрибута символа хранится в стеке непосредственно под этим символом (даже если самого символа в стеке еще нет, т. е. к нему не произведена свертка, его наследуемый атрибут уже присутствует в стеке непосредственно под нижним символом основы).

Обобщим рассмотренный пример до универсального метода. Для простоты мы примем, в дополнение к ограничениям, изложенным в § 5, одно необременительное предположение о рассматриваемой схеме трансляции:

$(\#)$ в любом правиле вывода (а оно имеет вид (2.1)) для любого $i = 1, \ldots, n$ действие с номером i присутствует тогда и только тогда, когда символ X_i имеет наследуемый атрибут.

Вначале приведем алгоритм маркировки.

Алгоритм 2.2. *Вставка маркеров в грамматику.*
Вход. Схема трансляции TG, основанная на грамматике G.
Выход. Грамматика с маркерами G_{M} такая, что $L(G_{M}) = L(G)$.

1. **для** `каждого правила вывода` $A \to X_1 \ldots X_k$ `из` G
2. **для** i `от 1 до` k **выполнять**
3. **если** `в схеме трансляции [действие]`$_i \neq \emptyset$
4. `вставить перед` X_i `уникальный маркер` M_i

5. `добавить правило` $M_i \to \varepsilon$ `в` G_M
6. `добавить полученное правило с маркерами в` G_M

Очевидно, что языки, задаваемые грамматиками G и G_M, действительно совпадают. Выясним пригодность грамматики G_M для синтаксического анализа.

Предложение 2.3. *Если G – LL(1)-грамматика, то G_M также является LL(1)-грамматикой.*

Доказательство. Таблица нисходящего анализа для грамматики G не содержит конфликтов. Структура таблицы для грамматики G_M отличается только наличием дополнительных строк для маркеров. Поскольку каждому маркеру соответствует только одно правило, конфликт в строке маркера невозможен. Покажем, что строки остальных нетерминалов в таблицах нисходящего анализа для G и G_M совпадают с точностью до замены каждого правила $A \to \alpha$ из G на преобразованное алгоритмом 2.2 правило $A \to \alpha_M$. В самом деле, таблица строится по множествам вида $\textsc{follow}(A)$ и $\textsc{first}(\alpha)$, соответственно для нетерминалов и правых частей правил (см. ч. 2, алгоритмы 2.3 и 2.4). Но $\textsc{first}(\alpha) = \textsc{first}(\alpha_M)$, а множество FOLLOW для любого нетерминала из G не изменилось от введения аннулируемых маркеров. Следовательно, таблица нисходящего анализа для грамматики G_M также не содержит конфликтов, т. е. G_M – LL(1)-грамматика. □

Замечание 2.1. Поскольку мы предполагаем анализировать построенную грамматику с маркерами восходящими методами, то нам достаточно, чтобы она была LR(1)-грамматикой (напомним, что $LL(1) \subset LR(1)$). Таким образом, класс «маркируемых» грамматик оказывается шире класса $LL(1)$, например, в него входит грамматика из примеров 2.2 и 2.7. Однако существуют LR(1)-грамматики, в результате маркировки которых алгоритмом 2.2 получаются не LR(1)-грамматики, см. пример 2.8 в следующем параграфе.

Замечание 2.2. Напомним, что аксиома грамматики, используемой для восходящего анализа, является «внешней», т. е. не появляется в правых частях правил вывода. Следовательно, значения ее наследуемых атрибутов не могут изменяться в ходе трансляции, поэтому в действиях их можно заменить на константы и считать, что *у аксиомы нет наследуемых атрибутов*. Именно такая замена произведена

в примере 2.2 для базового кегля: этот наследуемый атрибут аксиомы заменен на константу 10.

Итак, пусть для маркированной грамматики G_M построен LR-анализатор. Покажем, что в процессе LR-анализа можно вычислить в стеке все атрибуты из схемы трансляции TG. Шаг LR-анализатора грамматики G_M – это перенос, свертка к маркеру (т. е. добавление маркера в стек), или свертка по правилу вида $A \to \alpha_M$, полученному из правила $A \to \alpha$ грамматики G. Отметим, что таблица LR-анализа для G_M не содержит конфликтов по предложению 2.3, т. е. анализатор на шаге свертки к маркеру однозначно определяет, какой именно маркер нужно добавлять в стек. Кроме того, из (#) и алгоритма 2.2 немедленно следует, что если A имеет наследуемый атрибут, то в стеке непосредственно под A обязан находиться маркер. Как и в § 5, будем размещать в стеке рядом с символом (поле *state*) значение его синтезируемого атрибута (поле *val*). А значение наследуемого атрибута нетерминала будем размещать в поле *val* предыдущей записи стека (которая содержит маркер) и *вычислять в момент добавления маркера в стек*.

Пусть в схеме трансляции TG правило $A \to \alpha$ вместе с соответствующими семантическими действиями имеет вид (2.1), и, следовательно, $\alpha_M = M_1 X_1 \dots M_n X_n$ (отсутствие каких-либо из действий и сооветствующих маркеров не меняет наших рассуждений). Покажем, как все действия могут быть выполнены в стеке LR-анализатора при добавлении маркеров $M_1, \dots, M_n$ в стек и свертке по правилу $A \to \alpha_M$.

Действия с 1-го по n-е выполняются при добавлении маркеров. В j-м действии вычисляется наследуемый атрибут $X_j.i$, сопоставленный добавляемому в поле $state[top + 1]$ маркеру M_j. Этот атрибут может зависеть от значений атрибута $A.i$ и атрибутов $X_1.i$, $X_1.s$, $X_2.i$, …, $X_{j-1}.s$. Расположение атрибутов «иксов» в стеке определить легко. Поскольку на рассматриваемом шаге анализа верхним символом в стеке является X_{j-1}, то значение $X_{j-1}.s$ размещается в $val[top]$; предыдущая запись в стеке содержит M_{j-1} и значение $X_{j-1}.i$; и т. д. (если какие-то из маркеров отсутствуют, мы все равно можем определить положение значений всех атрибутов). На вопрос о положении в стеке атрибута $A.i$ отвечает следующая лемма.

Лемма 2.1. *Если наследуемый атрибут нетерминала A существует, то он находится в стеке непосредственно под первым символом основы.*

Доказательство. Символ A не является аксиомой по замечанию 2.2; следовательно, после свертки по правилу $A \to M_1X_1 \dots M_nX_n$ он окажется в основе для одной из последующих сверток. В силу наличия у A наследуемого атрибута, перед A в этой основе находится маркер, а с ним в стеке – атрибут $A.i$ (рис. 2.2). □

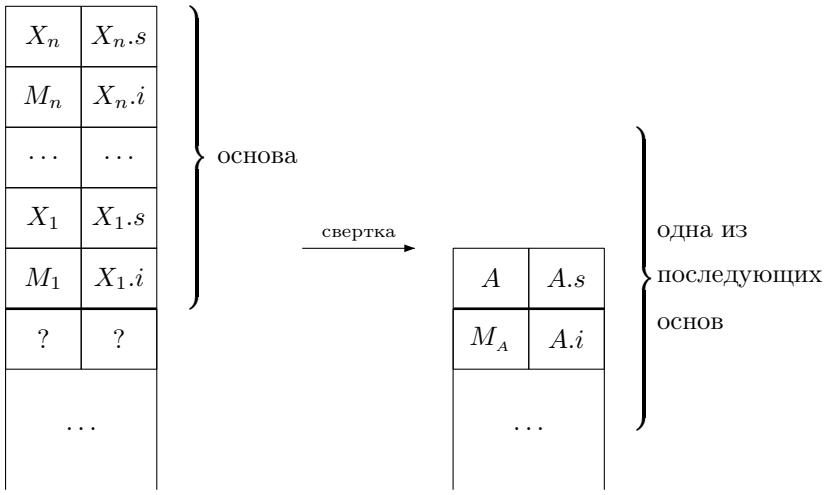


Рис. 2.2. *Изменение стека при свертке по правилу* $A \to M_1X_1 \dots M_nX_n$

С учетом леммы 2.1 мы можем найти в стеке всю информацию, необходимую для вычисления $X_j.i$. Отметим, что в j-м действии могут, кроме вычисления $X_j.i$, содержаться какие-то фрагменты кода, но эти фрагменты могут обращаться только к $X_j.i$ и атрибутам, использовавшимся при вычислении $X_j.i$ (см. правила размещения действий в схеме трансляции, § 5).

Последнее, $(n+1)$-ое действие очевидно может быть выполнено при свертке по правилу $A \to \alpha_{M}$, поскольку вся основа к этому моменту перенесена в стек, т. е. позиции всех необходимых для вычисления $A.s$ атрибутов в стеке известны. В итоге мы полностью реализовали трансляцию схемы TG в процессе LR-анализа маркированной грамматики G_{M}.

Замечание 2.3. Если наследуемый атрибут $X_1.i$ существует, то он зависит только от $A.i$; если при этом соответствующее действие име-

ет вид $X_1.i := A.i$, то маркер M_1 можно удалить из грамматики. В самом деле, в процессе разбора непосредственно под M_1 в стеке находится другой маркер, содержащий значение атрибута $A.i$ (см. лемму 2.1); это значение может выполнять и роль $X_1.i$. Этим наблюдением мы пользовались при маркировке грамматики в примере 2.7.

Замечание 2.4. В примере 2.7 мы построили схему трансляции по маркированной грамматике, используя атрибуты маркеров. Как видно из описания общего метода, такую схему трансляции строить не обязательно: для построения транслятора достаточно маркированной грамматики и исходной схемы трансляции.

§ 8. Более сложные атрибутные грамматики

В этом параграфе мы приведем несколько примеров атрибутных грамматик, не транслируемых в ходе синтаксического анализа, и обсудим возможные обобщения транслятора из § 6 для трансляции таких атрибутных грамматик.

Пример 2.8. В табл. 2.5 приведена L-атрибутная грамматика, основанная на очень простой LR(1)-грамматике. Тем не менее ее нельзя транслировать в ходе LR-анализа, поскольку первая свертка всегда производится по правилу $L \to \varepsilon$, а значение атрибута $L.i$, т. е. количество единиц в строке, в этот момент неизвестно.

Таблица 2.5. *«Несвоевременное» действие в атрибутной грамматике*

	Правило вывода	Семантические правила
1	$S \to L$	$L.i := 0$
2	$L \to L_1 1$	$L_1.i := L.i + 1$
3	$L \to \varepsilon$	$\textsc{print}(L.i)$

В данном случае, безусловно, наиболее простым решением является преобразование атрибутной грамматики, а не переход к более сложным методам трансляции. Вполне достаточно, к примеру, поменять во втором правиле L_1 и единицу местами, чтобы получить «транслируемую» атрибутную грамматику. Следующий пример несколько сложнее.

Пример 2.9. В табл. 2.6 приведен фрагмент не L-атрибутной грамматики. Для ее трансляции необходимо предварительно построить дерево вывода. Трансляция первого из правил требует стандартного обхода сыновей узла A слева направо, а второго правила – обхода справа налево, несовместимого с синтаксическим анализом (та часть входной строки, в которую разворачивается R_2, недоступна в момент применения правила $A \to R_1 R_2$).

Таблица 2.6. *Разнонаправленные обходы в атрибутной грамматике*

	Правило вывода	Семантические правила
1	$A \to L_1 L_2$	$L_1.i := f_1(A.i)$ $L_2.i := g_1(L_1.s)$ $A.s := h_1(L_2.s)$
2	$A \to R_1 R_2$	$R_2.i := f_2(A.i)$ $R_1.i := g_2(R_2.s)$ $A.s := h_2(R_1.s)$

В данном случае уместно применить второй из способов определения порядка выполнения семантических действий, изложенных в § 2: для каждого правила вывода порядок выполнения действий свой и определяется на стадии построения компилятора. На первом проходе компилятора строится дерево вывода, а на втором вычисляются атрибуты при помощи набора рекурсивных функций для нетерминалов. Эти функции очень похожи на функции нисходящего транслятора (§ 6), конструируемые алгоритмом 2.1. Основное отличие состоит в том, что синтаксический разбор уже выполнен, дерево построено, а для каждого узла известны примененное в нем правило вывода и упорядоченный список сыновей. Порядок обхода сыновей определяется только правилом вывода.

Ниже приведена функция для нетерминала A из табл. 2.6. Функция CHILD(n, m) возвращает m-го (по порядку в дереве вывода) сына n-го узла. Каждая функция для нетерминала имеет два аргумента: узел в дереве и наследуемый атрибут этого нетерминала. Заметим, что в функциях транслятора из § 6 роль первого из аргументов неявно выполняла глобальная переменная *lookahead*.

A(n : `узел`, ai : TYPE$(A.i)$) : TYPE$(A.s)$
1. `объявить` $li1, li2, ls1, ls2, ri1, ri2, rs1, rs2$

2. %% тело функции
3. **если** $A \to L_1 L_2$ - `правило вывода в узле` n
4. $\quad li1 \leftarrow f_1(ai)$
5. $\quad ls1 \leftarrow \mathrm{L}(\textsc{child}(n,1), li1)$
6. $\quad li2 \leftarrow g_1(ls1)$
7. $\quad ls2 \leftarrow \mathrm{L}(\textsc{child}(n,2), li2)$
8. $\quad \mathrm{A} \leftarrow h_1(ls2)$
9. **иначе если** $A \to R_1 R_2$ - `правило вывода в узле` n
10. $\quad\quad ri2 \leftarrow f_2(ai)$
11. $\quad\quad rs2 \leftarrow \mathrm{R}(\textsc{child}(n,2), ri2)$
12. $\quad\quad ri1 \leftarrow g_2(rs2)$
13. $\quad\quad rs1 \leftarrow \mathrm{R}(\textsc{child}(n,1), ri1)$
14. $\quad\quad \mathrm{A} \leftarrow h_2(rs1)$
15. $\quad$ **иначе** A $\leftarrow$ `error`

Наконец, рассмотрим наиболее сложный пример, когда все атрибуты не могут быть вычислены за один проход по готовому дереву вывода.

Пример 2.10. В следующей атрибутной грамматике (табл. 2.7) нетерминал E имеет два синтезируемых и один наследуемый атрибут. Легко видеть, что синтезируемый атрибут $.s$ в узле всегда зависит только от тех же атрибутов сыновей узла. Наследуемый атрибут $.i$ в узле зависит от того же атрибута родителя узла либо, если родитель – корень дерева, от атрибута $.s$ самого узла. Наконец, синтезируемый атрибут $.t$ зависит от тех же атрибутов сыновей, а в случае, когда сын является листом, – от атрибута $.i$ самого узла. Следовательно, для вычисления всех атрибутов нужно:

1) пройти по дереву вывода снизу вверх, вычисляя атрибут $.s$;

2) пройти по дереву вывода сверху вниз, вычисляя атрибут $.i$;

3) еще раз пройти снизу вверх, вычисляя атрибут $.t$.

На практике можно транслировать данную атрибутную грамматику в два прохода. Сначала провести синтаксический анализ с построением дерева и вычислением значений атрибута $.s$ во всех узлах, а затем по дереву рекурсивно вычислить значения атрибутов $.i$ и $.t$ (требуется один стандартный обход дерева в глубину).

Возникает естественный вопрос: а встречаются ли на практике такие вычурные грамматические конструкции? Оказывается, да.

Таблица 2.7. *Сложная зависимость атрибутов в грамматике*

	Правило вывода	Семантические правила
1	$S \to E$	$E.i := h_1(E.s)$ $S.t := E.t$
2	$E \to E_1 E_2$	$E.s := f_1(E_1.s, E_2.s)$ $E_1.i := g_1(E.i)$ $E_2.i := g_2(E.i)$ $E.t := f_2(E_1.t, E_2.t)$
3	$E \to \mathtt{x}$	$E.s := \mathtt{x}.s$ $E.t := h_2(E.i)$

Несложно привести пример задачи, для решения которой нужна атрибутная грамматика именно с такой структурой семантических действий, как в примере 2.10. Рассмотрим язык программирования, не требующий явного определения типов (подробнее о работе с типами будет рассказано в гл. 3). При компиляции программы нужно произвести разбор выражения и установить тип выражения и типы всех его подвыражений (на основании контекста, т. е. конкретного использования в тексте программы с операторами, константами, и т. п.), после чего внести информацию о типах в генерируемый код.

Для этого необходимо, во-первых, для каждого подвыражения определить множество допустимых типов (вычисляется снизу вверх по дереву – это синтезируемый атрибут $E.s$ из примера 2.10). Во-вторых, нужно выбрать из этого множества единственный тип (это производится сверху вниз – фиксируется допустимый тип для всего выражения, а потом на основании этого уточняются типы подвыражений; выбранный тип представлен в примере 2.10 наследуемым атрибутом $E.i$). Наконец, на основе выбранных типов снизу вверх по дереву генерируется код (в примере 2.10 – синтезируемый атрибут $E.t$).

Задачи

1. LR-анализатор грамматики арифметических выражений GA_2 обрабатывает строку $3*(2+5)$, вычисляя при этом значение выражения. Вычислить состояние стека (поля *state* и *val*) перед переносом закрывающей скобки.

2. Переписать атрибутную грамматику AGN_2 в виде схемы трансляции.

3. Устранить левую рекурсию
   а) из атрибутной грамматики AGN_1;
   б) из схемы трансляции, полученной в задаче 2.

4*. Построить нисходящий транслятор
   а) для атрибутной грамматики AGT (пример 2.2);
   б) для атрибутной грамматики из задачи 2 к гл. 1.

5*. Адаптировать атрибутную грамматику AGN_2 для восходящего анализа, построить LR-анализатор и восходящий транслятор на его основе.

Глава 3. Проверка семантической корректности программ

Эта глава почти полностью посвящена проверке типов – наиболее сложной из семантических проверок, выполняемых компилятором. Такая проверка основана на рассмотрении типов и множеств типов как атрибутов нетерминалов основной грамматики, поэтому мы будем активно пользоваться материалом двух предыдущих глав.

§ 9. Виды семантических проверок. Ошибки, не распознаваемые при компиляции

Компилятор, помимо проверки синтаксической правильности транслируемой программы, выполняет и часть проверки ее семантической корректности. Более точно, компилятор производит *статические* проверки – выявляет те ошибки, которые могут быть обнаружены без запуска транслированной программы. К таким проверкам относятся:

1. *Проверки типов.* Операторы должны применяться к совместимым с ними операндам, функции должны вызываться с нужным количеством аргументов, типы аргументов должны соответствовать объявленным типам параметров и т. д.

2. *Проверки управления.* Передача управления в программе должна осуществляться в существующее и разрешенное место. На-

пример, воспользоваться принудительным выходом из цикла или функции (в Паскале есть оператор `exit`, в C – `break`) можно только в точке внутри цикла/функции[†]. Обычно запрещено передавать управление (например, при помощи метки) внутрь цикла или функции, минуя заголовок.

3. *Проверки единственности.* В ряде языков допускается лишь однократное объявление имени. К примеру, в Паскале такое правило касается использования идентификаторов внутри группы объявлений, меток оператора `case`, а также констант в скалярном типе.

4. *Проверки имен.* В языках разметки, например в XML, каждый открывающий тэг должен иметь парный закрывающий тэг с тем же именем. В издательской системе TEX подчиненные конструкции, называемые окружениями (в том числе определяемые пользователем), должны быть заключены в «скобки» вида

   `\begin{имя_конструкции} \end{имя_конструкции}`.

   Это требование, в частности, позволяет локализовать многие ошибки и опечатки, ускоряя подготовку текстов.

Очевидно, что только статической проверки недостаточно, требуется еще *динамическая* – в ходе выполнения программы. В большинстве случаев только во время работы программы можно обнаружить такие ошибки, как

1. *Недопустимые операции* – например, деление на 0.

2. *Ошибки работы с памятью* – всевозможные переполнения, а также обращения к неопределенным динамическим объектам.

3. *Ошибки индексирования* – например, недопустимые значения индексов в массиве.

Когда речь идет о серьезных программных продуктах, семантика программы описывается *спецификацией* на специальном формализованном языке. Динамическая проверка семантической корректности в этом случае состоит в тестировании при помощи специальных

[†]Впрочем, в современных версиях Delphi – «наследника» Паскаля – оператор `exit`, примененный на верхнем уровне программы, не приведет к ошибке, а будет проигнорирован. Однако соответствующая проверка все равно необходима.

инструментов, проверяющих соответствие между работой готовой программы и спецификацией. Соответствующие технологии также имеют непосредственное отношение к языкам, грамматикам и автоматам, но находятся за рамками этой книги. Мы рассмотрим только статическую проверку – прерогативу компилятора, причем только наиболее нетривиальную ее часть – проверку типов.

§ 10. Типы данных и выражения типа

Любой элемент/блок данных имеет свой тип, определяющий структуру данных, способ представления в памяти, допустимые методы обработки и самое главное – семантику, т. е. смысл самих данных[†]. Тип данных задается *выражением типа*. Такие выражения получаются применением операторов, называемых *конструкторами типов*, к операндам, представляющим собой элементарные, или *базовые* типы, либо другие выражения типа. Кроме того, любое выражение типа можно снабдить именем (мы будем говорить о нем как об *имени типа*). Как мы увидим в дальнейшем, удобно ввести в рассмотрение *переменные типа* – переменные, значениями которых являются выражения типа.

Базовые типы не имеют внутренней структуры. К ним относятся числовые, байтовый и битовый типы; могут быть и другие, в зависимости от рассматриваемого языка (например, перечисляемые типы). В примерах мы будем рассматривать в качестве базовых типы `int, real, char` и `bool`. К ним добавляются два специальных типа – `type_error`, сигнализирующий об ошибке типов, а также `void`, указывающий на отсутствие возвращаемого значения и тем самым позволяющий анализировать инструкции и процедуры с точки зрения наличия ошибок типов. Обозначения базовых типов будем называть *константами типа*. Мы будем использовать пять конструкторов типов – *массив*, *произведение*, *запись*, *указатель* и *функция*. Дадим строгое индуктивное определение выражения типа.

Определение. 1. Константа типа, имя типа и переменная типа являются выражениями типа.
2. Результат применения конструктора типов к выражениям типа является выражением типа. Более точно:

[†]Даже в языке Lisp, одним из принципов которого является *отсутствие* типов у данных, типы на самом деле есть, только их вычисление производится уже в ходе работы программы.

1) *Массивы.* Если T – выражение типа и N – натуральное число, то $\texttt{array}(N, T)$ является выражением типа, задающим тип массива длины N с элементами типа T и индексами от 0 до $N-1$[†].

2) *Произведения.* Если T_1, T_2 – выражения типа, то их декартово произведение $T_1 \times T_2$ является выражением типа.

3) *Записи.* Если $T_1, \ldots T_k$ – выражения типа, $name_1, \ldots name_k$ – их имена, то

$$\texttt{record}(\,(name_1 \times T_1) \times \ldots \times (name_k \times T_k)\,)$$

является выражением типа, задающим тип записи с полями $T_1, \ldots T_k$, имеющими имена $name_1, \ldots name_k$ соответственно. Тип записи всегда имеет имя, которое задается при объявлении этого типа.

4) *Указатели.* Если T – выражение типа, то $\texttt{ptr}(T)$ является выражением типа, задающим тип «указатель на объект типа T».

5) *Функции.* Если T_1, T_2 – выражения типа, то $T_1 \to T_2$ является выражением типа, задающим тип функции, отображающей значения типа T_1 в значения типа T_2.

3. Других выражений типа нет.

Выражения типа, как и любые выражения, удобно представлять в виде синтаксических деревьев или дагов (см. § 3). Приведем пример.

Пример 3.1. Пусть функция f вычисляет по любой перестановке τ множества $\{1, \ldots, n\}$ квадрат этой перестановки. Тогда f имеет тип $(\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$, задаваемый синтаксическим деревом и дагом как на рис. 3.1, *а*, *б*[‡].

Более сложная ситуация получается, когда выражение типа является рекурсивным. Например, рекурсии не избежать при построении динамических списков: элементом списка является запись, одно или несколько полей которой – указатели на записи этого же типа.

[†]Можно использовать и любую другую трактовку индексного множества. Мы выбрали эту из-за ее простоты.

[‡]В большинстве современных языков программирования ограничения на типы аргументов и значения функций несущественны и подобные выражения типа могут быть легко реализованы.

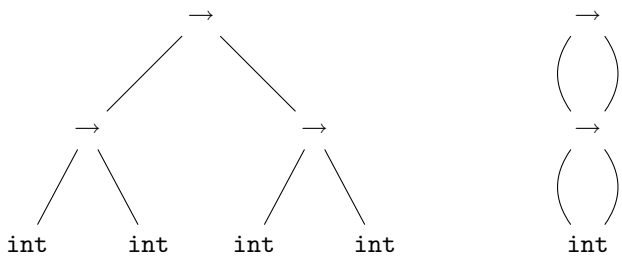


Рис. 3.1. *Синтаксическое дерево (а) и даг (б) для выражения типа* (`int` → `int`) → (`int` → `int`)

Пример 3.2. Рассмотрим объявление типа записи в C для построения двусвязного списка целых чисел:

`struct` node { `int` info; `struct` node ∗next, ∗prev; }

Соответствующее выражение типа можно представить в виде графа двумя способами: либо без циклов, но используя имя объявляемого типа в качестве меток листьев (рис. 3.2, *а*), либо вводя циклы как на рис. 3.2, *б*. Мы будем использовать ациклическое представление.

В заключение параграфа мы обсудим понятие эквивалентности выражений типа, необходимое для проверки типов. Интуитивно два выражения, имеющие одинаковую структуру, определяют один и тот же тип. Однако имеется одна тонкость, связанная с рекурсивными выражениями.

Определение. Выражения типа называются *структурно эквивалентными*, если представляющие их даги изоморфны[†].

Замечание 3.1. Следующее наблюдение легко преобразовать в рекурсивный алгоритм проверки изоморфизма дагов: два дага изоморфны, если их корни имеют одинаковые метки, а подграфы, порожденные соответствующими сыновьями корней, изоморфны.

Замечание 3.2. Рекурсивные выражения типа (как в примере 3.2) структурно не эквивалентны, если их имена различны. Это связано с тем, что имя типа появляется в качестве метки узла дага. Поскольку, естественно, одно и то же имя нельзя использовать для обозначения

[†]Напомним, что даг – упорядоченный ациклический помеченный орграф.

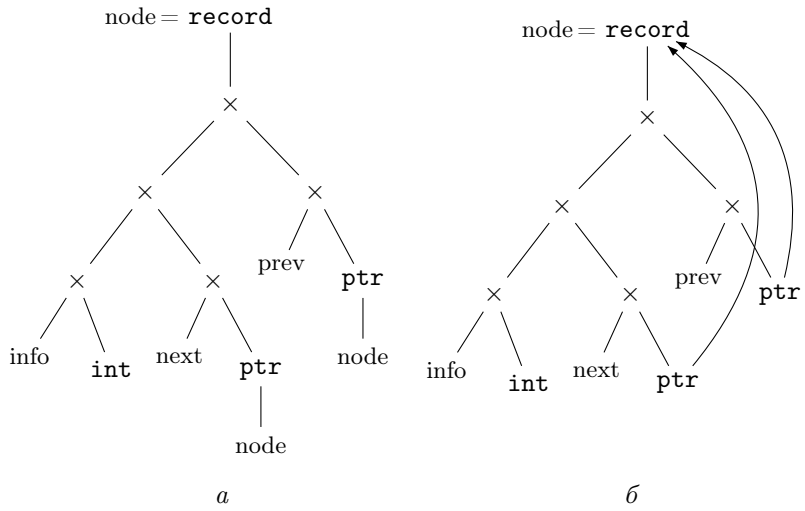


Рис. 3.2. *Дерево (а) и циклический граф (б) для представления записи*

разных типов, структурная эквивалентность рекурсивных выражений типа есть в точности равенство их имен.

В свете замечания 3.2, алгоритм проверки изоморфизма, предложенный в замечании 3.1, можно уточнить следующим образом: если метки корней проверяемых дагов совпадают, то

– в случае, когда эти метки представляют «потенциально рекурсивный» конструктор (например, запись), проверяется равенство имен дагов;

– если же корни помечены «нерекурсивным» конструктором, то выполняется проверка на изоморфизм дагов, порожденных соответствующими сыновьями.

§ 11. Атрибутная грамматика для проверки типов

Множество правил назначения типов частям программы называется *системой типов*. Статическая проверка типов в программе состоит в реализации этих правил. Фрагмент программы, получивший тип `type_error`, содержит ошибку типов; если же программа целиком имеет тип, отличный от `type_error`, то она успешно прошла

проверку. В этом параграфе мы изложим простую систему типов в виде атрибутной грамматики.

Рассмотрим грамматику, порождающую небольшой, но достаточно представительный фрагмент языка программирования (с «паскалеподобным» синтаксисом), содержащий объявления, выражения и инструкции[†]. Функции мы рассматриваем очень упрощенно, как объявление идентификатора с типом $T_1 \mapsto T_2$ (используется стрелка $\mapsto$, поскольку в той же формуле обычная стрелка $\rightarrow$ означает «вывод по правилу»). Программа (аксиома P) состоит из набора объявлений D, за которыми следует набор инструкций S. Объявление состоит в присвоении идентификатору типа, задаваемого выражением типа T. Инструкции могут содержать выражения E. Нетерминалы T, S, E, P имеют синтезируемый атрибут *type*, содержащий тип фрагмента программы, представляемого этим нетерминалом. Полностью атрибутная грамматика (будем обозначать ее APG) приведена в табл. 3.1.

Вычисление атрибутов $T.type$ производится в соответствии с определением выражения типа (см. § 9). Остановимся подробнее на вычислении типов используемых выражений и инструкций. Знак равенства между выражениями типа означает эквивалентность этих выражений.

Если выражение представляет собой константу, то его тип – это тип данной константы. В грамматике APG есть только целочисленные константы, поэтому тип присваивается сразу. Если выражение – это переменная, то его тип берется из таблицы символов (куда он попал после разбора объявления данной переменной, см. правило 3). Далее, выражение-неравенство имеет тип `bool`, если типы его частей позволяют эти части сравнивать, и тип ошибки в противном случае. Для простоты мы полагаем, что сравнивать можно только одинаковые базовые типы. О *преобразовании типов*, позволяющем расширить возможности применения различных операторов, см. следующий параграф. Выражение-ссылка по указателю имеет тип, для значений которого определен указатель. Выражение-обращение к элементу массива имеет, естественно, тип элемента этого массива; необходимым условием отсутствия ошибки является то, что «массив» E_1 действительно имеет тип массива, а «индекс» E_2 имеет целочислен-

[†]Подобная грамматика, написанная в синтаксисе C, будет существенно отличаться в части обработки объявлений. Понять эти отличия можно на примере объявления `int *x`.

Таблица 3.1. *Атрибутная грамматика для проверки типов*

	Правило вывода	Семантические правила
1	$P \to D; S$	$P.type{:=}S.type$
2	$D \to D; D$	
3	$D \to \mathtt{x} : T$	$\textsc{addtype}(\mathtt{x}.entry, T.type)$
4	$T \to \mathtt{real}$	$T.type{:=}\mathtt{real}$
5	$T \to \mathtt{int}$	$T.type{:=}\mathtt{int}$
6	$T \to \uparrow T_1$	$T.type{:=}\mathtt{ptr}(T_1.type)$
7	$T \to \mathtt{array[n]\ of}\ T_1$	$T.type{:=}\mathtt{array}(\mathtt{n}.val, T_1.type)$
8	$T \to T_1 \mapsto T_2$	$T.type{:=}T_1.type \to T_2.type$
9	$S \to S_1; S_2$	$S.type{:=}\begin{cases} S_2.type\ , & \text{если } S_1.type{=}\mathtt{void} \\ \mathtt{type\_error} & \text{иначе} \end{cases}$
10	$S \to \mathtt{x}{:=}E$	$S.type{:=}\begin{cases} \mathtt{void}, & \text{если } \textsc{type}(\mathtt{x}.entry){=}E.type \\ \mathtt{type\_error} & \text{иначе} \end{cases}$
11	$S \to \mathtt{if}\ E\ \mathtt{then}\ S_1$	$S.type{:=}\begin{cases} S_1.type\ , & \text{если } E.type{=}\mathtt{bool} \\ \mathtt{type\_error} & \text{иначе} \end{cases}$
12	$S \to \mathtt{while}\ E\ \mathtt{do}\ S_1$	$S.type{:=}\begin{cases} S_1.type\ , & \text{если } E.type{=}\mathtt{bool} \\ \mathtt{type\_error} & \text{иначе} \end{cases}$
13	$E \to \mathtt{n}$	$E.type{:=}\mathtt{int}$
14	$E \to \mathtt{x}$	$E.type{:=}\textsc{type}(\mathtt{x}.entry)$
15	$E \to E_1 < E_2$	$E.type{:=}\begin{cases} \mathtt{bool}, & \text{если } E_1.type{=}E_2.type{=}\mathtt{int} \\ & \text{или } E_1.type{=}E_2.type{=}\mathtt{real} \\ \mathtt{type\_error} & \text{иначе} \end{cases}$
16	$E \to E_1\uparrow$	$E.type{:=}\begin{cases} t\ , & \text{если } E_1.type{=}\mathtt{ptr}(t) \\ \mathtt{type\_error} & \text{иначе} \end{cases}$
17	$E \to E_1[E_2]$	$E.type{:=}\begin{cases} t\ , & \text{если } E_2.type{=}\mathtt{int}, \\ & E_1.type{=}\mathtt{array}(N,t) \\ \mathtt{type\_error} & \text{иначе} \end{cases}$
18	$E \to E_1(E_2)$	$E.type{:=}\begin{cases} t\ , & \text{если } E_2.type{=}s, \\ & E_1.type = s{\to}t \\ \mathtt{type\_error} & \text{иначе} \end{cases}$

ный тип. Наконец, выражение-вызов функции имеет тип значения этой функции; для корректности такого вызова нужно, чтобы E_1 действительно имело тип функции, а E_2 – тип, совпадающий с типом аргумента функции.

Инструкция имеет тип `void`, если она корректна с точки зрения типов, и тип `type_error` в случае наличия ошибки. Соответственно в инструкции присвоения проверяется эквивалентность типа присваиваемого выражения типу переменной, а в условной инструкции и инструкции цикла проверяется, что условие имеет булев тип.

Пример 3.3. Рассмотрим следующую «программу»:

```
y: int;
z: array[2] of int;
if z[1]<z[2] then y:=z[1] .
```

На рис. 3.3 приведено ее (не аннотированное) дерево вывода в грамматике *APG*. При обходе левого поддерева (с корнем D) в таблицу символов будут внесены записи о типах идентификаторов; это произойдет при выполнении процедур ADDTYPE(`y`, `int`) и ADDTYPE(`z`, `array`(2, `int`)). При обходе правого поддерева (с корнем S) на основе данных о типах идентификаторов будет вычислен тип самой программы; значения атрибута *type* в этом поддереве приведены на рис. 3.4.

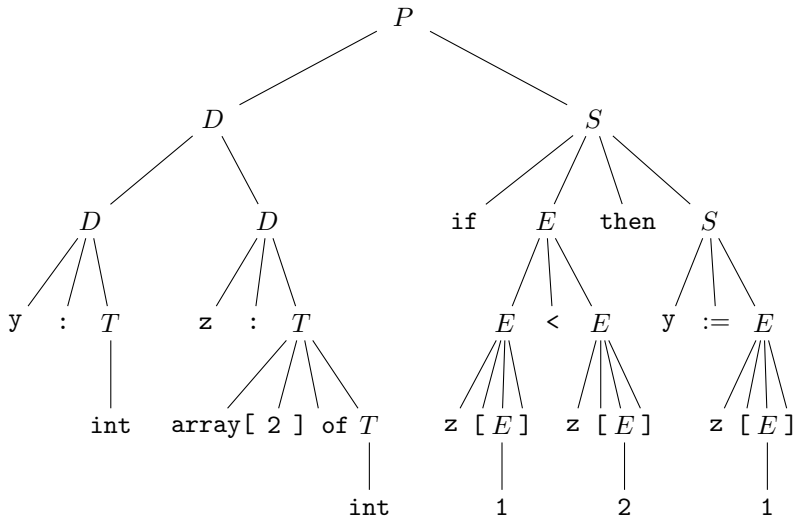


Рис. 3.3. *Дерево вывода в грамматике APG*

Замечание 3.3. В случае когда в языке программирования объявления типов отсутствуют или не обязательны, рассмотренная страте-

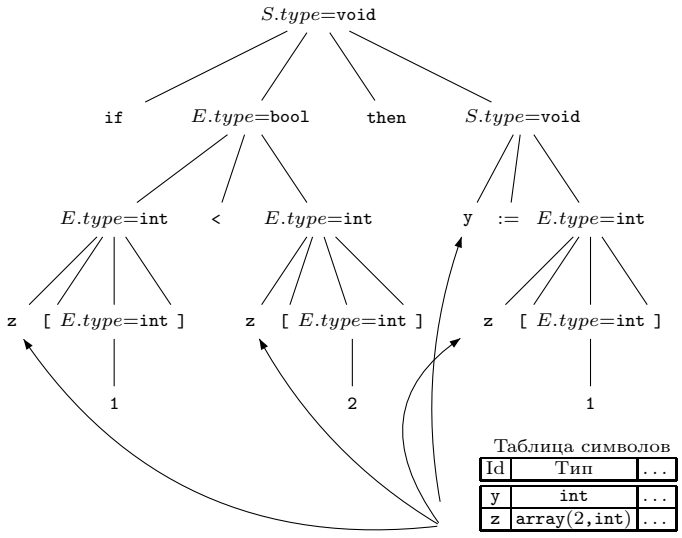


Рис. 3.4. *Значения типов выражений и инструкций*

гия вычисления типов частей программы по-прежнему применима. Однако вычисляться будут не типы выражений, а *множества допустимых типов* этих выражений; такие множества вычисляются в предположении, что программа корректна с точки зрения типов. Пустое множество допустимых типов сигнализирует об ошибке[†]. После вычисления всех множеств допустимых типов необходимо произвести выбор единственного типа для каждого выражения, после чего внести записи о типах в таблицу символов (см. пример 2.10 и комментарии после него).

§ 12. Преобразование типов

Разумеется, такие жесткие условия на типы аргументов выражений и инструкций, как приведенные в грамматике *APG*, не встречаются в реальных языках программирования высокого уровня, хотя на машинном уровне это необходимо. Процессор не может сложить целое число с действительным по двум причинам: действительные

[†]В языке Lisp такая ситуация обрабатывается по-другому и может привести к определению нового типа.

числа имеют иное внутреннее представление, нежели целые, а для их сложения используется другая машинная инструкция. Выход прост и хорошо известен: целое число надо преобразовать в действительный тип и выполнить сложение действительных чисел.

Общая концепция работы с типами в современных языках такова: в обращении с типами допускаются все «вольности», которые полезны для программистов с точки зрения создателей языка, при условии, что компилятор в состоянии распознать все разрешенные «несоответствия» типов и выполнить необходимые преобразования автоматически[†]. Действия компилятора разберем на примере.

Пример 3.4. Пусть в языке имеется три числовых типа: `int`, `real` и `double` («длинные» действительные числа). Для инструкции присвоения `z:=x+y` построим промежуточное представление (синтаксическое дерево) с учетом типов переменных. Пусть `x` имеет тип `int`, а `y` – тип `real`. Тогда сложение должно быть действительным, и выражение `x+y` получит тип `real`. Соответствующий фрагмент синтаксического дерева приведен на рис. 3.5, *а*.

Далее, рассмотрим тип переменной `z`. Если это `real`, то никаких преобразований больше не нужно, инструкция корректна. Если этот тип – `int`, то компилятор должен выдать сообщение об ошибке[‡]. Наконец, если `z` имеет тип `double`, то результат суммирования `x` и `y` нужно преобразовать к этому типу; в итоге получаем синтаксическое дерево как на рис. 3.5, *б*.

Рассмотренная в примере 3.4 инструкция содержит *перегруженный* оператор `'+'`: он означает различные операции над аргументами в зависимости от типов аргументов. Помимо целочисленного и действительного сложений `'+'` может означать, к примеру, конкатенацию строк. Естественно, компилятор должен определить, какая из операций имеется в виду, и учесть это при построении синтаксического дерева.

Без учета типов, правилу вывода вида $E \to E_1{+}E_2$ при постро-

[†]Преобразования, выполняемые компилятором, называются *неявными*, в отличие от *явных* – записанных в программе. Например, преобразование символа в его код, т. е. в целое число, в Паскале осуществляется явно – при помощи функции `ord`, а в C – неявно.

[‡]Другая возможность – выполнять преобразование действительного выражения к целочисленному типу – теоретически допустима, но может привести к трудноотслеживаемым ошибкам в работе программы. Нередко в компиляторах используется третья возможность: изменение типа переменной `z` с выдачей соответствующего предупреждения.

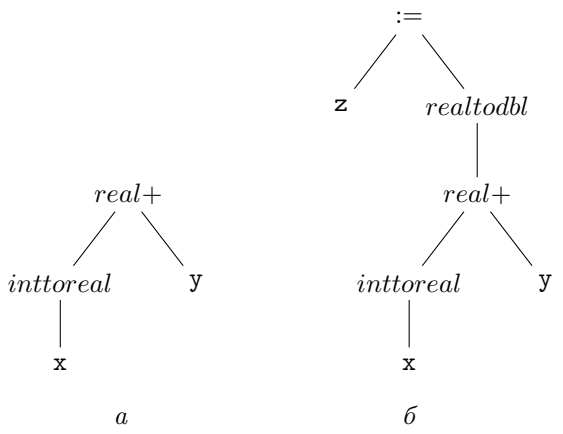


Рис. 3.5. *Синтаксические деревья для цепочек* `x+y` *(а) и* `z:=x+y` *(б) после преобразования компилятором*

ении синтаксического дерева сопоставляется семантическое правило $E.ptr :=$ MKNODE$(+, E_1.ptr, E_2.ptr)$ (ср. пример 1.8). Посмотрим, какие действия надо сопоставить этому правилу, если типы принимаются во внимание. Для простоты будем считать, что сложение можно применять только к типам `real` и `int`. Вместо одной функции MKNODE будем рассматривать две – одну для унарных, другую для бинарных операций. Результат запишем в виде `case`-структуры по значениям атрибутов $E_1.type$ и $E_2.type$:

`int`, `int`: $E.type :=$ `int`
$E.ptr :=$ MKNODE2$(int+, E_1.ptr, E_2.ptr)$
`int`, `real`: $E.type :=$ `real`
$E.ptr :=$
MKNODE2$(real+,$ MKNODE1$(inttoreal, E_1.ptr), E_2.ptr)$
`real`, `int`: $E.type :=$ `real`
$E.ptr :=$
MKNODE2$(real+, E_1.ptr,$ MKNODE1$(inttoreal, E_2.ptr))$
`real`, `real`: $E.type :=$ `real`
$E.ptr :=$ MKNODE2$(real+, E_1.ptr, E_2.ptr)$
иначе: $E.type :=$ `type_error`

Последнее замечание в этом параграфе касается автоматического преобразования констант. Многие учебники обошел классический

пример: если для массива `X[ ]` действительных чисел в Паскале написать присвоение

«`for i:=1 to n do X[i]:=1`»,

то полученный фрагмент исполняемого кода будет работать гораздо медленнее, чем для присвоения, записанного в виде

«`for i:=1 to n do X[i]:=1.0`».

Это связано с тем, что при прямолинейном преобразовании типов, как в примере 3.4, в каждой итерации цикла будет вызываться функция *inttoreal* (выполняющаяся к тому же в несколько раз медленнее самой инструкции присвоения).

Мораль приведенного примера проста: эффективный компилятор должен преобразовывать константы иначе, чем выражения, значения которых априори неизвестны. При необходимости преобразования константы должна быть изменена запись в таблице символов: указаны новый тип и новое значение.

§ 13. Полиморфизмы

В этом, последнем, параграфе мы кратко рассмотрим работу с выражениями типа, содержащими переменные.

Определение. Структурный элемент программы (выражение, инструкция, функция, класс, метод и т. п.) называется *полиморфным*, если он может однотипно выполняться с аргументами различных типов.

В любом «промышленном» языке программирования есть встроенные полиморфные операторы, например, для индексирования массивов, вызова функций или работы с указателями. Возможность применения такого оператора обычно определяется не типом аргумента, а только меткой корневого узла соответствующего выражения типа.

Пример 3.5. Рассмотрим еще раз атрибутную грамматику *APG* для проверки типов (табл. 3.1). Правило 17 показывает, как работает встроенный оператор индексирования массива. А именно, если выражение E_1 имеет тип массива с элементами некоторого типа t, а выражение E_2 имеет тип `int`, то выражение $E_1[E_2]$ получает тип t (и значение элемента массива E_1 с номером, равным значению E_2). Поскольку тип t, вообще говоря, может быть любым, естественно

рассматривать t как переменную типа; оператор индексирования задается выражением типа $(\texttt{array}(N, t) \times \texttt{int}) \to t$. Заметим, что наличие переменных типа – отличительная черта выражений типа с полиморфными операторами.

Правило 16 этой же грамматики определяет, как работает полиморфный оператор, возвращающий значение по указателю на него (этот оператор иногда называют *разыменованием указателя*). Как легко проверить, разыменование задается выражением типа $\texttt{ptr}(t) \to t$.

Частные случаи, или *примеры*, выражения типа с переменными получаются при замене переменных выражениями типа (все вхождения одной и той же переменной должны заменяться одинаково). Последнее выражение типа имеет примеры $\texttt{ptr}(\texttt{char}) \to \texttt{char}$, $\texttt{ptr}(\texttt{ptr}(t)) \to \texttt{ptr}(t)$ и т. п.

Замечание 3.4. С точки зрения математической логики каждая переменная типа в выражениях с полиморфизмами связана квантором всеобщности. Это, в частности, означает, что при переименовании переменных (без отождествления!) выражение типа не меняется. Дописывать кванторы в выражения типа мы не будем.

Пример 3.6. Обратимся еще раз к грамматике *APG* (табл. 3.1) и рассмотрим выражение `x[x[y]↑]↑` . Пусть мы ничего не знаем о типах идентификаторов `x` и `y`. Можно ли вычислить тип выражения? Оказывается, да.

Построим синтаксический даг для выражения. Внутренние узлы этого дага помечены одним из двух операторов – это бинарный оператор индексирования и унарный оператор разыменования, которые мы обозначим через `ind` и `deref` соответственно. Кроме того, сопоставим каждому узлу уникальную переменную типа, отвечающую за тип подвыражения с корнем в данном узле (рис. 3.6).

Проанализируем даг снизу вверх. При применении первого оператора `ind` должно получиться значение типа γ, т. е. γ играет роль переменной в выражении типа для `ind`. Из применимости `ind` к паре значений типов α и β следуют равенства $\alpha = \texttt{array}(N, \gamma)$ и $\beta = \texttt{int}$.

На следующем шаге применяется разыменование, дающее значение типа δ, откуда $\gamma = \texttt{ptr}(\delta)$. При рассмотрении второго оператора индексирования мы получаем $\delta = \texttt{int}$ и $\alpha = \texttt{array}(N, \varkappa)$. Но в этот момент мы уже знаем, что $\alpha = \texttt{array}(N, \gamma) = \texttt{array}(N, \texttt{ptr}(\texttt{int}))$; следовательно, $\varkappa = \texttt{ptr}(\texttt{int})$. Наконец, рассмотрев оператор `deref` в корне дага, мы получаем $\lambda = \texttt{int}$. Таким образом, тип выражения

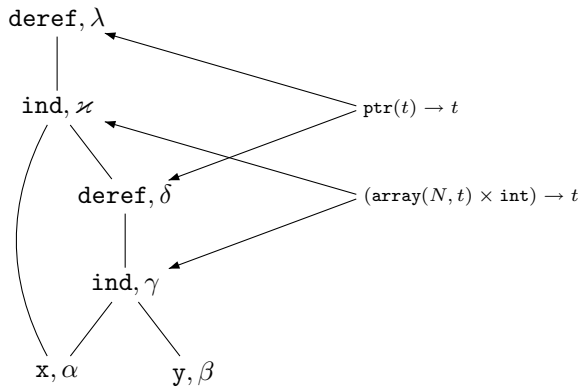


Рис. 3.6. *Синтаксический даг для определения типа выражения*

вычислен, а попутно установлены типы входящих в него идентификаторов (кроме размера массива `x`).

Процедура, которую мы применяли в каждом узле дага в последнем примере для уточнения типов выражений, называется *унификацией* выражений типа. Более точно, процедура унификации такова: имея два (или более) выражений типа, мы выполняем в них одновременно ряд замен переменных на выражения типа (для *всех* вхождений одной переменной во *все* исходные выражения выполняется *одна и та же* замена) так, чтобы полученные примеры исходных выражений типа совпали. Если такой набор замен существует, то выражения типов называются *унифицируемыми*, а любой подходящий набор замен – *унификатором*. Заметим, что выражения типа без переменных унифицируемы тогда и только тогда, когда они эквивалентны.

Пусть для фрагмента программы вычислены разными способами два выражения типа; если они унифицируемы, то в качестве выражения типа для этого фрагмента берется их общий пример[†]; если они неунифицируемы, то фрагмент содержит ошибку типов. Например, при анализе верхнего из узлов с меткой `ind` на рис. 3.6 мы унифицировали выражения $\texttt{array}(N, \varkappa)$ и $\texttt{array}(N, \texttt{ptr}(\texttt{int}))$ для типа идентификатора `x` заменой $\varkappa = \texttt{ptr}(\texttt{int})$. Если бы унификатора не

[†]Если таких примеров несколько, то берется «наиболее общий» из них, т. е. такой, для которого все остальные являются примерами.

существовало, то мы бы обнаружили, что использование идентификатора x некорректно с точки зрения типов.

Для унификации нециклических выражений типа (а мы договорились рассматривать только такие) можно применить алгоритм унификации термов из стандартного курса математической логики (см. например, [ЧЛ]).

Задачи

1. Записать выражения типов для идентификаторов u и v из приводимого ниже фрагмента объявления на C:

```
typedef struct {
   double a,b;
   } Point, *Pointer;
Point u[100];
Pointer v(x,y) int x; Point y {...}
```

2. Записать выражение типа для вершины детерминированного конечного автомата над двухбуквенным алфавитом.

3. Изобразить выражение типа в виде синтаксического дага:
   а) $\texttt{array}(10, \texttt{ptr}(\texttt{int} \times \texttt{int}))$;
   б) выражения типа из задачи 1;
   в) выражение типа из задачи 2.

4. При помощи атрибутной грамматики APG проверить типы в следующем фрагменте программы:

```
i: int;
y: int↦int;
z: array[10] of ↑int;
while i<y(i) do
   z[i]↑:=y(i); i:=y(i) .
```

5. Переписать часть атрибутной грамматики APG, описывающую объявления, в синтаксисе языка C.

6. Перестроить семантические правила для присвоения и сравнения в атрибутной грамматике APG с учетом преобразования типов.

7. Предполагая, что приводимая ниже инструкция присвоения корректна с точки зрения типов, вывести типы идентификаторов и присваиваемого выражения (используется синтаксис атрибутной грамматики APG):
   а) x↑:=y(y(z[1]))
   б) x↑↑:=z[y(x,x↑)]

Список обозначений

$\mathbb{N}_0$	множество натуральных чисел (с нулем)
$x \bmod y$	остаток от деления x на y
$\emptyset$	пустое множество
$P \times Q$	декартово произведение множеств P и Q
2^Q	булеан множества M
$\{2^Q\}_{fin}$	множество всех конечных подмножеств из M
$\|w\|$	длина цепочки w, 14
ε	пустая цепочка, 14
$\overleftarrow{w}$	цепочка $\mathtt{a}_n \dots \mathtt{a}_1$ ($w = \mathtt{a}_1 \dots \mathtt{a}_n$), 49
Σ^*	множество всех цепочек над алфавитом Σ, 14
Σ^+	множество всех непустых цепочек над алфавитом Σ, 14
$\dashv$	символ конца строки, 45
∇	символ дна стека, 45
$L(\mathcal{A})$	язык, распознаваемый автоматом $\mathcal{A}$, 21
$\mathcal{A}^{\mathtt{q}}$	автомат, полученный из автомата $\mathcal{A}$ заменой начального состояния на состояние $\mathtt{q}$, 23
$\mathtt{s} \sim \mathtt{t}$	эквивалентные состояния автомата, 23
$sep(\mathtt{s}, \mathtt{t})$	наименьшая длина слова, разделяющего состояния, 26
$Clo(\mathtt{q})$	замыкание состояния $\mathtt{q}$, 33
$\models$	переход в следующую конфигурацию МПА, 45
$\models^*$	переход в новую конфигурацию МПА за конечное число шагов, 45
$\checkmark$	команда допуска (в МПА), 48
$\Rightarrow_G$, $\Rightarrow$	непосредственная выводимость (в грамматике G), 57
$\Rightarrow^*_G$, $\Rightarrow^*$	выводимость (в грамматике G), 57
$\Rightarrow^+_G$, $\Rightarrow^+$	нетривиальная выводимость (в грамматике G), 57
$L(G)$	язык, порождаемый грамматикой G, 58
$Ann(G)$	множество аннулирующих символов грамматики, 90
$G_{_M}$	грамматика с маркерами, эквивалентная G, 213

$\lhd$	линейный порядок на узлах дерева вывода, 65
$\doteq, \lessdot, \gtrdot$	отношения предшествования в грамматике, 135, 147
$\leq$	объединение отношений $\doteq$ и $\lessdot$, 143
$\leftarrow$	операция переноса при LR-анализе, 154
$\otimes$	операция свертки при LR-анализе, 154
$\mathcal{A}_G$	LR(0)-автомат грамматики G, 161
$\mathcal{B}_G$	LR(1)-автомат грамматики G, 172
$\mathcal{I}_G$	автомат пунктов грамматики G, 157
LA	язык арифметических выражений, 15
LB	скобочный язык, 14
LD	язык описаний типов, 15
LL	язык списков, 14
LN	язык двоичных чисел, 15
LN'	язык «правильно записанных» двоичных чисел, 15
GA_1–GA_3	грамматики арифметических выражений, 70, 94
GB_1–GB_3	грамматики расстановок скобок, 69
GD	грамматика описаний типов, 70
GL_1–GL_3	грамматики списков, 69
GN_1,GN_2	грамматики двоичных чисел, 70
AGA_2	атрибутная грамматика арифметических выражений, 186
AGD	атрибутная грамматика описаний типов, 188
AGN_1,AGN_2	атрибутные грамматики двоичных чисел, 189
AGT	атрибутная грамматика фрагмента TEX′a, 203
APG	атрибутная грамматика для проверки типов, 228
$STGA_2$	атрибутная грамматика построения синтаксических деревьев арифметических выражений, 196
TGA_2,TGA_3	схемы трансляции арифметических выражений, 206
TGT	схема трансляции фрагмента TEX′a, 204

Предметный указатель

Литература

[CT] *Cooper K. D., Torczon L.* Engineering a Compiler. Morgan Kaufmann, 2004.

[Hand] Handbook of Formal Languages/ Eds. G. Rosenberg, A. Salomaa B.: Springer, 1997. Vol. 1–3.

[Kar] *Karhumaki Ju.* Automata and Formal Languages, Turku, 2005, http://www.math.utu.fi/en/home/karhumak/automata05.pdf

[Sak] *Sakarovitch J.* Elements of Automata Theory. Cambridge Univ. Press, 2006.

[Wi] *Wirth N.* Compiler Construction. Addison-Wesley, 1996.

[АУ1] *Ахо А., Ульман Дж.* Теория синтаксического анализа, перевода и компиляции. Т. 1. Синтаксический анализ. М., 1978.

[АУ2] *Ахо А., Ульман Дж.* Теория синтаксического анализа, перевода и компиляции. Т. 2. Компиляция. М., 1978.

[АСУ] *Ахо А., Сети Р., Ульман Дж.* Компиляторы: принципы, технологии, инструменты. СПб.: Издательский дом «Вильямс», 2001.

[Ги] *Гинзбург С.* Математическая теория контекстно-свободных языков. М., 1970.

[ЙВ] *Йенсен К., Вирт Н.* Паскаль. Руководство для пользователя. М.: Финансы и статистика, 1989.

[Ка] *Карпов Ю. Г.* Теория и технология программирования. Основы построения трансляторов. СПб.: БХВ-Петербург, 2005.

[КП] *Касьянов В. Н., Поттосин И. В.* Методы построения трансляторов. Новосибирск: Наука, 1986.

[Кнут] *Кнут Д.* Искусство программирования. Т. 3. Сортировка и поиск. М.: Диалектика, 2004.

[ЛРС] *Льюис Ф., Розенкранц Д., Стирнз Р.* Теоретические основы проектирования компиляторов. М., 1979.

[ОС] *Опалева Э. А., Самойленко В. П.* Языки программирования и методы трансляции. СПб.: БХВ-Петербург, 2005.

[Сал] *Саломаа А.* Жемчужины теории формальных языков. М., 1986.

[ЧЛ] *Чень Ч., Ли Р.* Математическая логика и автоматическое доказательство теорем. М.: Наука, 1983.

Оглавление

Учебное издание

Замятин Алексей Петрович

Шур Арсений Михайлович

Языки, грамматики, распознаватели

Учебное пособие

Редактор и корректор Т. А. Федорова
Компьютерная верстка А. М. Шур

Оригинал-макет подготовлен
с использованием издательского пакета LaTeX

Темплан 2007 г., поз. 64.
Подписано в печать 07.09.2007. Формат $60\times84\,{}^{1}/_{16}$. Бумага офсетная
Уч.изд. л. 14,2. Усл. печ. л. 14,4. Тираж 300 экз. Заказ
Издательство Уральского университета. 620083, Екатеринбург, пр. Ленина, 51.
Отпечатано в ИПЦ «Издательство УрГУ». 620083, Екатеринбург, ул. Тургенева, 4.