Ханспетер Мёссенбёк
Предисловие Никлауса Вирта

Конструирование компиляторов

Основания и приложения

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Ханспетер Мёссенбёк

Конструирование компиляторов

Hanspeter Mössenböck

Compiler Construction

Fundamentals and Applications

Springer

Ханспетер Мёссенбёк

Конструирование компиляторов

Основания и приложения

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2025

УДК 004.4'422
ББК 32.972
М53

Мёссенбёк Х.
М53 Конструирование компиляторов / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2025. – 276 с.: ил.

ISBN 978-5-93700-391-1

В книге рассматриваются практические основы конструирования компиляторов – от лексического и синтаксического анализа до семантической обработки и генерирования кода. В качестве сквозного примера описан и реализован компилятор простого Java-подобного языка программирования (MicroJava). Навыки конструирования компиляторов найдут применение всюду, где есть структурированные входные данные, которые можно описать грамматикой, – от простых наборов команд до системных журналов и файлов конфигурации.

Издание ориентировано на студентов факультетов информатики и смежных дисциплин, а также на программистов-практиков, которые хотят применять базовые методы компиляции в повседневной работе.

УДК 004.4'422
ББК 32.972

First published in the German language edition: “Compilerbau” by Hanspeter Mössenböck, © dpunkt.verlag 2024.

ISBN 978-3-031-84812-4 (англ.)

ISBN 978-5-93700-391-1 (рус.)

Содержание

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте *www.dmkpress.com*, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу *dmkpress@gmail.com*; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу *http://dmkpress.com/authors/publish_book/* или напишите в издательство по адресу *dmkpress@gmail.com*.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу *dmkpress@gmail.com*. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты *dmkpress@gmail.com*.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Вступительное слово Никлауса Вирта[1]

Мы не говорим на языках программирования – это формальные системы. Программы, которые транслируют тексты, написанные на языках программирования, в последовательность машинных команд, называются компиляторами. Это сложные программы. На заре развития компиляторов использовались языки Фортран (1957) и Алгол (1960). Над проектированием и реализацией компиляторов для них годами трудились большие коллективы программистов. Однако применение систематического подхода к конструированию компиляторов, описанного в этой книге, позволило сократить сроки до нескольких месяцев работы одного человека. Это был гигантский шаг вперед.

Принципиально новые универсальные языки программирования – редкость в наши дни, как и принципиально новые компьютерные архитектуры. Поэтому конструирование компиляторов кажется специализированной областью, в которой подвизаются немногие специалисты в крупных компаниях. Тогда зачем эта книга?

Причина проста. Любая программа, любое приложение строится по правилам, описывающим, как должны выглядеть предложения и объявления данных. Формализация этих правил делает приложение яснее и понятнее. Ключ ко всему – формализация, т. е. определение синтаксиса входных данных. Она позволяет производить синтаксический анализ, который лежит в основе конструирования компиляторов. Синтаксический анализ и другие методы, описанные в этой книге, полезен не только для обработки языков программирования, его можно применить и ко многим другим задачам, включающим систематическую обработку наборов данных или текстов, и к постоянно расширяющемуся множеству других структурированных входов. Эти методы вносят немалый вклад в правильность и понимание подобных приложений.

Пусть эта книга станет вашим помощником в этом многообещающем начинании!

Цюрих, Швейцария
Декабрь 2023

Заслуженный профессор, д-р Никлаус Вирт

[1] Профессор Никлаус Вирт скоропостижно скончался спустя неделю после того, как написал это вступительное слово, немного не дожив до своего девяностолетия.

Предисловие

Написать компилятор? Да ведь это под силу только таким крупным компаниям, как Microsoft, Google или Oracle, разве не так? Так-то оно так, но чуть ли не каждый специалист по информатике рано или поздно испытывает искушение спроектировать свой собственный язык программирования, пусть даже это всего лишь предметно-ориентированный язык или язык команд очень узкого назначения. И конечно, для этого языка нужен компилятор! Но эта мечта часто остается нереализованной из-за отсутствия знаний о конструировании компиляторов или же потому, что методы, описанные в учебниках, слишком сложны и зачастую касаются в основном таких продвинутых тем, как оптимизация, распределение регистров или детали генерирования кода.

По счастью, основы конструирования компиляторов просты – всякий может изучить их и включить в свой арсенал умений и навыков. Да, компиляторы для таких языков программирования, как Java или C/C++, действительно разрабатываются крупными компаниями, но есть много задач (даже выходящих за рамки собственно конструирования компиляторов), которые можно решить просто и элегантно, пользуясь лишь элементарными методами. В принципе, эти методы применимы всюду, где есть структурированные входные данные, которые можно описать грамматикой. Примерами могут служить простые языки команд, обработка конфигурационных файлов, журналов, списков деталей или рядов данных измерений.

В этой книге показано, как это делается. В ней рассматриваются практические основы конструирования компиляторов – от лексического и синтаксического анализа до семантической обработки и генерирования кода. Из других тем отметим описание процессов трансляции с помощью атрибутных грамматик и использование генератора компиляторов для автоматического генерирования основных частей компилятора. Эти две темы особенно важны на практике, хотя в большинстве книг по компиляторам не освещаются.

Для синтаксического анализа в этой книге используется идея рекурсивного спуска – простая техника нисходящего разбора, которую можно реализовать вручную (т. е. без применения инструментов). Но для полноты картины в конце книги представлен также восходящий синтаксический анализ – метод, более мощный, чем рекурсивный спуск, но и более сложный.

Любой метод можно полностью понять только в контексте конкретного примера, поэтому в качестве сквозного примера мы разработаем небольшой компилятор языка *MicroJava* – простого похожего на Java языка программирования, который транслируется в исполняемый байт-код, напоминающий байт-код Java. Полный исходный код этого компилятора можно скачать с со-

проводительного сайта [Download]. Сам компилятор, а также все примеры в этой книге написаны на Java.

Целевой машиной для компилятора является упрощенная *виртуальная Java-машина* (JVM), точнее *MicroJava Virtual Machine* (μJVM), не перегруженная излишними деталями, но в то же время достаточно реалистичная для иллюстрации и изучения методов генерирования кода. В роли системы команд μJVM выступает байт-код, основанный на байт-коде JVM. Для этого байт-кода предоставляется интерпретатор, позволяющий исполнять программы на компьютере. Изучив процесс генерирования кода, вы заодно узнаете, как работает компьютер, – еще одна причина потратить время на конструирование компилятора.

В конце каждой главы имеются упражнения. Решения некоторых из них можно найти на указанном ниже сайте [Download]. Рекомендуется проработать упражнения самостоятельно, так вы лучше усвоите описанные методы. Но даже если вы не сможете найти время для решения всех 70 с лишним упражнений, хотя бы ознакомьтесь с выборочными решениями, они послужат вам дополнительными примерами.

В основу книги положен курс по конструированию компиляторов, который я уже много лет читаю в университете им. Иоганна Кеплера в Линце и в Оксфордском университете Брукса в Англии. Ее можно использовать как учебник по начальному курсу конструирования компиляторов, за которым может последовать дополнительный курс, охватывающий такие темы, как оптимизация и распределение регистров. На сайте книги имеются также слайды, подготовленные для курса. Но книгу можно использовать и для самообразования, поскольку она ни от чего не зависит и в ней описываются все методы, необходимые для практического построения инструментов, напоминающих компилятор.

Хотел бы воспользоваться возможностью и отдать дань моему бывшему учителю и коллеге профессору Никлаусу Вирту (из Швейцарской высшей технической школы Цюриха), который написал вступительное слово к этой книге. Он был мастером конструирования компиляторов, у которого я научился многим методам. Также выражаю признательность издательству Springer-Verlag за поддержку этого проекта и д-ру Брайану Кирку, оказавшему огромную помощь в переводе книги с немецкого. Искреннее спасибо Дэвиду Лайтфуту и Полу Риду за помощь в уточнении перевода.

Линц, Австрия — Ханспетер МёссенБёк
Декабрь 2024

Сопроводительный сайт *https://ssw.jku.at/CompilerBook/*:

- слайды к лекциям;
- исходный код компилятора MicroJava;
- решения избранных упражнений;
- дополнительные материалы.

Глава 1

Общие сведения

Компилятор – это инструмент, который транслирует *исходную программу* (например, на языке Java, Pascal или C) в *целевую программу*. Языком целевой программы обычно является машинный код, например команды процессора или байт-код Java. Однако целью трансляции может быть и другой исходный язык. В таком случае компилятор называется *транспилятором*.

Помимо компиляторов в строгом смысле слова, имеются компиляторо-подобные инструменты, которые транслируют любой синтаксически структурированный вход в какую-то другую форму. Например, можно транслировать файл журнала в электронную таблицу, которая содержит фактически ту же информацию, но в форме, более удобной для последующего использования.

Вам, наверное, интересно, для чего могут пригодиться навыки построения компиляторов. Ведь настоящие компиляторы разрабатывают только крупные компании, такие как Microsoft, Google или Oracle. Для изучения компиляторов есть несколько причин:

- компиляторы – одни из самых часто используемых инструментов разработки программного обеспечения. Поэтому хорошо бы понимать, как они устроены и работают;
- изучая компиляторы, мы заодно узнаем, как работают компьютеры на машинном уровне. Мы должны разобраться с системой команд, регистрами, режимами адресации, а также с областями данных, например стеком вызовов и кучей. Это мост между оборудованием и программным обеспечением;
- зная, в какие команды транслируются те или иные языковые конструкции, вы начинаете лучше разбираться в вопросах эффективности программы;
- при конструировании компиляторов мы по необходимости имеем дело с грамматиками. Как и программирование, умение описывать структурированные данные с помощью грамматик является важным орудием в арсенале любого инженера-программиста.

Навыки конструирования компиляторов могут пригодиться и для программной инженерии вообще. И хотя лишь немногие занимаются созда-

нием компиляторов в строгом смысле слова, большинство программистов в какой-то момент сталкиваются с необходимостью разработать нечто похожее на компилятор. Это может быть разбор параметров командной строки, обработка списков деталей, языки описания документов (типа PDF или postscript) или средство статического анализа программ, которое вычисляет такие метрики, как сложность исходного кода. Во всех этих случаях умение конструировать компиляторы оказывается полезным.

В этой книге речь пойдет о конструировании компиляторов в строгом смысле, хотя рассмотренные методы применимы и к программной инженерии вообще. Описана полная реализация компилятора простого Java-подобного языка программирования (*MicroJava*), который генерирует исполняемый байт-код (похожий на байт-код Java). Книга рассчитана на читателей, проявляющих конкретный интерес к этому предмету. Формальная теория рассматривается только в той мере, которая необходима для понимания используемых методов. В книге сознательно опущены тонкие детали построения компиляторов, а включено лишь то, что требуется на практике. В частности, методы оптимизации едва затрагиваются, поскольку они легко могли бы составить полноценную книгу, а интересны лишь людям, пишущим компиляторы промышленного качества. Кроме того, рассматриваются только методы конструирования компиляторов императивных языков. Для функциональных и логических языков программирования необходимы другие методы, по крайней мере отчасти, но в практической программной инженерии они используются не так часто.

Существует много книг, посвященных продвинутому конструированию компиляторов (см., например, [ALSU06, Appe02, Coop22, FCL09, Much97]). В них в основном рассматриваются методы статического и динамического анализа кода, обширная тема оптимизации компиляторов и тонкости генерирования кода и распределения регистров. Эти книги интересны тем, кто хочет разрабатывать полноценные компиляторы для таких языков, как Java или C++. А для начинающих они зачастую слишком сложны.

В этой книге основное внимание уделено основам. Она написана на базе многолетнего курса по конструированию компиляторов и может использоваться в качестве учебника по этому предмету. Однако она также адресована программистам-практикам, которые хотят лучше понять, как работает компилятор, и включить соответствующие навыки в свой багаж знаний.

1.1. Краткая история компиляторов

Первые компиляторы появились в конце 1950-х годов. Тогда конструирование компиляторов было окутано тайной, а уровень ажиотажа можно сравнить с сегодняшним интересом к искусственному интеллекту. В то время лишь немногие что-то знали об этом, и широко известных методов компиляции языков программирования не существовало вовсе. Разработка пер-

вых компиляторов занимала много человеко-лет. Ныне конструирование компиляторов – один из наиболее изученных разделов информатики. Имеются хорошо отработанные методы синтаксического анализа, оптимизации и генерирования кода. В результате студенты могут реализовать (простой) компилятор в течение одного семестра.

В этой главе приводится краткий (и по необходимости неполный) очерк истории конструирования компиляторов, совпадающей с историей развития идей языков программирования. Изложение ведется на примере языков, которые считаются важными вехами и дают представление о развитии методов построения компиляторов в соответствующий период.

Fortran (1957). Одним из первых языков программирования высокого уровня стал Fortran [Back56], название которого является аббревиатурой от FORmula TRANslation. *Джон Бэкус*, работавший тогда в IBM, возглавил команду, которая поставила себе целью заменить язык ассемблера, превалировавший в то время, языком более высокого уровня и реализовать компилятор, который порождал бы машинный код, сравнимый по эффективности с написанными вручную ассемблерными программами. Многие основополагающие методы конструирования компиляторов были разработаны именно в то время. Например, в самом начале было далеко не очевидно, как транслировать арифметическое выражение (скажем, a + b * c) в машинные команды с сохранением стандартного предшествования (приоритета) арифметических действий. Кроме того, предстояло придумать методы трансляции условных предложений и циклов, а также механизмы вызова процедур и доступа к массивам.

1960: Algol. Algol60 [Naur64] стал знаменательным этапом в истории языков программирования. Он был разработан консорциумом специалистов из Европы и США, часть которых работала в университетах, а другая часть – в коммерческих компаниях. Была поставлена цель научиться выражать алгоритмы в ясной, краткой форме, допускающей машинное исполнение. Важными новшествами стали блочная структура с локальными и глобальными переменными, а также идея рекурсии, благодаря которой процедуры могли сохранять значения переменных в процессе рекурсивных вызовов. Algol60 был замечателен еще и тем, что стал первым языком программирования, синтаксис которого был формально определен грамматикой. *Джон Бэкус* и *Питер Наур* разработали для этой цели форму Бэкуса–Наура (БНФ), которая теперь входит в арсенал любого специалиста по компиляторам. БНФ сама по себе является языком и определяется грамматикой.

1970: Pascal. Преемником Algol60 стал Pascal [JW75], разработанный *Никлаусом Виртом* и *Тони Хоаром*. Хотя Pascal проектировался как простой язык для образовательных целей, он вобрал в себя важные новшества, которые нашли отражение и в конструкции его компилятора. Если в предыдущих языках использовались только предопределенные типы данных, такие как `integer` или `real`, то Pascal позволял программисту определять собственные типы данных, которые затем можно было использовать в объявлениях переменных. В дополнение к массивам – таблицам однородных данных появи-

лись записи (называемые также *структурами*), позволявшие группировать данные разных типов в новый тип. Хотя идея записей уже встречалась в языке Cobol, записи как тип данных появились только в Pascal. Для доступа к элементам таких структурных типов данных понадобилось разработать новые методы компиляции. Кроме того, первые компиляторы Pascal генерировали не машинный код, а так называемый «P-код», напоминающий современный байт-код Java. P-код был гораздо компактнее машинного кода, а это означало, что программы на Pascal можно было исполнять на тогдашних микрокомпьютерах с очень ограниченным объемом памяти. Эта идея способствовала распространению Pascal, но у нее был недостаток – P-код не мог быть выполнен непосредственно процессором, а должен был интерпретироваться, что замедляло работу программы.

1985: C++. C++ [Stro85] стал преемником языка C, который возник примерно в то же время, что и Pascal, и тоже воспринял идеи Algol60. В C++ соединились многие концепции языков программирования и построения компиляторов. Среди прочего были включены такие концепции, как *объектная ориентированность*, *обработка исключений* и *обобщенные* (т. е. параметризуемые) *типы данных*, которые уже существовали в других языках программирования, но популярность обрели только с появлением C++. Объектная ориентированность потребовала новых методов конструирования компиляторов, в частности представления иерархий наследования и трансляции вызовов виртуальных методов.

1995: Java. Java [AGH00] тоже был не совершенно новым языком, а конгломератом концепций существующих языков. Однако новым стал принцип *своевременной компиляции* (JIT-компиляции), благодаря которому Java-программы были переносимыми. Java-программа транслируется в команды байт-кода (похожего на Pascal'евский P-код), которые могут быть выполнены на любом компьютере, для которого имеется интерпретатор байт-кода. Однако наиболее часто выполняемые части программы транслируются в машинный код соответствующего компьютера во время выполнения (т. е. своевременно).

2005: Scala. Scala [Oder08] – язык, производный от Java, в качестве среды его выполнения выступает платформа Java (т. е. интерпретатор байт-кода и JIT-компилятор). Как и многие современные языки, Scala соединяет идеи функциональных языков с идеями императивного программирования. Функциональные языки включают *лямбда-выражения* (функции, которые рассматриваются как данные и могут передаваться другим функциям в качестве параметров), *ленивое выполнение* (откладывание вычисления выражения до того момента, когда в нем возникает необходимость) и *сопоставление с образцом* (распознавание образцов в последовательностях или древовидных структурах). Разумеется, для этих механизмов понадобились специальные методы конструирования компиляторов.

История языков программирования и компиляторов далека от завершения. Даже сегодня изобретаются новые концепции, которые ведут к новым методам компиляции. Но они выходят далеко за рамки этой книги, цель

которой – познакомить с фундаментальными методами конструирования компиляторов. Читатели, интересующиеся историей, могут найти информацию в трудах конференций по *истории языков программирования* ([HOPL-I, HOPL-II, HOPL-III]).

1.2. Динамическая структура компилятора

В качестве первого знакомства с принципами работы компиляторов рассмотрим их динамическую структуру, т. е. этапы компиляции (рис. 1.1).

Лексический анализ. Лексический анализ – первый этап компилятора. На нем поток литер исходной программы преобразуется в поток лексем. При этом отбрасываются незначащие литеры (например, пробелы), а остальные литеры объединяются в символы (например, имена, числа или операторы), которые называются *лексемами*. Каждому виду лексем сопоставляется уникальный код (например, 1 – идентификаторы, 2 – числа и т. д.). Для некоторых лексем имеется только код (например, для идентификации оператора `"*"` вполне достаточно кода лексемы. Для других лексем одним кодом не обойтись (например, существуют разные идентификаторы, скажем `"val"` и `"i"`). Поэтому в дополнение к коду (например, 1 = идентификатор) нам нужно еще и значение лексемы (например, `"val"`). Каждая лексема является объектом, который характеризуется кодом и факультативным значением. Поток лексем – абстракция потока литер, она упрощает последующие этапы компилятора. Часть компилятора, отвечающая за лексический анализ, называется *лексическим анализатором*, или *сканером*.

Синтаксический анализ. Следующий этап компилятора называется синтаксическим анализом. На нем поток лексем анализируется с точки зрения грамматики исходного языка и строится синтаксическое дерево, отражающее синтаксическую структуру исходной программы. Если все прошло без ошибок, то программа синтаксически правильна, и компилятор может переходить к следующему этапу. В противном случае имеется синтаксическая ошибка, компилятор сообщает о ней, а программист должен ее исправить и перекомпилировать программу. Часть компилятора, отвечающая за синтаксический анализ, называется *синтаксическим анализатором*, или *парсером*.

Семантический анализ. На этапе синтаксического анализа проверяется только синтаксическая правильность программы. Другие условия, например что все идентификаторы должны быть объявлены до использования или что типы данных операндов в операциях присваивания и в выражениях должны быть совместимы, проверяются на этапе семантического анализа. На этом этапе также строится *таблица символов*, в которой хранятся все объявленные имена и их свойства. Синтаксическое дерево и таблица символов образуют

следующую абстракцию исходной программы – промежуточное представление программы внутри компилятора.

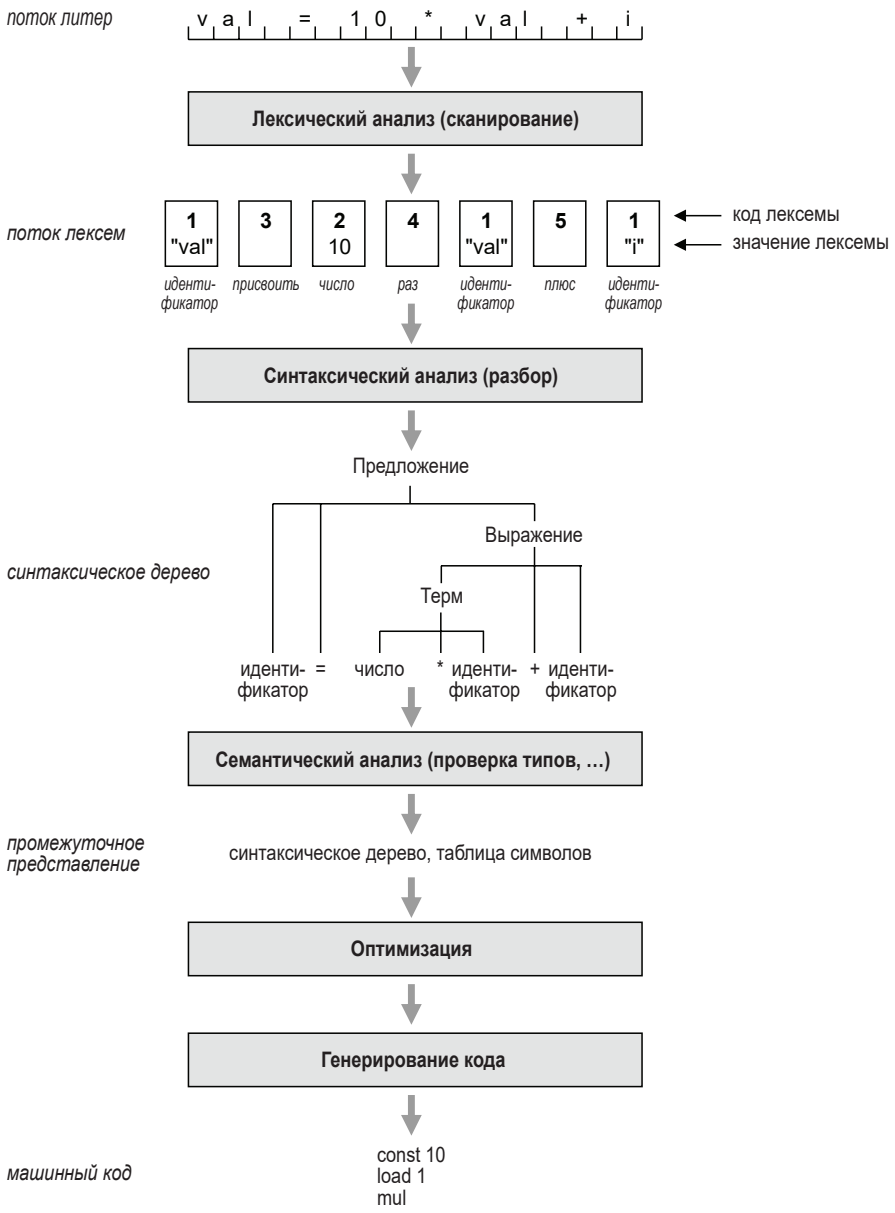


Рис. 1.1 ❖ Динамическая структура компилятора

Оптимизация. За семантическим анализом обычно следует этап оптимизации, на котором программа преобразуется в более быструю или в более компактную. Оптимизация – обширная тема, которой посвящены целые

книги, она важна для компиляторов промышленного качества. В этой книге оптимизация сознательно не рассматривается. Нас интересуют только базовые методы построения простых компиляторов или подобных им инструментов.

Генерирование кода. Последний этап компилятора – генерирование кода, когда по промежуточному представлению программы порождается код для целевой платформы.

1.2.1. Однопроходные и многопроходные компиляторы

Этапы компилятора можно объединять или выполнять строго последовательно. В первом случае компилятор называется однопроходным, а во втором – многопроходным.

В *однопроходном компиляторе* (рис. 1.2) сканер отдает следующую лексему, а парсер проверяет, допускает ли грамматика ее нахождение в текущей позиции. Затем лексему обрабатывает семантический анализатор, например проверяет типы, после чего кодогенератор порождает для нее машинный код. Далее цикл начинается сначала – и так, пока не будет откомпилирована вся программа.

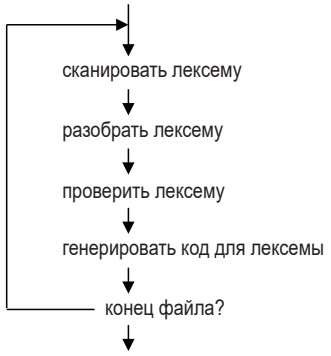


Рис. 1.2 ❖ Однопроходный компилятор

В *многопроходном компиляторе* (рис. 1.3) этапы следуют друг за другом. Сканер читает исходный файл и генерирует поток лексем, парсер проверяет синтаксическую правильность этого потока и порождает синтаксическое дерево, семантический анализатор выполняет дальнейшие проверки и строит таблицу символов, после чего кодогенератор порождает код для целевой платформы.

В прошлом многопроходные компиляторы часто были необходимы, потому что объем компьютерной памяти был невелик или потому что язык был настолько сложен, что казалось обязательным разбивать компилятор

на части. Ныне обе причины утратили актуальность, поэтому современные компиляторы обычно однопроходные. Однако для оптимизирующих компиляторов часто выбирается двухпроходный подход, когда фаза *анализа* (frontend) содержит сканер, парсер, семантический анализатор и простой кодогенератор, порождающий внутреннее представление, которое затем транслируется в целевое на фазе *синтеза* (backend) (рис. 1.4).

Рис. 1.3 ❖ Многопроходный компилятор

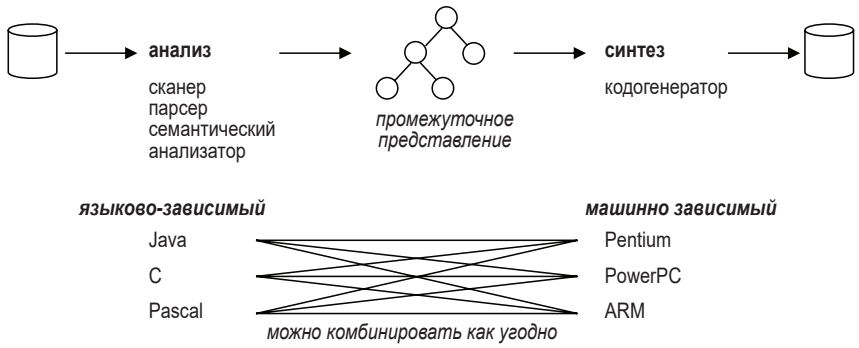


Рис. 1.4 ❖ Двухпроходный компилятор с промежуточным представлением

Фаза анализа языково-зависима, потому что для реализации различных языков, таких как Java, C или Pascal, нужны разные сканеры и парсеры. С другой стороны, фаза синтеза машинно зависима, потому что для различных процессоров, таких как Pentium, PowerPC или ARM, нужны разные кодогенераторы. Однако все анализаторы порождают одно и то же промежуточное представление, а все синтезаторы преобразуют одно и то же представление в код для соответствующей целевой платформы.

У такой декомпозиции есть несколько преимуществ. С одной стороны, она обеспечивает лучшую переносимость. Если мы хотим реализовать компилятор для нового языка, скажем C#, то нужно лишь написать для него фазу анализа. И тогда мы сразу получим компилятор C# для всех платформ, для которых существует синтезатор. Вообще, любую фазу анализа можно сочетать с любой фазой синтеза для получения компиляторов с разных исходных языков на разные целевые платформы. Но самое главное преимущество состоит в том, что оптимизацию гораздо проще осуществить для промежуточного представления, чем для исходного языка. Поэтому почти все оптимизирующие компиляторы двухпроходные.

Недостаток двухпроходных компиляторов в том, что они медленнее и потребляют больше памяти, чем однопроходные, потому что промежуточное

представление строится в памяти. Впрочем, для современных быстрых компьютеров с большим объемом памяти это обычно не проблема. Поскольку мы в этой книге не будем касаться оптимизации, наш компилятор MicroJava будет однопроходным.

1.2.2. Компилятор и интерпретатор

Компилятор транслирует исходную программу в машинный код, который затем можно загрузить и выполнить (рис. 1.5).

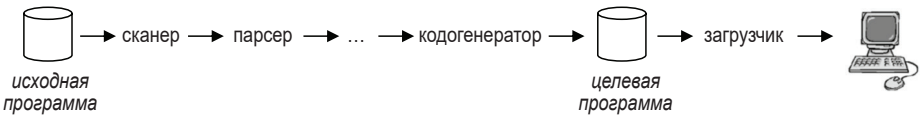


Рис. 1.5 ❖ Компилятор

С другой стороны, интерпретатор выполняет программу «непосредственно», не транслируя ее предварительно в машинный код. Однако сначала программу все равно нужно проанализировать, т. е. необходимы по меньшей мере сканер и парсер, которые распознают структуру программы. Но после того как структура известна, программу можно выполнить, т. е. интерпретировать (рис. 1.6).

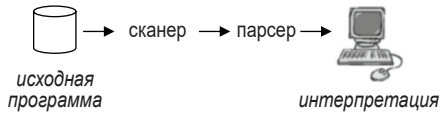


Рис. 1.6 ❖ Интерпретатор

При наличии интерпретатора полная компиляция не нужна. У пользователей складывается впечатление, что их программы выполняются немедленно. Однако интерпретация значительно медленнее, чем выполнение откомпилированной программы. Например, предложения, составляющие цикл, должны заново обрабатываться сканером и парсером на каждой итерации цикла. Поэтому многие языки (например, Java и MicroJava) применяют гибридный подход: компилятор генерирует не машинный код, а код для «виртуальной машины» (ВМ). Например, Java-программы транслируются в *байт-код* – простой формат команд, который затем может быть интерпретирован виртуальной машиной Java (рис. 1.7).

При таком подходе каждое предложение исходной программы нужно проанализировать и транслировать только один раз. Кроме того, интерпретация байт-кода эффективнее интерпретации исходного кода. ВМ «имитирует» фи-

зическую машину, исполняя байт-код вместо машинного кода. Достоинство еще и в том, что байт-код можно выполнить на любой машине, для которой имеется соответствующий интерпретатор. Это позволяет переносить программу на любой компьютер, который может выполнить ВМ.

Рис. 1.7 ❖ Интерпретация промежуточного кода

1.3. Статическая структура компилятора

Статическая структура компилятора описывает отношения между его компонентами (т. е. классами). В компиляторах промышленного качества есть много таких компонентов, но самые важные, встречающиеся в любом компиляторе (в т. ч. MicroJava), показаны на рис. 1.8.

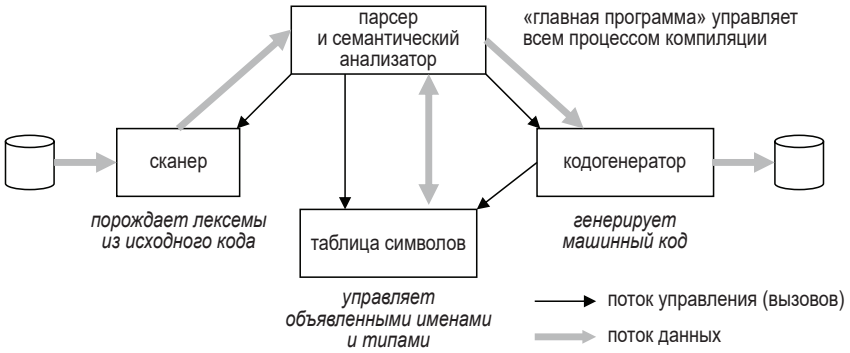


Рис. 1.8 ❖ Статическая структура компилятора

Парсер, объединенный с семантическим анализом, берет на себя роль главной программы и управляет всем процессом трансляции. Когда ему нужна лексема, он вызывает сканер, который выделяет следующую лексему из исходного кода и передает ее парсеру для синтаксического и семантического анализов. Объявленные имена и их свойства парсер помещает в таблицу символов, оттуда же он получает их по мере необходимости. Наконец, парсер вызывает методы кодогенератора для генерирования команд целевого кода.

Компилятор MicroJava устроен в соответствии с этой схемой. Сканер мы будем рассматривать в главе 2, парсер – в главе 3, таблицу символов – в главе 5 и, наконец, кодогенератор – в главе 6.

1.4. Грамматики

Языки программирования, как и естественные языки, обладают синтаксической структурой, которую можно описать грамматикой. Грамматика состоит из правил, которые описывают, как соотносятся друг с другом различные языковые конструкции. В примере ниже показано правило, описывающее структуру предложения `while`:

```
WhileStatement = "while" "(" Condition ")" Statement.
```

Предложение `while` начинается ключевым словом `"while"`, за которым следует условие в скобках и предложение, представляющее тело цикла. В общем случае грамматики состоят из следующих четырех частей.

- **Терминальные символы**: символы (лексемы), которые невозможно разбить на более мелкие; можно сказать, что они представляют «атомы» языка. К ним относятся ключевые слова, такие как `"while"` или `"if"`, операторы, такие как `"+"` или `"-"`, специальные символы, такие как `";"` или `","`, и, наконец, символы, такие как имена или числа, атомарные с точки зрения грамматики.
- **Нетерминальные символы**: символы, представляющие более крупные языковые конструкции, которые подлежат дальнейшему разложению на терминальные и нетерминальные символы. К ним относится, например, нетерминальный символ `WhileStatement` из предыдущего примера, а также `Condition` и `Statement`.
- **Продукции**: грамматические правила, которые описывают декомпозицию нетерминальных символов на терминальные и нетерминальные символы. В примере выше показан пример продукции (и ее декомпозиции) для нетерминального символа `WhileStatement`.
- **Начальный символ**: нетерминальный символ верхнего уровня, из которого выводится все остальное (т. е. весь язык). Совместно терминальные и нетерминальные символы образуют *алфавит* языка, описываемого грамматикой, т. е. множества символов, из которых состоит грамматика.

1.4.1. Нотация РБНФ для грамматик

Для записи грамматик существуют различные нотации. В этой книге мы будем пользоваться нотацией РБНФ (*расширенная форма Бэкуса–Наура*) [Wirt77], названной так в честь пионеров компиляторостроения *Джона Бэкуса* и *Питера Наура*. Это обобщение чистой БНФ, к которой мы вернемся в разделе 1.5.

Продукция РБНФ состоит из левой части и правой части, которые разделены знаком равенства. Каждая продукция завершается точкой:

```
WriteStatement = "write" ident "," Expression ";" .
```

В левой части находится нетерминальный символ. Правая часть состоит из последовательности терминальных и нетерминальных символов. Терминальными символами могут быть имена (например, `ident`) или литералы (например, `"write"`, `","`, `";"`), обозначающие сами себя. Нетерминальными символами всегда являются имена. По соглашению, терминальные символы записываются строчными буквами (например, `ident`), а нетерминальные начинаются с заглавной буквы (например, `Expression`). В нотации РБНФ правая часть продукции может также содержать метасимволы, которые разделяют альтернативы, представляют факультативные или итеративные части либо группируют несколько альтернатив, заключенных в скобки:

Метасимвол	Цель	Пример	Семантика
|	разделить альтернативы	a | b | c	a или b или c
(…)	сгруппировать альтернативы	a | (b | c)	ab или ac
[…]	факультативно	[a] b	ab | b
{…}	повторение (0..бесконечно много раз)	{a} b	b | ab | aab | aaab | …

1.4.2. Пример: грамматика арифметических выражений

В качестве примера рассмотрим РБНФ-грамматику арифметических выражений, в которой имена и числа (т. е. *операнды*) встречаются в сочетании с арифметическими операторами типа `"+"` и `"*"`:

```
Expr = ["+" | "-"] Term {("+" | "-") Term} .
Term = Factor {("*" | "/") Factor} .
Factor = ident | number | "(" Expr ")" .
```

Выражение (`Expr`) начинается факультативным *знаком* (`"+"` или `"-"`), за которым может следовать один или несколько *термов* (`Term`), разделенных знаками `"+"` или `"-"`. Терм состоит из одного или нескольких *множителей* (`Factor`), разделенных знаками `"*"` или `"/"`. Наконец, множитель – это имя (`ident`), число (`number`) или выражение в скобках. Отметим, что круглые скобки в `Factor` – терминальные символы, встречающиеся во входном языке (поэтому они заключены в кавычки), тогда как круглые скобки в продукциях `Expr` и `Term` – метасимволы, которые просто группируют альтернативы.

Грамматики можно также представлять графически *синтаксическими диаграммами* (рис. 1.9), на которых символы грамматики соединены линиями, проследовав по которым, можно определить, какие символы выводятся из нетерминального символа. Такие диаграммы легко читать, но они занимают много места, а их машинная обработка затруднена. Поэтому мы в этой книге будем использовать текстовую нотацию грамматик.

В приведенной ниже грамматике нетерминальными символами являются `Expr`, `Term` и `Factor`. Терминальные символы можно подразделить на *простые терминальные символы* (`"+"`, `"-"`, `"*"`, `"/"`, `"("`, `")"`), каждый из которых имеет

единственное представление, и *терминальные классы* (`ident`, `number`), которые могут иметь несколько представлений (например, `x`, `y`, `sum` – это всё имена, а `1`, `10`, `355` – числа). Начальным символом этой грамматики является `Expr`.

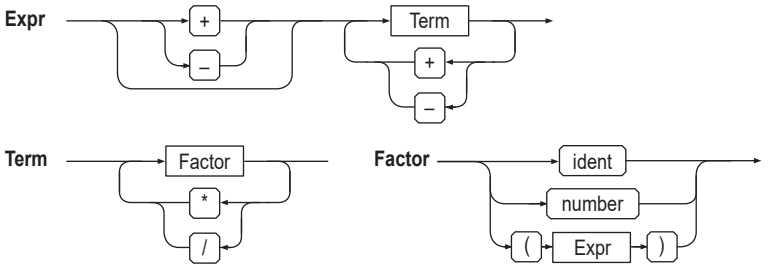


Рис. 1.9 ❖ Представление грамматики синтаксическими диаграммами

Грамматики можно использовать также для записи *правил предшествования* операторов. Например, в процессе анализа выражения

```
- a * 3 + b / 4 - c
```

согласно приведенной выше грамматике поток литер сначала преобразуется в терминальные символы, которые затем постепенно объединяются в нетерминальные (рис. 1.10).

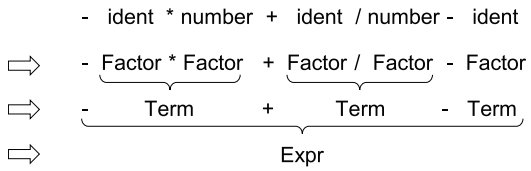


Рис. 1.10 ❖ Анализ выражения `- a * 3 + b / 4 - c`

Как видим, знак (`"-"`) здесь относится ко всему терму (`a * 3`), а не к одной лишь переменной `a`. Чтобы это изменить, нужно переместить факультативный символ знака в продукцию `Factor`, так чтобы `- ident` включался в `Factor` до того, как `Factor * Factor` будет распознано как `Term`:

```
Expr = Term {("+" | "-") Term}.
Term = Factor {("*" | "/") Factor}.
Factor = ["+" | "-"] (ident | number | "(" Expr ")").
```

Таким образом, на правила предшествования можно влиять, изменяя грамматику. Вообще, операторы на более низких уровнях грамматики предшествуют операторам на более высоких уровнях. Так, в приведенной выше грамматике унарный знак в продукции `Factor` имеет больший приоритет, чем бинарные операторы `"*"` и `"/"` в продукции `Term`, а те, в свою очередь, приоритетнее бинарных операторов `"+"` и `"-"` в продукции `Expr`.

1.4.3. Терминальные начальные символы нетерминальных символов

Для синтаксического анализа важно знать *терминальные начальные символы* нетерминальных символов, т. е. терминальные символы, которыми может начинаться нетерминальный. В нашей оригинальной грамматике

```
Expr = ["+" | "-"] Term {("+" | "-") Term}.
Term = Factor {("*" | "/") Factor}.
Factor = ident | number | "(" Expr ")".
```

в продукции `Factor` имеется три альтернативы. Первая начинается с `ident`, вторая – с `number`, а третья – с `"("`. Поэтому терминальными начальными символами `Factor` являются:

```
First(Factor) = ident, number, "("
```

Продукция `Term` начинается символом `Factor`, терминальные начальные символы которого нам уже известны. Поэтому

```
First(Term) = First(Factor) = ident, number, "("
```

Наконец, продукция `Expr` может начинаться знаком (`"+"` или `"-"`). Но поскольку знак необязателен, она может также начинаться символом `Term`, для которого мы уже знаем терминальные начальные символы:

```
First(Expr) = "+", "-", First(Term) = "+", "-", ident, number, "("
```

1.4.4. Терминальные последующие символы нетерминальных символов

Помимо терминальных начальных символов, парсеру необходимо знать *терминальные последующие символы* нетерминальных символов, т. е. терминальные символы, которые могут следовать за нетерминальным в любом контексте. Чтобы определить терминальные последующие символы `Expr`, мы должны посмотреть, когда символ `Expr` встречается в правой части продукции и какие терминальные символы могут следовать за ним. `Expr` встречается в продукции `Factor`, где за ним следует `")"`. И поскольку `Expr` является верхнеуровневым нетерминальным символом нашей грамматики (т. е. начальным символом), за ним также может следовать специальный символ `eof` (*конец файла*), обозначающий конец входного потока (т. е. выражения). Итак, терминальными последующими символами `Expr` являются:

```
Follow(Expr) = ")", eof
```

Нетерминальный символ `Term` встречается в двух местах продукции `Expr`. За первым вхождением следует повторение (`{...}`). Если мы вошли внутрь

повторения, то за ним будет следовать "+" или "–". Однако возможно и нуль повторений; в этом случае за Term следуют последующие символы повторения, т. е. последующие символы Expr, которые нам уже известны:

```
Follow(Term) = "+", "–", Follow(Expr) = "+", "–", ")", eof
```

Такая же ситуация складывается для нетерминального символа Factor, который встречается в двух местах продукции Term. За первым вхождением следует повторение, которое может начинаться с "*" или "/". Если повторение пропущено, то за Factor будут следовать терминальные последующие символы Term, которые мы уже знаем:

```
Follow(Factor) = "*", "/", Follow(Term) = "*", "/", "+", "–", ")", eof
```

1.4.5. Еще о терминологии формальных языков

С теоретической точки зрения, языки программирования являются формальными языками. И хотя в этой книге мы ограничиваемся только теми положениями теории, которые нужны нам для практического конструирования компилятора, кое-какую терминологию знать необходимо.

Мы уже упоминали, что множество терминальных и нетерминальных символов грамматики образует ее *алфавит*.

Термином *строка* описывается конечная последовательность терминальных или нетерминальных символов алфавита. Строки обозначаются греческими буквами. Ниже приведены примеры строк, образованных символами алфавита нашей грамматики Expr:

```
α = ident + number
β = – Term + Factor * number
```

Пустая строка, не содержащая ни одного символа, обозначается ε.

Если заменить нетерминальный символ в строке α правой частью его продукции, то получится новая строка β. Это называется *непосредственным выводом* и записывается $\alpha \Rightarrow \beta$. В примере ниже Factor заменен ident:

```
– Term + Factor * number ⇒ – Term + ident * number
```

Если вывод содержит несколько промежуточных стадий:

$$\alpha \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_n \Rightarrow \beta$$

то говорят, что вывод *нетривиальный*, и пишут $\alpha \Rightarrow^* \beta$. Если в выводе $\alpha \Rightarrow \beta$ заменяется самый левый нетерминальный символ, то говорят о *левостороннем выводе*, а если заменяется самый правый нетерминальный символ, то о *правостороннем выводе*.

Противоположность вывода называется *сверткой*. Если в строке β найдена последовательность символов, соответствующая правой части продукции,

и если эта последовательность заменена соответствующим нетерминальным символом, то говорят, что строка β свернута в строку α.

Парсеры обрабатывают текст либо *сверху вниз* (они называются *нисходящими*), выводя предложение языка из начального символа грамматики (см. раздел 3.2), либо *снизу вверх* (*восходящие* парсеры), сворачивая предложение языка в начальный символ (см. раздел 8.1).

Строка, которую можно вывести непосредственно или нетривиально из нетерминального символа, называется *фразой* этого символа. Например, следующие строки выводятся из `Term` и, стало быть, являются фразами `Term`:

```
Factor
Factor * Factor
ident * Factor
```

Фраза, выведенная из начального символа, называется *сентенциальной формой*. Например, из `Expr` можно вывести следующие сентенциальные формы:

```
Term + Term - Term
Term + Factor * ident - Term
...
```

Если сентенциальная форма состоит только из терминальных символов, то она называется *сентенцией* грамматики. Например, сентенциями нашей грамматики `Expr` являются:

```
ident * number + ident
number * ( ident + ident )
...
```

Все сентенции, которые можно вывести из начального символа грамматики, образуют (*формальный*) *язык* этой грамматики. Так, языком MicroJava является множество всех допустимых программ на MicroJava. Обычно число таких сентенций бесконечно.

В формальных языках имеется также концепция *удаляемости*. Строка α называется удаляемой, если из нее можно вывести пустую строку ($\alpha \Rightarrow^* \varepsilon$). В грамматике

```
X = Y Z.
Y = [b].
Z = c | d | .
```

строка `Y` удаляема, потому что `b` факультативен, и, следовательно, из `Y` можно вывести пустую строку. Продукция `Z` состоит их трех альтернатив, последняя из которых пуста; таким образом, из `Z` также можно вывести пустую строку, поэтому она удаляема. Так как `Y` и `Z` удаляемы, из `X` также можно вывести пустую строку, поэтому она тоже удаляема.

1.4.6. Рекурсия

Термин *рекурсия* (ссылка на себя) известен из математики и имеет важное значение в грамматиках. Продукция нетерминального символа X называется рекурсивной, если из X можно вывести строку, которая сама содержит X (X ⇒* ω_1 X ω_2). Строки ω_1 и ω_2 могут быть пустыми, что дает три возможные формы рекурсии:

Левая рекурсия X = b | X a. X ⇒ Xa ⇒ Xaa ⇒ Xaaa ⇒ ... ⇒ baaaaa
Правая рекурсия X = b | a X. X ⇒ aX ⇒ aaX ⇒ aaaX ⇒ ... ⇒ aaaaab
Центральная рекурсия X = b | "(" X ")". X ⇒ (X) ⇒ ((X)) ⇒ (((X))) ⇒... ⇒ (((((b)))))

В случае левой рекурсии из нетерминального символа можно вывести строку, которая начинается тем же символом (ω_1 пуста), а в случае правой рекурсии – строку, которая заканчивается тем же символом (ω_2 пуста). Как видно из примеров выше, это можно использовать для выражения повторений. С другой стороны, центральная рекурсия используется для выражения вложенных структур (ω_1 и ω_2 не пусты), когда вхождений ω_1 ровно столько, сколько вхождений ω_2.

Помимо прямой рекурсии, показанной в примерах выше, существует косвенная рекурсия. Следующая упрощенная грамматика Expr

```
Expr = Term {"+" Term}.
Term = Factor {"*" Factor}.
Factor = ident | "(" Expr ")".
```

является косвенно центрально рекурсивной (Expr ⇒* ω_1 Expr ω_2), поскольку из Expr можно за несколько шагов вывести строку, снова содержащую Expr:

```
Expr ⇒ Term ⇒ Factor ⇒ ( Expr )
```

1.4.7. Исключение левой рекурсии

Процессу нисходящего разбора, который мы будем рассматривать в главе 3, левая рекурсия мешает и должна быть исключена. Леворекурсивная продукция

```
X = b | X a.
```

состоит из двух альтернатив. Первая начинается с b, вторая – с X, но терминальным начальным символом X также является b. Поэтому если парсер хочет найти X и встречает во входном потоке a b, то он не может различить альтернативы, т. к. обе начинаются с b.

К счастью, левую рекурсию всегда можно преобразовать в повторение. Из нетерминального символа X можно вывести сентенцию следующим образом:

```
X ⇒ Xa ⇒ Xaa ⇒ ... ⇒ baaa... a
```

Легко видеть, что леворекурсивную продукцию X можно преобразовать в итеративную РБНФ-продукцию, не являющуюся рекурсивной:

```
X = b {a}.
```

Еще пример: следующая леворекурсивная продукция

```
Expr = Term | Expr "+" Term.
```

порождает вывод

```
Expr ⇒ Expr + Term ⇒ Expr + Term + Term ⇒ ... ⇒ Term + Term + ... + Term
```

который, в свою очередь, сворачивается в итеративную РБНФ-продукцию:

```
Expr = Term {"+" Term}.
```

1.4.8. Классификация грамматик по Хомскому

В 1950-х годах американский лингвист *Ноам Хомский* изучал грамматики как подстановочные системы, состоящие из правил α = β, позволяющих вывести строку β из строки α. В зависимости от форм α и β он различал четыре класса грамматик.

Класс 0: неограниченные грамматики. α и β могут быть произвольными строками, составленными из терминальных и нетерминальных символов, например:

```
X = a X b | Y c Y.
a Y c = d.
d Y = b b.
```

Здесь из X можно вывести сентенцию следующим образом:

```
X ⇒ aXb ⇒ aYcYb ⇒ dYb ⇒ bbb
```

Неограниченные грамматики – самый мощный класс, потому что могут порождать языки по сложным правилам. Однако общего алгоритма анализа таких грамматик не существует. Такие языки распознаются *машинами Тьюринга*.

Класс 1: контекстно-зависимые грамматики. Должно выполняться условие $|\alpha| \leqslant |\beta|$. Левая часть может быть строкой любой длины, например

```
a X = a b c.
```

но не должна содержать больше символов, чем правая часть. Как видим, здесь принимается во внимание контекст нетерминального символа: из X можно вывести b c, только если ему предшествует a. Контекстно-зависимые грамматики можно использовать для порождения языков, распознаваемых *линейно ограниченными автоматами* (разновидность машины Тьюринга).

Класс 2: контекстно-свободные грамматики. В этом случае строка α состоит из одного нетерминального символа, а строка β может быть любой, например

```
X = a b c.
```

В принципе, требуется также, чтобы β была непустой, но можно показать, что продукции, в которых β пуста, можно преобразовать в продукции с непустой β. Контекстно-свободные грамматики порождают языки, распознаваемые *автоматами с магазинной памятью* (см. главы 3 и 8).

Класс 3: регулярные грамматики. Здесь α снова состоит из одного нетерминального символа, но β может быть либо терминальным символом, либо терминальным символом, за которым следует нетерминальный, например

```
X = b.
X = b Y.
```

Регулярные грамматики порождают языки, распознаваемые *конечными автоматами* (см. главу 2).

С точки зрения компиляторов, интересны только контекстно-свободные и регулярные грамматики, потому что для их обработки существуют эффективные алгоритмы. Контекстно-свободные грамматики используются в синтаксическом анализе, а регулярные – в лексическом анализе.

1.5. Синтаксические деревья

Парсер анализирует фразу языка, сообразуясь с его грамматикой, и проверяет ее синтаксическую правильность. Это включает построение синтаксического дерева, описывающего разложение сентенции на составные части.

При построении синтаксического дерева проще работать с чистой БНФ (*формой Бэкуса–Наура*), в которой, в отличие от РБНФ, не используются скобки для группировки альтернатив ((...)), представления факультативности ([...]) и представления повторений ({...}). Факультативность необходимо выражать с помощью нескольких альтернатив, а повторение – с помощью левой рекурсии. Наша грамматика `Expr` записывается в виде чистой БНФ следующим образом:

```
Expr   = Sign Term | Expr AddOp Term.
Term   = Factor | Term MulOp Factor.
Factor = ident | number | "(" Expr ")".
Sign   = "+" | "-" | .
AddOp  = "+" | "-". MulOp = "*" | "/".
```

Например, для входной фразы `10 + 3 * x` будет построено синтаксическое дерево, показанное на рис. 1.11.

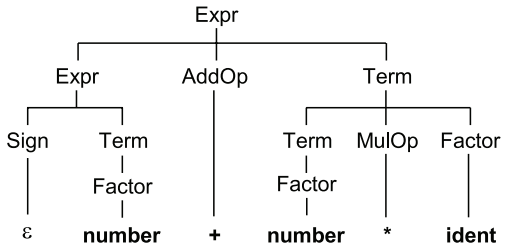


Рис. 1.11 ❖ Конкретное синтаксическое дерево для фразы `10 + 3 * x`

Оно отражает полный вывод от начального символа до нужной сентенции (`Expr` ⇒ `Expr AddOp Term`, `Term` ⇒ `Term MulOp Factor` и т. д.). Такое синтаксическое дерево называется *конкретным* (*деревом разбора*).

Помимо конкретных, существуют *абстрактные синтаксические деревья*, отражающие логическую структуру сентенции; они гораздо компактнее конкретных синтаксических деревьев. Листья абстрактного дерева представляют операнды, а внутренние узлы – операторы (рис. 1.12).

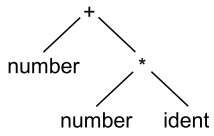


Рис. 1.12 ❖ Абстрактное синтаксическое дерево для фразы `10 + 3 * x`

Абстрактные синтаксические деревья часто используются в оптимизирующих компиляторах для внутреннего представления программ, подлежащих оптимизации.

1.5.1. Неоднозначность

Грамматика неоднозначна, если для некоторой сентенции можно построить несколько (конкретных) синтаксических деревьев. Например, из следующей грамматики (где `T` – сокращение `Term`, а `F` – сокращение `Factor`)

```
T = F | T "/" T.
F = id.
```

можно вывести фразу `id / id / id`:

```
T ⇒ T / T ⇒ T / T / T ⇒ F / F / F ⇒ id / id / id
```

Как видно по рис. 1.13, для этой фразы можно построить два разных синтаксических дерева.

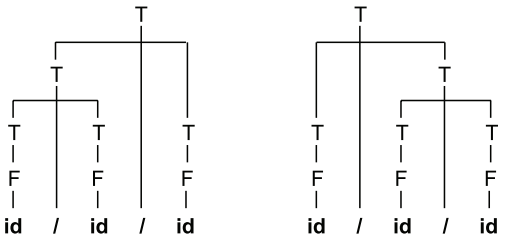


Рис. 1.13 ❖ Различные синтаксические деревья для фразы `id / id / id`

Неоднозначные грамматики непригодны для синтаксического анализа, потому что допускают различные интерпретации сентенции. Например, левое синтаксическое дерево на рис. 1.13 означает, что сначала делятся первые два вхождения `id`, тогда как правое дерево означает, что сначала делятся последние два вхождения.

По счастью, в этом примере неоднозначен не язык, а только грамматика. Если преобразовать грамматику к форме

```
T = F | T"/" F.
F = id.
```

то множество порождаемых сентенций будет тем же самым, но деление всегда выполняется слева направо. Таким образом, неоднозначность устранена. Однако существуют внутренне неоднозначные языки, и даже далеко ходить не придется. Большинство языков семейства C (C, C++, C# и Java) содержат неоднозначность, получившую название *висячее else*. В этих языках предложение `if` может принимать две формы:

```
Statement = "if" Condition Statement
          | "if" Condition Statement "else" Statement
          | ... .
```

В следующем вложенном предложении if:

```
if (a < b) if (b < c) x = c; else x = b;
```

ключевое слово `else` может относиться как к первому `if`, так и ко второму. Это означает, что можно построить два разных синтаксических дерева (см. рис. 1.14), что является неоднозначностью. В данном случае неоднозначен сам язык, а не грамматика. И невозможно преобразовать грамматику таким образом, чтобы эта неоднозначность исчезла.

Эта неоднозначность разрешается путем связывания `else` с непосредственно предшествующим ему `if`. Если парсер анализирует обе альтернативы предложения `if` и встречает во входном потоке `else`, то он обрабатывает вторую альтернативу, вместо того чтобы закончить первую. Таким образом, применяется нижнее синтаксическое дерево на рис. 1.14, но это всего лишь соглашение, которое в действительности не устраняет неоднозначность.

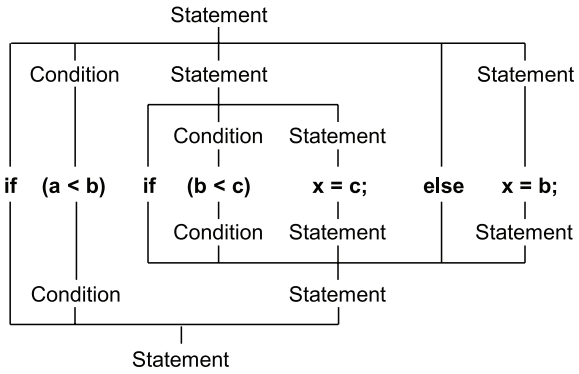


Рис. 1.14 ❖ Неоднозначность из-за «висячего else»

1.6. MicroJava

В этой книге демонстрируются принципы конструирования компиляторов на примере языка программирования *MicroJava*. Ниже приводится краткий обзор этого языка. Более подробное описание его синтаксиса и семантики см. в приложении A.

MicroJava – подмножество Java, достаточно простое, чтобы можно было разработать компилятор в течение односелестрового курса, но в то же время достаточно реалистичное для изучения наиболее важных принципов конструирования компиляторов.

Программа на MicroJava состоит из одного файла, который начинается ключевым словом `program` и может содержать глобальные данные и методы. Главный метод, с которого начинается выполнение программы, называется `main()`. В MicroJava имеются следующие элементы:

- *типы-значения*: `int` и `char`; значения типа `char` хранятся в одном байте (ASCII);
- *ссылочные типы*: это одномерные массивы, а также классы, которые могут содержать поля, но без методов и без наследования. Переменные этих типов содержат ссылки;
- *константы*: константы типа `int` (например, `3`) и типа `char` (например, `'x'`); строковые константы не поддерживаются;
- *переменные*: могут быть локальными в пределах метода или глобальными для всей программы;
- *методы*: глобальны для всей программы; не существует методов, локальных для класса;
- *комментарии*: однострочные комментарии от "//" до конца строки.

Ниже приведен пример программы на MicroJava со всеми основными элементами:

```
program P
   final int size = 10;       // объявление константы
  class Table {              // объявление класса (без методов)
    int[] pos;               // объявление  массива (одномерного)
    int[] neg;
  }
  Table val;                 // объявление глобальной переменной

{
  void main()
    int x, i;                // объявление локальных переменных
    //---------- инициализировать val ------------
    val = new Table;
    val.pos = new int[size];
    val.neg = new int[size];
    i = 0;
    while (i < size) {
      val.pos[i] = 0; val.neg[i] = 0;
      i++;
    }
    //---------- прочитать значения ---------
    read(x);
    while (x != 0) {
      if (0 <= x && x < size) {
          val.pos[x]++;
      } else if (-size < x && x < 0) {
          val.neg[-x]++;
      }
      read(x);
    }
  } // main
} // P
```

Программы на MicroJava транслируются в байт-код MicroJava, который можно выполнить на виртуальной машине MicroJava (см. главу 6).

1.7. Упражнения

Для всех упражнений в этой книге приведены примеры решений, их можно скачать с сопроводительного сайта [Download].

1. *Грамматика адресов электронной почты*. Напишите грамматику адресов электронной почты. Они состоят из адресной и доменной частей, разделенных знаком `"@"`. Обе части должны состоять из списка имен (`ident`), разделенных точками. Адресная часть может содержать одно имя, но доменная должна содержать по меньшей мере два, например:

   ```
   john.doe@some.company.com
   ```

2. *Грамматика упрощенных булевых выражений*. Напишите грамматику булевых выражений, в которой имеются идентификаторы (`ident`) и константы `true` и `false` в качестве операндов, а также операторы `&&`, `||` и `!` (причем `!` имеет больший приоритет, чем `&&`, а `&&` – больший приоритет, чем `||`). Должна быть возможность заключать подвыражения в скобки. За образец возьмите грамматику арифметических выражений из раздела 1.4. Пример:

   ```
   (big || small) && ready || big && ! ready
   ```

3. *Грамматика римских чисел*. Напишите грамматику римских чисел от 1 до 20 (т. е. I, II, III, IV, V, VI, VII, VIII, IX, X, XI, XII, XIII, XIV, XV, XVI, XVII, XVII, XVII, XVIII, XIX, XX). Терминальными символами являются I, V и X.

4. *Терминальные начальные и последующие символы (1)*. Перечислите терминальные начальные и последующие символы для всех нетерминальных символов следующей грамматики (примечание: имена, начинающиеся строчными буквами, являются терминальными символами):

   ```
   Course  = Intro Section {Section} Final.
   Intro   = lecture [questions].
   Section = {lecture | questions} (project | test).
   Final   = [panic] test.
   ```

5. *Терминальные начальные и последующие символы (2)*. Перечислите терминальные начальные и последующие символы для всех нетерминальных символов следующей грамматики:

   ```
   Message = Header [Data] Status.
   Header = "get" | "put" [number | "final"].
   Data = number {number}. Status = "ok" | Number.
   ```

6. *Исключение левой рекурсии (1)*. Преобразуйте следующую леворекурсивную грамматику в эквивалентную нерекурсивную грамматику, используя повторения в РБНФ:

   ```
   A = A B | a b.
   B = B d | c.
   ```

7. *Исключение левой рекурсии (2)*. Преобразуйте следующую леворекурсивную грамматику в эквивалентную нерекурсивную грамматику, используя повторения в РБНФ:

   ```
   List = List number | .
   ```

8. *Синтаксические деревья (1)*. Рассмотрим следующую БНФ-грамматику упрощенных арифметических выражений:

   ```
   Expr   = Term
   Expr   = Expr AddOp Term.
   Term   = Factor
   Term   = Term MulOp Factor.
   ```

```
Factor = number
Factor = "-" Factor
Factor = "(" Expr ")".
AddOp  = "+" AddOp = "-".
MulOp  = "*" MulOp = "/".
```

(a) Нарисуйте конкретное и абстрактное синтаксическое дерево для выражения `3 + 5`.
(b) Нарисуйте конкретное и абстрактное синтаксическое дерево для `(7 - 2) * 5 + 1`.

9. *Синтаксические деревья (2).* Рассмотрим следующую сильно упрощенную грамматику предложений:

```
Statement =
   ident "=" ident "-" ident
| "{" Statement {";" Statement} "}"
| "if" "(" ident ">" ident ")" Statement
| "while" "(" ident ">" ident ")" Statement.
```

Нарисуйте абстрактное синтаксическое дерево для следующей программы

```
if (a > b) { while (a > b) a = a - b; b = b - a }
```

где `"if"`, `"while"`, `"="`, `">"`, `"-"` и `";"` – операторы, а идентификаторы являются операндами. Оператор `";"` конкатенирует два предложения.

10. *Неоднозначность.* Грамматика

```
List = ident | List "," List.
```

неоднозначна, потому что для некоторых сентенций можно построить несколько разных синтаксических деревьев.
(a) Нарисуйте все возможные синтаксические деревья для сентенции `ident, ident, ident`.
(b) Преобразуйте грамматику, так чтобы она порождала тот же язык, но без неоднозначностей.

Глава 2

Лексический анализ

Первым этапом компиляции является лексический анализ (*сканирование*). На нем поток литер исходной программы преобразуется в последовательность терминальных символов, т. е. в поток лексем (рис. 2.1).

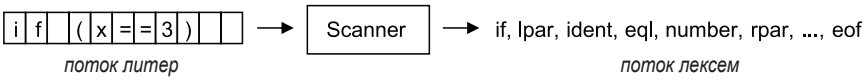


Рис. 2.1 ❖ Преобразование исходной программы в поток лексем

В этом примере литеры `"i"` и `"f"` становятся лексемой `if`, литера `"("` становится лексемой `lpar` (*левая скобка*), `"x"` – литерой `ident`, пара `"="`, `"="` – лексемой `eql`, а `")"` – лексемой `rpar`. Заметим, что поток лексем всегда завершается специальной лексемой `eof`, обозначающей конец ввода.

В главе 1 мы видели, что каждая лексема описывается кодом, определяющим ее вид (например, идентификатор, число, оператор). Лексемы, способные принимать несколько значений (например, идентификаторы), в дополнение к коду имеют еще и значение. В случае идентификатора значением лексемы является текст идентификатора.

Вторая задача лексического анализа – отфильтровать незначащие литеры (пробелы, табуляторы, концы строк и комментарии) из входного потока, поскольку для синтаксического анализа они не нужны. После этого поток лексем будет содержать только существенные символы. Такой поток парсеру проще обработать, чем оригинальный исходный код.

Лексические и синтаксические структуры

Структуру терминальных символов можно описать грамматикой, например:

```
ident  = letter {letter | digit}.
number = digit {digit}.
if     = "i" "f".
eql    = "=" "=".
...
```

Почему эти лексические структуры не рассматриваются как часть синтаксиса языка? Например, почему ident считается терминальным символом, а не нетерминальным символом, который можно разложить на буквы и цифры?

Тому есть несколько причин. Если бы синтаксический анализ производился на уровне символов, то было бы гораздо труднее отличить идентификаторы от ключевых слов. Например, следующую продукцию

```
Statement = ident "=" Expr ";"
          | "if" "(" Expr ")" Statement
          | ... .
```

пришлось бы записывать в виде

```
Statement = "i" ( "f" ( "(" Expr ")" Statement
                      | (letter | digit) {letter | digit} "=" Expr ";"
                      )
                | not_f {letter | digit} "=" Expr ";"
                )
          | not_i {letter | digit} "=" Expr ";"
          | ... .
```

Такую запись понять намного сложнее. Кроме того, пробелы и другие несущественные литеры пришлось бы рассматривать в самых разных местах грамматики, например:

```
Statement = "i" "f" {Blank} "(" {Blank} Expr {Blank} ")" {Blank} Statement {Blank} ... .
Blank     = " " | " \n" | "\t" | Comment.
```

Это привело бы к увеличению размера грамматики и затруднило бы ее чтение. Третья причина в том, что для разбора терминальных символов достаточно регулярных грамматик, анализировать которые эффективнее, чем контекстно-свободные грамматики, используемые в синтаксическом анализе.

По этим причинам лексические структуры распознаются сканером, который предоставляет их в виде потока лексем. Затем синтаксические структуры можно описать простой грамматикой, которая рассматривает лексические структуры как терминальные символы.

2.1. Регулярные грамматики и конечные автоматы

В главе 1 мы определили регулярные грамматики, сказав, что левая часть их продукций должна состоять из единственного нетерминального символа, а правая часть должна состоять из терминального символа, за которым может следовать нетерминальный символ:

```
X = a.
X = b Y.
```

Этим можно воспользоваться, чтобы написать регулярную грамматику для идентификаторов, например:

```
ident = letter
      | letter Rest.
Rest  = letter
      | digit
      | letter Rest
      | digit Rest.
```

Все эти продукции имеют нужную форму. Идентификатор `xy3` можно записать так:

```
ident ⇒ letter Rest ⇒ letter letter Rest ⇒ letter letter digit
```

Но для наших целей достаточно более простого определения регулярных грамматик: грамматика называется *регулярной*, если ее можно выразить *одним* правилом РБНФ *без рекурсии*. Например, грамматику идентификаторов можно выразить следующей нерекурсивной РБНФ-продукцией:

```
ident = letter {letter | digit}.
```

Таким образом, она регулярна. А как насчет следующей грамматики – регулярна она или нет?

```
E = T {"+" T}.
T = F {"*" F}.
F = ident.
```

Она состоит из нескольких продукций, но `F` можно подставить в продукцию `T`:

```
T = ident {"*" ident}.
```

а затем `T` в продукцию `E`:

```
E = ident {"*" ident} {"+" ident {"*" ident}}.
```

С помощью преобразований мы получили грамматику, которая состоит из единственного нерекурсивного правила РБНФ. Таким образом, мы показали, что эта грамматика регулярна. Рассмотрим еще один пример. Регулярна ли следующая грамматика?

```
E = F {"*" F}.
F = ident | "(" E ")".
```

Мы могли бы попробовать тот же прием – подстановку `F` в продукцию `E`:

```
E = (ident | "(" E ")") {"*" (ident | " (" E ")")}.
```

Это приводит к единственному правилу РБНФ, но оно рекурсивно (точнее, центрально-рекурсивно). Невозможно исключить рекурсию, продолжая подставлять `E` в эту продукцию. Мы увидели основное ограничение регулярных грамматик: они не способны справиться с центральной рекурсией.

2.1.1. Ограничения регулярных грамматик

Регулярные грамматики не способны справиться с центральной рекурсией. Иными словами, с их помощью нельзя выразить вложенные структуры. Однако вложенные структуры часто встречаются в языках программирования, например:

```
вложенные выражения:    Expr ⇒* "(" Expr ")"
вложенные предложения:  Statement ⇒ "do" Statement "while" "("Expr")"
внутренние классы:      Class ⇒ "class" "{" ... Class ... "}"
```

Таким образом, для синтаксического анализа языков программирования необходим следующий класс грамматик, а именно контекстно-свободные. Но для лексических структур регулярных грамматик достаточно, например:

```
имена                ident   = letter {letter | digit}.
числа                number  = digit {digit}.
литерные константы   charCon = "\'" noQuote "\'".
ключевые слова       keyword = letter {letter}
операторы            geq     = ">" "=".
```

Единственное исключение – вложенные комментарии:

```
/* ... /* ... */ ... */
```

Их нельзя описать регулярной грамматикой, поэтому сканер должен обрабатывать их специально. Поэтому во многих языках, включая Java, вложенные комментарии запрещены.

2.1.2. Детерминированные конечные автоматы

Детерминированный конечный автомат (ДКА) – это механизм распознавания регулярных языков. Он представляет собой совокупность состояний и переходов между ними (рис. 2.2).

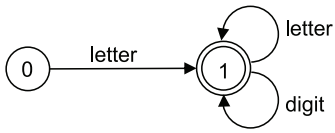


Рис. 2.2 ❖ Детерминированный конечный автомат для распознавания идентификатора

Состояния представлены пронумерованными кружочками, конечные состояния – двойными кружочками. По соглашению, начальное состояние имеет номер 0. Переходы между состояниями представлены помеченными стрелками. Если автомат на рис. 2.2 находится в состоянии 0 и следующей входной литерой является буква, то автомат принимает эту литеру и переходит в состояние 1. Дальнейшие буквы и цифры, поступившие автомату, находящемуся в состоянии 1, тоже принимаются, и ДКА остается в состоянии 1. Если в состоянии 1 поступила литера, отличная от буквы и цифр, то автомат завершает работу. Если автомат завершил работу в конечном состоянии (например, в состоянии 1), то он распознал сентенцию языка (в данном случае – идентификатор). С другой стороны, если бы литера, отличная от буквы, встретилась в состоянии 0, то это было бы ошибкой.

Переходы состояний можно также представить в виде таблицы (рис. 2.3), количество строк которой равно числу состояний, а количество столбцов – числу входных символов. Таблица представляет функцию перехода состояний δ: появление `letter` в состоянии 0 переводит нас в состояние 1, а появление `digit` – в состояние ошибки. Автомат называется «конечным», потому что δ можно описать таблицей конечного размера.

δ	`letter`	`digit`
s0	s1	ошибка
s1	s1	s1

Рис. 2.3 ❖ Функция перехода состояния δ представлена в виде таблицы

Таким образом, ДКА – это пятерка (Q, S, δ, s_0, F), где:

- Q – множество состояний;
- S – множество входных символов;
- $\delta: Q \times S \rightarrow Q$ – функция перехода состояний;
- s_0 – начальное состояние;
- F – множество конечных состояний.

Языком, распознаваемым ДКА, является множество всех последовательностей символов, которые ведут из начального состояния в одно из конечных состояний. Говорят, что ДКА распознал сентенцию, если:

- он находится в конечном состоянии *и*
- вход закончился *или* переход в новое состояние для следующего входного символа невозможен.

Как перейти от регулярной грамматики к ДКА? Мы не станем приводить соответствующий алгоритм, но легко видеть, что ДКА на рис. 2.2 выводится из продукции

```
ident = letter {letter | digit}.
```

Сначала должна быть распознана буква (`letter`), за которой может следовать произвольное число букв или цифр (в т. ч. ни одной). Это в точности поведение нашего ДКА.

2.2. Сканер как детерминированный конечный автомат

Сканер можно рассматривать как большой ДКА, вызываемый парсером для распознавания терминальных символов. При каждом вызове он начинает работу в состоянии 0, пропускает несущественные символы, а затем переходит к подавтомату, распознающему один из терминальных символов, и возвращает этот символ парсеру (рис. 2.4).

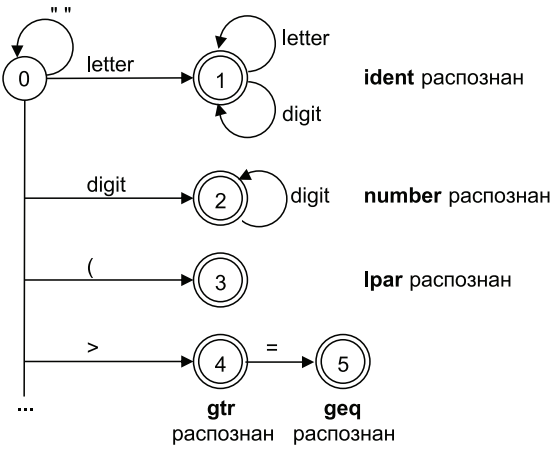


Рис. 2.4 ❖ Сканер как детерминированный конечный автомат (ДКА)

Сканер начинает работу в состоянии 0, переходит в состояние 1 после литеры `"m"` и остается в этом состоянии после литер `"a"` и `"x"`. Поскольку следующая литера – пробел и для нее не существует перехода из состояния 1, ДКА завершает работу. В этом конечном состоянии он распознал лексему `ident` со значением `"max"` и возвращает ее парсеру.

При следующем вызове сканер снова начинает работу в состоянии 0. Поскольку от предыдущего вызова остался пробел, он пропускает его и остается в состоянии 0. Увидев `">"`, он переходит в состояние 4, а после чтения `"="` совершает переход в состояние 5. Следующий символ – пробел, для которого из состояния 5 нет перехода. Поэтому ДКА завершает работу и возвращает парсеру лексему `geq` (*больше или равно*).

При третьем вызове сканер снова пропускает оставшийся пробел в состоянии 0, после чего переходит в состояние 2, увидев литеру `"3"`, и остается в нем

после чтения "0", т. к. это тоже цифра. Если следующая литера – пробел, то ДКА завершает работу (потому что для пробела нет перехода из состояния 2) и возвращает парсеру лексему number со значением 30.

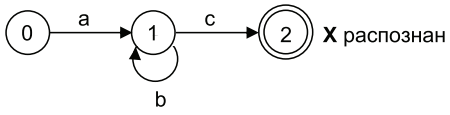
2.3. Реализация ДКА

ДКА может быть реализован как табличный алгоритм, в котором управляющая анализом функция описывается таблицей. Альтернатива – реализовать состояния и переходы прямо в коде.

Мы рассмотрим оба варианта. Сначала рассмотрим табличную реализацию на примере регулярной грамматики

```
X = a {b} c.
```

Этой грамматике соответствует ДКА, показанный на рис. 2.5 вместе с эквивалентной функцией перехода состояний, представленной в виде таблицы.



δ	a	b	c	
0	1	-	-	
1	-	1	2	
2	-	-	-	**X** распознан

Рис. 2.5 ❖ ДКА и его таблицы переходов состояний

Таблицу можно реализовать в виде двумерного массива, в котором ошибки закодированы числом –1. Это приводит нас к следующему алгоритму (на Java-подобном псевдокоде):

```
int[][] delta = { {1, -1, -1}, {-1, 1, 2}, {-1, -1, -1} }; // таблица переходов состояний
int state = 0, lastState;      // ДКА начинает работу в состоянии 0
do {
   char ch = read();           // читать следующую литеру
   lastState = state;
   state = delta[state][ch];   // искать последующее состояние
} while (state != -1);
assert(lastState является конечным состоянием);
return recognizedToken[lastState];
```

Следующая входная литера и текущее состояние используются в цикле для поиска следующего состояния и перехода в него. Цикл продолжается, пока не окажется, что очередной переход невозможен (state == –1). В этот момент ДКА должен находиться в конечном состоянии, иначе возникнет ошибка. Наконец, по таблице recognizedToken определяется распознанная в этом конечном состоянии (lastState) лексема и возвращается парсеру.

Заметим, что этот алгоритм «универсален» в том смысле, что его можно использовать для любой регулярной грамматики. Необходимо только ини-

циализировать таблицу переходов состояний для распознавания лексем грамматики. Однако этот алгоритм не особенно эффективен, поэтому мы рассмотрим другой алгоритм, в котором состояния и переходы состояний закодированы непосредственно.

Взяв в качестве примера ДКА на рис. 2.5, реализуем состояния как метки `case` в предложении `switch`.

```
int state = 0; loop:
for (;;) {
   char ch = read();
   switch (state) {
     case 0: if (ch == 'a') { state = 1; break; }       // перейти в состояние 1
             else break loop;
     case 1: if (ch == 'b') { state = 1; break; }       // перейти в состояние 1
             else if (ch == 'c') { state = 2; break; }  // перейти в состояние 2
             else break loop;
     case 2: return X;
   }
}
return errorToken;
```

Здесь никакие операции доступа к таблице не нужны. Мы просто ветвимся по текущему состоянию в предложении `switch` на каждой итерации цикла. В случае ошибки `break loop` осуществляет выход из цикла.

Но еще лучше вообще абстрагировать состояния и переходы между ними и реализовать ДКА следующим образом:

```
char ch = read();
if (ch == 'a') {
    ch = read();
    while (ch == 'b') ch = read();
    if (ch == 'c') { ch = read(); return X; }
    else return errorToken;
} else return errorToken;
```

Здесь ДКА неявный, но зато код очень эффективен. Поэтому при реализации сканера для MicroJava мы воспользуемся именно таким подходом.

2.3.1. Реализация сканера

Теперь рассмотрим реализацию сканера для MicroJava. Сканер – класс с двумя открытыми методами, объявленными `static`, потому что в компиляторе существует всего один экземпляр сканера.

```
public class Scanner {
  public static void init (Reader r) {...}
  public static Token next() {...}
}
```

Метод `init()` инициализирует сканер и передает ему объект `Reader`, представляющий входной поток (т. е. исходную программу). Обращение к `init()` могло бы выглядеть так:

```
InputStream s = new FileInputStream(sourceFileName);
Reader r = new InputStreamReader(s);
Scanner.init(r);
```

Метод `next()` повторно вызывается парсером и при каждом вызове возвращает очередную лексему. Лексема – объект следующего класса:

```
public class Token {
  public int    kind;    // код лексемы
  public int    line;    // строка лексемы (для сообщений об ошибках)
  public int    col;     // столбец лексемы (для сообщений об ошибках)
  public String val;     // значение лексемы
  public int    numVal;  // числовое значение лексемы (для number и charCon)
}
```

У каждой лексемы есть *код* (`kind`), определяющий ее вид. У некоторых лексем есть также *значение*, которое хранится в поле `numVal` для чисел и символьных констант и в поле `val` для идентификаторов. Для сообщений об ошибках в каждой лексеме хранятся также строка и столбец позиции исходного кода, в которой эта лексема начинается.

Каждому виду лексем сопоставляется уникальный код лексемы. Для MicroJava определены такие коды:

```
static final int
// лексема ошибки
none = 0,
// классы лексем  операторы и специальные символы                    ключевые слова
ident   = 1,      plus  = 4, /* + */      assign    = 17, /* = */     break_    = 29,
number  = 2,      minus = 5, /* - */      pplus     = 18, /* ++ */    class_    = 30,
charCon = 3,      times = 6, /* * */      mminus    = 19, /* --*/    else_     = 31,
                  slash = 7, /* / */      semicolon = 20, /* ; */     final_    = 32,
                  rem   = 8, /* % */      comma     = 21, /* , */     if_       = 33,
                  eql   = 9, /* == */     period    = 22, /* . */     new_      = 34,
                  neq   = 10, /* != */    lpar      = 23, /* ( */     print_    = 35,
                  lss   = 11, /* < */     rpar      = 24, /* ) */     program_  = 36,
                  leq   = 12, /* <= */    lbrack    = 25, /* [ */     read_     = 37,
                  gtr   = 13, /* > */     rbrack    = 26, /* ] */     return_   = 38,
                  geq   = 14, /* >= */    lbrace    = 27, /* { */     void_     = 39,
                  and   = 15; /* && */    rbrace    = 28, /* } */     while_    = 40,
                  or    = 16, /* || */
// лексема конца файла
eof = 41;
```

Текущее состояние сканера хранится в глобальных переменных:

```
static Reader in;                  // входной поток литер
static char ch;                    // следующая необработанная литера
```

```
static int line, col;                // строка и столбец ch
static final char eofCh = (char) -1; // литера, обозначающая конец файла
```

В каждый момент времени ch содержит следующую необработанную литеру, а line и col – координаты ее позиции в исходном коде; нумерация строк и столбцов начинается с 1.

Метод init() инициализирует глобальные переменные сканера и читает первую литеру исходного кода, вызывая вспомогательный метод nextCh():

```
public static void init (Reader r) {
  in = r;
  line = 1, col = 0;
  nextCh(); // читает первую литеру в ch и присваивает col значение 1
}
```

Метод nextCh() читает следующую литеру исходного кода, сохраняет ее в глобальной переменной ch и обновляет номера строки и столбца:

```
private static void nextCh() {
  try {
     ch = (char) in.read(); col++;
     if (ch == '\n') { line++; col = 0; }
  } catch (IOException e) { ch = eofCh; }
}
```

Сердцевиной сканера является метод next(), вызываемый парсером всякий раз, как ему нужна очередная лексема. Он реализует ДКА так, как было показано в конце предыдущего раздела. Ниже показан фрагмент этого метода.

```
public static Token next() {
  while (ch <= ' ') nextCh(); // пропустить пробелы, табуляторы, знаки конца строк
  Token t = new Token();
  t.line = line; t.col = col;
  switch (ch) {
    // идентификаторы и ключевые слова
    case 'a': case 'b': ... case 'z': case 'A': case 'B': ... case 'Z':
      readName(t); break;

    // числа
    case '0': case '1': ... case '9':
      readNumber(t); break;

    // литерные константы
    case '\'':
      readCharCon(t); break;

    // однолитерные лексемы
    case ';': nextCh(); t.kind = semicolon; break;
    case '.': nextCh(); t.kind = period; break;
    case eofCh: t.kind = eof; break; // больше нет литер
    ...
    // многолитерные лексемы
```

```
    case '=': nextCh();
      if (ch == '=') { nextCh(); t.kind = eql; } else t.kind = assign; break;
    case '&': nextCh();
      if (ch == '&') { nextCh(); t.kind = and; } else t.kind = none; break;
    ...
    // комментарий или оператор /
    case '/': nextCh();
      if (ch == '/') {
         do nextCh(); while (ch != '\n' && ch != eofCh); // пропустить комментарий
         t = next(); // рекурсивный вызов сканера
      } else t.kind = slash; break;

    // недопустимые символы
    default: nextCh(); t.kind = none; break;
  }
  return t;
} // в ch находится следующая, еще не обработанная литера
```

Метод сначала пропускает несущественные литеры (пробелы, табуляторы, знаки конца строки), все они меньше или равны `' '`. После этого в `ch` оказывается первая литера следующей лексемы. Поэтому создается новый объект лексемы `t`, а номерам ее строки и столбца присваиваются номера строки и столбца `ch`. Затем метод ветвится в зависимости от первой литеры лексемы. Если это буква, то вызывается метод `readName()`, который обрабатывает оставшиеся литеры имени или ключевого слова. Если это цифра, то вызывается метод `readNumber()`, который обрабатывает оставшиеся литеры числа, и так далее.

Для лексем, состоящих из одной литеры, код устанавливается сразу (например, `semicolon` для `';'`). Если же лексема состоит из нескольких литер, то необходимо продолжить чтение. Например, один знак `'='` представляет лексему `assign`, но два знака `'='` подряд – лексему `eql`. Важно после каждой прочитанной литеры вызывать метод `nextCh()`, который читает следующую лексему в `ch`, чем гарантирует продвижение сканера вперед.

В этом фрагменте показана также обработка комментариев. Один знак `'/'` означает лексему `slash`, а два знака `'/'` – начало комментария. Комментарии пропускаются до конца текущей строки. Но какую лексему потом следует вернуть? Ведь комментарии ничего не значат для парсера и не должны порождать лексему. Поэтому после чтения комментария нужно вернуть следующую за ним лексему, для чего рекурсивно вызывается метод `next()`. Если несколько комментариев следуют друг за другом, то `next()` будет рекурсивно вызываться, пока не встретится лексема (в худшем случае `eof`), которая затем и возвращается парсеру.

Если лексема не может начинаться литерой `ch`, то возвращается лексема ошибки `none`. Заметим, что в конце `next()` глобальная переменная `ch` содержит следующую необработанную литеру, с которой метод продолжит работу при следующем вызове.

Метод `next()` вызывает вспомогательные методы для распознавания имен, чисел и символьных констант. Перечислим их.

private static void readName (Token t)

При вызове `readName()` переменная `ch` содержит первую букву имени, а `t` уже инициализирована правильными номерами строки и столбца. Метод читает последующие буквы или цифры и сохраняет их в `t.val`. В конце `ch` будет содержать первую литеру после имени. Поскольку идентификаторы и ключевые слова имеют одинаковую структуру, необходимо проверять, является ли распознанное имя ключевым словом. Для этого имя ищется в таблице ключевых слов MicroJava. Если оно найдено, то в `t.kind` записывается код лексемы, соответствующей найденному ключевому слову, в противном случае – `ident`.

private static void readNumber (Token t)

При вызове `readName()` переменная `ch` содержит первую цифру числа. Метод читает последующие цифры и сохраняет их в `t.val`. Затем последовательность цифр преобразуется в число, которое сохраняется в `t.numVal`. Если число слишком велико для типа `int`, то следует сообщить об ошибке, но код лексемы `t.kind` все равно будет равен `number`, так чтобы парсер мог продолжить работу с распознанным числом. После того как число будет полностью распознано, `ch` содержит первую литеру после числа.

private static void readCharCon (Token t)

При вызове `readCharCon()` переменная `ch` содержит апостроф (`'\''`). Метод читает последующие литеры вплоть до закрывающего апострофа или до конца строки и сохраняет их в `t.val`. Затем эти литеры анализируются. Если прочитана допустимая литерная константа (например, `'x'`, `'\r'`, `'\n'` или `'\t'`), то в `t.kind` записывается `charCon`, а в `t.numVal` – значение соответствующей литеры. В случае недопустимой литерной константы (например, пустой, `'xy'`, `''` или `'x`) следует сообщить об ошибке, но код лексемы `t.kind` все равно будет равен `charCon`, так чтобы парсер мог продолжить работу с литерной константой. По завершении `ch` содержит первую литеру после литерной константы.

2.3.2. К вопросу об эффективности

Хотя лексический анализ – самый простой этап компилятора, он также один из самых времяёмких (если не считать оптимизации). Ведь нужно прочитать много литер и для каждого вызвать `nextCh()`. Поэтому эффективности лексического анализа следует уделить особое внимание.

По этой же причине очередной прочитанный символ не рассматривается как значение, возвращенное `nextCh()`, а хранится в глобальной переменной `ch`.

Для очень больших исходных программ с целью повышения производительности имеет смысл использовать буферизованное чтение, т. е. класс `BufferedReader`:

```
InputStream s = new FileInputStream(sourceFileName);
Reader r = new BufferedReader(new InputStreamReader(s));
Scanner.init(r);
```

Однако если программа мала (как в случае языка MicroJava), то разница в быстродействии будет едва заметна.

2.4. Упражнения

1. *Регулярные грамматики (1).* Напишите регулярную грамматику для невложенных комментариев, ограниченных с двух сторон (/* ... */). Можете обозначить `char` все литеры, отличные от `'/'` и `'*'`.
2. *Регулярные грамматики (2).* Является ли регулярной следующая грамматика для показаний часов? Если нет, можно ли преобразовать ее в регулярную?

   ```
   Time    = Hours ':' Minutes.
   Hours   = digit digit.
   Minutes = digit digit.
   ```

3. *Регулярные грамматики (3).* Является ли регулярной следующая грамматика для списков? Если нет, можно ли преобразовать ее в регулярную?

   ```
   List    = '(' Element {',' Element} ')'.
   Element = ident | List.
   ```

4. *Преобразование регулярной грамматики в ДКА.* Нарисуйте ДКА для следующей регулярной продукции, описывающей числа с плавающей точкой:

   ```
   Float = digit {digit} '.' {digit} ['E' ['+' | '-'] digit {digit}].
   ```

5. *Преобразование ДКА в регулярную грамматику.* Определите регулярную грамматику для ДКА на рис. 2.6, который распознает шестнадцатеричное число (например, `1A5X`), начинающееся десятичной цифрой.

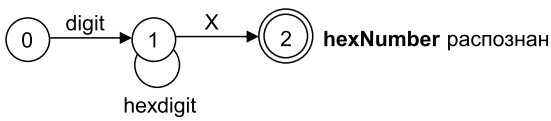


Рис. 2.6 ❖ ДКА для распознавания шестнадцатеричных чисел

6. *Введение новой лексемы.* Что нужно сделать для добавления в MicroJava новой лексемы **, обозначающей оператор возведения в степень (`x**y` = x^y)?
7. *Обработка вложенных комментариев, ограниченных с двух сторон.* Модифицируйте реализацию метода `next()`, так чтобы он распознавал не толь-

ко однострочные комментарии, но и вложенные блочные комментарии (/* ... /* ... */ ... */) и отфильтровывал их из потока лексем.

8. *Сканирование имен*. Реализуйте метод `readName()` согласно спецификации в предыдущем разделе. Этот метод вызывается из `next()` для распознавания имени или ключевого слова. Код лексемы должен быть сохранен в `t.kind`, а текст имени в `t.val`.
9. *Сканирование чисел*. Реализуйте метод `readNumber()` согласно спецификации в предыдущем разделе. Этот метод вызывается из `next()` для распознавания числа. Код лексемы `number` должен быть сохранен в `t.kind`, а значение числа в `t.numVal`. Если число слишком велико для представления типом `int`, должно быть напечатано сообщение об ошибке.
10. *Сканирование литерных констант*. Реализуйте метод `readCharCon()` согласно спецификации в предыдущем разделе. Этот метод вызывается из `next()` для распознавания литерной константы (например, `'x'`). Код лексемы `charCon` должен быть сохранен в `t.kind`, а числовое значение символьной константы в `t.numVal`. Если литерная константа недопустима (например, `''`, `'abc'`), должно быть напечатано сообщение об ошибке.
11. *Реализация сканера*. Реализуйте полный сканер MicroJava, как описано в этой главе, соблюдающий лексическую структуру MicroJava, описанную в приложении A. Не отклоняйтесь от интерфейса сканера (раздел 2.3) и реализуйте вспомогательные методы `nextCh()`, `readName()`, `readNumber()` и `readCharCon()` в дополнение к методам `init()` и `next()`. Затем скачайте с сопроводительного сайта файл `TestScanner.zip` и воспользуйтесь им для тестирования своего сканера. Должны распознаваться все допустимые терминальные символы, а для недопустимых должно печататься сообщение об ошибке.

Глава 3

Синтаксический анализ

После лексического анализа исходная программа представлена потоком лексем, содержащим только существенные лексемы; все пробелы, табуляторы, знаки конца строки и комментарии удалены.

Задача синтаксического анализатора (*парсера*) заключается в том, чтобы проанализировать поток лексем в соответствии с заданной грамматикой и построить (по крайней мере неявно) синтаксическое дерево. Если все пройдет успешно, значит, программа синтаксически правильна, и ее обработку можно продолжить. В противном случае необходимо сообщить о синтаксической ошибке, после чего синтаксический анализ можно продолжить с целью обнаружения других ошибок.

Как и в случае лексического анализа, мы начнем с краткого экскурса в теорию и рассмотрим контекстно-свободные грамматики, а также механизм их распознавания – автоматы с магазинной памятью.

3.1. Контекстно-свободные грамматики и автоматы с магазинной памятью

В главе 2 мы видели, что регулярные грамматики не могут выразить центральную рекурсию. Но центральная рекурсия встречается в языках программирования повсеместно, поэтому для описания таких языков нам необходима следующая в иерархии грамматика – *контекстно-свободная* (КСГ). Грамматика называется контекстно-свободной, если все ее продукции имеют вид

$x = \alpha$.

В левой части находится один нетерминальный символ, а правая часть α состоит из последовательности терминальных и нетерминальных символов; РБНФ-грамматики могут содержать также метасимволы, например |,

(...), [...] и {...}. Простой пример контекстно-свободной грамматики дает следующая косвенно центрально-рекурсивная грамматика арифметических выражений:

```
Expr = Term {("+" | "-") Term.
Term = Factor {("*" | "/") Factor}.
Factor = ident | number | "(" Expr ")".
```

Контекстно-свободные грамматики распознаются *автоматами с магазинной памятью*, который мы сейчас и рассмотрим.

3.1.1. Автоматы с магазинной памятью

Как и детерминированный конечный автомат (ДКА), автомат с магазинной памятью (АМП) описывается состояниями и переходами между ними. Однако в отличие от ДКА:

- АМП допускает переходы, помеченные не только терминальными, но и нетерминальными символами;
- АМП запоминает переходы состояний в стеке и потому может вернуться, распознав нетерминальный символ, и продолжить распознавание.

Рассмотрим в качестве примера очень простую центрально-рекурсивную грамматику:

```
E = x | "(" E ")".
```

Первая версия АМП для распознавания языка этой грамматики выглядит, как показано на рис. 3.1.

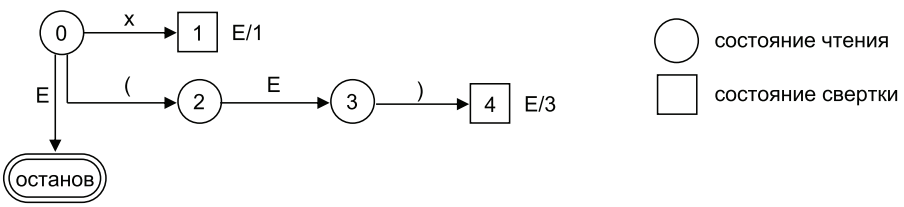


Рис. 3.1. Предварительная версия АМП для грамматики `E = x | "(" E ")"`

Мы видим, что существуют состояния двух типов. В *состоянии чтения* (представлены кружочками) читается следующий символ, как в ДКА, и автомат переходит в последующее состояние (например, из состояния 0 в состояние 1 при чтении x). В *состоянии свертки* (представлены квадратиками) распознан нетерминальный символ. Поэтому распознанные ранее символы сворачиваются в этот нетерминальный символ, вследствие чего АМП совершает частичный возврат и продолжает с распознанного нетерминального символа. Состояния свертки на рис. 3.1 помечены действием. В состоянии 4

действием является "E/3", означающее, что АМП распознал нетерминальный символ E и должен вернуться на 3 шага (в состояние 0). Оттуда он продолжает распознавание, имея символ E, и переходит в состояние останова. Это также объясняет семантику переходов, помеченных нетерминальными символами.

Существует также переход из состояния 2 в состояние 3, помеченный символом E. Но прежде чем такой переход станет возможным, должен быть распознан нетерминальный символ E. Можете считать, что в состоянии 2 автомат E вызывается рекурсивно. После распознавания E рекурсивно вызванный автомат возвращается в состояние 2, и производится переход, помеченный E (рис. 3.2).

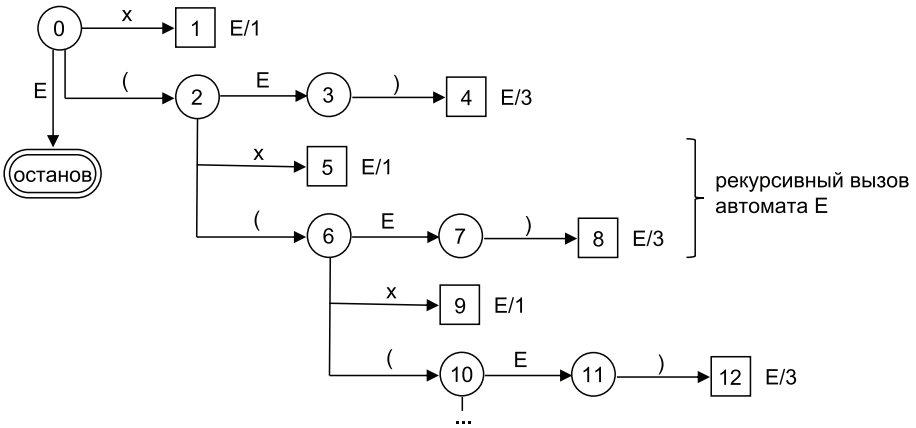


Рис. 3.2 ❖ Рекурсивный вызов автомата E в состоянии 2

Однако проблема осталась: в состоянии 6 имеет место еще один переход, помеченный E, так что автомат E следует вызвать еще раз, и так далее. Это привело бы к бесконечной рекурсии.

Но, присмотревшись внимательнее, мы увидим, что состояния 1 и 5 одинаковы, как и состояния 2 и 6. Следовательно, мы можем перейти из состояния 2 в состояние 1, прочитав x, и вернуться в состояние 2, прочитав "(". В итоге получаем окончательный АМП (рис. 3.3).

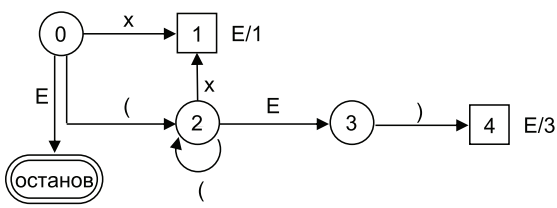


Рис. 3.3 ❖ АМП для грамматики E = x | "(" E ")"

При распознавании сентенции ((x)) АМП проходит состояния следующим образом:

```
стек      действие
0         перенос по ( в состояние 2
0 2       перенос по ( в состояние 2
0 2 2     перенос по x в состояние 1
0 2 2 1   свертка по x в E и возврат на один шаг
0 2 2     перенос по E в состояние 3
0 2 2 3   перенос по ) в состояние 4
0 2 2 3 4 свертка по (E) в E и возврат на 3 шага
0 2       перенос по E в состояние 3
0 2 3     перенос по ) в состояние 4
0 2 3 4   свертка по (E) в E и возврат на 3 шага
0         перенос по E в состояние останов
0 stop    сентенция распознана
```

Как видим, АМП запоминает путь, по которому прошел (т. е. посещенные состояния), в стеке, поэтому может вернуться по этому пути после свертки и продолжить работу с распознанного нетерминального символа. Поэтому АМП намного мощнее, чем ДКА, т. к. помнит «историю» и может возвращаться после свертки. И значит, в отличие от ДКА, он способен выразить центральную рекурсию.

3.1.2. Ограничения контекстно-свободных грамматик

Как мы видели, регулярные грамматики не способны обработать центральную рекурсию. Есть ли подобные ограничения для контекстно-свободных грамматик? Да – контекстно-свободная грамматика не может выразить контекстные условия.

Контекстно-свободная грамматика описывает только *синтаксис* языка, но не его (статическую) *семантику*. Семантика выражается в терминах контекстных условий, которые должны быть проверены компилятором. Приведем несколько примеров таких контекстных условий.

- *Каждое имя должно быть объявлено до использования.*
  Обычно имя объявляется за много строк до использования, так что объявление принадлежит контексту использования. Следовательно, предложение

  ```
  x = 3;
  ```

  может быть правильным или неправильным в зависимости от того, была объявлена переменная x или нет. Выразить такое условие с помощью контекстно-свободной грамматики невозможно.
- *Типы операндов в выражениях должны быть совместимы.*
  Типы операндов задаются там, где операнды объявлены, и, следовательно, принадлежат контексту выражения.

Как решить эту проблему? Один из вариантов – перейти к следующему в иерархии типу грамматик – контекстно-зависимым. Но они слишком сложны для конструирования компиляторов и не поддаются эффективному анализу.

Поэтому проблема решается путем откладывания проверки контекстных условий до этапа семантического анализа. Следовательно, программа

```
char x;
...
x = 3;
```

распознается парсером как синтаксически правильная. Несовместимость типов в операции присваивания обнаруживается и диагностируется только на этапе семантического анализа.

3.1.3. Контекстные условия

Для описания семантики языков программирования имеются формальные нотации (например, [Schm86, GTWW77]), но обычно они слишком сложны для написания и чтения. Поэтому мы будем использовать полуформальную нотацию, в которой подлежащие проверке контекстные условия формулируются на естественном языке для каждой продукции грамматики.

Ниже приведено несколько примеров. Полный список контекстных условий для MicroJava см. в приложении A.

Statement = Designator “=” Expr “;”.

- `Designator` должен обозначать переменную, элемент массива или поле объекта.
- Тип `Expr` должен быть *совместим по присваиванию* с типом `Designator`.

Factor = “new” ident “[” Expr “]”.

- `ident` должен обозначать тип.
- `Expr` должно иметь тип `int`.

Designator_0 = Designator_1 “[” Expr “]”.

- $\texttt{Designator}_1$ должен иметь тип массива.
- `Expr` должно иметь тип `int`.

Как видим, контекстные условия ссылаются на символы из продукций (например, `ident`, `Expr` и `Designator`). Если символ встречается несколько раз (как в третьей продукции), то для различения используются индексы (например, $\texttt{Designator}_1$).

Позже мы увидим, что семантический анализ – не отдельный этап компилятора, а объединен с парсером. Семантический анализ продукции выполняется одновременно с ее синтаксическим анализом. По ходу дела проверяются контекстные условия и диагностируются ошибки.

3.1.4. Сравнение регулярных и контекстно-свободных грамматик

Чтобы подвести итоги, еще раз сравним регулярные и контекстно-свободные грамматики, поговорим об их использовании, характеристиках и ограничениях (рис. 3.4).

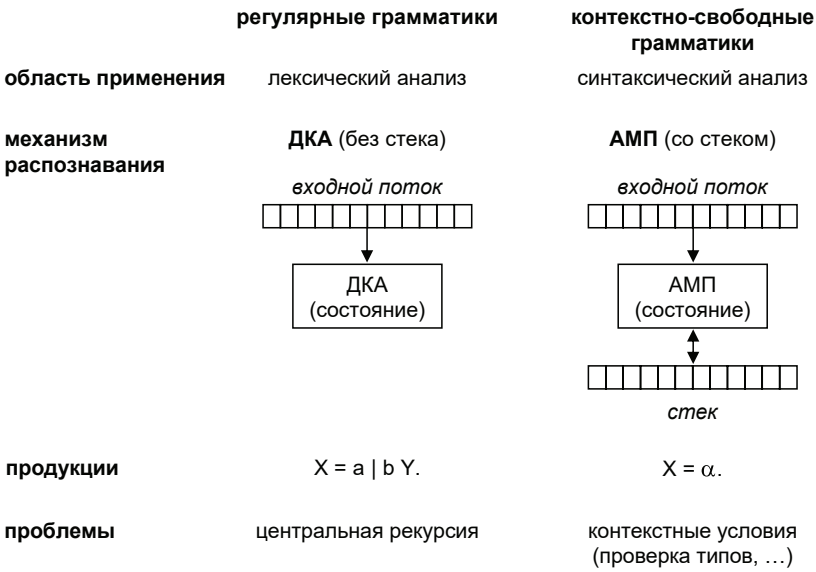


Рис. 3.4 ❖ Сравнение регулярных и контекстно-свободных грамматик

Регулярные грамматики применяются для лексического анализа, контекстно-свободные – для синтаксического анализа. Механизмом распознавания регулярных языков является детерминированный конечный автомат (ДКА), тогда как контекстно-свободных грамматик – автомат с магазинной памятью (АМП). АМП более мощный, чем ДКА, потому что запоминает «историю» анализа в стеке и, стало быть, способен обрабатывать вложенные структуры (с центральной рекурсией).

Продукции регулярных грамматик могут содержать в правой части только терминальный символ или терминальный, за которым следует нетерминальный (согласно другому определению, они могут состоять только из одного нерекурсивного правила РБНФ), тогда как продукции контекстно-свободных грамматик могут содержать в правой части любые РБНФ-конструкции.

Регулярные грамматики не могут выразить вложенные структуры и потому не способны справиться с центральной рекурсией. Для контекстно-свободных грамматик центральная рекурсия – не проблема, однако они не могут выразить контекстные условия, а значит, описать требования к семантической правильности программы.

3.2. Метод рекурсивного спуска

Мы переходим собственно к синтаксическому анализу, т. е. к реализации парсера для заданной грамматики. Есть несколько методов синтаксического анализа. Здесь мы опишем самый простой – и единственный, который можно реализовать «вручную», не пользуясь инструментами типа генератора парсеров. Другой метод описан в главе 8.

Описанный здесь метод называется *рекурсивным спуском*. Это нисходящий метод, который строит синтаксическое дерево для заданного входа сверху вниз. Для примера рассмотрим грамматику

```
X = a X c | b b.
```

и вход `a b b c`. Анализ начинается с начального символа грамматики (в данном случае `X`), показанного сверху, и входной последовательности `a b b c`, показанной снизу. Между ними должно быть построено синтаксическое дерево, которое отображает начальный символ на входную последовательность (рис. 3.5).

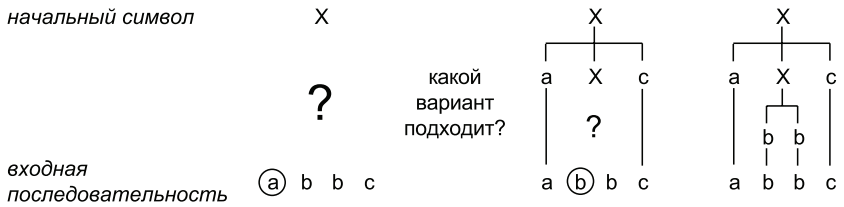


Рис. 3.5 ❖ Как работает нисходящий парсер

Первым входным символом является `a`. Какой из двух вариантов `X` подходит для него? Первая альтернатива – `a X c`, вторая – `b b`. Значит, подойти может только первая альтернатива, потому что лишь она начинается с `a`. Помещаем ее на следующий уровень после начального символа `X` и можем сопоставить первый и последний символы входной фразы с `a X c`.

В дереве остался нетерминальный символ `X` и еще ни с чем не сопоставленная часть входа `b b`. Первым символом остатка является `b`, и он сопоставляется только со второй альтернативой `X`. Поэтому используем альтернативу `b b` на следующем уровне синтаксического дерева и таким образом поглощаем остаток входа. Синтаксическое дерево построено, а вход распознан целиком и является синтаксически правильным.

В общем случае парсер всегда выбирает подходящий вариант нетерминального символа, исходя из следующего входного символа (*опережающего символа*) и *терминальных начальных символов* вариантов.

3.2.1. Парсер как класс

Как и сканер, парсер является классом с глобальными полями и методами. В любой момент времени он заглядывает вперед на одну лексему и использует ее для управления анализом. Эта лексема называется *опережающей*. Ее код хранится в глобальном поле `sym`:

```
private static int sym; // код опережающей лексемы
```

Парсер также запоминает две последние прочитанные лексемы, `t` и `la`:

```
private static Token t;  // последняя распознанная лексема
private static Token la; // опережающая лексема (еще не распознана)
```

`Token` – тип лексем, поставляемых сканером в качестве терминальных символов (см. главу 2). На рис. 3.6 показана связь между `t`, `la` и `sym`.

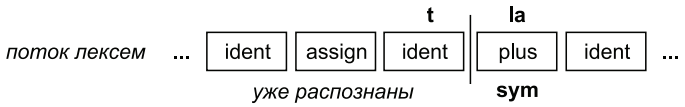


Рис. 3.6 ❖ Текущие лексемы в потоке лексем

Вертикальной чертой обозначена текущая позиция парсера. `t` – последняя распознанная лексема, `la` – опережающая еще не распознанная лексема, а `sym` – ее код. После распознавания каждой лексемы позиция парсера сдвигается вперед на одну лексему с помощью вспомогательного метода `scan()`:

```
private static void scan() {
  t = la;
  la = Scanner.next(); sym = la.kind;
}
```

В начале синтаксического анализа вызывается метод `scan()`, поэтому первая лексема из входного потока находится в `la`, а ее код – в `sym`; `t` пока не определена.

Далее мы рассмотрим, как систематически (почти механически) вывести парсер по заданной РБНФ-грамматике. Для каждого элемента грамматики (т. е. терминальных и нетерминальных символов, альтернатив, факультативных элементов и повторений) мы покажем, как преобразовать его во фрагмент кода парсера.

3.2.2. Разбор терминальных символов

Если в правой части продукции встречается терминальный символ `a`, то ему будет соответствовать действие парсера `check(a)`.

check() – вспомогательный метод, который принимает код подлежащей распознаванию лексемы и проверяет, совпадает ли она с опережающим символом sym. Если да, то у сканера запрашивается следующая лексема, иначе диагностируется ошибка.

```
private static void check (int expected) {
  if (sym == expected) scan();  // лексема распознана => сдвинуться вперед
  else error(name[expected] + "expected");
}
```

В качестве сообщения об ошибке мы просто выводим лексему, которую ожидали, но не получили. Для этого используется глобальный массив name, инициализированный именами лексем в порядке их кодов:

```
private static String[] name = {"?", "identifier", "number", ..., "+", "–", ...};
```

Метод error() печатает сообщение об ошибке, а также строку и столбец позиции, где она обнаружена; эти значения берутся из опережающей лексемы la:

```
private static void error (String msg) {
   System.out.println("line " + la.line + ", col " + la.col + ": " + msg);
   System.exit(1); // лучшее решение показано ниже
}
```

После выдачи сообщения об ошибке компиляция прекращается обращением к System.exit(1). Разумеется, это плохое решение, потому мы хотели бы за один прогон найти как можно больше ошибок. Мы вернемся к этому вопросу в разделе 3.4.

Коды лексем, передаваемые в качестве параметра методу check(), объявляются с помощью именованных констант, как в сканере (см. главу 2):

```
static final int none = 0, ident = 1, number = 2, ...;
```

3.2.3. Разбор нетерминальных символов

Если в правой части продукции встречается нетерминальный символ X, то ему будет соответствовать действие парсера X(a), разбирающее этот нетерминальный символ.

```
private static void X() {
   ... действия для разбора X ...
}
```

Таким образом, для каждого нетерминального символа существует одноименный метод парсера.

Начальным символом грамматики языка MicroJava является MicroJava, поэтому в парсере имеется метод с таким именем, который разбирает всю про-

грамму, вызывая другие методы парсера и распознавая терминальные символы с помощью метода `check()`. Ниже приведен главный метод класса `Parser`.

```
public static void parse() {
   scan();       // инициализировать la и sym
   MicroJava();  // разобрать всю программу на MicroJava
   check(eof);   // убедиться, что после разбора программы ничего не осталось
}
```

3.2.4. Разбор последовательностей

Теперь мы можем объединить оба паттерна и посмотреть, как разбираются последовательности терминальных и нетерминальных символов. Для грамматики

```
X = a Y c.
Y = b b.
```

мы должны написать методы парсера для `X` и `Y`. Метод для `X` выглядит следующим образом:

```
private static void X() {
   // sym содержит первую лексему X
   check(a);
   Y();
   check(c);
   // sym содержит первую лексему после X
}
```

Паттерны разбора терминальных и нетерминальных символов можно применять механически для разбора последовательности `a Y c`. На рис. 3.7 показано, как работают методы для `X` и `Y`.

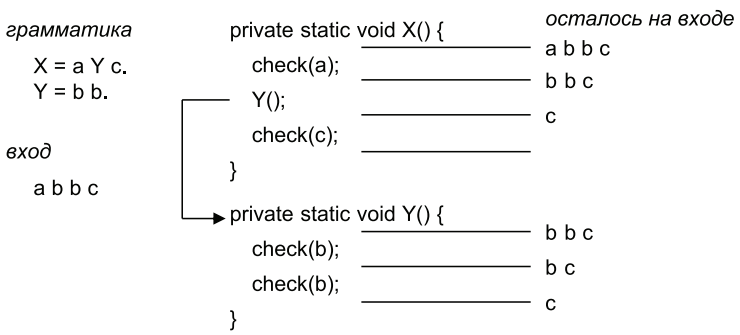


Рис. 3.7 ❖ Порядок работы методов для `X` и `Y`

Каждый вызов `check()` потребляет очередную лексему из входного потока, пока он не опустеет.

3.2.5. Разбор альтернатив

Если в правой части продукции имеются альтернативы α | β | γ (где α, β и γ обозначают произвольные РБНФ-выражения), то действие парсера (на псевдокоде) выглядит следующим образом:

```
if (sym ∈ First(α)) { ... разобрать α ... }
else if (sym ∈ First(β)) { ... разобрать β ... }
else if (sym ∈ First(γ)) { ... разобрать γ ... }
else error("..."); // напечатать подходящее к случаю сообщение об ошибке
```

Альтернативы проверяются по очереди в поисках той, что подойдет к опережающей лексеме `sym` (здесь `First(α)` обозначает множество терминальных начальных символов α). Если альтернатива найдена, то она разбирается. В противном случае выводится подходящее сообщение об ошибке. Рассмотрим пример. Для грамматики

```
X = a Y | Y b.
Y = c | d.
```

сначала определим терминальные начальные символы альтернатив:

```
First(c) = {c}
First(d) = {d}
First(aY) = {a}
First(Yb) = First(Y) = {c, d}
```

Теперь методы парсера для `X` и `Y` можно реализовать следующим образом:

```
private static void X() {
   if (sym == a) {                    // если sym соответствует первой альтернативе
      check(a);
      Y();
   } else id (sym == c || sym == d) { // если sym соответствует второй альтернативе
      Y();
      check(b);
   } else error("invalid start of X");
}

private static void Y() {
  if (sym == c) {                     // если sym соответствует первой альтернативе
     check(c);
  } else if (sym == d) {              // если sym соответствует второй альтернативе
     check(d);
  } else error("invalid start of Y");
}
```

Здесь сообщение говорит о том, что опережающий символ не является допустимым начальным ни для `X`, ни для `Y`.

3.2.6. Разбор факультативных элементов РБНФ

Если в правой части продукции встречается факультативное РБНФ-выражение [α], то действие парсера выглядит следующим образом:

```
if (sym ∈ First(α)) { ... разобрать α ... } // ветвь без ошибок!
```

Для продукции

```
X = [a b] c.
```

метод парсера для X выглядит так:

```
private static void X() {
   if (sym == a) {      // если sym совпадает с началом факультативного элемента, т. е. a
      check(a);
      check(b);
   }
   check(c);
}
```

Если на вход подан текст a b c, то мы входим в факультативный элемент и разбираем a и b. После этого разбирается c. С другой стороны, если на вход подан текст c, то факультативный элемент пропускается и разбирается только c. Таким образом, факультативная часть может отсутствовать.

3.2.7. Разбор повторений РБНФ

Если в правой части продукции встречается повторение {α} (где α – произвольное РБНФ-выражение), то действие парсера выглядит следующим образом:

```
while (sym ∈ First(α)) { ... разобрать α ... }
```

В качестве примера рассмотрим следующую грамматику:

```
X = a {Y b} c.
Y = d | e.
```

Терминальные начальные символы повторения {Y b} совпадают с терминальными начальными символами Y, т. е. это d и e. Поэтому метод парсера для X имеет вид:

```
private static void X() {
   check(a);
   while (sym == d || sym == e) { // пока sym совпадает с First(Y)
      Y();
      check(b);
   }
   check(c);
}
```

Если множество терминальных начальных символов повторения велико, то, возможно, будет эффективнее выполнять цикл до тех пор, пока не встретится терминальный *последующий* символ повторения. Но тогда в условие завершения цикла нужно будет добавить также `eof` – на всякий случай:

```
private static void X() {
   check(a);
   while (sym != c && sym != eof) {
      Y();
      check(b);
   }
   check(c);
}
```

Однако по возможности следует предпочесть первую форму цикла, иначе если терминальный последующий символ (в данном случае `c`) не встретится во входном потоке, то выход из цикла будет возможен только по `eof`.

3.2.8. Работа с большими множествами терминальных начальных символов

При разборе альтернатив, факультативных элементов и повторений опережающий символ `sym` необходимо сравнивать с терминальными начальными символами этих конструкций. Но количество терминальных начальных символов может быть очень велико. Как справиться с этой проблемой?

Эвристическое правило: если множество терминальных начальных символов содержит больше четырех элементов, то следует использовать класс `BitSet`. Например, если начальные символы нетерминальных символов `X` и `Y` таковы:

```
First(X) = {a, b, c, d, e}
First(Y) = {f, g, h, i, j}
```

то в парсере эти множества можно объявить следующим образом:

```
import java.util.BitSet;
...
private static BitSet firstX = new BitSet();
private static BitSet firstY = new BitSet();
```

и соответственно инициализировать их в начале парсера:

```
firstX.set(a); firstX.set(b); firstX.set(c); firstX.set(d); firstX.set(e);
firstY.set(f); firstY.set(g); firstY.set(h); firstY.set(i); firstY.set(j);
```

Тогда разбор продукции

```
Z = X | Y.
```

можно реализовать следующим образом:

```
private static void Z() {
   if (firstX.get(sym)) X();       // если sym ∈ First(X)
   else if (firstY.get(sym)) Y();  // если sym ∈ First(Y)
   else error("invalid start of Z");
}
```

Если множество терминальных начальных символов содержит меньше пяти элементов, то эффективнее проверять их непосредственно. Для множества

```
First(X) = {a, b, c}
```

будет более эффективно проверить элементы один за другим:

```
if (sym == a || sym == b || sym == c) ...
```

3.2.9. Как избежать нескольких проверок

До сих пор мы рассматривали схемы, которые можно систематически использовать для трансляции РБНФ-грамматики в код парсера. И хотя получающийся парсер правилен, он может быть неэффективен из-за того, что некоторые проверки производятся неоднократно. Это можно оптимизировать. Для продукции

```
X = a | b.
```

неоптимизированный метод парсера выглядит так:

```
private static void X() {
   if (sym == a) check(a);
   else if (sym == b) check(b);
   else error("invalid start of X");
}
```

Метод `check(a)` проверяет, верно ли, что `sym == a`, и если да, то вызывает `scan()` для получения следующей лексемы. Однако условие `sym == a` уже было проверено в методе `X()`, поэтому вторая проверка излишня. Вообще, любой вызов `check(x)` можно просто заменить вызовом `scan()`, если условие `sym == x` было проверено раньше. В итоге получаем такое оптимизированное решение:

```
private static void X() {
   if (sym == a) scan();
   else if (sym == b) scan();
   else error("invalid start of X");
}
```

Так мы можем избежать многих двойных проверок, как показывает пример следующей грамматики:

```
X = {a | Y d}.
Y = b | c.
```

Неоптимизированный метод парсера для X имеет вид:

```
private static void X() {
   while (sym == a || sym == b || sym == c) {
      if (sym == a) check(a);
      else if (sym == b || sym == c) { Y(); check(d); }
      else error("a, b or c expected");
   }
}
```

И снова check(a) можно заменить вызовом scan(). Но можно сделать еще больше: поскольку вход в цикл осуществляется, только если sym равно a, b или c, то проверка в первой ветви else излишня. Если переменная sym не равна a, то она должна быть равна b или c. Поэтому проверку можно исключить, да и ветвь, соответствующую ошибке, тоже убрать. Получаем такое оптимизированное решение:

```
private static void X() {
   while (sym == a || sym == b || sym == c) {
      if (sym == a) scan();
      else /* безусловно */ { Y(); check(d); } // ветви, соответствующей ошибке, нет
   }
}
```

Но даже это еще не оптимально. Здесь условие sym == a проверяется дважды, чего удается избежать в следующей реализации:

```
private static void X() {
   for (;;) { // бесконечный цикл
      if (sym == a) scan();
      else if (sym == b || sym == c) { Y(); check(d); }
      else break;
   }
}
```

В этом решении каждая проверка производится ровно один раз. Такую схему можно использовать всякий раз, как имеется повторение альтернатив, например $\{\alpha \mid \beta \mid \gamma\}$. В этом случае повторение реализуется бесконечным циклом, в котором альтернативы проверяются поочередно. Если совпадений не нашлось, производится выход из цикла.

3.2.10. И снова о вычислении терминальных начальных символов

В разделе 1.4 мы видели, как вычисляются терминальные начальные символы нетерминальных символов. В методах парсера из предыдущих раз-

делов мы также вычисляли терминальные начальные символы альтернатив, факультативных элементов и повторений. В принципе, ничего сложного тут нет, но нужно учитывать некоторые тонкости.

Если мы хотим вычислить терминальные начальные символы РБНФ-выражения $\alpha_0\alpha_1$ и при этом α_0 удаляема, то нужно принимать во внимание не только терминальные начальные символы α_0, но и терминальные последующие символы α_0, т. е. терминальные начальные символы α_1. В грамматике

```
X = Y a.
Y = {b} c // может начинаться с b и c
  | [d]   // может начинаться с d и a(!)
  | e.    // может начинаться с e
```

первая альтернатива Y начинается с удаляемого элемента {b}. Поэтому эту альтернативу следует выбирать, если на вход был подан b или c. Вторая альтернатива полностью удаляема. Ее следует выбирать, если парсер увидел d или последующий символ [d] (здесь a является последующим символом для Y). Таким образом, метод парсера для Y имеет вид:

```
private static void Y() {
   if (sym == b || sym == c) {
       while (sym == b) scan();
       check(c);
   }
   else if (sym == d || sym == a) {
      if (sym == d) scan();
   } else if (sym == e) {
      scan();
   } else error("invalid start of Y");
}
```

Если бы при входе во вторую альтернативу Y мы не проверяли также условие sym == a, то диагностировали бы ошибку, увидев a в начале Y. Но поскольку вход в эту альтернативу был произведен по a, то факультативный элемент [d] пропускается, и a распознается как последующий символ Y в методе парсера для X.

Еще пример: в грамматике

```
U = V e // может начинаться с d (т. е. First(V)) и e, т. к. V является удаляемым
  | f.  // может начинаться с f
V = {d}.
```

нетерминальный символ V является удаляемым. Поскольку первая альтернатива U начинается с V, то в нее следует зайти, увидев терминальный начальный символ V (в данном случае d) или терминальный последующий символ V (в данном случае e). Поэтому методы парсера для U и V имеют вид:

```
private static void U() {
   if (sym == d || sym == e) {
      V(); check(e);
```

```
   } else if (sym == f) {
      scan();
   } else error("invalid start of U");
}

private static void V() {
   while (sym == d) scan();
}
```

При входе в первую альтернативу `U` по символу `e` первым делом вызывается `V()`. Однако `V()` возвращает управление, не выполнив цикл, после чего в `U` распознается `e`.

Таким образом, при вычислении терминальных начальных символов РБНФ-выражения следует принимать во внимание удаляемость.

3.2.11. Синтаксическое дерево, получающееся в результате разбора методом рекурсивного спуска

В начале этой главы мы сказали, что парсер строит синтаксическое дерево по потоку лексем. Но где же это дерево? Описанные выше действия парсера ничего такого не строят.

Дело в том, что метод рекурсивного спуска строит синтаксическое дерево неявно – оно представлено активными в текущий момент методами парсера, или, иначе говоря, продукциями, которые разбираются в текущий момент. В качестве примера рассмотрим следующую грамматику:

```
X = a Y d. Y = b c.
```

При вызове метода парсера для `X` мы работаем с поддеревом с корнем `X` и дочерними узлами `a`, `Y` и `d` (см. рис. 3.8а).

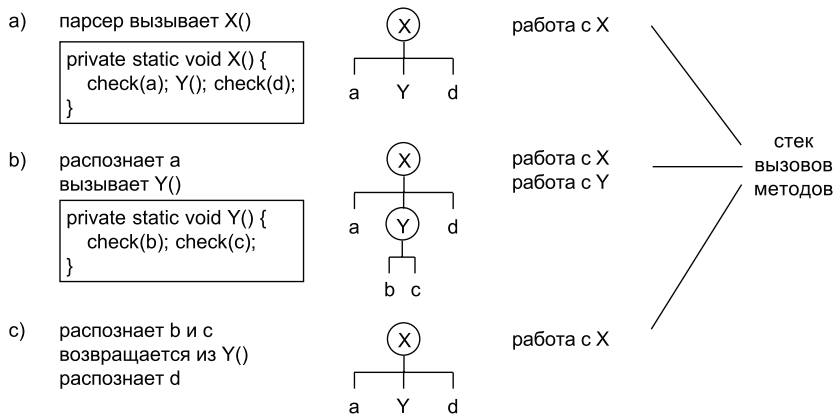


Рис. 3.8 ❖ Неявное построение синтаксического дерева при разборе методом рекурсивного спуска

Метод парсера `X()` распознает `a` и затем вызывает `Y()`, неявно добавляя в синтаксическое дерево еще один уровень с дочерними узлами `b` и `c` (рис. 3.8b).

`Y()` распознает `b` и `c`, а затем возвращается к `X`, в результате чего неявное дерево с корнем `Y` удаляется (рис. 3.8c).

Таким образом, синтаксическое дерево в процессе рекурсивного спуска явно не строится, а существует только в неявном виде, точнее в форме всех продукций, над которыми парсер работает в текущий момент. Кроме того, стек АМП существует только неявно, а именно в форме стека вызовов всех активных в текущий момент методов парсера. Если `X()` вызывает `Y()`, то стек вызовов растет. Когда `Y()` возвращает управление, стек снова сжимается. Можете представлять себе, что АМП – после распознавания `Y` – возвращается в начало `Y`, а затем продолжает работу, используя `Y` в методе парсера для `X`.

3.3. Свойство LL(1)

Чтобы грамматику можно было разобрать методом рекурсивного спуска, она должна обладать свойством LL(1). LL(1) означает, что сентенции грамматики должны поддаваться распознаванию **слева** направо с применением **левостороннего** вывода с **одним** опережающим символом. Однако это определение не особенно полезно. Поэтому мы будем пользоваться следующим определением:

- грамматика обладает свойством LL(1), если все ее продукции обладают этим свойством;
- продукция обладает свойством LL(1), если для всех альтернатив $\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ в ней выполняется следующее условие: $\forall\ i \neq j$: $First(\alpha_i) \cap First(\alpha_j) = \{\}$.

Иными словами, множества терминальных начальных символов всех альтернатив в продукции должны быть попарно непересекающимися. То есть все альтернативы должны начинаться разными терминальными символами, чтобы парсер мог однозначно выбрать альтернативу, видя только один опережающий символ. Если это условие не выполнено, имеет место LL(1)-конфликт.

К сожалению, многие грамматики, встречающиеся в описаниях языков или в интернете, не удовлетворяют этому требованию. Но обычно их можно преобразовать в форму LL(1), поэтому они пригодны для рекурсивного спуска.

3.3.1. Устранение LL(1)-конфликтов

Обычно LL(1)-конфликты можно устранить с помощью *факторизации*, т. е. выделения общих начал альтернатив. Рассмотрим продукцию предложения `if` в Java:

```
IfStatement = "if" "(" Expr ")" Statement
            | "if" "(" Expr ")" Statement "else" Statement.
```

Обе альтернативы в ней начинаются с `if`, поэтому продукция не удовлетворяет условию LL(1). Она также не удовлетворяет условию LL(*k*) ни для какого *k*, потому что `Expr` и `Statement` могут быть сколь угодно длинными, и, значит, в худшем случае придется читать неизвестно сколько лексем, чтобы понять, встретилось `else` или нет.

Однако если мы выделим общее начало альтернатив, а сами альтернативы будем начинать только после него, то получим

```
IfStatement = "if" "(" Expr ")" Statement (
                                          | "else" Statement
                                          ).
```

или в нотации РБНФ

```
IfStatement = "if" "(" Expr ")" Statement ["else" Statement].
```

Альтернативы исчезли, а вместе с ними исчез и LL(1)-конфликт. Однако позже мы увидим, что другой LL(1)-конфликт все же существует, а как с ним быть, узнаем в разделе, посвященном *висячему else*.

Чтобы увидеть LL(1)-конфликты и устранить их путем факторизации, иногда бывает полезно заменить нетерминальный символ правой частью его продукции. Рассмотрим следующую грамматику:

```
Statement  = Designator "=" Expr ";"
           | ident "(" [ActualParameters] ")" ";".
Designator = ident {"." ident}.
```

Здесь `Statement` может быть присваиванием или вызовом метода с факультативными параметрами. Обе альтернативы начинаются символом `ident`, но увидеть это будет проще, если подставить `Designator` в `Statement`:

```
Statement = ident {"." ident} "=" Expr ";"
          | ident "(" [ActualParameters] ")" ";".
```

Теперь тот факт, что обе альтернативы начинаются `ident`, очевиден. Выделив `ident`, мы устраним LL(1)-конфликт:

```
Statement = ident ( {"." ident} "=" Expr ";"
          | "(" [ActualParameters] ")" ";"
          ).
```

Продукция по-прежнему содержит альтернативы, но первая начинается с `"."` или `"="`, а вторая – с `"("`. Таким образом, множества терминальных начальных символов альтернатив не пересекаются, и, стало быть, LL(1)-конфликт устранен. Парсер может решить, какую альтернативу выбрать, зная только один опережающий символ.

3.3.2. Исключение левой рекурсии

Левая рекурсия всегда является LL(1)-конфликтом и потому должна быть устранена, если мы хотим применить к грамматике метод рекурсивного спуска. В главе 1 мы видели, что это всегда возможно. Рассмотрим следующую леворекурсивную грамматику:

```
IdentList = ident | IdentList "," ident.
```

Она порождает следующие сентенции:

```
ident ident "," ident ident "," ident "," ident ...
```

Легко видеть, что те же самые сентенции порождает следующая РБНФ-грамматика:

```
identlist = ident {"," ident}.
```

Таким образом, левую рекурсию всегда можно заменить РБНФ-повторением.

3.3.3. Скрытые LL(1)-конфликты в РБНФ-конструкциях

Свойство LL(1) требует, чтобы парсер всегда мог выбрать альтернативу, зная только один опережающий символ. Помимо явных альтернатив, которые мы видели в примерах выше, существуют альтернативы неявные, скрытые за факультативными элементами или повторениями РБНФ.

Встретив конструкцию [α] β, парсер может либо распознать факультативный элемент α, либо пропустить его и продолжить распознавание β. Тем самым мы имеем неявные альтернативы α β | β. Поэтому множества терминальных начальных символов α и β должны быть непересекающимися.

То же самое относится к повторениям. Встретив конструкцию {α} β, парсер может либо войти в повторение и распознать α, либо пропустить его и продолжить распознавание β (напомним, что повторений может быть 0 или больше). Поэтому множества терминальных начальных символов α и β должны быть непересекающимися.

Таким образом, для любого вхождения факультативного элемента или повторения в грамматику необходимо проверять следующее условие:

```
[α] β   First(α) ∩ First(β) должно быть {}
{α} β   First(α) ∩ First(β) должно быть {}
```

Если факультативный элемент или повторение встречаются в конце продукции, то множество ее терминальных начальных символов не должно пересекаться с множествами последующих символов этой продукции:

```
X = α [β].   First(β) ∩ Follow(X) должно быть {}
X = α {β}.   First(β) ∩ Follow(X) должно быть {}
```

Это относится также к частному случаю, когда в продукции имеется пустая альтернатива:

```
X = α | .   First(α) ∩ Follow(X) должно быть {}
```

По счастью, такие скрытые LL(1)-конфликты обычно можно устранить преобразованием грамматики. Например, в продукции

```
Name = [ident "."] ident.
```

скрытый LL(1)-конфликт возникает из-за того, что факультативный элемент начинается с `ident` и за ним также следует `ident`. Поэтому если парсер находится перед таким элементом и опережающий символ равен `ident`, то он не может решить, входить в него или пропустить. Чтобы устранить этот LL(1)-конфликт, мы должны преобразовать продукцию. Подумаем, какие сентенции могут быть сгенерированы:

```
ident "." ident
ident
```

Поэтому продукцию можно переписать так:

```
Name = ident ["." ident].
```

LL(1)-конфликт исчез, но мы по-прежнему должны проверить, пересекаются ли множества терминальных начальных символов факультативного элемента (т. е. `"."`) и терминальных последующих символов `Name`. Для этого нужно было бы вычислить множество терминальных последующих символов `Name`, которое здесь не показано.

Вот еще один, не столь тривиальный пример:

```
Program = Declarations ";" Statements.
Declarations = Decl {";" Decl}.
```

Встретив повторение, мы должны проверить, пересекаются ли множества его терминальных начальных символов (т. е. `";"`) и последующих символов (здесь – последующих символов `Declarations`). В данном случае пересекаются, потому что за `Declarations` следует `";"`. Разглядеть LL(1)-конфликт будет проще, если подставить `Declarations` в `Program`:

```
Program = Decl {";" Decl} ";" Statements.
```

Если парсер находится перед повторением и опережающий символ равен `";"`, то он не может решить, входить в повторение или пропустить его. Однако продукцию `Program` можно преобразовать к виду

```
Program = Decl ";" {Decl ";"} Statements.
```

LL(1)-конфликты исчезли, но мы по-прежнему должны проверять, что `First(Decl) ∩ First(Statements) = {}`, для чего следовало бы рассмотреть продукции `Decl` и `Statements`, которые здесь не показаны.

Проверка свойства LL(1) грамматики занимает много времени, но может быть поддержана инструментами (см. главу 7). В грамматике MicroJava (приложение А) все LL(1)-конфликты уже устранены. Однако если грамматика получена из внешнего источника, то свойство LL(1) необходимо проверить и устранить все LL(1)-конфликты, прежде чем пытаться применить к ее разбору метод рекурсивного спуска.

3.3.4. Висячее else

В главе 1 мы видели, что во вложенных предложениях if, таких как

```
if (a < b) if (b < c) x = c; else x = b;
```

возникает неоднозначность, называемая *висячим else*: символ `else` можно считать принадлежащим как внешнему, так и внутреннему `if`. По соглашению, эта неоднозначность разрешается путем отнесения `else` к непосредственно предшествующему ему `if`.

Однако висячее else также представляет LL(1)-конфликт. Рассмотрим грамматику

```
Statement = "if" "(" Expr ")" Statement ["else" Statement] | ... .
```

Эта грамматика содержит факультативный элемент в конце продукции. Поэтому множества терминальных начальных символов этого элемента (т. е. `else`) и терминальных последующих символов `Statement` не должны пересекаться. Однако, как мы видим, за `Statement` может следовать `else`, из-за чего возникает LL(1)-конфликт. Если парсер находится в начале факультативного элемента и опережающим символом является `else`, то парсер не может решить, то ли ему войти в факультативный элемент и, стало быть, отнести `else` к внутреннему `if`, то ли пропустить факультативный элемент и, значит, отнести `else` к внешнему `if`.

К сожалению, этот LL(1)-конфликт невозможно устранить путем преобразования грамматики – он внутренне присущ ей. Тогда как же решить проблему? Рассмотрим, что делает парсер, если LL(1)-конфликт не устранен. Если опережающий символ соответствует нескольким альтернативам, то парсер всегда выбирает первую подходящую. В грамматике

```
X = a b c
  | a d.
```

обе альтернативы начинаются с `a`. Если опережающий символ равен `a`, то парсер выберет первую альтернативу, а вторая никогда не будет исследоваться. Очевидно, что такое поведение нежелательно, и этот LL(1)-конфликт необходимо устранить, преобразовав грамматику. Это несложно:

```
X = a (b c | d).
```

А как насчет висячего else? Если парсер находится перед факультативным элементом в грамматике

```
Statement = "if" "(" Expr ")" Statement ["else" Statement]
          | ... .
```

и должен решить, входить в факультативный элемент или пропустить его, увидев опережающий символ `else`, то он выберет первый вариант: войти в элемент и, следовательно, отнести `else` к внутреннему `if`. Это именно то, чего мы хотим, поэтому в данном случае устранять LL(1)-конфликт нет нужды.

Таким образом, LL(1)-конфликт просто сводится к предупреждению о том, что парсер выберет первую из нескольких подходящих альтернатив. Если такое поведение нас устраивает, как в случае висячего else, то LL(1)-конфликт можно игнорировать, в противном случае его необходимо устранить.

3.3.5. Другие требования к грамматике

Помимо LL(1), грамматика должна обладать и другими свойствами, чтобы быть пригодной для синтаксического анализа. Они относятся не только к методу рекурсивного спуска, но и ко всем вообще методам синтаксического анализа.

- **Полнота.** Для каждого нетерминального символа должна существовать продукция. Например, грамматика

  ```
  X = a Y Z.
  Y = b b.
  ```

  неполна, потому что отсутствует продукция для `Z`.

- **Конечная сводимость.** Каждый нетерминальный символ должен допускать прямое или косвенное сведение к последовательности терминальных символов. Например, грамматика

  ```
  X = a Y | c.
  Y = b Y.
  ```

  не является конечно сводимой, потому что попытка свести `Y` к последовательности терминальных символов ведет к бесконечной рекурсии.

- **Отсутствие циклических зависимостей.** Нетерминальный символ не должен сводиться к самому себе, т. е. не должно существовать вывода вида $X \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow X$. Например, грамматика

  ```
  X = a | Y.
  Y = b | X.
  ```

  циклическая, потому что в ней имеется вывод $X \Rightarrow Y \Rightarrow X$.

3.4. Обработка синтаксических ошибок

Обнаружив синтаксическую ошибку, парсер должен сообщить о ней. Но потом он должен продолжить разбор, чтобы можно было найти другие синтаксические ошибки. Вообще, хороший алгоритм обработки синтаксических ошибок должен удовлетворять следующим требованиям:

- парсер должен находить как можно больше ошибок за один прогон;
- парсер не должен аварийно завершаться даже при наличии серьезных ошибок;
- механизм обработки ошибок не должен замедлять разбор синтаксически правильной программы;
- обработка ошибок не должна приводить к непомерному разбуханию кода парсера.

Иногда эти требования оказываются противоречивыми. Чем более качественной мы хотим сделать обработку ошибок, тем сложнее становится парсер, а значит, разбухает его код и обычно замедляется анализ синтаксически правильных программ. Для рекурсивного спуска существуют различные методы обработки ошибок, в той или иной степени отвечающие сформулированным требованиям:

- обработка ошибок в *режиме паники*;
- обработка ошибок с *общими якорями*;
- обработка ошибок со *специальными якорями*.

Рассмотрим эти методы по порядку. Обработка ошибок в режиме паники – самый простой из них, но он может обнаружить только одну ошибку за прогон. Другие методы способны обнаруживать несколько ошибок, но они более сложны и потому последним двум требованиям отвечают не в полной мере.

3.4.1. Обработка ошибок в режиме паники

До сих пор мы пользовались именно этим методом. Обнаружив ошибку, парсер сообщает о ней, вызывая метод `error()`, после чего прекращает разбор.

```
private static void error (String msg) {
   System.out.println("line " + la.line + ", col " + la.col + ": " + msg);
   System.exit(1); // прекратить разбор
}
```

Достоинства этой техники в том, что она обходится дешево и не приводит к разбуханию кода парсера, поскольку вся обработка ошибок сводится к вызову метода `error()`. Если в программе нет ошибок, то ее разбор вообще не замедляется. Но зато обнаруживается только первая ошибка. Исправив ее, компиляцию придется запустить заново для поиска последующих ошибок. Для тех языков, на которых пишутся короткие программы (например, языков

команд), эта техника вполне приемлема. Но для таких языков, как Java или MicroJava, применяемых для написания более длинных программ, она не годится. Тут нужна одна из двух других техник.

3.4.2. Обработка ошибок с общими якорями

В этом случае после обнаружения синтаксической ошибки производится *восстановление после ошибки*, т. е. парсер синхронизируется с некорректной программой, так чтобы можно было продолжить разбор и найти последующие ошибки. Для этого строится множество *якорей*, за которые парсер может «ухватиться» после ошибки и таким образом синхронизироваться с исходной программой. Рассмотрим пример. Пусть имеется грамматика

```
X = a Y e f.
Y = b c d.
```

и вход

```
a b x y e f eof
```

Как видно по рис. 3.9, вызывается `X` и распознается `a`. Затем вызывается `Y` и распознается `b`, после чего парсер обнаруживает, что следующий символ `x` не совпадает с ожидаемым символом `c`. Диагностируется ошибка. Затем парсер собирает все терминальные символы, которые могут следовать в грамматике после того места, где произошла ошибка. Эти символы образуют множество *якорей*, начиная с которых можно продолжить разбор.

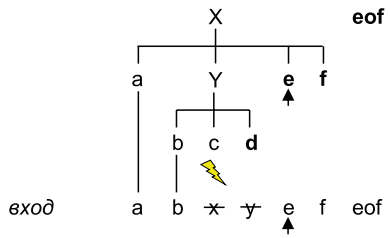


Рис. 3.9 ❖ Построение множества якорей

Рассмотрим процесс восстановления более пристально. Он состоит из трех шагов.

1. По грамматике вычисляется множество якорей, которое затем можно использовать для возобновления анализа после места ошибки. Ошибка была обнаружена, потому что ожидаемый символ `c` не совпал с входным символом `x`. Якоря – это все терминальные символы, следующие за символами, над которыми сейчас работает парсер. Парсер работает с `c`, для него последующим является `d`. Он также работает с символом `Y`,

для которого последующими являются `e` и `f`, и с `X`, для которого последующим является `eof`. Итак, множество якорей – `{d, e, f, eof}`.

2. Теперь все символы входа, не являющиеся якорями, пропускаются, т. е. символы `x` и `y` пропускаются, пока не встретится `e`, входящий в множество якорей. Таким образом, входной поток синхронизирован: следующий символ `e` можно использовать для продолжения разбора с некоторой точки грамматики.
3. На последнем шаге парсер нужно продвинуть до места грамматики, в котором ожидается якорь `e`. Для этого парсер просто продолжает работать, пока не достигнет этого места (см. стрелку на рис. 3.9). Теперь вход синхронизирован с грамматикой, и разбор можно продолжить.

Чтобы вычислить множество якорей, каждому методу парсера передается список последующих символов для соответствующего нетерминального символа в качестве параметра, например:

```
private static void X (BitSet sux) {
   ...
}
```

В зависимости от контекста, в котором встретился нетерминальный символ, множество последующих символов может быть разным, как видно на рис. 3.10.

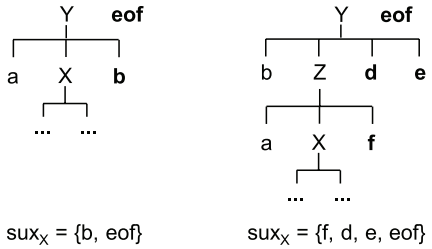


Рис. 3.10 ❖ Последующие символы для нетерминального символа `X` в различных контекстах

Поэтому контекстно-зависимые последующие символы нетерминального символа необходимо передавать в виде параметра методу парсера. Отметим, что символ `eof` также принадлежит множеству последующих, потому что он является последующим для начального символа. Это гарантирует, что в худшем случае синхронизация наступит в `eof`.

Для обработки ошибок с общими якорями необходимо модифицировать схемы разбора терминальных и нетерминальных символов, альтернатив, факультативных элементов и повторений; этим мы и займемся дальше.

Разбор терминальных символов

Когда в правой части продукции встречается терминальный символ **a**, за которым следуют элементы α_i (символы, альтернативы, факультативные элементы или повторения)

X = ... a α_1 α_2 ... α_n.

действие парсера выглядит следующим образом (на псевдокоде):

check(a, First(α_1) ∪ First(α_2) ∪ ... ∪ First(α_n) ∪ sux_x);

Иными словами, методу check() в качестве второго параметра передаются последующие символы терминального символа a:

```
private static void check (int expected, BitSet sux) {
   if (sym == expected) scan();
   else error(name[expected] + " expected", sux); // передать sux методу error
}
```

Первую часть множества последующих символов (First(α_1) ∪ First(α_2) ∪ ... ∪ First(α_n)) можно вычислить статически по грамматике. Однако множество последующих символов нетерминального символа (sux_x) необходимо добавить во время выполнения. Поэтому метод парсера для продукции

X = a b c.

на псевдокоде имеет вид:

```
private static void X (BitSet sux) {
   check(a, {b, c} ∪ sux);
   check(b, {c} ∪ sux);
   check(c, sux);
}
```

Множества, представленные классом BitSet, объявляются и инициализируются в начале парсера. Например, множество {b, c} создается следующим образом:

```
BitSet fs1 = new BitSet();
...
fs1.add(b); fs1.add(c);
```

Тогда первое обращение к check() в методе парсера X() на Java записывается таким образом:

```
check(a, ((BitSet)fs1.clone().or(sux));
```

Заметим, что сначала необходимо создать копию fs1 (чтобы не уничтожить fs1), а только потом добавить в нее sux. Разумеется, этот код утомительно писать, он увеличивает размер парсера и замедляет разбор.

Разбор нетерминальных символов

Разбор нетерминальных символов похож на описанный выше. Если нетерминальный символ Y встречается в правой части продукции (и за ним следуют элементы α_i)

X = ... Y α_1 α_2 ... α_n.

то методу парсера Y() в качестве параметра передается множество последующих символов Y:

```
Y(First(α1) ∪ First(α2) ∪ ... ∪ First(αn) ∪ suxX);
```

И снова первую часть (First(α_1) ∪ First(α_2) ∪ ... ∪ First(α_n)) можно вычислить статически по грамматике, а sux_x необходимо добавлять во время выполнения. Продукция

```
X = a Y c. // предполагается, что First(Y) = {b}
```

разбирается следующим образом (на псевдокоде):

```
private static void X (BitSet sux) {
   check(a, {b, c) ∪ sux);
   Y({c} ∪ sux);
   check(c, sux);
}
```

Начальный символ MicroJava разбирается путем вызова MicroJava({eof}).

Пропуск ошибочных символов

Обнаружив синтаксическую ошибку, парсер вызывает метод error(), которому передает сообщение об ошибке и множество якорей:

```
private static void error (String msg, BitSet sux) {
   System.out.println("line " + la.line + ", col " + la.col + ": " + msg);
   errors++;
   while (!sux.get(sym)) scan(); // while (sym ∉ sux) scan();
   // sym ∈ sux
}
```

В методе error() используется scan() для пропуска лексем из входного потока, пока не встретится якорь из множества sux. Поскольку теперь мы можем обнаружить несколько ошибок, их нужно подсчитывать в переменной errors, которая объявлена в начале парсера следующим образом:

```
public static int errors = 0; // счетчик ошибок
```

Эту переменную можно опросить в конце разбора и узнать, сколько встретилось синтаксических ошибок.

После синхронизации с входным потоком в методе error() опережающий символ sym содержит якорь, с которого можно продолжить разбор.

Синхронизация с грамматикой

После пропуска ошибочных символов парсер должен в конечном итоге достичь такого места грамматики, откуда можно продолжить разбор с найденного якоря. Для этого парсер просто продолжает читать символы (и, возможно, выдавать наведенные сообщения об ошибках), пока не дойдет до

места грамматики, где ожидается найденный якорь. Рассмотрим пример: для продукции

```
X = a b c.
```

псевдокод метода парсера выглядит так:

```
private static void X (BitSet sux) {
   check(a, {b, c} ∪ sux);
   check(b, {c} ∪ sux);
   check(c, sux);
}
```

Предположим, что на входе была не последовательность a b c, а последовательность x y c. Парсер пытается распознать a, вызвав check(a, {b, c} ∪ sux), но следующим входным символом является x, а не a, поэтому диагностируется ошибка. Множество якорей равно {b, c} ∪ sux. Метод error() пропускает все символы, не входящие в это множество. Поэтому x и y пропускаются, но c является якорем, поэтому на нем пропускание останавливается (sym == c).

Теперь парсер просто продолжает работу с вызова check(b, {c} ∪ sux). Распознавание b также завершается ошибкой, потому что sym содержит символ c. Теперь множество якорей равно {c} ∪ sux. Но на этот раз ничего не пропускается, потому что опережающий символ c принадлежит этому множеству.

Если парсер продолжит работу с вызова check(c, sux), то будет распознан символ c, так что синхронизация завершена. Парсер может возобновить разбор и попытаться найти дополнительные ошибки.

Подавление наведенных ошибок

Как мы видели, после обнаружения синтаксической ошибки парсер продолжает работать, пока не дойдет до места в грамматике, где ожидается один из найденных якорей. При этом могут возникать наведенные ошибки, которые, конечно же, нежелательны, т. к. являются не настоящими ошибками, а следствиями первоначальной.

Сообщения о наведенных ошибках можно подавить с помощью простой эвристики: ошибка диагностируется, только если с момента возникновения последней ошибки было правильно распознано по меньшей мере три символа. Для этого заведем глобальную переменную errDist, в которой хранится расстояние до последней диагностированной ошибки:

```
private static int errDist = 3;
```

В начальный момент мы считаем, что последнее сообщение об ошибке было напечатано три символа назад. После каждого правильно распознанного и потребленного символа (т. е. вызова scan()) эта переменная увеличивается:

```
private static void scan() {
   ...
```

```
   errDist++; // еще одна лексема корректно разобрана
}
```

В методе `error()` мы сообщаем об ошибке, только если `errDist >= 3`. После пропуска ошибочных символов `errDist` сбрасывается в 0 и подсчет начинается заново:

```
private static void error (String msg, BitSet sux) {
   if (errDist >= 3) {
      System.out.println("line " + la.line + ", col " + la.col + ": " + msg);
      errors++;
   }
   while (!sux.get(sym)) scan();
   errDist = 0; // начать подсчет заново
}
```

Эта простая эвристика подавляет наведенные сообщения. Но если между двумя синтаксическими ошибками находится меньше трех правильно распознанных символов, то вторая ошибка останется недиагностированной. Впрочем, в большинстве случаев синтаксические ошибки встречаются не так часто, так что на практике эта эвристика работает хорошо.

Разбор альтернатив

Схему разбора альтернатив также необходимо модифицировать для обработки ошибок с общими якорями. До сих пор для разбора альтернатив вида

`X = α | β | γ.`

мы использовали следующую схему (на псевдокоде):

```
private static void X() {
   if (sym ∈ First(α)) ... разобрать α ...
   else if (sym ∈ First(β)) ... разобрать β ...
   else if (sym ∈ First(γ)) ... разобрать γ ...
   else error("invalid X"); }
```

Если `sym` не соответствует ни одной альтернативе, то в последнем `else` будет напечатано сообщение об ошибке, и ошибочные символы будут пропущены, но потом мы уже не сможем войти ни в одну из альтернатив. Поэтому лучше проверить, совпадает ли `sym` с какой-нибудь альтернативой, еще до их разбора. Если нет, то мы уже можем диагностировать в этом месте ошибку, пропустить ошибочные символы, а затем все-таки войти в одну из альтернатив. Поэтому псевдокод новой схемы выглядит так:

```
private static void X (BitSet sux) {
   if (sym ∉ First(α) ∪ First(β) ∪ First(γ))
      error("invalid X", First(α) ∪ First(β) ∪ First(γ) ∪ sux);
   // sym соответствует одной из альтернатив или sux
   if (sym ∈ First(α)) ... разобрать α ...
   else if (sym ∈ First(β)) ... разобрать β ...
```

```
   else if (sym ∈ First(γ)) ... разобрать γ ...
   // ветви с error нет
}
```

Метод `error()` пропускает ошибочные символы, пока не встретится какой-нибудь символ из переданного множества. Это будет либо начальный символ одной из альтернатив, либо один из последующих символов `X`. Множество `First(α) ∪ First(β) ∪ First(γ)` снова можно вычислить статически по грамматике, а `sux` придется добавлять во время выполнения. Как видим, нам больше не нужна ветвь с вызовом `error()`, потому что ошибка, если она имеется, была диагностирована раньше.

Разбор факультативных элементов РБНФ

Аналогично для факультативных элементов РБНФ мы должны заранее проверить, соответствует ли опережающий символ самому факультативному элементу или следующим за ним, и если нет, то выдать сообщение об ошибке. Для продукции

```
X = [α] β.
```

псевдокод новой схемы выглядит так:

```
private static void X (BitSet sux) {
   if (sym ∉ First(α) ∪ First(β)) error("invalid X", First(α) ∪ First(β) ∪ sux);
   if (sym ∈ First(α)) ... разобрать α ...
   ... разобрать β ...
}
```

Разбор повторений РБНФ

Для повторений РБНФ используется совершенно новая схема – цикл, в который мы заходим в любом случае. В цикле мы проверяем, соответствует ли опережающий символ повторяемому символу или следующим за ним. Если нет, то мы сообщаем об ошибке, но остаемся в цикле, чтобы можно было продолжить для следующего повторения после пропуска ошибочных символов. Для продукции

```
X = {α} β.
```

псевдокод этой схемы выглядит так:

```
private static void X (BitSet sux) {
   for (;;) { // войти в цикл, даже если sym ∉ First(α)
      if (sym ∈ First(α)) ... разобрать α ... // правильный случай 1: обработать α
      else if (sym ∈ First(β) ∪ sux) break;   // правильный случай 2: выйти из цикла
                                              // и обработать β
      else error("invalid X", First(α) ∪ First(β) ∪ sux); // ошибка: синхронизироваться
                                              // и остаться в цикле
   }
   ... разобрать β ...
}
```

Метод `error()` сообщает об ошибке, а затем пропускает символы до тех пор, пока не встретится терминальный начальный символ α (с которого начинается следующее повторение), или терминальный начальный символ β (на котором повторение заканчивается), или символ из `sux` (на нем повторение также заканчивается; в этом случае печатались бы наведенные сообщения об ошибке при разборе β, но они подавляются).

Пример

Наконец, рассмотрим пример, демонстрирующий схемы разбора всех РБНФ-конструкций (альтернатив, факультативных элементов и повторений). Для грамматики

```
X = a Y | b {c d}. Y = [b] d.
```

методы парсера с обработкой ошибок выглядят так (простоты ради мы снова используем псевдокод для вычисления множеств якорей):

```
private static void X (BitSet sux) {
   if (sym != a && sym != b) error("invalid X", {a, b} ∪ sux);
   if (sym == a) {
      scan(); Y(sux);
   } else if (sym == b) {
      scan();
      for (;;) {
         if (sym == c) {
            scan();
            check(d, {c} ∪ sux); // с также является последующим символом для d
         } else if (sym ∈ sux) {
            break;
         } else {
            error("c expected", {c} ∪ sux);
         }
       }
    }
}

private static void Y (BitSet sux) {
   if (sym != b & sym != d) error("invalid Y", {b, d} ∪ sux);
   if (sym == b) scan();
   check(d, sux);
}
```

Резюме

Обработка ошибок с общими якорями является хорошим и быстрым методом восстановления после ошибки. Его можно применять систематически, пользуясь схемами для разбора терминальных символов, нетерминальных символов, альтернатив, факультативных элементов и повторений.

Но у него есть и недостатки: он сложен, приводит к разбуханию кода парсера и замедляет разбор даже синтаксически правильных программ, поскольку

множества якорей нужно вычислять и передавать методам парсера во время выполнения.

3.4.3. Обработка ошибок со специальными якорями

В этом случае восстановление после синтаксических ошибок производится только в особо «безопасных» точках синхронизации, а именно там, где ожидаются ключевые слова, больше нигде в грамматике не встречающиеся. К таким точкам относятся, например:

- начало *предложения*, где ожидаются такие ключевые слова, как `if` или `while`, или
- начало *объявления*, где ожидаются такие ключевые слова, как `public` или `void`.

Видя `if`, мы знаем, что находимся в начале предложения, потому что нигде больше это ключевое слово встречаться не может. Видя `void`, мы знаем, что начинается объявление.

Для каждого из таких мест мы вычисляем множество якорей, т. е. множество всех символов, которые там ожидаются. Однако в начале предложения или объявления может встречаться также символ `ident`, не являющийся безопасным якорем, потому что он встречается и во многих других местах грамматики. Поэтому `ident` из множества якорей удаляется.

В каждой из этих точек синхронизации мы вставляем в парсер такой (псевдо)код:

```
if (sym ∉ expectedSymbols) {
   error(...); // вызывается без параметра, содержащего множество последующих символов;
               // в методе error синхронизация не производится
   while (sym ∉ anchors ∪ {eof}) scan(); // синхронизация
   errDist = 0;
}
```

Если `sym` не является символом, ожидаемым в точке синхронизации, то мы сообщаем об ошибке. Тогда символы исходной программы пропускаются до тех пор, пока не встретится ожидаемый якорь. И с этого места можно возобновить разбор. Процесс пропуска символов останавливается также после обнаружения `eof`, чтобы не зациклиться.

Преимущество этого метода в том, что ни методы парсера, ни метод `error()` не нуждаются в передаче множества последующих символов. Якоря в точках синхронизации можно статически вычислить по грамматике и не вычислять во время выполнения, что уменьшает размер парсера и повышает его эффективность. Обнаружив синтаксическую ошибку, парсер просто продолжает работать, пока не дойдет до точки синхронизации, в которой производится восстановление.

Для примера рассмотрим точку синхронизации в начале предложения. Ожидаемые там символы (ident, if, while, ...) хранятся в глобальной переменной firstStat:

```
private static BitSet firstStat = new BitSet();
...
firstStat.set(ident); firstStat.set(if_); firstStat.set(while_);
...
```

Затем с помощью метода syncStat мы создаем множество «безопасных» якорей; оно отличается от firstStat только отсутствием ident, потому что такой якорь был бы небезопасен, вместо него добавляется якорь eof:

```
private static BitSet syncStat;
...
syncStat = (BitSet) firstStat.clone();
syncStat.clear(ident); syncStat.add(eof);
```

Метод парсера для Statement теперь выглядит так:

```
private static void Statement() {
  if (! firstStat.get(sym)) { // синхронизация
     error("invalid start of statement");
     while (! syncStat.get(sym)) scan();
     errDist = 0;
   }
   // разобрать предложение
   if (sym == if_) {
      scan();
      check(lpar);
      Condition();
      check(rpar);
      Statement();
      if (sym == else_) { scan(); Statement(); }
   } else if (sym == while_) {
      ...
   }
```

Как видим, обработка ошибок и синхронизация производятся только в начале Statement. Больше в коде парсера ничего не изменилось. В нем не нужно вычислять множества якорей и передавать их в качестве параметров методам парсера или методу check(). Поэтому код парсера не разбухает и остается эффективным.

Кроме того, методу error() не нужно передавать множество якорей в качестве параметра, и он не пропускает ошибочные символы, т. к. это происходит в точках синхронизации (т. е. в начале Statement). Для подавления сообщений о наведенных ошибках используется эвристика расстояния между ошибками, описанная выше.

```
private static void error (String msg) {
   if (errDist >= 3) {
```

```
      System.out.println("line " + la.line + ", col " + la.col + ": " + msg);
      errors++;
   }
   errDist = 0;
}
```

Рассмотрим, как работает обработка ошибок со специальными якорями в процессе анализа следующего некорректного предложения на MicroJava:

```
if a > b , max = a;
```

Парсер входит в `Statement()` (см. выше), когда опережающим символом является `if`. Поскольку ключевое слово `if` может встречаться в начале предложения, ошибка не диагностируется и `if` потребляется методом `scan()`. При вызове `check(lpar)` обнаруживается ошибка, потому что следующим символом является `ident`. В сообщении об ошибке печатается `"( expected"`. Далее парсер продолжает работать и вызывает метод `Condition()`, который без ошибок разбирает `a > b`. Следующий вызов `check(rpar)` печатает сообщение об ошибке `(") expected")`. Парсер работает дальше, вызывает `Statement()` и приходит в точку синхронизации. Следующий символ `comma` не может встречаться в начале предложения, поэтому снова вызывается `error()`.

Но поскольку последняя ошибка была обнаружена менее чем три символа назад, сообщение об ошибке (наведенной) подавляется. Теперь парсер пропускает символы до тех пор, пока не встретится якорь из множества `syncStat` – в данном случае точка с запятой. Точка с запятой является допустимым началом предложения (пустого), поэтому ее можно использовать для продолжения разбора. Восстановление после ошибки прошло успешно, и отсутствие скобок вокруг `a > b` было диагностировано как ошибка. Фрагмент кода `", max = a"` был пропущен, и разбор продолжился с точки с запятой.

Как видим, обработка ошибок со специальными якорями синхронизируется только в нескольких местах, и до успешного восстановления некоторые участки кода могут быть пропущены. Однако эта техника гораздо проще обработки ошибок с общими якорями, и парсер получается более компактным и эффективным.

Синхронизация в начале повторения

Описанный выше метод можно улучшить. Если точка синхронизации находится в начале повторения, то восстановление может оказаться неоптимальным. Например, рассмотрим продукцию `Block`:

```
Block = "{" {Statement} "}".
```

Псевдокод метода парсера для `Block` выглядит так:

```
private static void Block() {
   check(lbrace);
   while (sym ∈ First(Statement)) {
     Statement();
```

```
   }
   check(rbrace);
}
```

Но если за `lbrace` не следует начало предложения, то входа в цикл не произойдет и точка синхронизации в `Statement()` достигнута не будет. Это препятствует синхронизации. Исправить это можно, переписав метод парсера следующим образом:

```
private static void Block() {
   check(lbrace);
   while (sym ∉ {rbrace, eof}) {
      Statement();
   }
   check(rbrace);
}
```

Вместо того чтобы проверять, является ли `sym` терминальным начальным символом `Statement`, мы проверяем, является ли он терминальным символом, который может следовать за последовательностью предложений (или `eof`, чтобы избежать зацикливания). В результате мы войдем в цикл и тогда, когда `sym` не является терминальным начальным символом `Statement`, и точка синхронизации в `Statement()` будет достигнута. В случае ошибки парсер будет пропускать символы до тех пор, пока не встретит начало следующего предложения, после чего синтаксический анализ можно будет продолжить. Так что немножко «подкрутить» здесь нужно.

Резюме

Обработка ошибок со специальными якорями требует небольшой модификации кода парсера. Синхронизация производится только в местах, где могут находиться особо «безопасные» символы, нигде больше в грамматике не встречающиеся. Весь остальной код парсера не меняется, поэтому парсер остается компактным и эффективным. Однако эта техника не такая систематическая, как обработка ошибок с общими якорями. Она требует определенного опыта и иногда нуждается в «подкручивании» для вычисления правильного множества якорей. В парсере MicroJava используется обработка ошибок со специальными якорями.

3.5. Упражнения

1. *Автомат с магазинной памятью*. Нарисуйте автомат с магазинной памятью для грамматики

   ```
   E = x | E "+" x.
   ```

   Как в примере на рис. 3.3, опишите, как разбирается вход `x + x` и какие состояния находятся в стеке на каждом шаге анализа.

2. *Рекурсивный спуск (1).* Напишите метод парсера для продукции `Assignment = ident "=" Expr ";".` В этом и последующих упражнениях используйте коды лексем, объявленные в разделе 2.3. Реализовывать обработку ошибок необязательно.
3. *Рекурсивный спуск (2).* Напишите методы парсера для следующих продукций:

```
VarDecl = ("static" | "final") Type ident {"," ident} ";".
Type    = ident.
```

4. *Рекурсивный спуск (3).* Напишите методы парсера для следующих продукций:

```
MethodCall = ident "(" [Parameters] ")" ";".
Parameters = Param {"," Param}.
Param      = ["out" | "ref"] ident.
```

5. *Рекурсивный спуск (4).* Напишите методы парсера для следующих продукций:

```
Signature = ["public" | "private"] ("void" | Type) ident "(" [ Param {"," Param} ]
            ")" ";".
Type      = ident ["[""]"]. Param = Type ident.
```

6. *Рекурсивный спуск (5).* Напишите методы парсера для следующих продукций:

```
EnumType   = Visibility "enum" ident "{" [IdentList] "}".
Visibility = [ "public" | "private" ].
IdentList  = ident {"," ident}.
```

7. *Свойство LL(1) (1).* Пусть дана следующая грамматика:

```
Decl = Type Variable ";"
     | Decl Type Variable ";".
Type = "int"
     | "string"
     | Type "*".
Variable = ident
     | ident "[" "]".
```

   (a) Найдите все LL(1)-конфликты.
   (b) Преобразуйте грамматику, так чтобы она обладала свойством LL(1).
8. *Свойство LL(1) (2).* Пусть дана следующая грамматика:

```
DeclStatList = DeclStat | DeclStatList DeclStat.
DeclStat     = Declaration | Statement.
Declaration  = Type ident ";".
Type         = "int" | ident.
Statement    = ident "=" Expr ";".
```

   (a) Найдите все LL(1)-конфликты.
   (b) Преобразуйте грамматику, так чтобы она обладала свойством LL(1).

9. *Свойство LL(1) (3)*. Пусть дана следующая грамматика (back, contents, cover, index, intro, page и preface – терминальные символы):

```
Book    = cover Heading Body back.
Heading = [preface] contents.
Body    = {Chapter} Chapter [index].
Chapter = intro | Chapter page.
```

   (a) Найдите все LL(1)-конфликты.
   (b) Преобразуйте грамматику, так чтобы она обладала свойством LL(1).

10. *Обработка синтаксических ошибок с общими якорями.* Напишите методы парсера для следующей грамматики и реализуйте обработку ошибок с общими якорями (раздел 3.4.2). Операции над множествами можете записывать на псевдокоде.

```
A = { b C | d }. C = [ c ] e.
```

    A – начальный символ этой грамматики, для которого последующим является eof.

11. *Обработка синтаксических ошибок со специальными якорями.* Напишите методы парсера для следующей грамматики и реализуйте обработку ошибок со специальными якорями (раздел 3.4.3).

```
Program = "program" ident {Declaration} "{" {Statement} "}".
Declaration = ("public" | "private") Type ident ";".
Type = ident ["[""]"].
```

    Поскольку ключевые слова public и private могут встречаться только в начале объявления, последнее является хорошей точкой синхронизации. Реализовывать метод парсера для Statement не нужно.

12. *Реализация парсера*. Реализуйте полный парсер MicroJava согласно грамматике в приложении А. Реализуйте обработку ошибок со специальными якорями и возьмите в качестве точек синхронизации места начала объявлений (в Program) и начала Statement.
    Затем скачайте с сопроводительного сайта файл TestParser.zip и протестируйте на нем свой парсер. Все синтаксические ошибки в программе BuggyParserInput.mj (находится в архиве TestParser.zip) должны быть диагностированы.

Глава 4

Атрибутные грамматики

Парсер только проверяет синтаксическую правильность исходной программы, трансляция – не его задача. Это делается на следующем этапе процесса компиляции, который называется *семантическим анализом*. Необходимо выполнить следующие действия.

- **Проверка контекстных условий**. Компилятор должен проверить условия, которые невозможно описать контекстно-свободными грамматиками, например требование о том, что каждое имя должно быть объявлено до использования или что типы операндов в выражениях должны быть совместимы. В результате могут быть найдены дополнительные (семантические) ошибки.
- **Управление таблицей символов**. Необходимо создать каталог всех объявленных имен, содержащий их свойства, например типы и адреса. Эта важнейшая структура данных компилятора называется *таблицей символов* (см. главу 5). В ней также хранятся типы данных, например массив или класс, и области видимости имен.
- **Генерирование кода**. Если ни синтаксических, ни семантических ошибок не обнаружено, то можно перейти к генерированию целевого кода. Для этого вызываются методы, которые генерируют команды для целевой машины, управляют метками переходов и обеспечивают выделение памяти для локальных переменных всех вызываемых в программе методов.

В однопроходном компиляторе (каковым является компилятор MicroJava) эти действия включены прямо в парсер. Мы обсудим их более подробно в главах 5 и 6. Но чтобы специфицировать эти действия, нам сначала придется ввести нотацию, которую можно будет использовать для описания процессов трансляции в простом и компактном виде, а именно *атрибутные грамматики*.

Атрибутные грамматики (АГ) предложил *Дональд Кнут* в работе [Knut68]. В оригинальной формулировке предполагается, что по исходной программе строится синтаксическое дерево, а затем оно обходится один или несколько раз, и по ходу дела его узлы снабжаются такими атрибутами, как типы или

адреса. Далее по этому снабженному атрибутами синтаксическому дереву генерируется код. Однако в этой книге мы будем использовать более простую форму атрибутных грамматик, в которой семантические действия выполняются прямо в процессе синтаксического анализа. Такая форма атрибутных грамматик называется *синтаксически управляемой трансляцией*. Она особенно хорошо подходит однопроходным компиляторам.

4.1. Компоненты атрибутных грамматик

Сначала мы рассмотрим компоненты атрибутных грамматик, а затем на примерах покажем, как их использовать (в том числе за пределами конструирования компиляторов в строгом смысле). В последующих главах, особенно в тех, что посвящены генерированию кода, мы будем широко пользоваться атрибутными грамматиками для описания трансляции программ на MicroJava.

4.1.1. Семантические действия

Семантические действия – это фрагменты кода (например, на Java), которые вставляются в грамматику в точке, где они должны быть выполнены во время разбора. Рассмотрим пример:

```
Number = digit {digit}.
```

Число состоит из последовательности цифр. В принципе, числа являются частью лексической структуры языка, но мы будем использовать их так, будто они составляют часть синтаксиса языка, чтобы пример получился проще.

Предположим, что «трансляция» чисел заключается в подсчете числа цифр в них. Это приводит к следующей грамматике:

```
Number =
   digit    (. int n = 1; .)
   { digit  (. n++; .)
   }        (. System.out.println(n); .) .
```

Как видим, семантические действия вставляются в программу в виде кусков кода, заключенных в скобки `(.` и `.)`, а затем выполняются парсером в той точке, где они оказались. После того как первая цифра распознана, счетчику `n` присваивается значение 1. Для каждой последующей цифры `n` увеличивается на единицу. В конце последовательности цифр `n` печатается.

Для простоты восприятия атрибутные грамматики записываются в табличной форме: синтаксическая конструкция слева, а семантические действия, выполняемые в случае распознавания соответствующих символов, – справа. Наша «трансляция» чисел дает следующие результаты:

```
123  => 3
4711 => 4
```

4.1.2. Выходные атрибуты

Синтаксические символы могут поставлять значения (*выходные атрибуты*), которые затем можно использовать в семантических действиях. Такие значения играют роль выходных параметров распознанных символов. Например, в результате распознавания цифры может быть порождено значение этой цифры в виде выходного атрибута `val`:

```
digit <↑val>
```

Атрибуты заключаются в угловые скобки (< и >). Направление потока обозначается стрелкой. Так, `digit` возвращает выходной атрибут `val`.

Теперь мы можем сделать трансляцию чуть более интересной – не просто подсчитывать цифры в числе, а вычислить и напечатать значение числа:

```
Number           (. int val, n; .)
= digit <↑val>
  { digit <↑n>  (. val = 10 * val + n; .)
  }              (. System.out.println(val); .) .
```

Первая цифра предоставляет значение `val`. Каждая последующая цифра предоставляет значение `n`, которое прибавляется к `val`, умноженному на 10. И в конце печатается значение числа.

4.1.3. Входные атрибуты

У нетерминальных символов могут быть также *входные атрибуты*, т. е. параметры, передаваемые им продукцией, в которой они встретились. Это позволяет параметризовать нетерминальные символы. В нашем примере входной атрибут `base` можно использовать для задания основания системы счисления (например, 10 или 16), в которой следует вычислять число:

```
Number <↓base, ↑val> (. int base, val, n; .)
= digit <↑val>
  { digit <↑n>        (. val = base * val + n; .)
  }.
```

`Number` имеет входной атрибут `base` и выходной атрибут `val`. Первая цифра дает начальное значение `val`, а каждая последующая – значение `n`, которое прибавляется к числу по формуле `base * val + n`. Результат не печатается, а возвращается продукцией `Number` в виде выходного атрибута `val`. Ниже показаны результаты такой трансляции:

```
base = 10, digit sequence = 123  => val = 123
base = 16, digit sequence = 123  => val = 291 (= 1*16^2 + 2*16^1 + 3*16^0)
```

Итак, атрибутная грамматика – это компактная нотация для описания процессов трансляции. Она состоит из трех частей.

- **Продукции в нотации РБНФ**. Например:

```
IdentList = ident {"," ident}.
```

- **Атрибуты** как параметры синтаксических символов, например:

```
ident <↑name>       // выходной атрибут (содержит результат трансляции)
IdentList <↓type>   // входной атрибут (предоставляет контекст)
```

- **Семантические действия** как произвольные последовательности предложений (например, на языке Java):

```
(. произвольная последовательность предложений  .)
```

4.2. Примеры

Атрибутные грамматики – чрезвычайно полезная и компактная нотация. Мы будем часто использовать их в последующих главах для описания трансляции программ на MicroJava. Но они полезны не только для конструирования компилятора как такового, поэтому ниже мы рассмотрим несколько примеров, демонстрирующих и такое использование.

4.2.1. Обработка объявлений переменных

Начнем с примера из компилятора MicroJava, показывающего, как обрабатываются объявления переменных. Объявления вида

```
int a, b, c;
```

начинаются типом, за которым следует список имен. Результатом объявления должна быть вставка объявленных имен и их типов в таблицу символов (Tab). Соответствующая атрибутная грамматика имеет вид:

```
VarDecl                 (. Struct type; .)
= Type <↑type>
  IdentList <↓type>
  ";" .

IdentList <↓type>      (. Struct type; String name; .)
= ident <↑name>        (. Tab.insert(name, type); .)
  { "," ident <↑name>  (. Tab.insert(name, type); .)
  } .
```

Type предоставляет описание типа в форме выходного атрибута типа Struct, который передается IdentList. В IdentList каждое объявленное имя вместе с его типом помещается в таблицу символов методом Tab.insert().

Но как атрибутная грамматика отображается в код компилятора? Для этого мы просто включаем атрибуты и семантические действия в код парсера. Атрибуты становятся параметрами методов парсера, а семантические действия вставляются в методы парсера как фрагменты кода. Результат выглядит следующим образом (атрибуты и семантические действия показаны серым шрифтом):

```
private static void VarDecl() {
   Struct type;
   type = Type();
   IdentList(type);
   check(semicolon);
}

private static void IdentList (Struct type) {
   String name;
   check(ident); name = t.val;
   Tab.insert(name, type);
   while (sym == comma) {
      scan();
      check(ident); name = t.val;
      Tab.insert(name, type);
   }
}
```

Входные атрибуты становятся параметрами, а выходные – возвращаемыми значениями методов. Выходные атрибуты терминальных символов берутся из последней распознанной лексемы `t`, например:

```
check(ident); name = t.val;
```

Как видим, атрибутные грамматики более компактны и их проще читать, чем методы парсера. Синтаксис более четко отделен от семантики. Представление альтернатив, факультативных элементов и повторений РБНФ компактнее, чем в коде парсера. Поэтому в последующих главах мы будем описывать процессы трансляции атрибутными грамматиками.

4.2.2. Вычисление константных выражений

В этом примере мы хотим разобрать и вычислить константное выражение, например:

```
3 * (2 + 4)
```

Это можно описать следующей атрибутной грамматикой:

```
Expr <↑val>            (. int val, val1; .)
= Term <↑val>
  { "+" Term <↑val1>   (. val = val + val1; .)
```

```
  | "-" Term <↑val1>   (. val = val - val1; .)
  }.

Term <↑val> (. int val, val1; .)
= Factor <↑val>
  { "*" Factor <↑val1> (. val = val * val1; .)
  | "/" Factor <↑val1> (. val = val / val1; .)
} .

Factor <↑val>          (. int val; .)
= number <↑val>
| "(" Expr <↑val> ")" .
```

Продукции `Expr`, `Term` и `Factor` возвращают значение в виде выходного атрибута. Затем эти значения комбинируются с другими значениями согласно указанному оператору. На рис. 4.1 показано воображаемое синтаксическое дерево константного выражения, в котором нетерминальные символы аннотированы значениями своих выходных атрибутов.

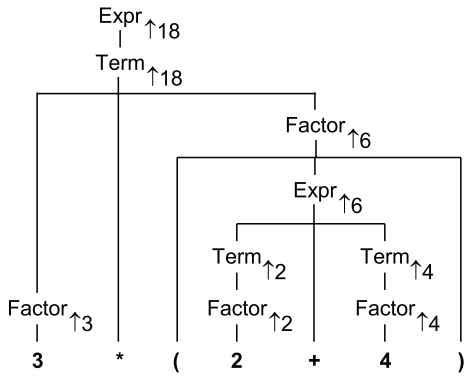


Рис. 4.1 ❖ Синтаксическое дерево константного выражения (с выходными атрибутами)

Снова рассмотрим, как атрибутные грамматики реализуются методами парсера. Для примера мы выбрали продукцию `Expr`:

```
private static int Expr() {
   int val, val1;
   val = Term();
   for (;;) {
      if (sym == plus) {
         scan(); val1 = Term(); val = val + val1;
      } else if (sym == minus) {
         scan(); val = Term(); val = val - val1;
      } else break;
   }
   return val;
}
```

Заметим, что выходной атрибут Expr возвращается в качестве значения функции. Поэтому метод Expr должен быть реализован как функциональный метод[1].

4.2.3. Статистика продаж

Следующий пример не относится к конструированию компиляторов, а описывает процесс включения данных о продажах в статистику продаж. Тем не менее атрибутные грамматики с успехом применимы и здесь.

Пусть имеется файл товаров (обозначаемых своими числовыми артикулами) с указанием объема продаж каждого. Строки файла заканчиваются точкой с запятой:

```
3451 2 5 3 7 ;
3452 4 5 1 ;
3453 1 1 ;
...
```

Мы хотим просуммировать и напечатать объемы продаж каждого товара:

```
3451 17
3452 10
3453 2
```

Входной файл обладает синтаксической структурой, а именно:

```
File = {Article}.
Article = Code {Amount} ";".
Code = number.
Amount = number.
```

При условии что вход имеет правильную структуру, мы можем обработать его, применяя методы конструирования компиляторов. Для этого сначала напишем атрибутную грамматику:

```
File                          (. int code, amount; .)
= { Article <↑code, ↑amount>  (. System.out.println(code + " " + amount); .)
  }.

Article <↑code, ↑amount>      (. int code, x, amount = 0; .)
= number <↑code>
  { number <↑x>               (. amount += x; .)
  }
  ";".
```

Теперь эту грамматику можно транслировать в методы парсера. В продукции Article два выходных атрибута. Но поскольку метод парсера может

[1] Функциональным автор называет метод, возвращающий значение (в отличие от метода, объявленного как void). – *Прим. перев.*

возвращать только одно значение, придется объединить оба атрибута в объект вспомогательного класса `ArticleInfo`:

```
class ArticleInfo {
  int code, amount;
}
```

Теперь можно реализовать методы парсера (атрибуты и семантические действия снова показаны серым шрифтом):

```
private static void File() {
   while (sym == number) {
      ArticleInfo a = Article();
      System.out.println(a.code + " " + a.amount);
   }
   check(eof);
}

private static ArticleInfo Article() {
   ArticleInfo a = new ArticleInfo();
   check(number); a.code = t.numVal; // значение числа берется из t.numVal
   while (sym == number) {
      scan(); int x = t.numVal;
      a.amount += x;
   }
   check(semicolon);
   return a;
}
```

Осталось только написать мини-сканер, который будет возвращать терминальные символы `number`, `semicolon` и `eof`. Но его-то реализовать очень просто.

Это простой пример, однако техника конструирования компиляторов помогла нам реализовать его систематически, так что программа получилась весьма компактной. Одна из целей данной книги – рассказать читателям о таких ситуациях. Всякий раз, как возникает необходимость в обработке структурированных данных, можно призвать на помощь методы конструирования компиляторов.

4.2.4. Язык описания изображений

Этот пример снова нельзя назвать компилятором, мы будем иметь дело с языком описания изображений. Изображение может состоять из нескольких фигур (прямоугольников, окружностей, многоугольников и т. д.), описанных в специальном формате. Здесь мы ограничимся (замкнутыми) многоугольниками. Многоугольник кодируется следующим образом:

```
POLY (10, 30) (50, 50) (40, 30) (50, 10) END
```

Он состоит из нескольких точек, определенных координатами x и y. Многоугольники такого вида можно прочитать из файла и нарисовать на экране (см. рис. 4.2).

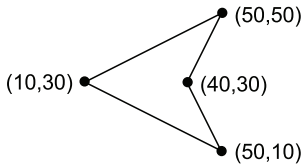


Рис. 4.2 ❖ Представление многоугольника на экране

Для рисования мы будем пользоваться так называемой «черепашьей графикой», когда перо (наподобие черепашки) бегает по поверхности и проводит линию. Нам понадобятся две операции:

```
Turtle.start(p);  устанавливает перо в точку p
Turtle.move(q);   перемещает перо в точку q, рисуя при этом линию.
```

Как опытные конструкторы компиляторов, мы начинаем с написания контекстно-свободной грамматики для описания многоугольников:

```
Polygon = "POLY" Point {Point} "END".
Point = "(" number "," number ")".
```

Теперь мы можем преобразовать эту грамматику в атрибутную, где Point должна возвращать выходной атрибут типа

```
class Pt { int x, y; }
```

Атрибутная грамматика выглядит следующим образом:

```
Polygon          (. Pt p, q; .)
= "POLY"
   Point <↑p>    (. Turtle.start(p); .)
   { Point <↑q>  (. Turtle.move(q); .)
   }
   "END"         (. Turtle.move(p); .) .

 Point <↑p>      (. Pt p = new Pt(); .)
 = "(" number <↑p.x>
   "," number <↑p.y>
   ")" .
```

Первая точка определяет начало многоугольника. Все остальные точки говорят, что перо нужно переместить в эту позицию. В конце мы перемещаем перо в начальную точку и тем самым замыкаем многоугольник. Точка определяется двумя числами, которые присваиваются координатам p.x и p.y.

Теперь осталось транслировать эту атрибутную грамматику в методы парсера и написать похожие продукции для других фигур (прямоугольников, окружностей и т. д.), после чего обработку языка описания изображений можно считать законченной. Как видим, техника конструирования компиляторов пригодилась и здесь тоже.

4.2.5. Преобразование из инфиксной в постфиксную нотацию

Последний пример будет несколько ближе к конструированию компиляторов. Мы хотим преобразовать арифметическое выражение из инфиксной нотации в эквивалентную постфиксную. В инфиксной нотации операторы ставятся между операндами, а в постфиксной – после операндов, например:

инфиксная нотация	*постфиксная нотация*
`3 + 4 * 2`	`3 4 2 * +`
`(3 + 4) * 2`	`3 4 + 2 *`

В первом постфиксном выражении сначала перемножаются операнды 4 и 2, а затем к результату прибавляется операнд 3. Эта задача может показаться сложной, но, вооружившись знаниями о конструировании компиляторов, мы легко решим ее. Идея в том, чтобы каждый операнд выводить немедленно, а вывод операторов отложить до тех пор, пока не станут известны оба операнда. Это позволяет сразу же написать атрибутную грамматику:

```
Expr
= Term
  { "+" Term     (. print("+ "); .)
  | "-" Term     (. print("- "); .)
 }.

Term
= Factor
  { "*" Factor   (. print("* "); .)
  | "/" Factor   (. print("/ "); .)
  }.

Factor
= number <↑val>  (. print(val + " "); .)
| "(" Expr ")".
```

Если множитель является числом, то его значение печатается немедленно. Если терм состоит из произведения `Factor * Factor`, то оба множителя уже были напечатаны в продукции `Factor`, так что оператор `"*"` нужно напечатать после них в продукции `Term`. Для термов ситуация аналогична: если мы имеем сумму `Term + Term`, то оба терма уже были напечатаны в продукции `Term`, поэтому оператор `"+"` печатается после них в продукции `Expr`. Если множитель

является выражением в скобках, то для его печати вызывается метод Expr, который, в свою очередь, вызывает методы Term и Factor для печати частей выражения. Чтобы понять, как работает эта атрибутная грамматика, взгляните на рис. 4.3, где изображено (воображаемое) синтаксическое дерево выражения

```
(3 + 4) * 2
```

аннотированное выходами элементов постфиксного выражения.

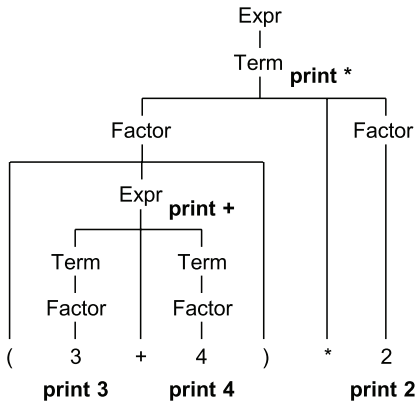


Рис. 4.3 ❖ Преобразование инфиксного выражения (3 + 4) * 2 в постфиксное выражение

Expr вызывает Term, а тот – Factor. Factor распознает открывающую скобку и вызывает Expr. Expr вызывает Term, а тот – снова Factor. Factor распознает число 3 и печатает его. Затем Factor и Term возвращаются. Expr распознает оператор "+", но пока не печатает его. Вместо этого он снова вызывает Term и Factor. Factor распознает число 4 и печатает его. Затем Factor и Term возвращаются. Теперь оба терма напечатаны, поэтому Expr выводит отложенный оператор "+". И обработка продолжается далее. Итогом является постфиксное выражение 3 4 + 2 *.

4.3. Упражнения

1. *Освоение атрибутных грамматик*. Пусть дана следующая атрибутная грамматика:

```
N <↑n>     (. int n, x; .)
= D <↑n>
  { D <↑x> (. n = 2 * n + x; .)
  }.
D <↑x>     (. int x; .)
```

```
= "0"      (. x = 0; .)
| "1"      (. x = 1; .).
```

(a) Что делает эта грамматика, т. е. что возвращает `N` в качестве выходного атрибута?

(b) Транслируйте обе продукции в методы парсера, которые также содержат атрибуты и семантические действия. Терминальные символы `"0"` и `"1"` можно обозначить в парсере лексемами `zero` и `one`.

2. *Римские числа.* Рассмотрим следующую контекстно-свободную грамматику, описывающую в упрощенной форме римские числа от 1 до 19 (`"I"` означает 1, `"V"` означает 5, а `"X"` означает 10):

```
RomanNumber = OneOrMore | FiveOrMore | TenOrMore.
OneOrMore   = "I" ["I" ["I" ["I"]]].
FiveOrMore  = "V" [OneOrMore].
TenOrMore   = "X" [OneOrMore | FiveOrMore].
```

Преобразуйте эту грамматику в атрибутную, так чтобы продукция `RomanNumber` возвращала целое значение римского числа в выходном атрибуте.

3. *Операции со временем.* Следующая контекстно-свободная грамматика описывает выражения для сложения и вычитания моментов времени:

```
TimeExpr = Time { "+" Time | "-" Time }.
Time     = Hours ":" Minutes.
Hours    = digit [digit].
Minutes  = digit digit.
```

Преобразуйте эту грамматику в атрибутную, которая возвращает число минут в выходном атрибуте `TimeExpr`, например:

```
3:24 + 1:00          => 264
21:15 + 0:30 - 12:00 => 585
```

Грамматика должна также проверять следующие контекстные условия:

- часы должны принадлежать диапазону 0..23;
- минуты должны принадлежать диапазону 0..59.

Можете предполагать, что `digit` возвращает выходной атрибут `val`.

4. *Присваивание адреса в объявлениях переменных.* Рассмотрим следующую контекстно-свободную грамматику для объявления переменных:

```
VarDeclList = VarDecl {VarDecl}.
VarDecl     = Type ident {"," ident} ";".
Type        = "boolean" | "char" | "int".
```

Преобразуйте эту грамматику в атрибутную, которая печатает адрес каждой объявленной переменной, а в конце общий объем памяти, необходимой для размещения всех переменных. Для переменных типа `boolean` требуется 1 байт, типа `char` – 2 байта, типа `int` – 4 байта. Для объявлений

```
boolean x, y;
int b;
char c;
```

вывод должен иметь вид:

```
x: 0
y: 1
b: 2
c: 6
Total = 8 bytes
```

5. *Курсы и слушатели*. Пусть дан текстовый файл, в котором описаны учебные курсы и их слушатели. У каждого курса есть имя и список слушателей согласно следующей грамматике:

```
Courses      = {Course}.
Course       = "course" CourseName ":" Participants.
Participants = {Participant}.
CourseName   = string.
Participant  = string.
```

   Преобразуйте эту грамматику в атрибутную, которая строит глобальную хеш-таблицу, содержащую для каждого слушателя список посещаемых им курсов. Используйте следующую глобальную структуру данных:

```
HashMap<String, ArrayList<String>> participants;
```

   В конце напечатайте этот список. Значение терминального символа `string` берите из `t.val`.

6. *Вычисление префиксных выражений*. Арифметические выражения можно записывать в префиксной нотации, когда оператор *предшествует* своим операндам (например, `+ a b`). Вот несколько примеров префиксных выражений и их эквивалентов в инфиксной нотации:

```
+ 3 * 6 2     = 3 + (6 * 2)
* + 1 2 3     = (1 + 2) * 3
+ * 2 4 * 6 8 = (2 * 4) + (6 * 8)
```

   (a) Напишите контекстно-свободную грамматику для префиксных выражений. Префиксное выражение либо является числом (терминальный символ `number`), либо начинается оператором `"+"` или `"*"`, за которым следуют два префиксных выражения (используется рекурсия). Правила предшествования для `"+"` и `"*"` можно игнорировать, потому что они неявно встроены в префиксную нотацию.

   (b) Преобразуйте свою грамматику в атрибутную, которая возвращает в выходном атрибуте значение префиксного выражения. Значение `number` можно брать из `t.numVal`.

7. *Скриптовый язык булевых выражений*. Следующая грамматика описывает небольшой язык, который можно использовать для вычисления значений булевых выражений, сохранения их в переменных и печати:

```
Program   = { Statement }.
Statement = ident "=" Expr ";"
          | "print" Expr ";".
```

```
Expr      = Term { "||" Term }.
Term      = Factor { "&&" Factor }.
Factor    = "true" | "false" | ident | "(" Expr ")" | "!" Factor.
```

Преобразуйте эту грамматику в атрибутную, которая позволяет вычислить значение булева выражения, сохранить его в переменной или напечатать. Храните переменные и их значения в глобальной хеш-таблице (`HashMap`). Ниже приведен пример программы:

```
big = true;
small = ! big;
ready = false;
print (big || small) && ready || big && ! ready;
```

8. *Трансляция XML в Json*. И XML (Extensible Markup Language), и JSON (JavaScript Object Notation) – языки для описания размеченных наборов данных. Набор данных XML, состоящий из нескольких элементов XML:

```
<person>
  <name>Miller</name>
  <city>London</city>
</person>
<person>
  <name>Smith</name>
  <city>Paris</city>
</person>
```

в нотации JSON представляется следующим образом:

```
[
  person: {
    name: Miller,
    city: London
  },
  person: {
    name: Smith,
    city: Paris }
]
```

В этом примере мы будем использовать упрощенную грамматику XML:

```
XML     = { Element } .
Element = "<" ident ">"
          ( ident
          | { Element }
          )
          "</" ident ">".
```

Преобразуйте эту грамматику в атрибутную, которая преобразует набор данных XML в набор данных JSON. Также проверяйте, чтобы имена тегов в `<ident>` и `</ident>` совпадали.

Глава 5

Таблица символов

Таблица символов – одна из главных структур данных компилятора, на нее возложены следующие задачи.

- Управление всеми объявленными именами и их свойствами, а именно:
  - тип данных;
  - для констант – значение;
  - для переменных, полей и методов – адрес;
  - для методов – параметры.

  Всякий раз, как имя используется в программе, производится его поиск в таблице символов и возвращаются его свойства (тип, значение, адрес, параметры).
- **Управление типами данных.** Простые типы данных (`int` и `char`) обозначаются кодом типа. Для структурных типов (массивов и классов) необходимо хранить также структуру.
- **Управление областями видимости**. С каждым именем ассоциирована область видимости, в которой оно объявлено. Например, имена могут быть локальными, глобальными или принадлежать классу.

Мы реализуем таблицу символов как класс `Tab` с такими методами, как `insert()` для вставки имен и `find()` для поиска имен. Таблица символов – динамическая структура данных, содержащая записи трех типов:

- *записи объектов*: в них хранится информация об объявленных именах;
- *записи структур*: в них хранится информация о типах данных;
- *записи областей видимости*: для управления областями видимости.

Для реализации таблицы символов можно использовать различные структуры данных: линейный список, двоичное дерево или хеш-таблицу. Двоичные деревья и хеш-таблицы хороши тем, что поиск в них эффективен, но за это приходится расплачиваться потерей информации о порядке следования имен. Поэтому в нашем компиляторе MicroJava с учетом того, что программы на этом языке короткие и объявлений немного, будут использоваться линейные списки – они проще и позволяют сохранить порядок объявлений.

Таблица символов как линейный список

Как уже было сказано, в нашей реализации все объявленные имена хранятся в связанном списке. Если имеются следующие объявления:

```
final int n = 10;
class T { ... }
int a, b, c;
void m() { ... }
```

то для каждого объявленного имени создается и добавляется в список запись объекта. Получается структура данных, показанная на рис. 5.1.

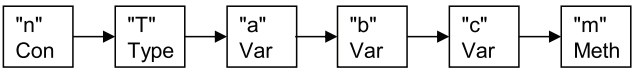


Рис. 5.1 ❖ Таблица символов как линейный список записей объектов

Линейными списками легко управлять, а поскольку в программах на MicroJava обычно очень мало объявлений, эти списки короткие, и поиск в них эффективен. И порядок имен сохраняется, что очень удобно для присваивания переменным последовательных адресов.

Теперь рассмотрим различные типы записей в таблице символов, их содержимое и способы управления ими.

5.1. Записи объектов

Для каждого объявленного имени компилятор создает объект и вставляет его в таблицу символов. В MicroJava имеются следующие виды объектов, каждому из которых присваивается свой числовой код:

```
static final int // виды объектов
   Con = 0,      // константа
   Var = 1,      // переменная или поле класса
   Type = 2,     // тип
   Meth = 3,     // метод
   Prog = 4;     // программа
```

Состав хранимых свойств зависит от вида объекта.

- Для всех объектов: имя, вид объекта, тип.
- Для констант: значение.
- Для переменных: адрес, уровень объявления (локальный или глобальный).
- Для типов: – (ничего; свойства хранятся в записях структур).
- Для методов: адрес, число параметров, список формальных параметров.
- Для программы: – (ничего).

Поскольку есть разные виды записей объектов, имеет смысл создать иерархию классов с общим суперклассом `Obj` и подклассами для различных видов объектов (рис. 5.2).

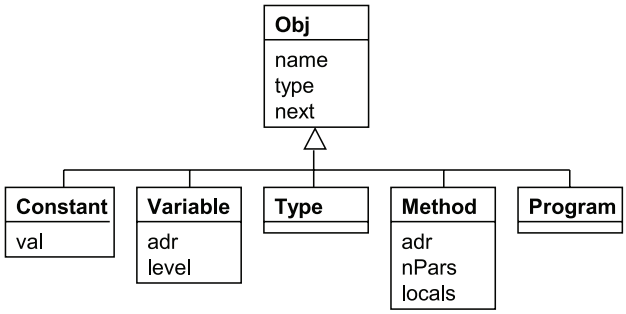


Рис. 5.2 ❖ Возможная иерархия классов для записей объектов

Однако использовать такой объектно ориентированный дизайн было бы неудобно из-за необходимости постоянно производить проверку и приведение типов, например:

```
Obj obj = Tab.find("someVar");
if (obj instanceof Variable) {
   Variable v = (Variable) obj;
   v.adr = ...;
   v.level = ...;
}
```

В результате поиска в таблице символов возвращается объект типа `Obj`, который затем обычно необходимо преобразовать в фактический тип объекта для дальнейшего использования. Проверки и приведения типов требуют времени и затрудняют чтение кода.

Поэтому для компилятора MicroJava мы решили остановиться на «плоской» структуре и объявить все поля экземплярами единственного класса `Obj`. При этом используются только поля, относящиеся к конкретному виду объекта.

```
class Obj {
   static final int Con = 0, Var = 1, Type = 2, Meth = 3, Prog = 4;
   int    kind;    // вид объекта: Con, Var, Type, Meth, Prog
   String name;    // имя объекта
   Struct type;    // тип объекта (см. ниже)
   Obj    next;
   int    val;     // для Con: значение
   int    adr;     // для Var и Meth: адрес
   int    level;   // для Var: 0 = глобальная, 1 = локальная
   int    nPars;   // для Meth: число параметров
   Obj    locals;  // для Meth: список параметров и локальных объектов
}
```

Снова рассмотрим следующие глобальные объявления на MicroJava:

```
final int n = 10;
class T { ... }
int a, b, c;
void m (int x) { ... }
```

Для каждого объявленного имени создается запись типа `Obj`, так что в результате получается следующий список (рис. 5.3):

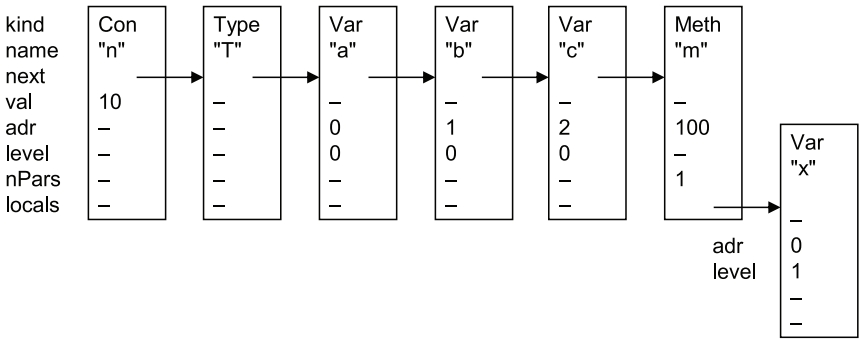


Рис. 5.3 ❖ Таблица символов и содержимое записей объектов

Мы видим, что поле `val` используется только для констант, а поле `adr` – только для переменных и методов. Адреса переменных определяют их местоположение в памяти. Они назначаются последовательно, детали этой процедуры мы рассмотрим в следующем разделе. Адреса методов определяют их местоположение в коде. В данном примере мы предполагаем, что метод `m()` имеет адрес 100. Поле `level` определяет уровень, на котором объявлены переменные. В предположении, что все показанные выше объявления глобальны, `level` равно 0. У метода `m()` всего один параметр, о чем говорит поле `nPars`. Поле `locals` указывает на формальные параметры метода, т. е. на список записей объектов. В методах назначение адресов параметрам и локальным переменным каждый раз начинается с 0, так что `adr` равно 0. Уровень объявления параметра `x` равен 1 (локальный).

В MicroJava глобальные и локальные переменные хранятся в разных областях памяти. Далее мы рассмотрим эти области.

5.1.1. Глобальные переменные

Глобальные переменные во время выполнения хранятся в *области глобальных данных* ВМ MicroJava. Каждая переменная занимает 1 слово (4 байта). Адреса назначаются последовательно в порядке объявления. Это номера слов относительно начала области данных. На рис. 5.4 приведен пример размещения глобальных переменных в области глобальных данных.

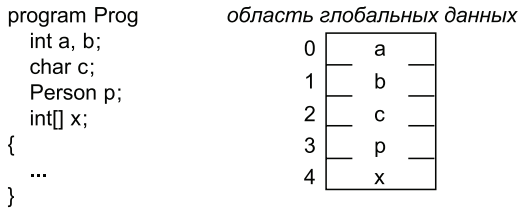


Рис. 5.4 ❖ Глобальные переменные и их размещение в области глобальных данных

Переменные типа `char` занимают полное слово, как и переменные, содержащие ссылку на объект класса (например, `Person`) или на массив.

5.1.2. Локальные переменные

Локальные переменные во время выполнения хранятся в *кадре стека* своего метода. При каждом вызове метода выделяется память для нового кадра стека, в котором будут находиться локальные переменные. Когда метод возвращает управление, эта память освобождается. Таким образом, кадры стека создаются динамически и организуются в виде стека. На рис. 5.5 показано, как локальные переменные размещаются в стеке метода.

Рис. 5.5 ❖ Локальные переменные и их размещение в стеке метода

Кадры стека выделяются в *стеке вызова методов* ВМ. На начало кадра текущего работающего метода указывает регистр *указателя кадра* (`fp`), а на его конец – регистр *указателя стека* (`sp`), который также указывает на текущий конец стека. Над кадром стека текущего метода находится кадр стека вызвавшего его метода, над ним – кадр стека вызвавшего его метода и так далее вплоть до самого первого метода `main()`.

В каждом кадре стека хранятся локальные переменные соответствующего метода в порядке их объявления. Каждая переменная занимает ровно одно слово (4 байта). Адресами являются номера слов относительно указателя кадра.

5.1.3. Вставка имен в таблицу символов

Каждое объявленное имя добавляется в таблицу символов методом `Tab.insert()`:

```
Obj obj = Tab.insert(kind, name, type);
```

Параметр `kind` – вид объекта имени (`Con`, `Var`, `Type`, `Meth`, `Prog`), а `type` – тип данных имени. Метод `insert()` создает новую запись объекта с указанными свойствами, проверяет, было ли уже объявлено такое имя, а затем добавляет запись в конец списка. Переменным и полям назначаются последовательные адреса, а для переменных задается еще и уровень объявления (глобальный или локальный). Метод возвращает запись объекта, чтобы вызывающая сторона могла продолжить работу с ней. Следующая атрибутная грамматика показывает, как обрабатываются объявления переменных.

```
VarDecl                (. Struct type; String name; .)
= Type <↑type>
  ident <↑name>        (. Tab.insert(Obj.Var, name, type); .)
  { "," ident <↑name> (. Tab.insert(Obj.Var, name, type); .)
  } ";" .
```

Параметр `kind` метода `insert()` принимает значение `Obj.Var`, потому что это объявления переменных. Тип данных переменных определяется нетерминальным символом `Type`. Это объект типа `Struct`, содержащий информацию о типе (см. раздел 5.3).

5.1.4. Предопределенные имена

В любом языке программирования есть имена с предопределенным значением. В MicroJava это:

- предопределенные типы: `int`, `char`;
- предопределенные константы: `null`;
- предопределенные методы: `ord(ch)`, `chr(i)`, `len(arr)`.

Эти имена вставляются в таблицу символов в самом начале компиляции (см. рис. 5.6). Можно сказать, что они известны заранее. Если впоследствии какое-то из них встретится в коде, то в таблице символов будет найдена и возвращена соответствующая ему запись объекта. С точки зрения использования, предопределенные имена ничем не отличаются от объявленных пользователем.

Можно было бы вместо этого рассматривать предопределенные имена как ключевые слова. Но это потребовало бы специальной обработки в компиляторе: если имя типа является ключевым словом, то с ним нужно ассоциировать предопределенный тип, тогда как для имени, объявленного пользователем, тип следует искать в таблице символов:

```
Type <↑type>
= ident <↑name> (. Obj obj = Tab.find(name); // имя типа – искать в таблице
                   type = obj.type; .)
| "int"         (. type = Tab.intType; .)    // ключевое слово – ассоциировать
                                               // предопределенный тип
| "char"        (. type = Tab.charType; .) . // ключевое слово – ассоциировать
                                               // предопределенный тип
```

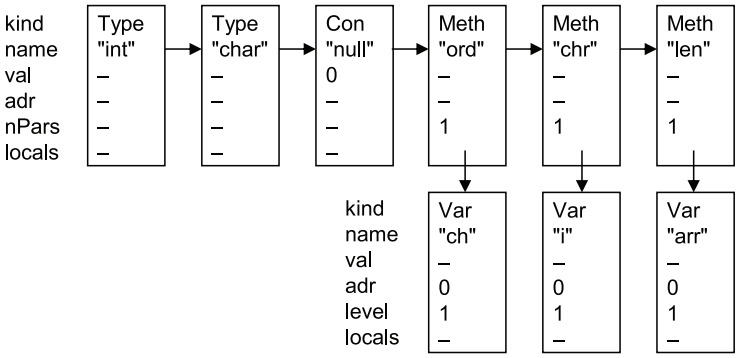


Рис. 5.6 ❖ Предопределенные имена в таблице символов

Поэтому проще обращаться с предопределенными и объявленными пользователем именами одинаково, как показано в следующей атрибутной грамматике, где обе разновидности имен типов представлены просто символом `ident`:

```
Type <↑type>
= ident <↑name> (. Obj obj = Tab.find(name); type = obj.type; .) .
```

5.2. Записи областей видимости

Любое объявленное имя видимо только к некоторой части программы (области видимости). Так, в MicroJava имеются глобальные имена, видимые во всей программе, локальные имена, видимые только в том методе, в котором объявлены, и имена полей, видимые только в своем классе. Наконец, предопределенные имена принадлежат специальной области видимости, которая называется универсальной. Итак, мы имеем следующие области видимости:

- область видимости программы: содержит глобальные имена;
- область видимости метода: содержит локальные имена;
- область видимости класса: содержит поля;
- универсальная область видимости: содержит предопределенные имена.

На рис. 5.7 показан пример программы на MicroJava и области видимости, которым принадлежат объявленные имена.

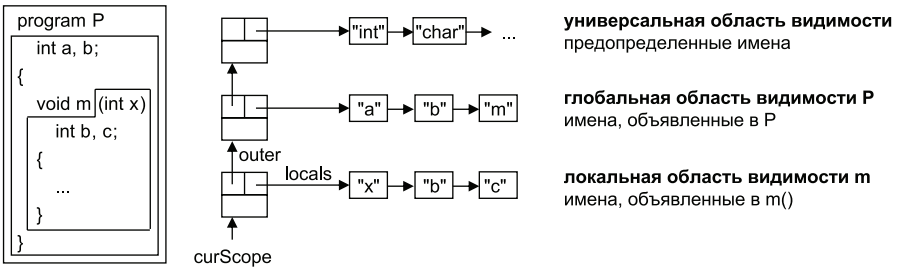


Рис. 5.7 ❖ Пример программы, области видимости и их содержимое

Как видим, таблица символов состоит из нескольких подсписков – по одному для каждой области видимости. Когда компилятор, проходя по исходной программе, оказывается в методе `m()`, текущей областью видимости (`curScope`) является область видимости `m()`; она содержит записи объектов для локальных переменных `x`, `b` и `c`. Ее объемлет область видимости программы `P`, содержащая записи объектов для глобальных переменных `a` и `b` и для метода `m()`. Наконец, самой внешней является универсальная область видимости, которая содержит записи объектов для всех предопределенных имен. Для каждой области видимости имеется запись `Scope`, которая указывает на локальные объекты, принадлежащие этой области видимости (`locals`), и на следующую объемлющую область видимости (`outer`).

Встретив в коде имя, компилятор ищет его во всех открытых областях видимости. Поиск всегда начинается в текущей области (`curScope`) и продолжается в объемлющих областях по указателям `outer`. Например, имя `b` на рис. 5.7 будет найдено уже в области видимости метода `m()`. Это ожидаемое поведение: локальная переменная `b` скрывает глобальную переменную `b`. Имя `a` не будет найдено в области видимости `m()`, но найдется в области видимости `P`. Наконец, имя `int` не будет найдено в областях видимости `m()` и `P`, но найдется в универсальной области видимости. Подсписки связаны записями областей видимости следующим образом:

```
class Scope {
   Scope outer;   // на объемлющую область видимости
   Obj locals;    // на локальные объекты в этой области видимости
   int nVars;     // число переменных, объявленных в этой области видимости
}
```

Когда начинается новая область видимости (в начале программы, метода или класса), вызывается метод `openScope()`, который создает пустую область видимости и связывает ее с существующими. Новая область видимости становится текущей (`curScope`), а ее уровень объявления `curLevel` увеличивается на единицу. `curScope` и `curLevel` – глобальные переменные класса `Tab`.

```
static void openScope() {
   Scope s = new Scope();
```

```
   s.outer = curScope;
   curScope = s;
   curLevel++;
}
```

В конце программы, метода или класса текущая область видимости закрывается методом `closeScope()`. В этот момент `curScope` снова указывает на объемлющую область видимости, а `curLevel` уменьшается на единицу. Иными словами, множество областей видимости управляется, как стек.

```
static void closeScope() {
   curScope = curScope.outer;
   curLevel--;
}
```

Методы `openScope()` и `closeScope()`, как и `insert()` и `find()`, принадлежат классу `Tab`. Метод `insert()` всегда вставляет имена в текущую область видимости (`curScope`).

Следующая атрибутная грамматика показывает, где именно в объявлениях методов вызываются `openScope()` и `closeScope()`.

```
MethodDecl      (. Struct type; String name; .)
= Type <↑type>
  ident <↑name> (. curMethod = Tab.insert(Obj.Meth, name, type);
                   Tab.openScope(); .)
  "(" ... ")"
  "{"           (. curMethod.locals = Tab.curScope.locals; .)
  ...
  "}"           (. Tab.closeScope(); .) .
```

Имя метода вставляется в область видимости программы (его вид равен `Meth`) еще до открытия новой области видимости. Параметры и локальные переменные методы вставляются уже в новую область видимости в момент своего объявления. Еще до обработки предложений метода список локальных переменных связывается с `curMethod.locals`, чтобы он не потерялся при закрытии области видимости в конце метода. Глобальная переменная `curMethod` ссылается на метод, над которым сейчас работает компилятор.

Для иллюстрации вышесказанного рассмотрим шаги обработки программы на рис. 5.7. В начале компиляции создается универсальная область видимости, которая становится текущей (рис. 5.8).

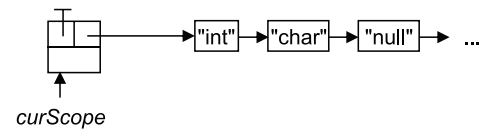


Рис. 5.8 ❖ Таблица символов в начале компиляции
(содержит предопределенные имена в универсальной области видимости)

После обработки ключевого слова `program` и имени программы `P` вызывается метод `openScope()`, и в новую область видимости вставляются обе глобальные переменные `a` и `b`, а также метод `m()` (рис. 5.9).

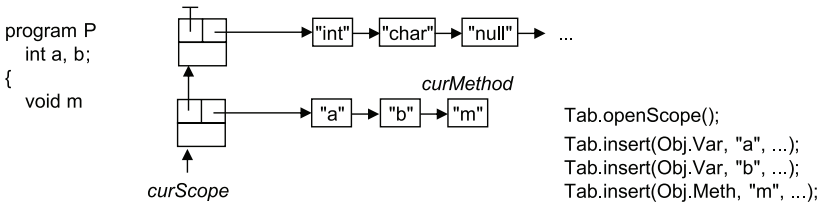


Рис. 5.9 ❖ Таблица символов после объявлений `a`, `b` и `m()`

После объявления имени метода `m` снова вызывается `openScope()`, и в новую область видимости вставляется параметр `x` и две локальные переменные `b` и `c`. Перед обработкой предложений `m()` объекты в текущей области видимости связываются с `curMethod.locals`, чтобы они не стали недоступными, когда впоследствии область видимости будет закрыта (рис. 5.10).

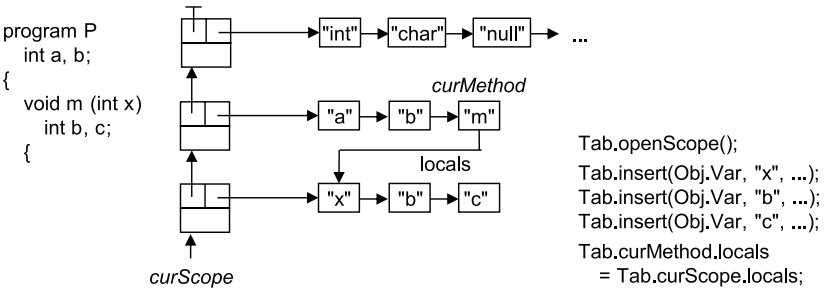


Рис. 5.10 ❖ Таблица символов
в начале обработки предложений метода `m()`

В конце метода `m()` область видимости закрывается методом `closeScope()`. Теперь `curScope` снова указывает на область видимости `P`. Однако локальные объекты `m()` все еще доступны по указателю `curMethod.locals` (рис. 5.11).

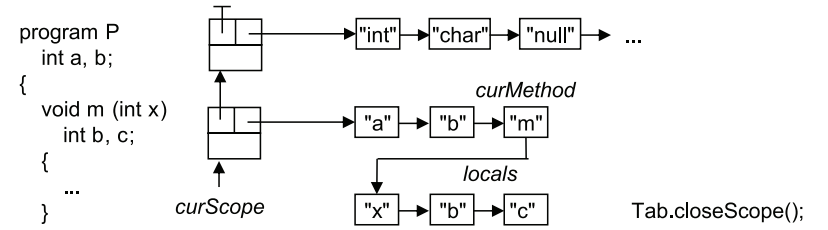


Рис. 5.11 ❖ Таблица символов по достижении конца `m()`

Если бы дальше следовало объявление другого метода, то была бы открыта новая область видимости, а по достижении конца метода она была бы закрыта. Иными словами, области видимости управляются, как стек. В нашем примере других объявлений методов нет, и программа на этом заканчивается. Таким образом, область видимости `P` закрывается методом `closeScope()`, и мы возвращаемся к состоянию, показанному на рис. 5.8.

5.2.1. Вставка имен в текущую область видимости

Теперь, зная, как управляются области видимости, рассмотрим реализацию метода `insert()`, который вставляет имена в текущую область видимости в момент объявления.

```
public static Obj insert (int kind, String name, Struct type) {
   //--- создать и инициализировать запись объекта
   Obj obj = new Obj(kind, name, type);
   if (kind == Obj.Var) {
      obj.adr = curScope.nVars; curScope.nVars++;
      obj.level = curLevel;
   }
   //--- вставить запись объекта в текущую область видимости
   Obj p = curScope.locals, last = null;
   while (p != null) {
      if (p.name.equals(name)) error(name + "declared twice");
      last = p; p = p.next;
   }
   if (last == null) curScope.locals = obj; else last.next = obj;
   return obj;
}
```

Объект создается и инициализируется своим видом (`Con`, `Var`, `Type`, `Meth`, `Prog`), именем и типом. Переменным назначается адрес в соответствии с текущим числом переменных в текущей области видимости (`curScope.nVars`). Так, первая переменная получает адрес 0, вторая – адрес 1 и т. д. Кроме того, уровень объявления переменных устанавливается равным `curLevel` – эта глобальная переменная увеличивается в `openScope()` и уменьшается в `closeScope()`.

Затем запись объекта вставляется в текущую область видимости (`curScope`). Для этого мы обходим список объектов и проверяем, нет ли в нем уже объявления такого имени. После этого объект добавляется в конец списка и возвращается вызывающей стороне для дальнейшей обработки.

5.2.2. Поиск имен в областях видимости

Встретив имя в исходной программе, компилятор ищет его во всех открытых областях видимости с помощью метода `find()`.

```
Obj obj = Tab.find(name);
```

Поиск начинается в текущей области видимости (curScope). Если там имя не найдено, то поиск продолжается в объемлющей области (см. рис. 5.7).

```
public static Obj find (String name) {
   for (Scope s = curScope; s != null; s = s.outer) { // для всех открытых областей
                                                      // видимости
      for (Obj p = curScope.locals; p != null; p = p.next) { // для всех объектов в этой
                                                             // области видимости
         if (p.name.equals(name) return p;
      }
   }
   error(name + " is undeclared");
   return noObj;
}
```

Если имя не найдено ни в одной области видимости, то вместо null возвращается объект ошибки (noObj). Это упрощает компилятор, потому что в противном случае ему пришлось бы при каждом доступе к объекту делать проверку на null, чтобы избежать некорректного доступа. Объект ошибки инициализируется таким образом, чтобы при последующих операциях доступа вероятность появления наведенных ошибок была мала. Например, в качестве вида объекта задается Obj.Var, в качестве имени – "$none" (такое имя недопустимо в программах на MicroJava), а в качестве типа – int.

5.3. Записи структур

В записях структур хранится информация о типе имени и в особенности о его структуре: для классов запоминаются поля, для массивов – тип элементов. В MicroJava существуют следующие типы, идентифицируемые кодом типа:

```
static final int
   None = 0,   // используется для void-методов
   Int = 1,    // int (примитивный тип)
   Char = 2,   // char (примитивный тип)
   Arr = 3,    // array (структурный тип)
   Class = 4;  // class (структурный тип)
```

Это ведет к следующему классу для записи структуры:

```
class Struct {
 static final int None = 0, Int = 1, Char = 2, Arr = 3, Class = 4; // виды типов
 int    kind;     // None, Int, Char, Arr, Class
 Struct elemType; // для Arr: тип элемента
 int    nFields;  // для Class: число полей
 Obj    fields;   // для Class: список полей
}
```

Некоторые поля `Struct` используются только для определенных типов: поле `elemType` – только для массивов, поля `nFields` и `fields` – только для классов. Для следующих (глобальных) объявлений на MicroJava:

```
int a, b;
char c;
```

– на рис. 5.12 показаны записи структур для типов `int` и `char`.

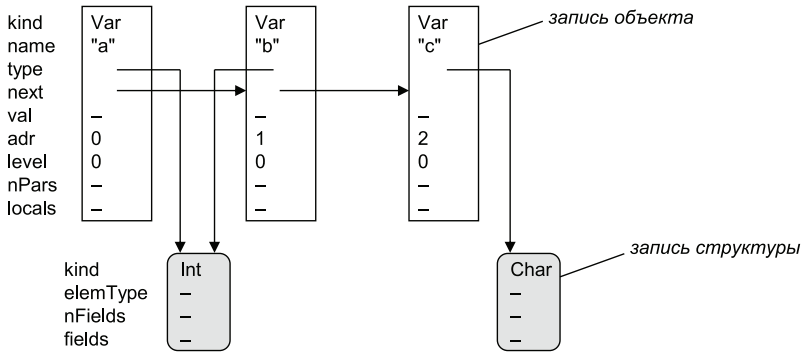


Рис. 5.12 ❖ Записи структур для `int` и `char`

Для каждого типа существует только одна запись структуры во всей таблице символов (это относится также к `int` и `char`). Все записи объектов, ссылающиеся на этот тип, указывают на одну и ту же запись структуры.

На рис. 5.13 показаны записи структур для массивов, соответствующие следующим (глобальным) объявлениям:

```
int[] a;
int b;
```

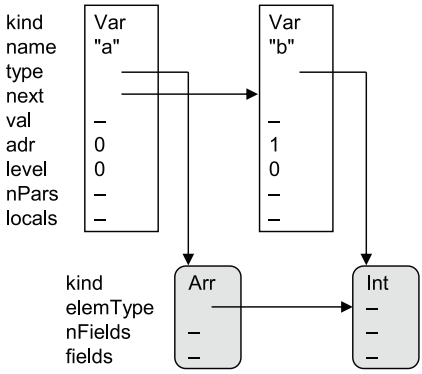


Рис. 5.13 ❖ Записи структуры для массивов

Переменная `a` имеет тип «массив `int`», тогда как тип `b` – просто `int`. Как легко видеть, имеется всего одна запись структуры для `int`. В MicroJava (как и в Java) длина массива неизвестна на этапе компиляции. Она определяется только на этапе выполнения и хранится вместе с массивом в куче.

Наконец, на рис. 5.14 показана запись структуры для класса со следующим (глобальным) объявлением:

```
class C {
   int x, y, z;
}
C obj;
```

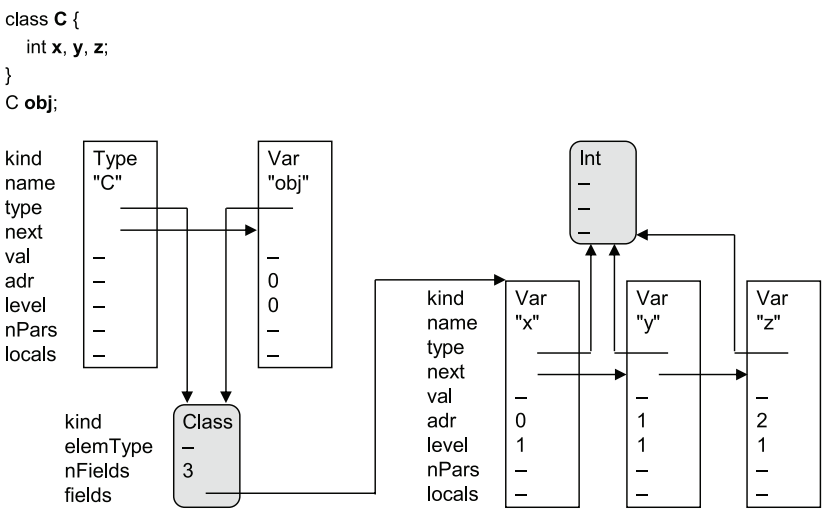


Рис. 5.14 ❖ Запись структуры для класса

Тип `C` – класс с тремя полями, представленными списком записей объектов. Адреса полей начинаются с 0, а уровень объявления равен 1. Здесь все три поля имеют тип `int`, а потому указывают на одну и ту же запись структуры для `int`. Переменная `obj` имеет тип `C`, поэтому структура объекта `obj` указывает на запись `Class` класса `C`.

Заметим, что именованные типы, такие как `C`, представлены двумя записями таблицы символов: *записью объекта*, описывающей имя `"C"`, и записью структуры, описывающей структуру `C`.

5.4. Проверка типов

Одна из задач компилятора – проверять совместимость типов операндов в присваиваниях, при передаче параметров, в выражениях и в других операциях. Это поднимает вопрос о том, когда можно считать типы эквива-

лентными. Существует два подхода: *эквивалентность по именам* и *структурная эквивалентность*. Мы рассмотрим их и сравним преимущества и недостатки.

5.4.1. Эквивалентность по именам

В этом случае два типа считаются одинаковыми, если они обозначаются одним и тем же именем типа. В следующих объявлениях:

```
class T { ... }
T x;
T y;
```

– переменные x и y имеют тип T. Имя типа одинаково, а значит, переменные принадлежат одному типу. На рис. 5.15 показано, что записи объектов x и y указывают на одну и ту же запись структуры (относящуюся к типу T). Поэтому для проверки того, принадлежат ли x и y одному типу, достаточно сравнить указатели: `objx.type == objy.type`; это просто и эффективно.

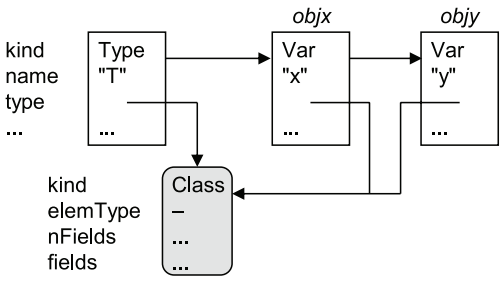


Рис. 5.15 ❖ Эквивалентность по именам – проверка типов: `objx.type == objy.type`

5.4.2. Структурная эквивалентность

В этом случае два типа считаются одинаковыми, если они имеют одинаковую *структуру* (для классов это значит, что одинаково количество полей и типы соответственных полей; для массивов – что одинаковы типы элементов). В следующих объявлениях:

```
class T1 { int a, b; }
class T2 { int c, d; }
T1 x;
T2 y;
```

– типы x и y структурно одинаковы. Хотя имена у них разные, структура одна и та же – каждый класс имеет два поля типа `int` (рис. 5.16).

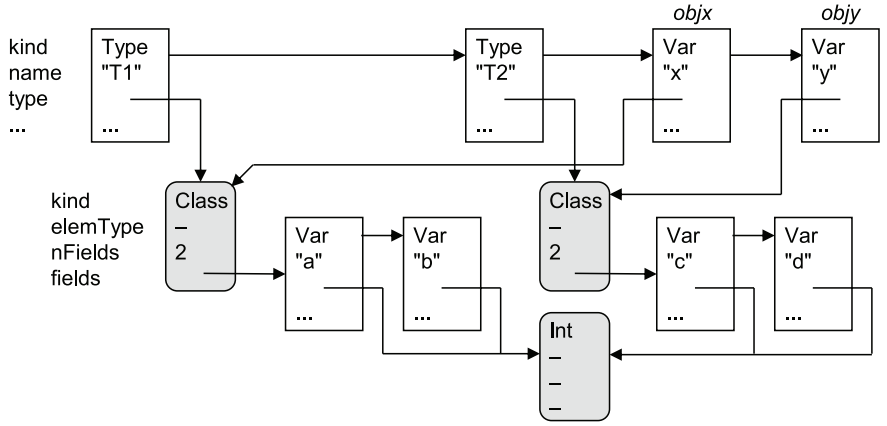


Рис. 5.16 ❖ Структурная эквивалентность.
Проверка типов: `objx.type.structEquivalentTo(objy.type)`

Проверка структурной эквивалентности гораздо сложнее проверки эквивалентности по именам. Если вы хотите убедиться, что типы `x` и `y` на рис. 5.16 структурно эквивалентны, то необходимо проверить, что у них одинаковое число полей структурно эквивалентных типов или (для массивов) что типы элементов структурно эквивалентны. Эта проверка может оказаться рекурсивной и обходится дорого.

Поэтому в большинстве языков программирования (например, в Pascal, C/C++, C# и Java) используется эквивалентность по именам. И лишь в немногих языках, например Algol68 или Modula-3, применяется структурная эквивалентность. В MicroJava мы также используем эквивалентность по именам, за одним исключением: для типов массивов применяется структурная эквивалентность. Для массивов

```
int[] x;
int[] y;
```

имена типов переменных `x` и `y` не совпадают, поэтому их записи объектов указывают на разные записи структур (см. рис. 5.17). Тем не менее массивы с одинаковыми типами элементов в MicroJava (и в Java) считаются эквивалентными.

5.4.3. Варианты совместимости типов

MicroJava требует разных видов совместимости типов в разных местах. Для арифметических операций типы операндов должны быть *одинаковы*, для присваиваний правая часть должна *допускать присваивание* левой части, а для сравнений типы операндов должны быть *сравнимы*. Поэтому мы объявляем в классе `Struct` разные методы для проверки этих вариантов совмес-

тимости типов. Метод `equals()` проверяет, что два типа одинаковы (структурно эквивалентны для массивов и эквивалентны по именам в остальных случаях).

```
public boolean equals (Struct other) { // проверяет, что this.equals(other)
   if (this.kind == Arr)
      return other.kind == Arr & other.elemType == this.elemType;
   else
      return other == this;
}
```

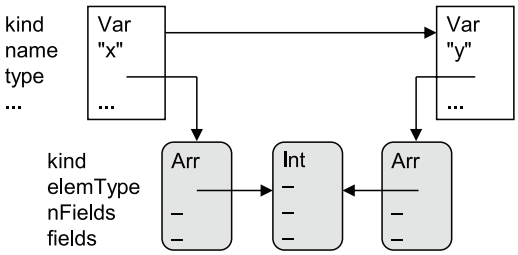


Рис. 5.17 ❖ Структурная эквивалентность массивов с одинаковыми типами элементов

Для присваиваний метод `assignableTo()` проверяет, что правая часть допускает присваивание левой. Так бывает, когда типы левой и правой частей одинаковы, когда правая часть равна `null`, а левая имеет ссылочный тип (класс или массив) или когда обе части являются массивами и элементы массива в левой части имеют специальный тип `Tab.noType`.

Последний случай в методе `assignableTo()` нуждается в объяснении: предопределенный метод `len(arr)` возвращает длину массива и должен быть применим к массивам элементов любого типа. Поэтому формальный параметр `len()` определен в таблице символов как «массив `noType`». Поскольку `assignableTo()` допускает присваивание любого массива «массиву `noType`», метод `len()` можно вызывать для любого массива.

```
public boolean assignableTo (Struct dest) { // проверяет, что this.assignableTo(dest)
   return this.equals(dest)
   || this == Tab.nullType && dest.isRefType()
   || this.kind == Arr & dest.kind == Arr & dest.elemType == Tab.noType;
}

public boolean isRefType() {
   return this.kind == Class || this.kind == Arr;
}
```

Для операций сравнения метод `compatibleWith()` проверяет, что оба операнда сравнимы. Это так, если оба типа одинаковы или если один операнд – `null`, а другой имеет ссылочный тип.

```
public boolean compatibleWith (Struct other) { // проверяет, что this.compatibleWith(other)
   return this.equals(other)
   || this == Tab.nullType && other.isRefType()
   || other == Tab.nullType && this.isRefType();
}
```

5.5. Разрешение LL(1)-конфликтов с помощью таблицы символов

Синтаксис методов в MicroJava нетрадиционный, поскольку объявление локальных переменных должно предшествовать блоку предложений:

```
void foo()
   int a;
{ a = 0; ...
}
```

Более привычно объявлять методы, как в Java:

```
void foo() {
   int a;
   a = 0; ...
}
```

Но это привело бы к LL(1)-конфликту, потому что тогда блок мог бы содержать и объявления, и предложения:

```
Block = "{" { VarDecl | Statement } "}".
```

И `VarDecl`, и `Statement` могут начинаться с `ident`, поэтому парсер, увидев `ident`, не смог бы решить, что идет дальше: объявление или предложение. Этот конфликт нелегко устранить путем преобразования грамматики – понять такую грамматику читателю было бы очень трудно.

Однако мы можем разрешить его, воспользовавшись семантической информацией в таблице символов. Объявления начинаются именем типа, а предложения – именем переменной или метода. Поэтому мы можем написать вспомогательный метод, который проверяет, что опережающий символ `sym` относится к имени типа:

```
private static boolean nextTokenIsType() {
   if (sym != ident) return false;
   Obj obj = Tab.find(la.val);   // искать ident в таблице символов ...
   return obj.kind == Obj.Type;  // ... и вернуть true, если он обозначает тип
}
```

Теперь парсер может вызвать этот метод, чтобы отличить объявления от предложений в методе `Block()` (ниже показан псевдокод):

```
private static void Block() {
   check(lbrace);
   for (;;) {
      if (nextTokenIsType()) VarDecl();
      else if (sym ∈ First(Statement)) Statement();
      else if (sym ∈ {rbrace, eof}) break;
      else {
         error("invalid declaration or statement");
         while (sym ∉ First(Statement) – {ident} ∪ {rbrace, eof}) scan(); // синхронизация
         errDist = 0;
      } // else
   } // for
   check(rbrace);
}
```

Увидев имя типа, парсер вызывает метод `VarDecl()`. Увидев допустимое начало предложения, он вызывает метод `Statement()`. Встретив закрывающую скобку или `eof`, парсер выходит из цикла. А во всех остальных случаях сообщает об ошибке и производит синхронизацию.

Как видим, иногда LL(1)-конфликты можно разрешить, используя семантическую информацию, ставшую доступной в процессе компиляции. Это позволяет анализировать почти любую грамматику методом рекурсивного спуска, даже если изначально она не обладает свойством LL(1).

5.6. Инициализация таблицы символов

В начале компиляции таблицу символов необходимо инициализировать, записав в нее предопределенные имена, и лучше всего сделать это в конструкторе класса `Tab`. Иными словами, нужно построить универсальную область видимости. Ее форма показана на рис. 5.18.

Некоторые записи объектов и структур экспортируются под специальными именами (например, `chrObj`, `intType`, ...), что подсказывает следующий интерфейс класса таблицы символов:

```
public class Tab {
   public static Scope curScope;  // текущая область видимости
   public static int curLevel;    // уровень вложенности текущей области видимости
   public static Struct intType, charType, nullType, noType; // предопределенные типы
   public static Obj chrObj, ordObj, lenObj, noObj;          // предопределенные объекты
   public static Obj insert (int kind, String name, Struct type) { ... }
   public static Obj find (String name) { ... }
   public static Obj findField (String name, Struct type) { ... }
   public static void openScope() { ... }
   public static void closeScope() { ... }
}
```

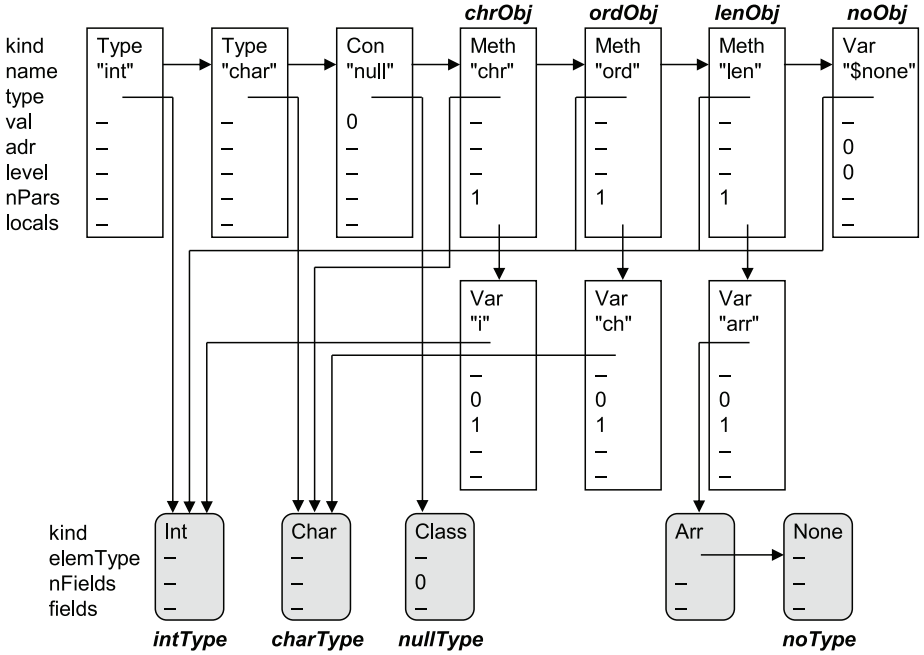


Рис. 5.18 ❖ Инициализация универсальной области видимости

5.7. Упражнения

1. *Таблица символов с простыми типами*. Пусть дана следующая программа:

```
program P
   final int a = 16;
   char b;
{
   void foo (int c, int d)
      char e;
   { ... // <== 1
   }
   void bar()
      int f, g;
   { ... // <== 2
   }
}
```

   (a) Нарисуйте таблицу символов (как на рис. 5.7) в момент, когда компилятор находится в точке 1 (в `foo()`). Нарисуйте все области видимости с записями объектов и структур. Также укажите значения полей в записях объектов и структур.

   (b) Нарисуйте таблицу символов в момент, когда компилятор находится в точке 2 (в `bar()`).

2. *Таблица символов с массивами*. Пусть дан следующий метод:

```
void foo()
   int[] a, b;
   int c;
   char[] d, e;
{ ... // <== 1
}
```

   Нарисуйте таблицу символов в момент, когда компилятор находится в точке 1. Вам нужно нарисовать только область видимости метода `foo()`.

3. *Таблица символов с классами*. Пусть дана следующая программа:

```
program P
   class C1 { int f, g; }
   class C2 { int h; char ch; }
{
   void foo()
      C1 a, b;
      C2 c;
   { ... // <== 1
   }
}
```

   Нарисуйте таблицу символов в момент, когда компилятор находится в точке 1. Нарисуйте области видимости `P` и `foo()`. Рисовать универсальную область видимости необязательно.

4. *Записи объектов и структур*. Почему для именованных типов, таких как `int` и `Person`, нужны и запись объекта, и запись структуры? Какая информация хранится в записи объекта, а какая в записи структуры?

5. *Проверка типов*. Объясните своими словами различие между эквивалентностью по именам и структурной эквивалентностью при проверке типов. Почему в Java (и MicroJava) в большинстве случаев используется эквивалентность по именам, но для массивов – структурная эквивалентность?

6. *Реализация структурной эквивалентности*. Реализуйте метод `t1.equals(t2)` в классе `Struct`, так чтобы он проверял структурную эквивалентность типов `t1` и `t2`.

7. *Параметры*. Почему список параметров хранится в записи объекта для метода и почему он сохраняется после закрытия области видимости метода?

8. *Косвенная рекурсия*. Язык MicroJava поддерживает прямую рекурсию, когда метод вызывает сам себя. Но он не поддерживает косвенную рекурсию, когда, например, метод `foo()` вызывает метод `bar()`, а тот снова вызывает `foo()`. Почему косвенная рекурсия не поддерживается в MicroJava?

9. *Реализация таблицы символов*. Реализуйте класс таблицы символов `Tab` для компилятора MicroJava, а заодно классы `Obj` для записей объектов, `Struct` для записей структур и `Scope` для записей областей видимости. Пользуйтесь описаниями в этой главе и интерфейсами классов в приложении B.

Глава 6

Генерирование кода

Мы подошли к последнему этапу процесса компиляции – генерированию кода. Это самый сложный и длительный этап, гораздо менее систематический, чем лексический или синтаксический анализ. Отчасти это связано с целевыми машинами и их системами команд, которые обычно переполнены тонкими деталями и зачастую весьма нерегулярны. Однако для своего компилятора мы используем в качестве целевой *виртуальную машину MicroJava* (μJVM) – простую и регулярную. Поэтому многих сложностей мы избежим. Перечислим основные задачи кодогенератора.

- **Генерирование машинных команд.** Для каждой исходной операции должны быть выбраны машинные команды, позволяющие добиться желаемого поведения. Иногда операцию можно реализовать разными командами, возможно, различающимися по эффективности. Кроме того, необходимо выбрать режим адресации для доступа к операндам. Например, к ячейке памяти можно обратиться по абсолютному адресу, по адресу относительно некоторой базы или с помощью индексирования.
- **Трансляция ветвлений и циклов в команды перехода.** В системе команд целевой машины нет таких конструкций исходного языка, как `if` и `while`. Поэтому поток управления необходимо выражать с помощью команд перехода.
- **Управление кадрами стека для локальных переменных.** У каждого метода имеются локальные переменные. При вызове метода необходимо выделить память для нового кадра стека, где эти переменные будут храниться, а в конце метода эту память нужно освободить. Это делается с помощью специальных команд, которые компилятору предстоит сгенерировать.
- **Возможные оптимизации**. Оптимизация – очень обширная тема. Обычно для нее предназначен отдельный этап, предшествующий генерированию кода, и обработке подвергается промежуточное представление программы (например, абстрактное синтаксическое дерево). Но в компиляторе MicroJava мы решили отказаться от сложных

оптимизаций, ограничившись только несколькими простыми, которые можно выполнить «на лету», в процессе генерирования кода. О более полной оптимизации см., например, работы [ALSU06, Appe02, Coop22, FCL09, Much97].

- **Вывод объектного файла.** В конце компиляции сгенерированный машинный код необходимо записать в файл, который затем может быть выполнен. Формат этого файла может быть весьма сложным, но для MicroJava, как мы увидим, он очень простой. Если в процессе компиляции были обнаружены ошибки, то объектный файл не создается.

Ниже описана общая процедура реализации кодогенератора.

1. **Изучение целевой машины.** Прежде всего вы должны досконально изучить конструкцию целевой машины. Для этого возьмите техническое руководство. Какие имеются регистры и для каких целей они предназначены? Какие поддерживаются форматы данных (целых чисел, чисел с плавающей точкой, битовых масок и т. д.)? Какие имеются режимы адресации? Какие есть команды и как они кодируются?
2. **Определение структур данных времени выполнения.** Размещение кадров стека, глобальных данных и объектов в куче определяется не целевой машиной, а проектировщиком компилятора, хотя на принятие решений могут оказать влияние вопросы интероперабельности с другими программами, работающими на той же машине.
3. **Управление областью кода.** Машинный код не записывается сразу в файл, а сначала сохраняется в области кода, потому что конечные адреса команд перехода иногда становятся известны позднее и только тогда могут быть включены в код. К управлению областью кода относится также кодировка команд в двоичном формате.
4. **Распределение регистров**. Регистровые машины имеют набор регистров, которыми нужно эффективно управлять. Однако ВМ MicroJava является *стековой машиной*, в которой вместо регистров используется *стек выражений*. Поэтому компилятор MicroJava не нуждается в распределении регистров.
5. **Генерирование кода**. Обычно это делается снизу вверх в следующем порядке:
   - загрузка переменных и констант в стек выражений;
   - обработка полей объектов и элементов массивов (например, `x.y` и `a[i]`);
   - обработка выражений (например, `a + 2 * b`);
   - управление переходами и метками;
   - обработка предложений;
   - обработка методов и параметров.

Мы будем придерживаться этого порядка и начнем с изучения целевой машины, т. е. ВМ MicroJava, ее памяти и системы команд. Дизайн ВМ MicroJava основан на упрощенной версии ВМ Java. В процессе изучения этой ВМ мы получим ценные знания о работе виртуальных машин вообще.

6.1. ВМ MicroJava

ВМ MicroJava (*μJVM*) – это *виртуальная машина*, процессор которой реализован программно и расположен между фактическим оборудованием (например, процессором Intel) и программой на MicroJava (рис. 6.1).

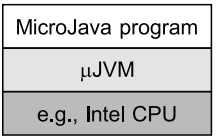


Рис. 6.1 ❖ μJVM – слой между оборудованием и программой на MicroJava

μJVM обладает своей собственной системой команд и ее интерпретатором. Программы на MicroJava транслируются в эти команды и исполняются интерпретатором μJVM.

Преимущество размещения ВМ между оборудованием и прикладным слоем заключается в том, что программы могут работать на любом оборудовании, коль скоро для него имеется реализация ВМ. Именно поэтому многие современные языки программирования, в частности Java и Python, используют ВМ в качестве исполняющей среды.

Как работает μJVM

μJVM – *стековая машина*, т. е. у нее нет регистров, а есть *стек выражений* (*EStack*), в который необходимо помещать значения операндов перед обработкой. Команды извлекают свои операнды из EStack и сохраняют результаты тоже в EStack. По сути дела, EStack используется как стек псевдорегистров. Рассмотрим пример с двумя локальными переменными `i` и `j`. Как мы увидим ниже, они находятся по некоторым адресам (в нашем случае 0 и 1) в *кадре* текущего метода в стеке вызовов методов (*MStack*). Для определенности предположим, что во время выполнения `i` и `j` принимают значения 3 и 4 соответственно (рис. 6.2).

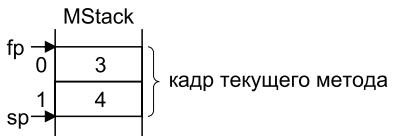


Рис. 6.2 ❖ Кадр текущего метода

Предложение MicroJava `i = i + j * 5;` транслируется в последовательность команд (см. также раздел 6.1.2), показанную на рис. 6.3 вместе с их воздействием на EStack:

команда	*EStac*	*объяснение*
load0	3	загрузить локальную переменную с адресом 0 (т. е. переменную i)
load1	3 4	загрузить локальную переменную с адресом 1 (т. е. переменную j)
const5	3 4 5	загрузить константу 5
mul	3 20	перемножить два верхних элемента стека
add	23	сложить два верхних элемента стека
store0		сохранить верхний элемент стека по адресу 0

Рис. 6.3 ❖ Выполнение команды `i = i + j * 5;`

Первые три команды загружают локальные переменные `i` и `j`, а также константу 5 в EStack. Команда `mul` перемножает два верхних элемента стека 4 и 5 и помещает результат 20 обратно в EStack. Команда `add` складывает два верхних элемента стека, 3 и 20, и помещает результат 23 обратно в EStack. Наконец, команда `store0` сохраняет верхний элемент стека по адресу 0, т. е. локальную переменную `i`.

Как видим, EStack используется только для хранения операндов *во время вычислений*, а в конце каждого предложения опустошается. Ячейки EStack играют роль регистров в регистровых машинах.

6.1.1. Области памяти

В µJVM имеются области памяти для данных и кода, их мы сейчас и рассмотрим.

Глобальные переменные

Глобальные переменные размещаются в массиве 32-разрядных слов фиксированного размера и хранятся на протяжении всего времени выполнения программы (рис. 6.4). Размер массива определяется компилятором, исходя из количества глобальных переменных.

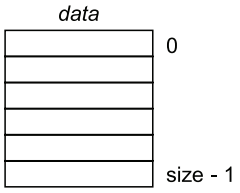


Рис. 6.4 ❖ Глобальные переменные

Каждая переменная занимает ровно одно слово (32 бита). Переменные адресуются номерами слов. Например, команда `getstatic 2` загружает глобальную переменную из слова 2 в области глобальных данных (`data`) в EStack.

Локальные переменные

Локальные переменные размещаются в *кадре стека* метода. При каждом вызове метода создается новый кадр в *стеке вызовов методов* (*MStack*), а по завершении метода этот кадр освобождается. Иными словами, кадры организованы в виде стека (рис. 6.5).

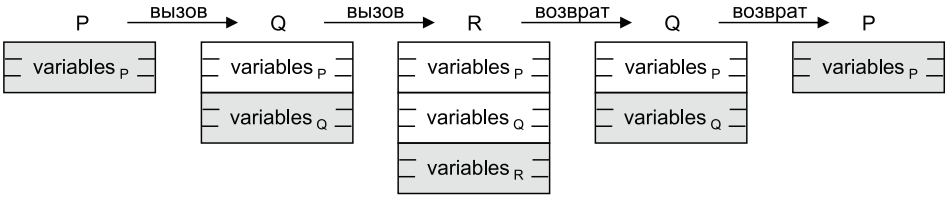


Рис. 6.5 ❖ Управление кадрами для локальных переменных по принципу стека

На рис. 6.6 показана организация стека вызовов методов.

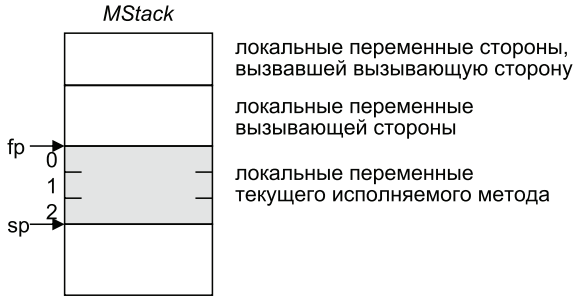


Рис. 6.6 ❖ Организация стека вызовов методов (MStack)

Кадр текущего исполняемого метода ограничен с одной стороны *указателем кадра* (`fp`), а с другой – *указателем стека* (`sp`). Последний также отмечает текущий конец MStack. Каждый кадр содержит локальные переменные метода, каждая из которых занимает ровно 1 слово (4 байта). Адресами являются номера слов относительно `fp`. Например, команда `load0` загружает локальную переменную с адресом 0 из кадра текущего исполняемого метода в EStack.

Куча

Куча содержит динамически выделенные объекты классов и массивы. Это массив слов фиксированной длины (см. рис. 6.7). Память для новых объектов выделяется командами `new` и `newarray`, начиная с позиции `free`, после чего `free` увеличивается на соответствующую величину. Ссылками на объекты служат адреса слов относительно начала кучи. В отличие от Java, в µJVM не предусмотрен сборщик мусора. Объекты, выделенные в куче, существуют на протяжении всего времени выполнения программы.

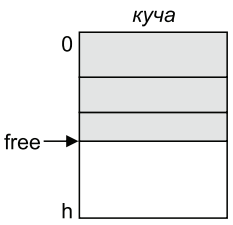


Рис. 6.7 ❖ Куча

Объекты классов. На рис. 6.8 показано размещение объекта класса C с тремя полями.

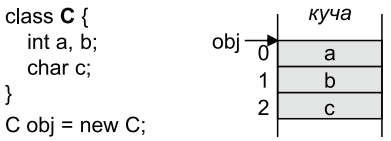


Рис. 6.8 ❖ Размещение объектов класса

Каждое поле занимает ровно одно слово (4 байта). Адресами полей служат номера слов относительно начала объекта.

Массивы. В первом слове массива хранится его длина, а затем следуют элементы. Размещение массивов слов и массивов `char` различается (рис. 6.9).

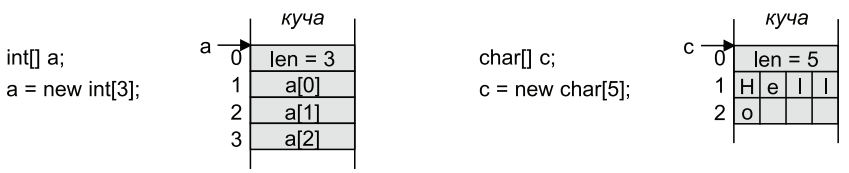


Рис. 6.9 ❖ Размещение массивов

В массиве слов каждый элемент занимает одно слово и адресуется номером слова. В массиве `char` каждый элемент занимает один байт и адресуется номером байта. Однако размер массива всегда кратен 4 байтам.

Код

Область кода в μJVM представляет собой массив байтов фиксированного размера, в котором команды методов хранятся в порядке объявления методов. Размер области кода определяется суммарным размером объявленных методов. Специальный регистр `pc` (*счетчик программы*) виртуальной машины всегда указывает на текущую выполняемую команду. Метод `main()` размещается по адресу `mainPc`, с него начинается выполнение программы (рис. 6.10).

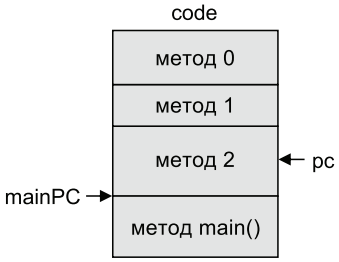


Рис. 6.10 ❖ Область кода

6.1.2. Система команд

µJVM обладает собственной системой команд, на которую транслируются программы на MicroJava. В основу положена система команд BM Java (JVM), но число команд меньше и в некоторых случаях они упрощены.

Команды µJVM очень компактны и зачастую занимают всего один байт, поэтому машинный код также называют *байт-кодом*. Но, в отличие от JVM, команды µJVM нетипизированные, т. е. типы операндов в команде не закодированы (рис. 6.11).

µJVM	JVM	
	для int	для float
load0	iload0	fload0
load1	iload1	fload1
add	iadd	fadd

Рис. 6.11 ❖ Сравнение нетипизированных команд µJVM с нетипизированными командами JVM

Например, чтобы сложить два значения, JVM выбирает различные команды в зависимости от того, принадлежат числа типу `int` или `float`. Это необходимо, потому что в состав JVM входит *верификатор байт-кода*, который проверяет, что команды применяются к операндам правильного типа, и только в этом случае разрешает их выполнение. Хотя проверка типов уже проделана компилятором, JVM проверяет типы еще раз, поскольку байт-код мог быть изменен. µJVM для простоты опускает эту проверку.

Формат команд

Формат команд µJVM гораздо проще, чем во многих других машинах. Программа состоит из последовательности команд. Каждая команда начинается байтом кода команды (`opcode`) и может иметь не более двух операндов размера 1, 2 или 4 байта:

```
Code        = {Instruction}.
Instruction = opcode {operand}.
```

Ниже приведены команды с разным числом операндов.

0 операндов	add	Два неявных операнда в EStack
1 операнд	load 7	Загружает локальную переменную по адресу 7
2 операнда	enter 1, 4	Выделяет память для кадра стека с 1 параметром и 4 локальными переменными

Команды без явных операндов (например, add) обычно имеют неявные операнды в EStack.

Режимы адресации

Команды обращаются к операндам, которые адресуются по-разному в зависимости от команды. μJVM поддерживает шесть режимов адресации, которые перечислены ниже вместе с примерами. В каждом режиме адресации семантика операндов различается.

- **Непосредственная** `const 7` *для констант*
  Операнд является константой и обозначает сам себя.
- **Локальная** `load 3` *для локальных переменных в MStack*
  Операнд является адресом локальной переменной относительно `fp`.
- **Статическая** `getstatic 3` *для глобальных переменных в* `data`
  Операнд является адресом глобальной переменной в `data`.
- **Стековая** `add` *для значений, загруженных в EStack*
  Явных операндов нет, но неявные операнды берутся из EStack. Например, `add` ожидает, что в EStack находятся два значения, подлежащих сложению.
- **Относительная** `getfield 3` *для полей объекта*
  Операндом является смещение поля объекта. Команда ожидает, что базовый адрес объекта находится в EStack (см. рис. 6.12).
- **Индексная** `aload` *для элементов массива*
  Команда ожидает, что неявные операнды – базовый адрес массива и значение индекса – находятся в EStack (см. рис. 6.12).

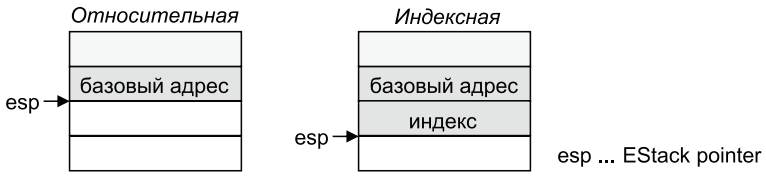


Рис. 6.12 ❖ EStack перед выполнением команды с относительной и индексной адресацией

В таблицах ниже показаны команды μJVM (второй столбец) вместе с кодом команды (первый столбец) и эффектом (четвертый столбец). В третьем столбце каждой таблицы показано содержимое EStack до и после команды. Формат

```
..., val, val
..., val
```

означает, что команда выбирает два значения из EStack и помещает в EStack новое значение. Операнды команд интерпретируются следующим образом:

`b`	байт (8-разрядный, со знаком)
`s`	короткий (16-разрядный, со знаком)
`w`	слов (32-разрядный, со знаком)

Операции `push()` и `pop()` ссылаются на EStack. `local[b]` означает локальную переменную по адресу `b` в текущем кадре, а `data[s]` – глобальную переменную по адресу `s`.

Загрузка и сохранение локальных переменных

Для доступа к локальным переменным предназначены следующие команды:

```
1     load b    ...          Загрузить локальную переменную
                ...,val        push(local[b]);
2..5  loadn     ...          Загрузить локальную переменную (короткая форма, n = 0..3)
                ..., val       push(local[n]);
6     store b   ..., val     Сохранить локальную переменную
                ...            local[b] = pop();
7..10 storen    ..., val     Сохранить локальную переменную (короткая форма, n = 0..3)
                ...            local[n] = pop();
```

Поскольку в большинстве методов есть всего несколько локальных переменных, младшие адреса встречаются гораздо чаще старших. Поэтому существуют короткие формы этих команд для адресов от 0 до 3 (`load0`, `load1`, `load2`, `load3`, `store0`, `store1`, `store2`, `store3`), которые кодируются одним байтом. Для больших адресов следует использовать длинные формы `load b` и `store b`, в которых адрес закодирован в дополнительном байте `b`. Поэтому длинная форма занимает 2 байта. Поскольку адрес локальной переменной кодируется байтом со знаком, всего можно адресовать не более 128 локальных переменных. Переменные типа `char` хранятся в младшем байте слова.

Загрузка и сохранение глобальных переменных

Следующие команды предоставляют доступ к глобальным переменным:

```
11    getstatic s   ...          Загрузить глобальную переменную
                    ...,val        push(data[s]);
12    putstatic s   ..., val     Сохранить глобальную переменную
                    ...            data[s] = pop();
```

Поскольку глобальных переменных часто больше, чем локальных, их адрес кодируется двумя байтами.

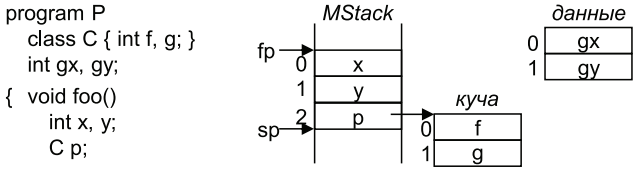
Загрузка и сохранение полей объектов

Для доступа к полям объектов предназначены следующие команды:

```
13    getfield s    ..., adr       Загрузить поле объекта
                    ..., val         adr = pop(); push(heap[adr+s]);
```

```
14    putfield s    ..., adr, val   Сохранить поле объекта
                    ...               val = pop(); adr = pop();
                                      heap[adr+s] = val;
```

getfield и putfield ожидают, что адрес объекта находится в EStack, а putfield дополнительно ожидает, что там же находится подлежащее сохранению значение. На рис. 6.13 показана трансляция операций присваивания локальным и глобальным переменным, а также полям объектов.



присваивание	*код*	*байтов*	*EStack*
x = y;	load1	1	y
	store0	1	–
gx = gy;	getstatic 1	3	gy
	putstatic 0	3	–
p.f = p.g;	load2	1	p
	load2	1	p p
	getfield 1	3	p p.g
	putfield 0	3	–

Рис. 6.13 ❖ Примеры присваивания переменным разных видов

Как видим, присваивание одной локальной переменной другой требует всего двух байт, тогда как присваивание одной глобальной переменной другой занимает шесть байт, а присваивание одного поля объекта другому – восемь байт. Это типичная и веская причина использовать локальные переменные всюду, где возможно.

Загрузка констант

Для загрузки констант предназначены следующие команды:

```
15..20  constn    ...          Загрузить константу (короткая форма, n = 0..5)
                  ..., val       push(n);
21      const_m1  ...          Загрузить минус единицу (короткая форма)
                  ..., -1        push(-1);
22      const w   ...          Загрузить константу
                  ..., val       push(w);
```

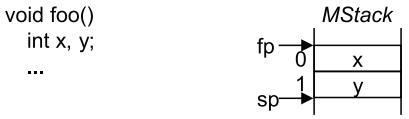
Для загрузки небольших констант (от 0 до 5) тоже существуют короткие формы, const0 ... const5, которые кодируются одним байтом, тогда как для загрузки больших констант следует использовать длинную форму, занимающую 5 байт. Поскольку константа –1 встречается часто, существует короткая форма const_m1 для загрузки значения –1.

Арифметические операции

Для арифметических операций и сдвигов предназначены следующие команды:

```
23  add          ..., val1, val2     Сложение
                 ..., val1+val2        push(pop() + pop());
24  sub          ..., val1, val2     Вычитание
                 ..., val1-val2        push(-pop() + pop());
25  mul          ..., val1, val2     Умножение
                 ..., val1*val2        push(pop() * pop());
26  div          ..., val1, val2     Деление
                 ..., val1/val2        x = pop(); push(pop() / x);
27  rem          ..., val1, val2     Остаток от деления
                 ..., val1%val2        x = pop(); push(pop() % x);
28  neg          ..., val            Отрицание
                 ..., – val            push(-pop());
29  shl          ..., val, x         Сдвиг влево
                 ..., val1             x = pop(); push(pop() << x);
30  shr          ..., val, x         Сдвиг вправо (арифметический)
                 ..., val1             x = pop(); push(pop() >> x);
31  inc b1, b2   ...                 Инкремент локальной переменной
                 ...                   local[b1] = local[b1] + b2;
```

Команда `inc b1, b2` служит для инкремента локальной переменной (например, в выражении `x++;`). Байт `b2` рассматривается как значение со знаком, т. е. команда позволяет также выполнять декремент (`x--;`). На рис. 6.14 показано несколько примеров арифметических операций и их трансляция в байт-код.



операция	*код*	*байтов*	*EStack*
x + y * 3	load0	1	x
	load1	1	x y
	const3	1	x y 3
	mul	1	x y*3
	add	1	x+y*3
x++;	inc 0,1	3	–
x--;	inc 0,-1	3	–

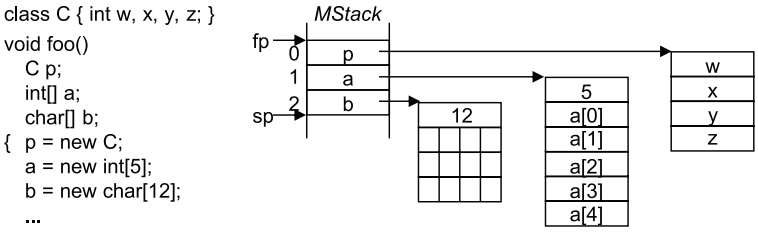
Рис. 6.14 ❖ Примеры арифметических операций

Выделение памяти для объекта

В MicroJava имеются объекты классов и массивов. Память для них выделяется следующими командами:

```
32  new s        ...          Новый объект класса
                 ..., adr       выделить область памяти размером s слов;
                                инициализировать область нулями;
                                push(adr(area));
33  newarray b   ..., len     Новый объект массива
                 ..., adr       len = pop();
                                if (b==0) выделить массив длиной len байт (+ слово длины);
                                else if (b==1)
                                  выделить массив длиной len слов (+ слово длины);
                                инициализировать массив нулями;
                                сохранить len в первом слове объекта массива;
                                push(adr(array));
```

При выделении памяти для объекта класса (`new s`) требуемый размер задается в словах. При выделении памяти для объекта массива (`newarray b`) добавляется переключатель `b`, который определяет, что создавать: массив слов или массив байтов. Длина массива не задается в явном операнде команды, а должна быть загружена в EStack заранее, потому что часто бывает выражением, значение которого неизвестно на этапе компиляции. Длина массива хранится в первом слове нового объекта массива и используется ВМ MicroJava для проверки индексов при доступе к массиву. На рис. 6.15 показаны примеры выделения памяти для объектов класса и массива (байтов и слов).



предложение	*код*	*байтов*	*EStack*
p = new C;	new 4	3	p
	store0	1	–
a = new int[5];	const5	1	5
	newarray 1	2	a
	store1	1	–
b = new char[12];	const 12	5	12
	newarray 0	2	b
	store2	1	–

Рис. 6.15 ❖ Примеры выделения памяти для объектов класса и массива

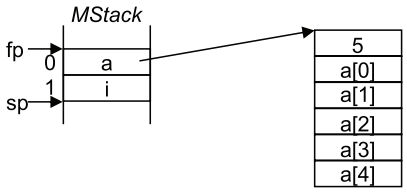
Доступ к массиву

Для доступа к массиву предназначены следующие команды:

```
34  aload        ..., adr, index     Загрузить элемент массива
                 ..., val              i = pop(); adr = pop();
                                       push(heap[adr+1+i]);
```

```
35  astore        ..., adr, index, val  Сохранить элемент массива
                  ...                     val = pop(); i = pop(); adr = pop();
                                          heap[adr+1+i] = val;
36  baload        ..., adr, index       Загрузить элемент массива байтов
                  ...,                    val i = pop(); adr = pop();
                                          x = heap[adr+1+i/4];
                                          push(byte i%4 of x);
37  bastore       ..., adr, index, val  Сохранить элемент массива байтов
                  ...                     val = pop(); i = pop(); adr = pop();
                  ...                     x = heap[adr+1+i/4];
                  ...                     присвоить байт i%4 в x значение val;
                  ...                     heap[adr+1+i/4] = x;
38  arraylength   ..., adr              Получить длину массива
                  ..., len                adr = pop();
                  ...                     push(heap[adr]);
```

Команды `aload` и `astore` обращаются к элементам массива слов, а команды `baload` и `bastore` – к элементам массива байтов. Они ожидают, что базовый адрес массива и индекс находятся в EStack, а `astore` и `bastore` дополнительно ожидают, что там же находится значение, подлежащее сохранению. Проверку индекса μJVM производит во время выполнения, исходя из хранимой длины массива. Команда `arraylength` возвращает длину массива. На рис. 6.16 показан пример операции доступа к массиву.



предложение	*код*	*байтов*	*EStack*
a[i] = a[i+1];	load0	1	a
	load1	1	a i
	load0	1	a i a
	load1	1	a i a i
	const1	1	a i a i 1
	add	1	a i a i+1
	aload	1	a i a[i+1]
	astore	1	–

Рис. 6.16 ❖ Примеры доступа к массиву

Как видим, необходимо предварительно загрузить операнды команды в EStack в правильном порядке. То, что здесь кажется «волшебством», обернется простой и систематической процедурой позже, когда мы перейдем к подробному обсуждению генерирования кода доступа к массиву.

Манипулирование стеком выражений

Следующие команды предназначены для манипулирования содержимым EStack:

```
39  pop   ..., val              Удалить верхний элемент стека
          ...                     dummy = pop();
40  dup   ..., val              Продублировать верхний элемент стека
          ..., val, val           x = pop(); push(x); push(x);
41  dup2  ..., v1, v2           Продублировать два верхних элемента стека
          ..., v1, v2, v1, v2     y = pop(); x = pop();
                                  push(x); push(y); push(x); push(y);
```

Команда `pop` используется, например, чтобы удалить из стека выражений значение, возвращенное функциональным методом, если оно не используется. Пример использования `dup2` показан на рис. 6.17 (переменные те же, что на рис. 6.16).

предложение	*код*	*байтов*	*EStack*
a[i]++;	load0	1	a
	load1	1	a i
	dup2	1	a i a i
	aload	1	a i a[i]
	const1	1	a i a[i] 1
	add	1	a i a[i]+1
	astore	1	–

Рис. 6.17 ❖ Пример использования `dup2`

Переходы

Как и любой компьютер, µJVM располагает командами условного и безусловного перехода:

```
42      jmp s                       Безусловный переход
                                      pc = pc + s;
43..48  j<cond> s   ..., val1, val2 Условный переход (cond = eq, ne, lt, le, gt, ge)
                      ...             y = pop(); x = pop();
                                      if (x cond y) pc = pc + s;
```

Безусловный переход выполняется всегда, а *условный переход* – только если ранее вычисленное условие истинно. Есть шесть видов команды условного перехода:

`jeq` перейти, если равно
`jne` перейти, если не равно
`jlt` перейти, если меньше
`jle` перейти, если меньше или равно
`jgt` перейти, если больше
`jge` перейти, если больше или равно

Команда условного перехода ожидает два значения в EStack, которые сравнивает в соответствии со своим условием и производит переход, если условие истинно. Длина перехода отсчитывается от счетчика программы `pc` (т. е. от начала команды перехода) и может быть положительной или отрицательной. Таким образом, возможны переходы как вперед, так и назад.

На рис. 6.18 показан пример условного перехода в предложении `if`. Если условие `if` равно `false`, то программа обходит ветвь `if`, в противном случае заходит в нее. Более подробно мы остановимся на этом вопросе в разделе 6.8.2. В этом примере `x` и `y` – локальные переменные с адресами 0 и 1.

предложение	*код*	*байтов*	*EStack*
if (x > y) y = x;	load0	1	x
	load1	1	x y
	jle 5	3	–
	load0	1	x
	store1	1	–

Рис. 6.18 ❖ Пример условного перехода
(длина перехода = 5 байт относительно команды перехода)

Вызовы методов

Следующие команды используются для вызова методов и возврата из них, а также для выделения и освобождения памяти для кадра стека, где находятся локальные переменные вызванного метода. Операции `PUSH()` и `POP()` ссылаются на MStack, а `push()` и `pop()` – на EStack.

```
49  call s          Вызвать метод
                        PUSH(pc+3); // адрес возврата
                        pc = pc + s;
50  return                    Вернуться
                        pc = POP();
51  enter b1, b2  ..., params   Войти в метод
                      ...  psize = b1; lsize = b2; // в словах
                        PUSH(fp); fp = sp; sp = sp + lsize;
                        initialize frame to 0;
                        for (i = psize-1; i >= 0; i--) local[i] = pop();
52  exit            Выйти из метода
                        sp = fp; fp = POP();
```

Функциональность этих команд довольно сложна, поэтому она иллюстрируется на рис. 6.19.

При вызове метода вызывающая сторона выполняет команду `call`, которая помещает адрес возврата `ra` (= `pc` + 3) в MStack и переходит на начало вызванного метода. Длина перехода отсчитывается от начала команды `call` и может быть положительной или отрицательной.

Первой командой каждого метода является `enter`. Она помещает указатель на кадр стека вызывающей стороны в MStack. µJVM использует этот

указатель, который называется *динамической связью* (dynamic link – `dl`), чтобы знать, как вернуться к кадру вызывающей стороны, когда метод вернет управление. Затем регистр `fp` устанавливается на конец MStack, и `sp` увеличивается на количество переменных в вызванном методе (`lsize`). Это действие создает новый кадр стека для локальных переменных метода. Содержимое кадра очищается, и все параметры (в количестве `psize`) копируются из EStack в начало кадра (см. раздел 6.9).

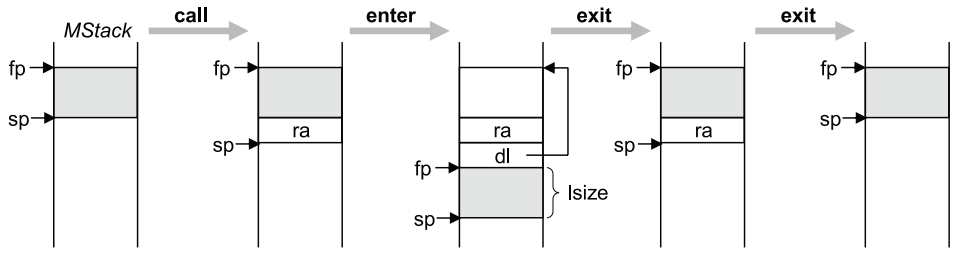


Рис. 6.19 ❖ Вызов метода и выделение памяти под новый кадр стека для локальных переменных

В конце метода выполняется команда `exit`, которая освобождает его кадр и – по динамической связи – устанавливает `fp` на начало кадра вызывающей стороны. Таким образом, восстанавливается состояние, имевшее место до выполнения `enter`.

Последняя команда метода – `return`. Она извлекает адрес возврата командой `POP()` и переходит по этому адресу, т. е. в точку после команды `call`. Теперь MStack снова находится в том состоянии, которое имело место перед выполнением команды `call`.

Ввод-вывод

µJVM поддерживает ввод и вывод целых чисел и литер на консоль. Для этого предназначены следующие команды:

```
53  read     ...                Читать целое
             ..., val             readInt(x); push(x);
54  print    ..., val, width    Печатать целое
             ...                  width = pop(); writeInt(pop(), width);
55  bread    ...                Читать литеру
             ..., val             readChar(ch); push(ch);
56  bprint   ..., val, width    Печатать литеру
             ...                  width = pop(); writeChar(pop(), width);
```

Команды `print` и `bprint` ожидают увидеть в EStack печатаемое значение и ширину поля. При печати значение выравнивается на правую границу этого поля. Если ширина поля слишком мала, то значение выводится в предположении минимально необходимой ширины.

Ошибки времени выполнения

Следующая команда выводит ошибку во время выполнения, а затем завершает программу:

```
57  trap b        Генерировать ошибку времени выполнения
                    напечатать сообщение об ошибке, зависящее от b;
                    прервать выполнение;
```

В программах на MicroJava есть только одно место, где генерируется команда `trap`, а именно когда функциональный метод достигает конца, не вернув значение с помощью `return`.

Пример

В заключение этого раздела рассмотрим полный метод и сгенерированный для него код (рис. 6.20). В методе объявлены четыре локальные переменные, размещенные в его кадре по адресам от 0 до 3. В начале метода для кадра выделяется память размером 4 слова с помощью команды `enter`. Поскольку у метода нет параметров, первый операнд команды `enter` равен 0. Как видим, предложения `if` и `while` транслируются в команды перехода. Как генерируются отдельные команды – тема последующих разделов.

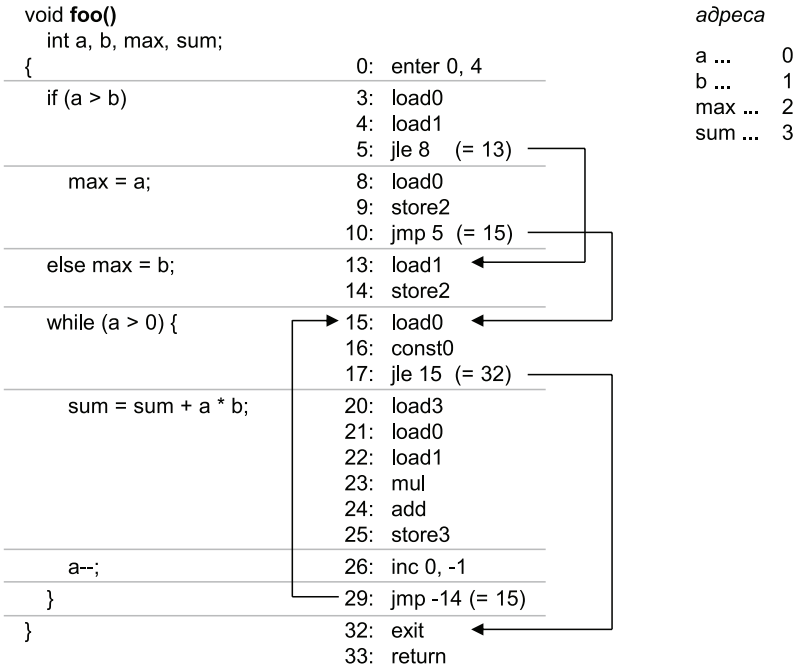


Рис. 6.20 ❖ Пример метода и сгенерированного для него кода

6.2. Буфер кода

Команды, сгенерированные компилятором, записываются не сразу в файл, а в буфер кода, поскольку некоторые команды перехода позже нужно будет скорректировать, вставив правильные длины переходов.

Буфер кода – это массив байтов внутри кодогенератора (класс `Code`). Переменная `pc` указывает на следующую свободную ячейку в буфере кода. Для генерирования команд предназначены методы `put()`, `put2()` и `put4()`, которые добавляют 1, 2 или 4 байта в буфер кода.

```
class Code {
   static final int // коды команды
      load = 1, load0 = 2, load1 = 3, load2 = 4, load3 = 5,
      store = 6, store0 = 7, store1 = 8, store2 = 9, store3 = 10,
      getstatic = 11, putstatic = 12, ...;
   private static byte[] code = new byte[10000]; // буфер кода
   public static int pc = 0;
   public static void put (int x) { code[pc++] = (byte) x; }
   public static void put2 (int x) { put(x >> 8); put(x); }
   public static void put4 (int x) { put2(x >> 16); put2(x); }
   ...
}
```

Благодаря простому формату команд µJVM генерирование команд также не вызывает затруднений. Например, команды `load0`, `store 7`, `getstatic 2` и `const 10` можно сгенерировать следующим образом:

```
Code.put(Code.load0);                    // load0
Code.put(Code.store); Code.put(7);       // store 7
Code.put(Code.getstatic); Code.put2(2);  // getstatic 2
Code.put(const); Code.put4(10);          // const 10
```

6.3. Дескрипторы операндов

Вот мы и подошли собственно к генерированию кода и начнем с понятия, важного для систематического подхода: *дескрипторы операндов*. Чтобы сгенерировать код конкретной операции, нам нужна информация о ее операндах. Например, общая схема сложения двух значений такова:

```
... загрузить операнд 1 ...
... загрузить операнд 2 ...
сложить
```

Но какие команды следует сгенерировать для загрузки операндов? Конечно, это зависит от операндов; для операндов разного вида нужны разные команды загрузки.

вид операнда	*требуемая команда загрузки*
константа c	`const c`
локальная переменная по адресу a	`load a`
глобальная переменная по адресу a	`getstatic a`
поле со смещением a	`getfield a`
элемент массива	`aload`
значение в EStack	`-` (ничего)
функциональный метод по адресу a	`call a`

Таким образом, дескриптор необходим, чтобы сообщить нам о виде операнда (а значит, и его режиме адресации), а также передать другую информацию: адрес (для переменных), значение (для констант) и т. д. В следующей таблице показано, какая информация необходима для каждого вида операнда (см. также описание режимов адресации в разделе 6.1.2).

вид операнда	*обозначается*	*требуемая информация*
константа	`Con`	значение константы
локальная переменная	`Local`	адрес в кадре стека
глобальная переменная	`Static`	адрес в `data`
поле	`Fld`	смещение относительно начала объекта
элемент массива	`Elem`	– (информация находится в EStack)
значение в EStack	`Stack`	– (информация находится в EStack)
метод	`Meth`	адрес в коде, объект метода

Эта информация определена в классе `Operand`:

```
class Operand {
   static final int   // виды операндов
      Con = 0, Local = 1, Static = 2, Stack = 3, Fld = 4, Elem = 5, Meth = 6;
   int kind;          // Con, Local, Static, Stack, Fld, Elem, Meth
   Struct type;       // тип операнда
   int val;           // для Con: значение константы
   int adr;           // для Local, Static, Fld, Meth: адрес
   Obj obj;           // для Meth: объект метода с формальными параметрами
}
```

Как и в случае таблицы символов в главе 5, некоторые поля используются только для определенных видов операндов, например `val` – только для констант, а `obj` – только для методов.

Обычно дескрипторы операндов получают свои данные от объектов в таблице символов. Когда компилятор видит имя в программе, он ищет его в таблице символов и создает дескриптор операнда по найденному объекту, пользуясь следующим конструктором:

```
public Operand (Obj obj) {
   type = obj.type; val = obj.val; adr = obj.adr;
   switch (obj.kind) {
      case Obj.Con: kind = Con; break;
```

```
      case Obj.Var: if (obj.level == 0) kind = Static; else kind = Local; break;
      case Obj.Meth: kind = Meth; this.obj = obj; break;
      case Obj.Type: error("a type is not a valid operand"); break;
      case Obj.Prog: error("the program cannot be used as an operand"); break;
   }
}
```

Существует еще один конструктор, который можно использовать для создания дескриптора операнда по целочисленной константе:

```
public Operand (int val) {
   kind = Con; type = Tab.intType; this.val = val;
}
```

Если дескрипторы операндов все равно создаются из записей объектов, то почему бы не использовать сами эти записи вместо дескрипторов? Тому есть две причины. Во-первых, операнды могут представлять не только простые имена, хранящиеся в таблице символов, но и такие определители, как `a[i]` или `x.f`. Во-вторых, операнды описывают сущности, местоположение которых может изменяться в процессе генерирования кода. Например, локальная переменная первоначально находится в MStack. Но после загрузки она находится уже в EStack. Следовательно, местоположение изменилось, что выражается изменившимся дескриптором операнда (см. рис. 6.21). С другой стороны, записи объектов в таблице символов не изменяются никогда.

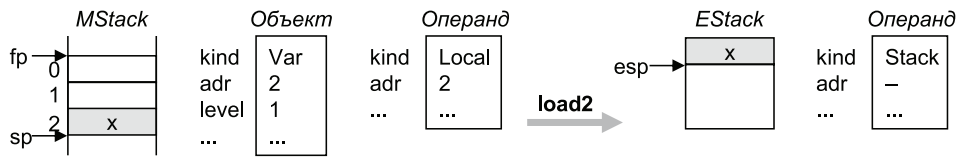


Рис. 6.21 ❖ Переменная x в MStack и после ее загрузки в EStack

Дескрипторы операндов создаются в процессе генерирования кода и описывают вид и местоположение операндов. Большинство методов парсера возвращают дескриптор операнда в качестве результата трансляции. Затем этот дескриптор используется для генерирования кода стороной, вызывающей метод парсера; это может привести к созданию нового дескриптора операнда, который снова возвращается соответствующим методом парсера. На рис. 6.22 показаны иерархия вызовов методов парсера и возвращаемые ими дескрипторы операндов.

```
x = y + z * 3;
```

где `x`, `y` и `z` – локальные переменные с адресами 0, 1 и 2.

`Statement` вызывает `Designator`, который распознает имя `x`, ищет его в таблице символов и преобразует найденный объект в дескриптор операнда вида `Local` с адресом 0 (`Local0`). Этот операнд возвращается методом `Designa-`

tor. Затем парсер продолжает работу, распознает "=", вызывает метод Expr, а вслед за ним Term и Factor. Метод Factor распознает имя y и создает дескриптор операнда Local1, который возвращают методы Factor и Term. Прежде чем продолжить, парсер генерирует команду load1 в точке ①, которая загружает операнд Local1 в EStack и изменяет вид своего операнда на Stack.

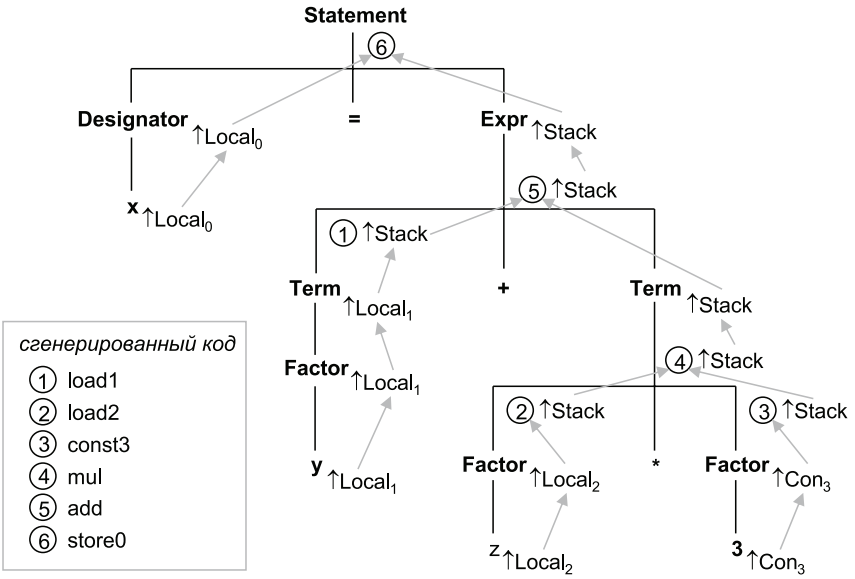


Рис. 6.22 ❖ Иерархия вызовов методов парсера и возвращаемые ими операнды

Теперь парсер распознает "+", вызывает метод Term, который вызывает Factor. Последний распознает имя z и создает для него дескриптор операнда Local2. Прежде чем продолжить, парсер генерирует команду load2 в точке ②, которая загружает операнд Local2 в EStack, снова создавая операнд вида Stack. Затем парсер распознает "*" и вызывает метод Factor. Тот обнаруживает константу 3 и создает для нее дескриптор операнда Con3, который загружается в EStack в точке ③ командой const3.

Теперь оба множителя находятся в EStack, так что их можно перемножить командой mul в точке ④. Результат умножения находится в EStack, поэтому Term возвращает операнд вида Stack. Теперь оба терма находятся в EStack, так что их можно сложить командой add в точке ⑤. Результат сложения снова находится в EStack, поэтому Expr возвращает операнд вида Stack. Так как метод Designator вернул операнд Local0, компилятор знает, что должен присвоить значение, находящееся в EStack, локальной переменной по адресу 0; это делается командой store0 в точке ⑥.

Как видим, дескрипторы операндов предоставляют компилятору всю информацию, необходимую для генерирования команд. Это общий принцип генерирования кода, который мы будем использовать в последующих разделах.

6.4. Загрузка значений

Прежде чем μJVM сможет обработать значения, их необходимо загрузить в EStack. Показанный ниже метод `load()` из класса `Code` принимает операнд любого вида и генерирует команду для загрузки его в EStack.

```
public static void load (Operand x) { // в классе Code
   switch (x.kind) {
      case Operand.Con:
         if (0 <= x.val && x.val <= 5) put(const0 + x.val);
         else if (x.val == -1) put(const_m1);
         else { put(const); put4(x.val); }
         break;
      case Operand.Static:
         put(getstatic);
         put2(x.adr);
         break;
      case Operand.Local:
         if (0 <= x.adr && x.adr <= 3) put(load0 + x.adr);
         else { put(load); put(x.adr); }
         break;
      case Operand.Fld: // assert: базовый адрес объекта находится в EStack
         put(getfield); put2(x.adr);
         break;
      case Operand.Elem: // assert: базовый адрес и индекс массива находятся в EStack
         if (x.type ==Tab.charType) put(baload); else put(aload);
         break;
      case Operand.Stack: break; // ничего (уже загружен)
      case Operand.Meth: error("cannot load a method");
   }
   x.kind = Operand.Stack; // операнд загружен и находится в EStack
}
```

Для генерирования кода типично, что выбор команды сводится к рассмотрению многочисленных случаев. Например, метод `load()` сначала делает выбор, зависящий от вида операнда. Если это константа, то метод должен сделать следующий выбор: между константами в диапазоне 0–5, которые можно загрузить короткой формой команды (`const0`, ..., `const5`), константой –1, для загрузки которой есть специальная команда `const_m1`, и остальными константами, которые загружаются длинной формой команды (`const w`). Похожая ситуация складывается и с операндами других видов. Отметим, что для операндов вида `Stack` никакой код не генерируется, потому что они уже загружены. В конце метода `load()` операнд считается загруженным, поэтому его вид устанавливается равным `Stack`.

Метод `load()` очень полезен. Мы можем использовать его для загрузки операндов любого вида, не задумываясь о деталях генерирования необходимых команд загрузки.

6.4.1. Загрузка переменных

Переменные встречаются в выражениях, точнее во множителях. Рассмотрим атрибутную грамматику `Factor`, чтобы понять, как загружаются переменные.

```
Factor <↑x>     (. String name; .)
= ident <↑name> (. Obj obj = Tab.find(name);       // obj.kind = Var | Con
                   Operand x = new Operand(obj);   // x.kind = Local | Static | Con
                   Code.load(x); .)                // x.kind = Stack
| ... .
```

Имя ищется в таблице символов, а его запись объекта `obj` преобразуется в операнд `x`, который затем загружается. На рис. 6.23 это показано на примере локальной переменной `v` по адресу 2.

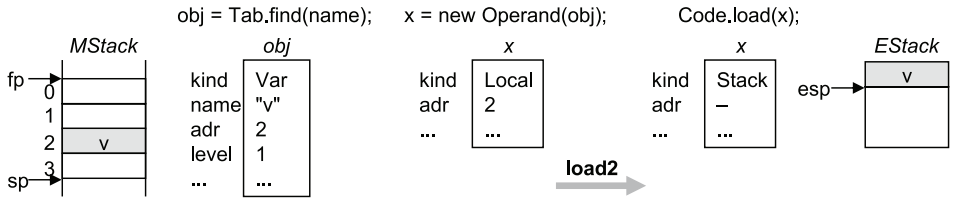


Рис. 6.23 ❖ Загрузка переменной

6.4.2. Загрузка констант

Как и переменные, константы встречаются в продукции `Factor`. Рассмотрим, как они загружаются.

```
Factor <↑x>      (. int val;.)
= ...
| number <↑val>  (. Operand x = new Operand(val);    // x.kind = Con
                   Code.load(x); .) .               // x.kind = Stack
```

Значение числа `val` преобразуется в операнд, который затем загружается. Этот процесс показан на рис. 6.24.

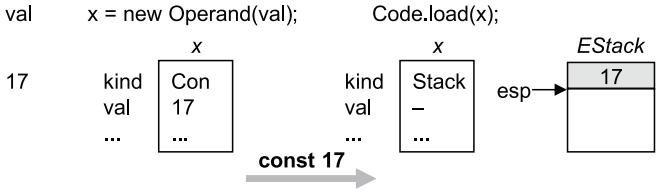


Рис. 6.24 ❖ Загрузка константы

6.4.3. Загрузка полей объекта

Доступ к полям объекта, например `var.f`, описывается в продукции `Designator`. Для загрузки таких полей сначала нужно проверить соответствующие контекстные условия (см. приложение А).

$Designator_0$ = $Designator_1$ "." ident.

- Типом $Designator_1$ должен быть класс (контекстное условие 1).
- `ident` должен быть полем $Designator_1$ (контекстное условие 2).

И снова процесс загрузки описывается атрибутной грамматикой.

```
Designator <↑x>          (. String name, fname; .)
= ident <↑name>          (. Obj obj = Tab.find(name);
                            Operand x = new Operand(obj); .)
{ "." ident <↑fname>     (. if (x.type.kind == Struct.Class) { // контекстное условие 1
                                Code.load(x);
                                Obj fld = Tab.findField(fname, x.type); // проверить также ку 2
                                x.kind = Operand.Fld; // изменить вид операнда на Fld
                                x.adr = fld.adr;
                                x.type = fld.type;
                             } else error(name + " is not an object"); .)
| ...
} .
```

Для определителя `var.f` имя `var` ищется в таблице символов и преобразуется в операнд `x`. Если дальше следует имя поля (например, `.f`), то мы сначала проверяем, что типом `var` является класс (контекстное условие 1). Затем загружается `var`, потому что базовый адрес объекта должен находиться в EStack до того, как загружать поле. Потом имя поля ищется в классе `var` методом `Tab.findField()`, который возвращает соответствующую запись объекта `fld`. Метод `findField()` похож на `find()`, но производит поиск только среди полей указанного типа. Если поле не найдено, то `findField()` сообщает об ошибке. Это соответствует проверке второго контекстного условия.

Наконец, для дескриптора операнда `x` устанавливается вид `Fld`; его адрес и тип берутся из записи объекта `fld`. Если обнаружены ошибки, не важно, какой код сгенерирован – позже он все равно будет отброшен.

Доступ к полям может продолжаться (например, `var.f.g`). В таком случае процесс будет повторен, как описано в атрибутной грамматике. Если больше операций доступа к полям нет, то `Designator` возвращает операнд `x` вида `Fld`. Если впоследствии `x` понадобится загрузить, то будет сгенерирована команда `getfield`. Заметим, что базовый адрес объекта уже находится в EStack в полной готовности к такому развитию событий. На рис. 6.25 показаны загрузка полей объектов и последовательность создаваемых при этом дескрипторов операндов.

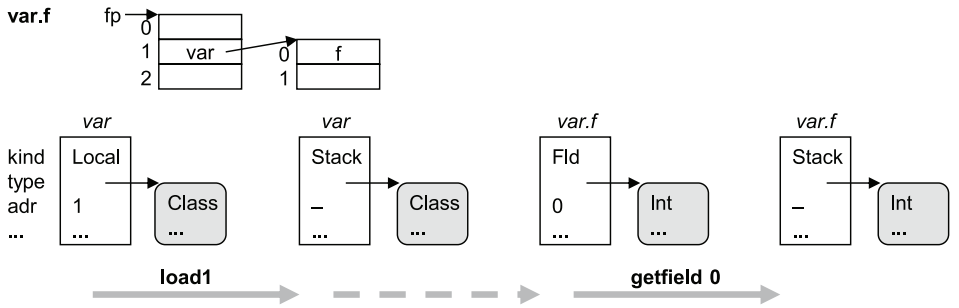


Рис. 6.25 ❖ Загрузка поля объекта – последовательность дескрипторов операндов

6.4.4. Загрузка элементов массива

Далее рассмотрим загрузку элемента массива `a[i]`. Здесь мы снова должны проверять контекстные условия (см. также приложение А).

`Designator`$_0$ `= Designator`$_1$ `"[" Expr "]".`

- Типом `Designator`$_1$ должен быть массив (контекстное условие 1).
- Типом `Expr` должен быть `int` (контекстное условие 2).

Опишем этот процесс атрибутной грамматикой.

```
Designator <↑x>   (. String name; Operand x, y; .)
= ident <↑name>   (. Obj obj = Tab.find(name); Operand x = new Operand(obj); .)
  { ...
  | "["           (. Code.load(x); // адрес массива
                     if (x.type.kind != Struct.Arr) error(name + " is not an array"); .)
                           // контекстное условие 1
    Expr <↑y>     (. if (y.type != Tab.intType) error("index must be of type int");
                           // контекстное условие 2
                     Code.load(y); // index
                     x.kind = Operand.Elem; // изменить вид операнда на Elem
                     x.type = x.type.elemType; .)
   "]"
} .
```

Имя массива `a` ищется в таблице символов и преобразуется в операнд `x`. Если далее следует открывающая квадратная скобка, то `x` загружается, потому что адрес массива должен находиться в EStack для доступа к массиву. Мы также проверяем, что `x` представляет массив (контекстное условие 1). Индексное выражение возвращает операнд `y`, типом которого должен быть `int` (контекстное условие 2). Затем загружается индекс. Адрес массива и индекс теперь находятся в EStack. Вид `x` устанавливается равным `Elem`, а тип `x` – типу элемента массива. Если впоследствии `x` понадобится загрузить, то будет сгенерирована команда `aload` или `baload`.

На рис. 6.26 показаны этот процесс и последовательность созданных дескрипторов операндов.

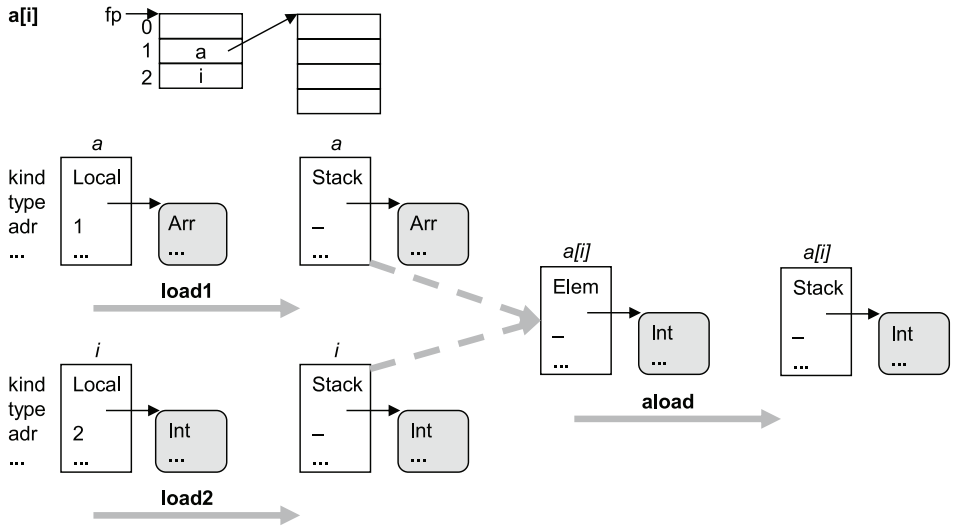


Рис. 6.26 ❖ Загрузка элемента массива – последовательность созданных дескрипторов операндов

Разумеется, доступ к массиву и к полю может комбинироваться, например `a[i].f`, при условии что `a[i]` указывает на объект, имеющий поле `f`. На рис. 6.27 показаны дескрипторы операндов, возникающие в этом процессе (альтернативный способ визуализации).

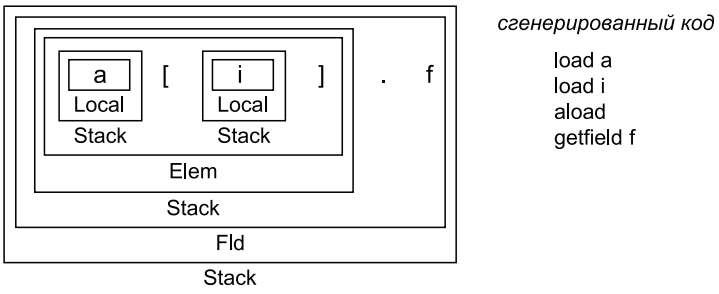


Рис. 6.24 ❖ Дескрипторы операндов для загрузки `a[i].f`

Переменная `a` представлена операндом вида `Local`, который загружается в EStack (`load a`) и таким образом становится операндом вида `Stack`. То же самое происходит с `i` (`load i`). Оба операнда вида `Stack` затем комбинируются в операнд вида `Elem`, который также загружается (`aload`) и, значит, становится операндом вида `Stack`. Это базовый адрес объекта. Вместе с полем `f` он превращается в операнд вида `Fld`, который загружается последним (`getfield f`).

6.5. Выражения

Научившись загружать простые и структурные переменные и константы, обратимся к генерированию кода для арифметических выражений. В стековой машине, каковой является μJVM, операнды выражения всегда должны загружаться в стек, прежде чем будет сгенерирована команда для оператора. Например, схема вычисления `x + y` такова:

```
... load x ...
... load y ...
add
```

Выражения описываются продукцией `Expr`, для которой должны быть проверены следующие контекстные условия:

`Expr = "-" Term.`

- `Term` должен иметь тип `int` (контекстное условие 1).

$\texttt{Expr}_0$ `= Expr` $_1$ `AddOp Term.`

- $\texttt{Expr}_1$ и `Term` должны иметь тип `int` (контекстное условие 2).

Генерирование кода для `Expr` описывается следующей атрибутной грамматикой:

```
Expr <↑x>           (. Operand x, y; int op; .)
= ( Term <↑x>
  | "-" Term <↑x>   (. if (x.type != Tab.intType) // контекстное условие 1
                          error("operand must be of type int");
                       if (x.kind == Operand.Con) x.val = - x.val;
                       else {
                         Code.load(x); Code.put(neg);
                       } .)
  )
  {  ( "+"          (. op = Code.add;.)
     | "-"          (. op = Code.sub; .)
     )              (. Code.load(x); .) // загрузить первый операнд
   Term <↑y>        (. Code.load(y);    // загрузить второй операнд
                       if (x.type != Tab.intType || y.type != Tab.intType)
                             // контекстное условие 2
                          error("operands must be of type int");
                       Code.put(op); .)
  } .
```

Оба терма предоставляют операнды `x` и `y`. Если перед первым термом находится `"-"`, то необходимо взять его со знаком минус. Если терм – константа, то противоположное число можно вычислить на этапе компиляции, в противном случае нужно загрузить операнд `x` и применить к нему команду `neg`. Это одна из немногих оптимизаций, выполняемых компилятором MicroJava.

За первым термом могут следовать другие, соединяемые операторами "+" или "-". Мы запоминаем команду (add или sub), которую нужно впоследствии сгенерировать для оператора, в переменной op. Прежде чем разбирать второй терм, необходимо загрузить первый, потому что термы в EStack должны находиться в правильном порядке. Поэтому первый терм следует загрузить еще до того, как будет сгенерирован код в процессе разбора второго терма. Наконец, мы загружаем и второй терм y, после чего генерируем запомненную команду, соответствующую оператору.

Теперь рассмотрим генерирование кода для термов. Здесь тоже имеется контекстное условие:

$Term_0$ = $Term_1$ MulOp Factor.

- $Term_1$ и Factor должны иметь тип int.

Генерирование кода для Term описывается следующей атрибутной грамматикой:

```
Term <↑x>       (. Operand x, y; int op; .)
= Factor <↑x>
  { ( "*"       (. op = Code.mul; .)
    | "/"       (. op = Code.div; .)
    | "%"       (. op = Code.rem; .)
    )           (. Code.load(x); .)
    Factor <↑y> (. Code.load(y);
                   if (x.type != Tab.intType || y.type != Tab.intType)
                                                  // контекстное условие
                       error("operands must be of type int"); Code.put(op); .) } .
```

Обработка такая же, как для Expr. Оба множителя предоставляют операнды x и y, которые загружаются в EStack. Мы запоминаем команду (mul, div или rem), которую предстоит сгенерировать в переменной op, и генерируем ее после загрузки второго множителя.

Не хватает только генерирования кода для множителей. Пока что забудем про вызовы функций, которые тоже являются множителями и будут обсуждаться в разделе 6.9. Необходимо проверить следующие контекстные условия:

Factor = "new" ident.

- ident должен обозначать класс (контекстное условие 1);

Factor = "new" ident "[" Expr "]".

- ident должен обозначать тип (контекстное условие 2);
- Expr должно иметь тип int (контекстное условие 3).

Тогда атрибутная грамматика имеет вид:

```
Factor <↑x>           (. Operand x; int val; String name; Struct type; .)
= Designator <↑x>      // для вызовов функций, см. ниже
| number <↑val>       (. x = new Operand(val); .)
```

```
| charCon <↑val>       (. x = new Operand(val); x.type = Tab.charType; .)
| "(" Expr <↑x> ")"
| "new" ident <↑name>  (. Obj obj = Tab.find(name); type = obj.type; .)
  ( "[" Expr <↑x> "]"  (. if (obj.kind != Obj.Type) error ("type expected"); // ку 2
                          if (x.type != Tab.intType)
                              error("array size must be of type int"); // ку 3
                          Code.load(x); // длина массива
                          Code.put(Code.newarray);
                          if (type = Tab.charType) Code.put(0); else Code.put(1);
                          type = new Struct(Struct.Arr, type); .)
  |                       (. if (obj.kind != Obj.Type || type.kind != Struct.Class) // сс 1
                                error("class type expected")
                             Code.put(Code.new_); Code.put2(type.nFields); .)
  )                       (. x = new Operand(); x.kind = Operand.Stack; x.type = type; .) .
```

Эта грамматика несколько сложнее предыдущих. Но сначала рассмотрим первые четыре альтернативы, все еще простые. `Designator` возвращает операнд вида `Local`, `Static`, `Fld` или `Elem`. Код доступа к полю и элементу массива уже был сгенерирован в `Designator`. `number` и `charCon` возвращают значение константы `val`, которое преобразуется в операнд. `Expr` также возвращает операнд. Операнд, являющийся результатом этих четырех альтернатив, просто возвращается `Factor`.

Теперь перейдем к пятой альтернативе, которая описывает выделение памяти для объектов массива и класса. Имя в `new ident` ищется в таблице символов, а его тип сохраняется в переменной `type`. За ним может следовать выражение длины или ничего.

Если следует выражение длины, то речь идет о выделении памяти для объекта массива. Мы должны проверить, обозначает ли `ident` тип (контекстное условие 2) и имеет ли заданная длина массива тип `int` (контекстное условие 3). Затем загружается длина массива и генерируется команда `newarray 0` или `newarray 1` в зависимости от того, что содержит массив: байты или слова. Тип выделенного объекта изменяется с `type` на `"Arr of type"`.

Если выражение длины отсутствует, то это выделение памяти для объекта класса. Мы должны проверить, что `ident` обозначает не просто тип, а тип класса (контекстное условие 1). Затем генерируется команда `new`, а число подлежащих выделению слов берется из `type.nFields`.

В обоих случаях создается новый операнд вида `Stack` с подходящим типом, который затем возвращается из `Factor`.

Теперь мы рассмотрим обобщающий пример, показывающий, какие методы парсера вызываются в процессе генерирования кода для выражения `var.f + 2 * var.g`, а также получающиеся в результате дескрипторы операндов (рис. 6.28).

`Expr` вызывает `Term`, `Factor` и `Designator`, где `var` распознается и преобразуется в операнд `Local0`. Он загружается в стек в точке ①, потому что адрес объекта должен находиться в EStack для последующего доступа к полю. В классе переменной `var` ищется поле `f` и создается операнд $\texttt{Fld}_1$ (смещение `f` равно 1), который возвращается методами `Designator`, `Factor` и `Term`. Прежде чем обрабатывать оператор `"+"` в `Expr`, необходимо загрузить первый терм в точке ②.

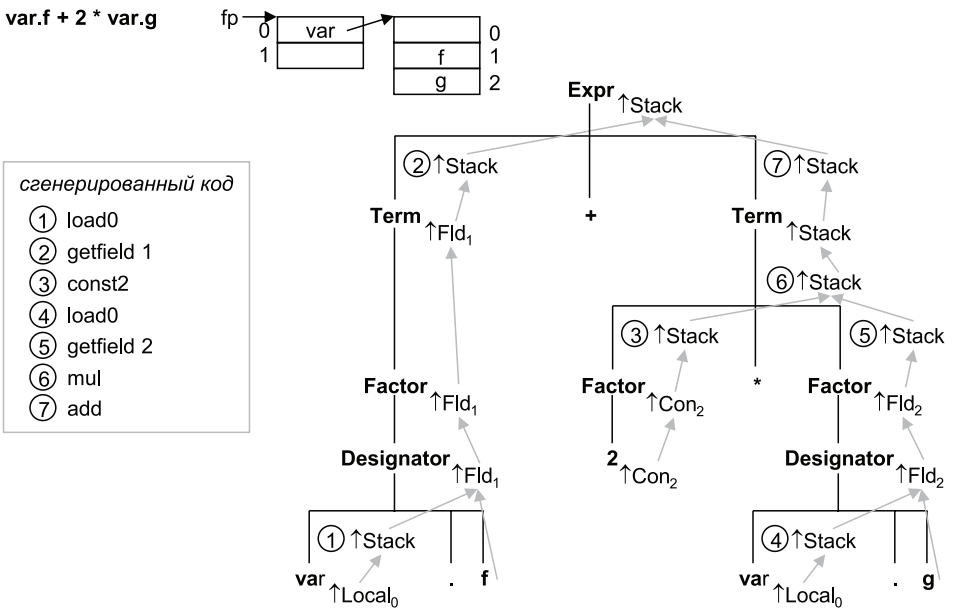


Рис. 6.28 ❖ Методы парсера и операнды в процессе трансляции выражения

Далее обрабатывается второй терм. `Factor` обнаруживает константу 2 и преобразует ее в операнд $\texttt{Con}_2$, который возвращается из `Factor`. Прежде чем `Term` продолжит обработку оператора `"*"`, первый множитель должен быть загружен в EStack (точка ③). Затем обрабатывается второй множитель, который предоставляет операнд $\texttt{Fld}_2$ для `var.g`, как было описано выше. Он также загружается в стек (точка ⑤). Поскольку теперь оба множителя находятся в EStack, в точке ⑥ генерируется команда `mul`. Результат умножения находится в EStack и возвращается методом `Term` в виде операнда `Stack`. В точке ⑦ оба терма, находящихся в EStack, складываются командой `add`, и `Expr` возвращает операнд `Stack` в качестве результата сложения.

На рис. 6.28 показан также код, сгенерированный в каждой точке. Как видно из примера, генерирование кода производится систематически в методах парсера, при этом дескрипторы операндов содержат всю информацию, необходимую для генерирования команд.

6.6. Присваивания

Теперь обратимся к генерированию кода предложений и начнем с самого простого предложения – присваивания. Присваивания имеют вид

```
designator = expr;
```

В зависимости от вида определителя в левой части (локальная или глобальная переменная, поле объекта или элемент массива) нужно генерировать различные команды (рис. 6.29).

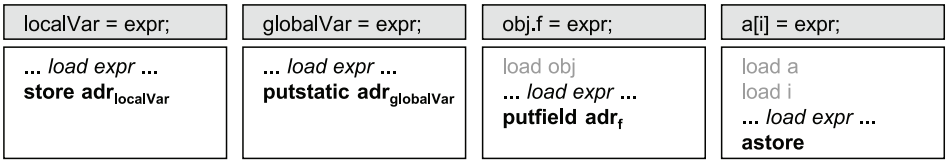


Рис. 6.29 ❖ Варианты кода присваиваний для различных определителей в левой части

В каждом из четырех случаев сначала нужно загрузить в EStack выражение в правой части присваивания. Затем, в зависимости от вида левой части, следует сгенерировать команду `store`, `putstatic`, `putfield` или `astore`. Набранные серым шрифтом команды в последних двух случаях на рис. 6.29 уже сгенерированы классом `Designator` (см. раздел 6.4). Необходимо также проверить контекстные условия.

```
Statement = Designator "=" Expr ";".
```

- `Designator` должен обозначать переменную, элемент массива или поле объекта (контекстное условие 1).
- Тип `Expr` должен быть *совместим по присваиванию* с типом `Designator` (контекстное условие 2).

Трансляция присваиваний описывается следующей атрибутной грамматикой:

```
Statement          (. Operand x, y; .)
= Designator <↑x>  // возможно, уже сгенерировал код
  "=" Expr <↑y>    (. Code.load(y);
                      if (y.type.assignableTo(x.type)) // контекстное условие 1
                         Code.assignTo(x); // x: Local | Static | Fld | Elem
                      else error("incompatible types in assignment"); .)
  ";" .
```

Совместимость по присваиванию (контекстное условие 2) проверяется в методе `assignableTo()` (см. раздел 5.4). Проверка первого контекстного условия и генерирование соответствующей команды присваивания производятся в методе `assignTo()`, показанном ниже.

```
public static void assignTo (Operand x) {
   // x обозначает левую часть присваивания; правая часть уже загружена
   switch (x.kind) {
      case Operand.Local:
        if (x.adr <= 3) put(store0 + x.adr) else { put(store); put(x.adr); }
        break;
      case Operand.Static:
```

```
        put(putstatic); put2(x.adr);
        break;
      case Operand.Fld: // базовый адрес объекта уже загружен
        put(putfield); put2(x.adr);
        break;
      case Operand.Elem: // адрес и индекс массива уже загружены
        if (x.type == Tab.charType) put(bastore); else put(astore);
        break;
     default:
        error("invalid left-hand side in assignment");
   }
}
```

6.6.1. Предложения инкремента и декремента

Предложение инкремента `x++;` и предложение декремента `x--;` – специальные формы присваивания, им соответствуют более длинные формы `x = x + 1;` и `x = x - 1;` соответственно. И на этот раз нужно проверить контекстные условия:

```
Statement = Designator ( "++" | "--" ) ";".
```

- `Designator` должен обозначать переменную, элемент массива или поле объекта (контекстное условие 1);
- `Designator` должен иметь тип `int` (контекстное условие 2).

Как транслируются эти предложения, показано ниже.

```
IncDecStatement     (. Operand x, y; .)
= Designator <↑x>   (. if (x.type != Tab.intType) error("type int expected"); // ку 2
                       if (x.kind == Operand.Fld) Code.put(Code.dup);
                       else if (x.kind == Operand.Elem) Code.put(Code.dup2);
                       y = (Operand) x.clone();
                       Code.load(y);
                       Code.put(Code.const1); .)
  ( "++"            (. Code.put(Code.add); .)
  | "--"            (. Code.put(Code.sub); .)
  ) (. Code.assignTo(x); .) // checks also cc1
  ";" .
```

Если `Designator` – поле объекта или элемент массива, то части адреса уже были загружены в EStack в методе `Designator`. Теперь эти части дублируются с помощью команды `dup` или `dup2`. Затем загружается дубликат и, наконец, прибавляется или вычитается значение 1, после чего результат присваивается определителю методом `assignTo()`.

Предложение `a[i]++;` было бы транслировано в следующий код (`a` и `i` – локальные переменные с адресами 0 и 1):

```
          EStack
load0     a
```

```
load1     a i
dup2      a i a i
aload     a i a[i]
const1    a i a[i] 1
add       a i a[i]+1
astore    -
```

Заметим, что в приведенной выше реализации не используется команда µJVM `inc`, применимая только к локальным переменным. Однако ее легко можно бы встроить, оставляем это читателю в качестве упражнения.

6.7. Переходы и метки

Прежде чем рассматривать предложения потока управления, такие как `if` и `while`, мы должны подумать о том, как собираемся реализовывать переходы и метки. Как и любая машина, µJVM располагает командой *безусловного перехода*

```
jmp offset
```

и командами *условного перехода*, которые выполняются, только если ранее вычисленное условие истинно, например:

```
... load operand1 ...
... load operand2 ...
jeq offset              // if (operand1 == operand2) jmp offset
```

Как уже было сказано, существует шесть видов условных переходов, соответствующих шести возможным операторам сравнения (рис. 6.30).

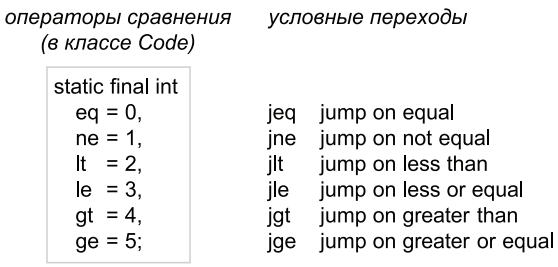


Рис. 6.30 ❖ Операторы сравнения и условные переходы

Безусловные переходы генерируются так:

```
Code.put(Code.jmp); Code.put2(offset);
```

Условные переходы генерируются путем прибавления оператора сравнения к команде `jeq`, что дает подходящую команду перехода:

```
Code.put(Code.jeq + operator); Code.put2(offset);
```

В обоих случаях длина перехода равна смещению на `offset` (2 байта) от начала команды перехода.

6.7.1. Прямые и обратные переходы

Длина перехода может быть положительной или отрицательной, поэтому переходы могут быть прямыми или обратными. *Обратные переходы* генерировать проще, потому что конечный адрес перехода уже известен в силу последовательного характера генерирования кода, и длину перехода можно вычислить непосредственно в момент генерирования команды перехода (рис. 6.31).

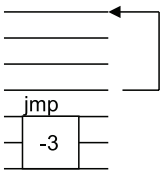


Рис. 6.31 ❖ Обратные переходы

В случае *прямого перехода* конечный адрес в момент генерирования команды перехода еще неизвестен. Поэтому два байта, отведенных под длину перехода, придется оставить пустыми и заполнить («скорректировать») позже, когда конечный адрес станет известен (рис. 6.32).

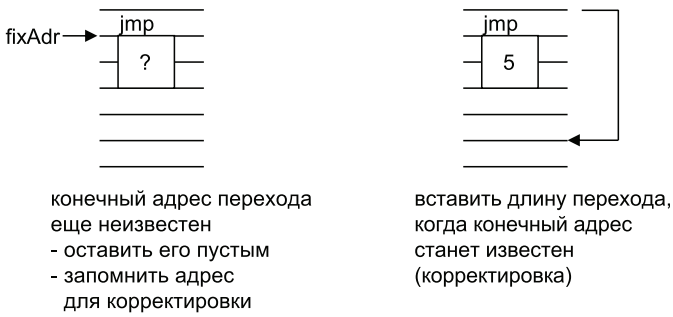


Рис. 6.32 ❖ Прямые переходы

Иногда несколько прямых переходов ведут в одно и то же место, например при выходе из цикла с помощью предложения `break`. Другой пример – составные булевы выражения, построенные с использованием операторов `&&` и `||`. Мы обсудим их в разделе 6.8.4. Если несколько прямых переходов ведут в одно место, то адреса, нуждающиеся в корректировке, следует со-

брать в список. Как только конечный адрес становится известен, все ячейки с хранящимися в списке адресами корректируются – в них подставляется правильная длина перехода (рис. 6.33).

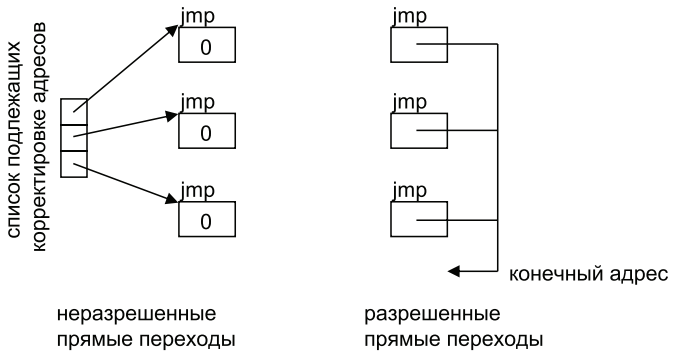


Рис. 6.33 ❖ Список подлежащих корректировке адресов для нескольких неразрешенных прямых переходов

6.7.2. Метки

Конечные адреса переходов определены метками, которые реализуются классом `Label` с двумя операциями:

- `label.here()` – определяет метку, соответствующую текущему значению `pc`;
- `label.putAdr()` – генерирует место для длины перехода на метку `label` в текущей позиции программы.

Эти операции используются следующим образом:

```
Label label = new Label(); // создает новую, еще не определенную метку
...
Code.put(Code.jmp);        // прямой переход на эту метку
label.putAdr();            // генерирует пустую ячейку и добавляет ее адрес в список
                           // подлежащих корректировке команд
...
label.here();              // определяет метку в текущей позиции pc и разрешает список
                           // подлежащих корректировке команд
```

У каждой метки есть адрес (отрицательный, если метка еще не определена) и список подлежащих корректировке адресов. Ниже показана реализация класса `Label`:

```
public class Label {
   private int adr;                        // adr >= 0: адрес метки (уже определен)
                                           // adr < 0: метка еще не определена
```

```
   private ArrayList<Integer> fixupList;     // список подлежащих корректировке адресов
   public Label() {
      adr = -1;                              // еще не определен
      fixupList = new ArrayList<Integer>();  // пустой список
   }

   public void putAdr() {
      if (adr >= 0) { // обратный переход; метка уже определена
         Code.put2(adr - (Code.pc - 1));
      } else {        // прямой переход; метка еще не определена
         fixupList.add(Code.pc);
         Code.put2(0);
      }
   }

   public void here() {
      if (adr >= 0) error("label defined twice");
      for (Integer pos: fixupList) {
         Code.put2(pos, Code.pc - (pos - 1));
      }
      adr = Code.pc;
   }
}
```

Метод `putAdr()` генерирует длину перехода на соответствующую метку. В случае обратных переходов метка уже определена (`adr >= 0`), поэтому генерируется расстояние от начала команды перехода (`Code.pc - 1`) до адреса метки. Для прямых переходов генерируется двухбайтовая ячейка со значением 0, и адрес этой ячейки добавляется в список подлежащих корректировке адресов (рис. 6.34).

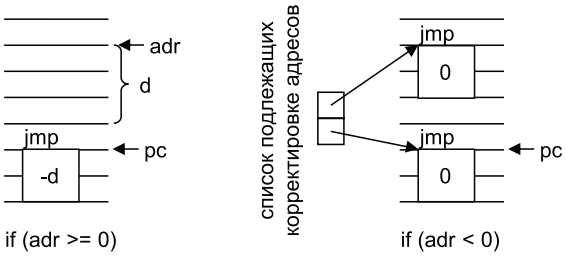


Рис. 6.34 ❖ Действие `putAdr()`

Метод `here()` разрешает список подлежащих корректировке адресов, вставляя расстояние между началом команды перехода (`pos - 1`) и текущей позицией `pc` в каждую ячейку, ожидающую корректировки. Затем определяется метка в текущей позиции `pc` (рис. 6.35).

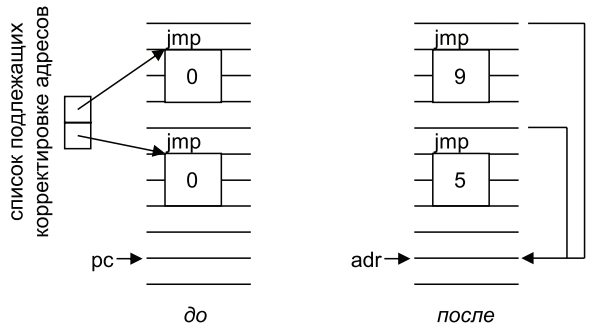


Рис. 6.35 ❖ Действие `here()`

6.7.3. Условия

В предложениях `if` и `while` проверяются условия, например

```
if (a > b) ...
```

Если условие принимает значение `true`, то программа входит в ветвь `if`, иначе обходит ее. Сгенерированный код должен выглядеть так:

```
... load a ...
  ... load b ...
jle ...
```

Но тут есть небольшая проблема: в µJVM нет явных команд сравнения, они выполняются как часть команд условного перехода. Поэтому информацию о сравнении, которое предстоит выполнить, следует как-то передать в команду перехода.

Мы решим эту проблему, заставив нетерминальный символ `Condition`, который обрабатывает условия, предоставлять новый вид операнда, `Cond`, который содержит не только оператор сравнения, участвующий в условии, но и две целевые метки для еще не разрешенных прямых переходов, следующих за условием (отметим, что это довольно хитроумный прием, который требует внимательного чтения). Таким образом, в класс `Operand` добавляются следующие элементы (выделены полужирным шрифтом):

```
class Operand {
   static final int Con = 0, Local = 1, Static = 2, Stack = 3, Fld = 4, Elem = 5, Meth = 6,
                    Cond = 7;
   ...
   int op;          // для Cond: оператор в условии
   Label tLabel;    // для Cond: конечная метка перехода, если true
   Label fLabel;    // для Cond: конечная метка перехода, если false
 }
```

Метки `tLabel` и `fLabel` используются только для составных булевых выражений, содержащих операторы `||` или `&&`. Для простых условий они не ис-

пользуются. На рис. 6.36 показано, как устанавливаются поля операнда вида Cond для составного булева выражения (см. раздел 6.8.4). Поле op – оператор в последнем сравнении в условии. Поля fLabel и tLabel – метки, на которые нужно перейти, если условие равно false и true соответственно.

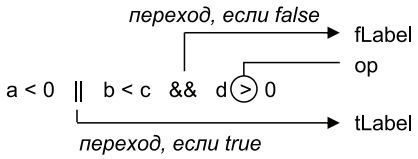


Рис. 6.36 ❖ Поля операнда вида Cond для составного булева выражения

Если первый операнд выражения x || y равен true, то и все выражение равно true, поэтому второй операнд вычислять необязательно. Это называется *закороченным вычислением* (см. раздел 6.8.4). Таким образом, *переход, если true* имеет место в позиции оператора || в булевом выражении на рис. 6.36: если a < 0, то производится переход на tLabel. Аналогично если первый операнд выражения x && y равен false, то и все выражение равно false. Следовательно, на рис. 6.36 в позиции оператора && имеет место *переход, если false*: если условие b < c равно false, то производится переход на fLabel.

Чтобы иметь возможность инициализировать операнды вида Cond в момент создания, мы объявим еще один конструктор в классе Operand:

```
public Operand (int kind, int val, Struct type) {
   this.kind = kind; this.val = val; this.type = type;
   if (kind == Cond) {
      op = val;
      tLabel = new Label(); // еще не определена
      fLabel = new Label(); // еще не определена
   }
}
```

Мы также объявим несколько вспомогательных методов в классе Code для создания команд безусловного и условного перехода.

```
public class Code {
   private static final int eq = 0, ne = 1, lt = 2, le = 3, gt = 4, ge = 5;
   private static int inverse[] = {ne, eq, ge, gt, le, lt}; // противоположные eq, ne, lt,
                                                            // le, gt, ge
   ...
   public static void jump (Label label) { // безусловный переход
      put(jmp); label.putAdr();
   }
   public static void tJump (int op, Label label) { // условный переход, если true
      put(jeq + op); // jeq, jne, jlt, jle, jgt, jge
      label.putAdr();
   }
   public static void fJump (int op, Label label) { // условный переход, если false
```

```
      put(jeq + inverse[op]); // jne, jeq, jge, jgt, jle, jlt
      label.putAdr();
   }
}
```

В следующем примере показано использование метода `fJump()` в (упрощенном) предложении `if`, где `Condition` возвращает операнд `x` вида `Cond`. Генерируется команда *перехода, если false*, которая осуществляет переход на `x.fLabel` в зависимости от `x.op`. Метка `x.fLabel` устанавливается в конце ветви `if`, и там же разрешается (т. е. корректируется) адрес перехода.

```
IfStatement                    (. Operand x; .)
= "if" "(" Condition <↑x> ")"  (. Code.fJump(x.op, x.fLabel); .)
  Statement                    (. x.fLabel.here(); .) .
```

6.8. Управление потоком

Теперь, когда мы знаем, как обращаться с переходами, рассмотрим конструкции управления потоком, такие как `if` и `while`, которые реализуются с помощью переходов. Самая простая из таких конструкций – предложение `while`, с него мы и начнем.

6.8.1. Предложение while

Предложение `while` исполняет цикл до тех пор, пока выполнено условие, проверяемое в начале каждой итерации.

Собираясь сгенерировать код для какой-то языковой конструкции, полезно сначала сопоставить код на исходном и целевом языке. На рис. 6.37 это показано для предложения `while`.

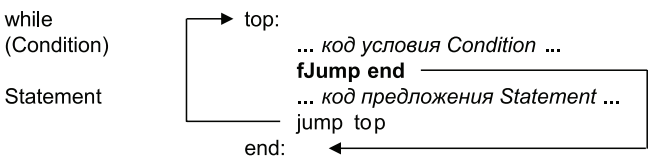


Рис. 6.37 ❖ Схема трансляции предложений `while`

В начале цикла устанавливается метка `top`, на которую производится переход по достижении конца цикла. Если условие в начале цикла равно `false`, то происходит переход на метку `end`, поставленную в конце цикла. Для `Condition` и `Statement` генерируется код, являющийся частью этой схемы, но не влияющий на поток управления. Всю схему можно реализовать следующей атрибутной грамматикой:

```
WhileStatement              (. Operand x; .)
= "while"                   (. Label top = new Label();
                               top.here(); .)
 (" Condition <↑x> ")"      (. Code.fJump(x.op, x.fLabel); .)
 Statement                  (. Code.jump(top);
                               x.fLabel.here(); .)
```

`Condition` возвращает операнд `x` вида `Cond`. Затем метод `Code.fJump()` переходит на `x.fLabel` в зависимости от оператора `x.op`. Эта метка соответствует метке `end` на рис. 6.37. Она устанавливается методом `x.fLabel.here()` после обратного перехода на метку `top`. На рис. 6.38 показан простой цикл `while` и сгенерированный для него код.

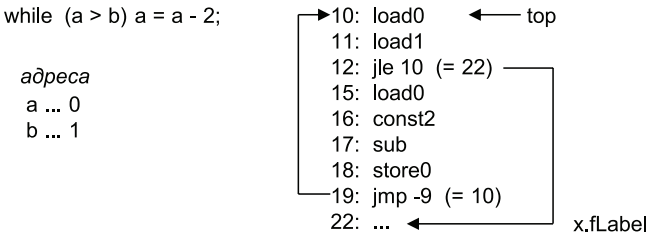


Рис. 6.38 ❖ Пример цикла `while` и сгенерированного для него кода

Поскольку условие имеет вид `a > b`, команда `jle` генерируется как *переход, если false*, т. е. производит переход, когда `a <= b`. Так как это прямой переход, длина перехода будет подставлена, только когда будет определена `x.fLabel`, – после цикла. Напомним, что длина перехода – это разность между конечным адресом перехода (здесь 22) и началом команды перехода (здесь 12).

6.8.2. Предложение if

Предложение `if` сложнее, чем `while`, потому что у него есть две формы: с `else` и без `else`. Снова рассмотрим оба варианта схематически, прежде чем приступить к генерированию кода для них (рис. 6.39).

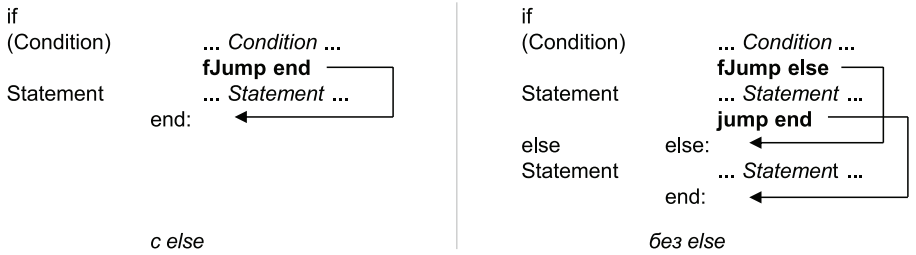


Рис. 6.39 ❖ Схема трансляции предложения `if` (без `else` и с `else`)

Если условие равно `false`, то мы должны обойти ветвь `if`, перейдя либо на конец предложения, либо на ветвь `else`. Если ветвь `else` имеется, то в конце ветви `if` необходимо создать команду безусловного перехода, которая перескакивает через `else`. Таким образом, мы имеем два (переплетенных) прямых перехода, которые должны быть разрешены в надлежащем месте. Мы покажем реализацию предложения `if` для обоих вариантов с использованием атрибутной грамматики.

```
IfStatement                (. Operand x; Label end; .)
= "if"
  "(" Condition <↑x> ")"   (. Code.fJump(x.op, x.fLabel); .)
  Statement
  ( "else"                 (. end = new Label(); Code.jump(end);
                              x.fLabel.here(); .)
    Statement              (. end.here(); .)
  |                        (. x.fLabel.here(); .)
) .
```

Факультативная ветвь `else` описывается двумя альтернативами, вторая из которых пуста и состоит только из семантического действия.

`Condition` снова возвращает операнд `x` вида `Cond`. В зависимости от `x.op` имеется *переход, если false* на метку `x.fLabel`. Если ветвь `else` присутствует, то эта метка устанавливается перед предложениями в этой ветви `else` (т. е. переход ведет на ветвь `else`), в противном случае – на пустую ветвь `else` (т. е. переход ведет на конец предложения `if`).

В начале ветви `else` генерируется команда безусловного перехода на метку `end` (т. е. на конец ветви `else`), потому что когда программа дойдет до этой точки, она уже выполнила ветвь `if` и должна обойти ветвь `else`. На рис. 6.40 показано простое предложение `if` и сгенерированный для него код.

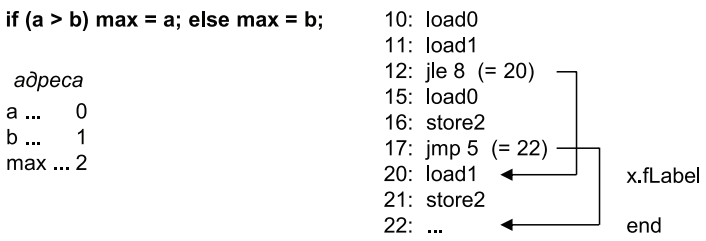


Рис. 6.40 ❖ Пример предложения `if` и сгенерированного для него кода

Мы видим два прямых перехода: *переход, если false* в обход ветви `if` и безусловный переход в обход ветви `else`.

6.8.3. Предложение break

Предложение `break` используется в MicroJava для выхода из цикла. Сгенерировать код для него просто: `break` транслируется в безусловный переход в точку после объемлющего цикла, где ставится метка выхода.

Однако имеется проблема, когда циклы вложенные: у каждого цикла должна быть своя метка выхода, и мы должны гарантировать, что метка выхода из внутреннего цикла не затирает метку выхода из внешнего цикла. Мы решим эту проблему с помощью глобального *стека меток*.

```
Label         curBreakLab = null;          // текущая метка выхода
Stack<Label>  breakLabs = new Stack<>();   // метки выхода внешних циклов
```

Когда компилятор доходит до начала цикла, он помещает текущую метку выхода в стек, а в конце цикла извлекает ее из стека. Это показано в следующей атрибутной грамматике, которая описывает трансляцию предложения `break`.

```
Statement
= "while"                 (. breakLabs.push(curBreakLab); // сохранить внешнюю метку выхода
                             curBreakLab = new Label();
          ... .)
  "(" Condition <↑x> ")"  (. ... .)
  Statement               (. ...
                             curBreakLab.here();  // break'и ведут в это место
                             curBreakLab = breakLabs.pop(); .) // восстановить внешнюю
                                                               // метку выхода
| "break" ";"             (. if (curBreakLab == null) error("break outside of a loop");
                             Code.jump(curBreakLab); .)
| ... .
```

В Java циклы могут быть снабжены меткой, на которую можно сослаться в предложении `break` (например, `break outerLab;`). Поэтому из внешнего цикла можно выйти, находясь во внутреннем цикле. Если вы захотите реализовать то же самое в MicroJava, то должны будете хранить имена меток (например, `outerLab`) в виде объектов в таблице символов, вместе с адресами. Если впоследствии программа сошлется на имя метки в предложении `break`, то компилятор поищет его в таблице символов, получит адрес и сгенерирует команду перехода на него.

6.8.4. Закороченное вычисление составных булевых выражений

В разделе 6.7 мы видели, что булевы выражения, содержащие операторы `||` и `&&`, вычисляются в «режиме закорачивания», т. е. вычисление останавли-

вается, как только результат становится известен. На псевдокоде это можно выразить так:

```
a || b   ⇔  if (a) true else b
a && b   ⇔  if (!a) false else b
```

Если a в выражении a || b уже истинно, то и все выражение истинно. Если a в выражении a && b уже ложно, то и все выражение ложно.

Закороченное вычисление можно реализовать путем вставки *перехода, если true* и *перехода, если false* в позиции операторов || и && соответственно (см. рис. 6.41; буквы a–f обозначают сравнения вида x > y).

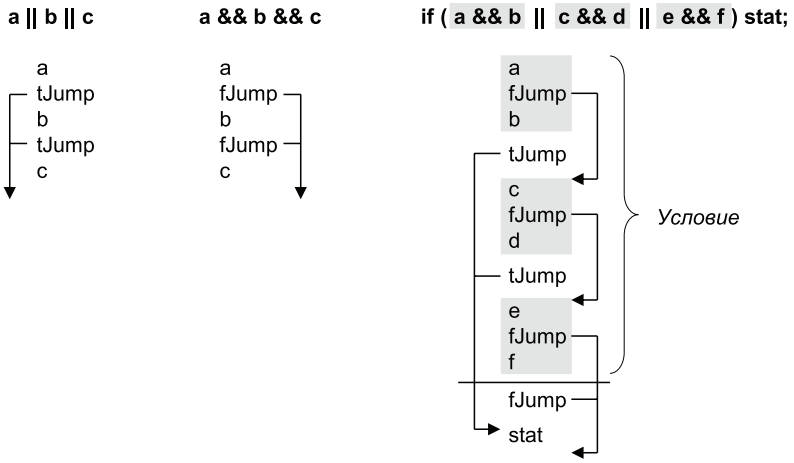


Рис. 6.41 ❖ Закороченное вычисление булевых выражений

Как видим, tJump генерируется для каждого оператора ||, а fJump для каждого оператора &&. В правой части рис. 6.41 показана структура, которая создается, когда в выражении имеется сочетание || и &&. В этом случае переходы fJump, соответствующие операторам &&, следуют за переходами tJump операторов ||.

Штриховая линия обозначает конец условия Condition в предложении if. Видно, что в этой точке может быть два открытых списка адресов, нуждающихся в корректировке: один для *переходов, если true*, другой для *переходов, если false*. После Condition в предложении if генерируется fJump, который обходит ветвь if. Он добавляется в список fJump. Все имеющиеся переходы tJump указывают на позицию сразу после этого fJump, т. к. теперь начинается ветвь if, которая должна быть выполнена, если Condition возвращает true. Переходы fJump ведут на позицию после ветви if.

В условии могут быть *переходы, если true*, и мы должны учесть это в атрибутной грамматике предложений while и if. Эти *переходы, если true* должны вести на начало тела цикла или на ветвь if:

```
WhileStatement              (. Operand x; .)
= "while"                   (. Label top = new Label();
                               top.here(); .)
  "(" Condition <↑x> ")"    (. Code.fJump(x.op, x.fLabel);
                               x.tLabel.here(); .) // переход, если true из Condition ведет
сюда
  Statement                 (. Code.jump(top);
                               x.fLabel.here(); .) .
```

Составные булевы выражения определяются следующей грамматикой:

```
Condition  = CondTerm {"||" CondTerm}.
CondTerm   = CondFactor {"&&" CondFactor}.
CondFactor = Expr RelOp Expr.
RelOp      = "==" | "!=" | ">" | ">=" | "<" | "<=".
```

Теперь рассмотрим генерирование кода для булевых выражений со следующими контекстными условиями для `CondFactor`:

`CondFactor = Expr RelOp Expr.`

- типы обоих выражений должны быть *совместимы* (контекстное условие 1);
- классы и массивы можно сравнивать только на равенство и неравенство (контекстное условие 2).

Ниже приведена атрибутная грамматика для `CondFactor`:

```
CondFactor <↑x>       (. Operand x, y; int op; .)
 = Expr <↑x>          (. Code.load(x); .)
   RelOp <↑op>
   Expr <↑y>          (. Code.load(y);
                         if (! x.type.compatibleWith(y.type)) error("type mismatch"); // ку 1
                         if (x.type.isRefType() && op != Code.eq && op != Code.ne)    // ку 2
                             error("invalid compare");
                         x = new Operand(Operand.Cond, op, null); .) .
```

Загружаются оба выражения, но сравнение будет произведено позже, в команде перехода. Поэтому `CondFactor` возвращает операнд `Cond x`, где `x.op` содержит оператор сравнения, а метки `x.tLabel` и `x.fLabel` еще не определены. Затем `CondTerm` реализует операцию `&&`:

```
CondTerm <↑x>         (. Operand x, y; .)
= CondFactor <↑x>
  { "&&"              (. Code.fJump(x.op, x.fLabel); .)
    CondFactor <↑y>   (. x.op = y.op; .)
} .
```

Для каждого оператора `&&` генерируется переход `fJump`. И снова `CondTerm` возвращает операнд `Cond x`, где `x.op` всегда содержит оператор последнего `CondFactor` и могут существовать переходы `fJump` на адрес `x.fLabel`, хранящийся в списке корректировки метки. Наконец, `Condition` реализуется следующим образом:

```
Condition <↑x>    (. Operand x, y; .)
= CondTerm <↑x>
  { "||"           (. Code.tJump(x.op, x.tLabel);
                      x.fLabel.here(); .)
  CondTerm <↑y>    (. x.op = y.op;
                      x.fLabel = y.fLabel;.)
} .
```

Для каждого оператора || генерируется переход tJump и потенциально существующий список *переходов, если false* разрешается, так что все они ведут на позицию после него. Condition возвращает операнд x вида Cond, который может содержать открытые списки *переходов, если true* и *переходов, если false*. x.op – это последний оператор в последнем CondTerm, а x.fLabel – потенциальный список *переходов, если false*, исходящий из последнего CondTerm. Поэтому y.op и y.fLabel необходимо скопировать в x.op и x.fLabel.

6.9. Методы

Нам осталось рассмотреть реализацию методов, т. е. их вызов, включая передачу параметров, объявление, управление кадрами для размещения локальных переменных и предложение return.

6.9.1. Вызов методов типа void

Методы, объявленные void, вызываются так же, как предложения. Вызов

```
m(a, b);
```

транслируется по схеме:

```
... load a ...
... load b ...
call m
```

Параметры загружаются в EStack перед вызовом. Затем идет команда вызова, которая помещает в MStack адрес возврата и производит переход по адресу m. Реализация проста:

```
Statement              (. Operand x, y; )
= Designator <↑x>
  ( ActPars <↓x>       (. Code.callMethod(x);
                          if (x.type != Tab.noType) Code.put(Code.pop); .)
  | "=" Expr <↑y>      (. ... .)
  ) ";"
| ... .
```

Если предложение начинается определителем Designator, то оно может быть вызовом метода или присваиванием. За вызовом метода следует список параметров, а за присваиванием – оператор присваивания. ActPars загружает параметры в EStack, а callMethod() генерирует команду вызова (см. ниже).

Если тип вызванного метода не void (x.type != Tab.noType), т. е. это функциональный метод, то возвращенное им в EStack значение должно быть снято со стека. В этом случае генерируется команда pop.

6.9.2. Вызов функциональных методов

Функциональные методы вызываются как часть выражений. Вызов

```
c = m(a, b);
```

транслируется по схеме:

```
... load a ...
... load b ...
call m
... store c ...
```

Параметры передаются в EStack. Возвращенное значение также помещается в EStack и затем может быть обработано или сохранено в памяти. Функциональные вызовы реализуются в продукции Factor:

```
Factor <↑x>          (. Operand x; .)
= Designator <↑x>
  [ ActPars <↓x>     (. if (x.type == Tab.noType) error("void method called as a function");
                        Code.callMethod(x);
                        x.kind = Operand.Stack; .)
  ]
 | ... .
```

Если множитель является определителем Designator с параметрами, то он обозначает вызов функционального метода. ActPars загружает фактические параметры в EStack. После проверки того, что метод не void, методом callMethod() генерируется команда call. Поскольку функциональный метод оставляет возвращенное значение в EStack, вид операнда x, возвращенного Factor, устанавливается равным Operand.Stack.

Метод callMethod() генерирует команду call. Но он также работает со стандартными методами ord(), chr() и len(), для которых команда вызова не генерируется, а встраивается в код.

```
public static void callMethod (Operand m) {
   if (m.obj == Tab.ordObj || m.obj == Tab.chrObj) ; // ничего не делать
   else if (m.obj == Tab.lenObj) // встраиваемая функция len()
      put(arraylength);
```

```
   else { // сгенерировать команду вызова с относительным смещением
      put(call); put2(m.adr - (pc - 1));
   }
}
```

Обработка стандартных методов нуждается в пояснениях. Например, если вызван метод `ord('a')`, то `ActPars` загрузит значение `'a'` в EStack. Поскольку функция `ord()` имеет тип `int`, операнд `x`, возвращенный `Designator` и затем `Factor`, также имеет тип `int`. Поэтому достаточно обращаться с загруженным параметром `'a'`, как со значением типа `int`. Таким образом, `callMethod()` не должен делать ничего. `Factor` просто возвращает уже загруженный параметр как операнд вида `Stack` и типа `int`. То же самое происходит при вызове `chr(i)`: параметр `i` загружается, и `Factor` возвращает операнд вида `Stack` и типа `char`.

При вызове `len(a)` `ActPars` загружает в EStack адрес массива. Затем `callMethod()` генерирует команду `arraylength`, которая загружает длину массива в EStack. Наконец, `Factor` возвращает операнд вида `Stack` и типа `int` (т. е. тип метода `len()`), который соответствует длине массива.

6.9.3. Кадры стека

При вызове метода необходимо выделить память для кадра стека, в котором будут размещены его локальные переменные. А в конце метода эту память необходимо освободить. Последовательность команд `call-enter-exit-return`, уже описанная на рис. 6.19, повторена еще раз на рис. 6.42.

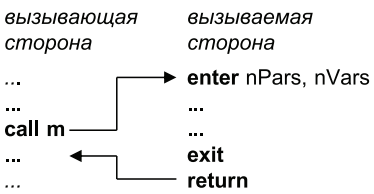


Рис. 6.42 ❖ Вызов метода – участвующие команды

Первой командой каждого метода является `enter`, она выделяет для кадра память размера `nVars` слов и связывает ее с кадром вызывающей стороны посредством *динамической связи*. Параметры (в количестве `nPars`) перемещаются из EStack в начало MStack, так что в начале работы метода EStack оказывается пуст (рис. 6.43).

Перед возвратом метод выполняет команду `exit`, которая освобождает текущий кадр и переустанавливает регистры `fp` и `sp` на кадр вызывающей стороны. Таким образом, `exit` выполняет действия, противоположные `enter`. Если метод функциональный, то возвращенное им значение находится в EStack (рис. 6.44).

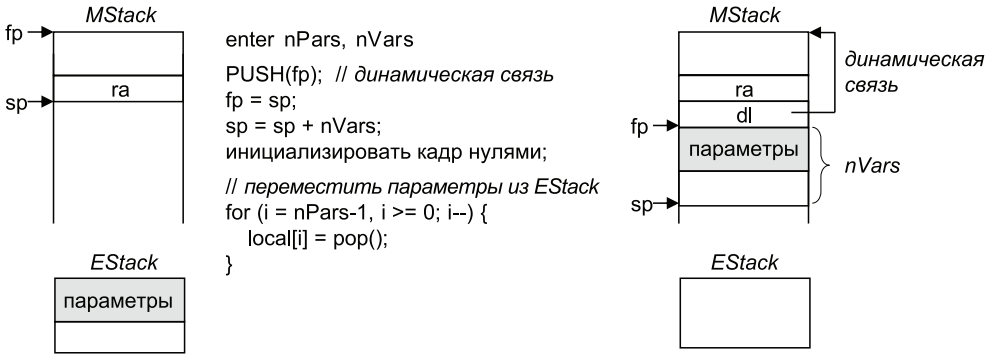


Рис. 6.43 ❖ Действие команды `enter`

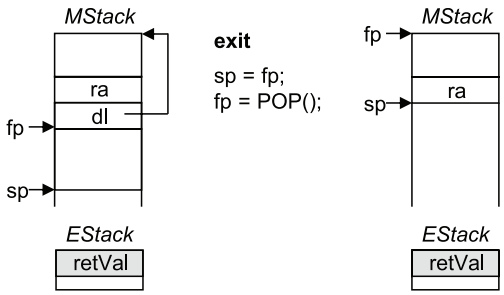


Рис. 6.44 ❖ Действие команды `exit`

6.9.4. Объявления методов

Объявление метода состоит из имени и формальных параметров. Метод может содержать локальные переменные, для которых создается новая область видимости в таблице символов, а в MStack выделяется кадр. Наконец, метод содержит еще и предложения, которые транслируются в команды, хранящиеся в буфере кода. Следующая атрибутная грамматика описывает обработку объявления метода.

```
MethodDecl                (. Struct type; String name; int n; .)
= ( Type <↑type>          (. if (type.isRefType())
                             error("methods may only return int or char"); .)
  | "void"                (. type = Tab.noType; .)
  )
  ident <↑name>           (. curMethod = Tab.insert(Obj.Meth, name, type);
                             Tab.openScope(); .)
  "(" FormPars <↑n> ")"   (. curMethod.nPars = n;
                             if (name.equals("main")) {
                                Code.mainPc = Code.pc;
                                if (curMethod.type != Tab.noType) error("main must be void");
                                if (curMethod.nPars != 0)
```

```
                                error("main must not have parameters"); } .)
  { VarDecl }                (. curMethod.locals = Tab.curScope.locals;
                                   // сохранить эту область видимости
                                curMethod.adr = Code.pc;
                                Code.put(Code.enter);
                                Code.put(curMethod.nPars);
                                Code.put(Tab.curScope.nVars); .)
  "{" { Statement } "}"      (. if (curMethod.type == Tab.noType) {
                                   Code.put(Code.exit);
                                   Code.put(Code.return_);
                                } else { // выход из функции без return
                                   Code.put(Code.trap); Code.put(1);
                                }
                                Tab.closeScope(); .) .
```

Объявление метода начинается типом функции или ключевым словом `void`. В MicroJava функции могут возвращать только значения типа `int` или `char`. Методы, объявленные как `void`, имеют тип `Tab.noType`.

Имя метода вносится в текущую область видимости (т. е. область видимости программы), затем открывается новая область видимости, в которую вносятся параметры и локальные переменные. В конце метода эта область видимости закрывается.

Адрес метода `main()` хранится в `Code.mainPc`, а позднее записывается в объектный файл. Мы также должны проверить контекстные условия: метод `main()` должен быть объявлен без параметров и типа `void` (см. приложение А).

Прежде чем начинать трансляцию предложений метода, адрес метода сохраняется в поле `curMethod.adr` и генерируется команда `enter`. После всех предложений мы генерируем команду `exit` и команду `return` для методов типа `void`. Если функциональный метод доходит до конца, не вернув значение с помощью предложения `return` (см. ниже), то генерируется команда `trap`, которая сообщает об этой ошибке и аварийно завершает программу.

6.9.5. Формальные параметры

Формальные параметры трактуются как локальные переменные и вносятся в область видимости своего метода. Кроме того, подсчитывается их количество.

```
FormPars <↑n>      (. int n = 0; .)
= [ FormPar        (. n++; .)
    { "," FormPar  (. n++; .)
    }
 ] .
FormPar            (. Struct type; String name; .)
= Type <↑type>
  ident <↑name>    (. Tab.insert(Obj.Var, name, type); .) .
```

6.9.6. Фактические параметры

Фактические параметры методов должны быть загружены в EStack. Для каждого фактического параметра необходимо также проверить, является ли он *совместимым по присваиванию* с соответствующим формальным параметром. Наконец, мы должны проверить, что количество формальных и фактических параметров совпадает. Это реализуется следующей атрибутной грамматикой:

```
ActPars <↓m>          (. Operand m, ap; .)
= "("                 (. if (m.kind != Operand.Meth) {
                            error("not a method");
                            m.obj = Tab.noObj;
                         }
                         int aPars = 0;            // число фактических параметров
                         int fPars = m.obj.nPars;  // число формальных параметров
                         Obj fp = m.obj.locals; .) // первый формальный параметр
  [ Expr <↑ap>        (. Code.load(ap); aPars++;   // загрузить и посчитать формальный
                                                   // параметр
                        if (fp != null && !ap.type.assignableTo(fp.type))
                           error("parameter type mismatch"); .)
  { "," Expr <↑ap>    (. Code.load(ap); aPars++;   // загрузить и посчитать фактический
                                                   // параметр
                         fp = fp.next;             // следующий формальный параметр
                         if (fp != null && !ap.type.assignableTo(fp.type))
                            error("parameter type mismatch"); .)
  }
 ]                    (. if (aPars > fPars)        // проверить число параметров
                            error("too many actual parameters");
                         else if (aPars < fPars)
                            error("too few actual parameters"); .)
")" .
```

Количество формальных параметров (`fPars`) берется из `m.obj.nPars`, количество фактических параметров (`aPars`) подсчитывается. Переменная `fp` пробегает по множеству формальных параметров, а атрибут `ap` – по множеству фактических параметров. Для каждого фактического параметра проверяется *совместимость по присваиванию* с соответствующим формальным параметром, после чего он загружается в EStack. В конце мы проверяем, совпадает ли число формальных и фактических параметров. На рис. 6.45 этот процесс показан для метода, объявленного следующим образом:

```
void foo (int a, char b) { ... }
```

и его вызова:

```
foo(3, 'x');
```

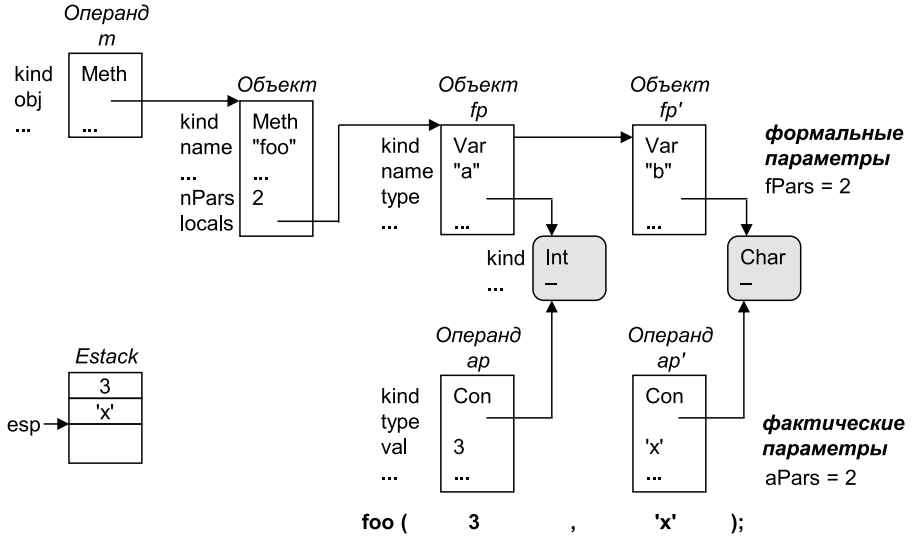


Рис. 6.45 ❖ Обработка и проверка фактических параметров

6.9.7. Предложение return

Предложение `return` существует в двух вариантах. В функциональных методах оно возвращает значение, которое необходимо загрузить в EStack и проверить на *совместимость по присваиванию* с функциональным типом текущего метода. В методах типа `void` предложение `return` не возвращает значения.

В обоих случаях генерируются команды `exit` и `return`, т. к. метод должен вернуть управление вызывающей стороне. Процесс реализуется следующей атрибутной грамматикой:

```
Statement          (. Operand x; ... .)
= ...
| "return"
  ( Expr <↑x>      (. Code.load(x); // загрузить возвращенное значение
                        if (curMethod.type == Tab.noType)
                            error("void method must not return a value");
                        else if (!x.type.assignableTo(curMethod.type))
                            error("type of return value must match method type"); .)
  |                (. if (curMethod.type != Tab.noType) error("return value expected"); .)
  )                (. Code.put(Code.exit);
                      Code.put(Code.return_); .)
";" .
```

6.10. Объектный файл

Последняя задача кодогенератора – вывести объектный файл. Помимо буфера кода, в начало объектного файла записывается следующая информация, необходимая загрузчику μJVM:

- размер кода (в байтах), полученный как длина буфера кода;
- размер области глобальных данных (в словах), полученный, исходя из количества глобальных переменных, хранящегося в поле `nVars` глобальной области видимости;
- адрес метода `main()`, который хранится в `Code.mainPc`.

На рис. 6.46 показано содержимое объектного файла. Первые два байта всегда равны «MJ», их можно использовать для проверки того, что это действительно объектный файл MicroJava.

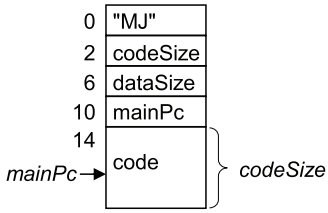


Рис. 6.46 ❖ Содержимое объектного файла

Загружая объектный файл, μJVM выделяет память для кода, области глобальных данных, стека вызовов методов и кучи. Затем она загружает код в область кода и начинает его интерпретацию с адреса `mainPc`.

Формат объектного файла MicroJava очень прост. В реальных компиляторах объектный файл обычно содержит гораздо больше информации, и его формат обычно определяется стандартом операционной системы, например Windows или Linux. Такие объектные файлы могут содержать следующую информацию.

- *Список экспортированных имен с адресами*. Если программа состоит из нескольких файлов, то компоновщику эта информация необходима для разрешения внешних ссылок на имена из других файлов.
- *Список импортированных имен*. Если в исходном коде используется имя, объявленное в каком-то другом файле, то компоновщик должен разрешить ссылку на него, вставив адрес этого имени.
- *Метаинформация для отладчика*. Отладчику необходима информация об именах, адресах и типах всех объявленных в программе элементов, чтобы он мог отобразить двоичное состояние машины (стека и кучи) в понятной человеку форме.

6.11. Упражнения

1. *Операнды при генерировании кода (1).* По аналогии с рис. 6.25 покажите, какие операнды создаются при компиляции выражения `3 + p.f` и какой при этом код генерируется. `p` – локальная переменная с адресом 0, которая указывает на объект, имеющий поле `f` со смещением 1.
2. *Операнды при генерировании кода (2).* По аналогии с рис. 6.25 покажите, какие операнды создаются при компиляции выражения `a[2*i] + i` и какой при этом код генерируется. `a` – локальная переменная с адресом 0, которая указывает на массив, а `i` – локальная переменная с адресом 1.
3. *Управление потоком (1).* Определите байт-код MicroJava для следующего фрагмента кода:

   ```
   while (x > 0) {
      sum = sum + x;
      x--;
   }
   ```

   `sum` и `x` – локальные переменные с адресами 0 и 1.
4. *Управление потоком (2).* Определите байт-код MicroJava для следующего фрагмента кода:

   ```
   if (x > 0) y = 1;
   else if (x < 0) y = -1;
   else y = 0;
   ```

   `x` и `y` – локальные переменные с адресами 0 и 1.
5. *Составные булевы выражения.* Определите байт-код MicroJava для следующего фрагмента кода:

   ```
   if (a == 10 || 0 < a && a < b) a = 0; else a = 1;
   ```

   `a` и `b` – локальные переменные с адресами 0 и 1. Обратите внимание на закороченное вычисление в составном булевом выражении.
6. *Объявление метода.* Рассмотрим следующее объявление метода:

   ```
   void foo (char ch, int n)
      int x;
   { ... }
   ```

   (a) Определите байт-код MicroJava для этого метода (`enter`, `exit` и `return`), где тело метода можно обозначить многоточием ... .
   (b) Определите байт-код MicroJava для вызова `foo('x', i+1);`, где `i` – локальная переменная типа `int` с адресом 0.
7. *Вызов функционального метода.* Рассмотрим следующее объявление функционального метода:

   ```
   int max (int a, int b) {
      if (a > b) return a; else return b;
   }
   ```

(a) Определите байт-код MicroJava для этого метода.
(b) Определите байт-код MicroJava для вызова `sum = max(x, 10);`, где `sum` и `x` – локальные переменные типа `int` с адресами 0 и 1.

8. *Предложения инкремента и декремента*. В разделе 6.6 мы обсудили генерирование кода для предложений инкремента и декремента (`x++;` и `x--;`), но без использования команды μJVM `inc`. Напишите атрибутную грамматику для этих предложений, которая генерирует код с использованием команды `inc` (см. раздел 6.1.2). Для этого напишите метод `Code.inc(x, val)`, который увеличивает определитель, описываемый операндом `x`, на величину `val` (равную 1 или –1). Также обратите внимание на контекстные условия (см. приложение A) и на то, что команду `inc` можно использовать только для локальных переменных.

9. *Расширение языка: специальный цикл while*. Расширьте предложение MicroJava `while`, так чтобы цикл мог иметь несколько заголовков и тел, например:

```
while (n % 2 == 0) // заголовок цикла 1
   n = n / 2;
|| (n % 3 == 0)    // заголовок цикла 2
   n = n - 1;
```

Первый заголовок цикла начинается ключевым словом `while`, остальные – лексемой `||`. Условия цикла проверяются последовательно. Если какое-нибудь из них равно `true`, то выполняется соответствующее предложение, и после этого предложения программа возвращается в начало цикла. Цикл завершается, если все условия цикла равны `false`. Контекстно-свободная грамматика этого предложения имеет вид:

```
WhileStat = "while" "(" Condition ")" Statement { "||" "(" Condition ")" Statement } .
```

Преобразуйте эту грамматику в атрибутную, так чтобы генерировались правильные команды перехода. Для этого воспользуйтесь вспомогательными методами для меток и переходов, описанными в разделе 6.7 (`Code.jump()`, `Code.fJump()` и `label.here()`). Заметим, что `Condition` возвращает операнд `Cond`.

10. *Расширение языка: типы перечислений*. Какие изменения необходимо было бы внести в компилятор, если бы в MicroJava были типы перечислений? Объявление

```
enum Color {RED, BLUE, GREEN}
```

определяло бы тип перечисления с тремя элементами, которые на внутреннем уровне представляются значениями 0, 1 и 2. С помощью атрибутной грамматики покажите, как следует обрабатывать тип перечисления и доступ к его элементам (например, `Color.RED`). Изменения необходимо будет внести в сканер, парсер, таблицу символов и кодогенератор.

11. *Расширение языка: тип данных `boolean`*. Какие изменения необходимо было бы внести в компилятор, если бы в MicroJava был тип данных `boo-`

lean, принимающий значения true и false? Должна быть возможность сохранить результат булева выражения в переменной типа boolean, например:

```
boolVar = a < b && b < c;
```

Должна быть также возможность использовать булевы переменные в условиях и в качестве операндов булевых выражений, например:

```
if (boolVar) ... if (boolVar || a < b) ...
```

Наконец, должна быть возможность выводить булевы выражения с помощью предложения print.

12. *Реализация кодогенератора*. Реализуйте кодогенератор компилятора MicroJava в виде класса Code, а также класс дескрипторов операндов Operand и класс Label для управления метками перехода, как описано в этой главе и в интерфейсах этих классов (см. приложение B).

Глава 7

Генератор компиляторов Coco/R

В предыдущих главах вы научились реализовывать простой компилятор «вручную» (без применения инструментов). Такая ручная реализация вполне пригодна для небольших компиляторов или для компиляторооподобных инструментов. Но вы, наверное, обратили внимание, что многие методы конструирования компиляторов (особенно относящиеся к лексическому и синтаксическому анализам) весьма систематические и могут применяться почти механически. Поэтому неудивительно, что эти методы и процессы поддаются автоматизации.

Инструменты, которые генерируют части компилятора по компактной спецификации, называются *генераторами компиляторов*. Большинство их генерируют сканеры и парсеры, тогда как остальные части компилятора приходится писать вручную. Существуют также инструменты для генерирования кода и оптимизации, но здесь мы их обсуждать не будем. Нас интересует прежде всего генерирование сканеров и парсеров (рис. 7.1), поскольку они присутствуют во всех компиляторах.

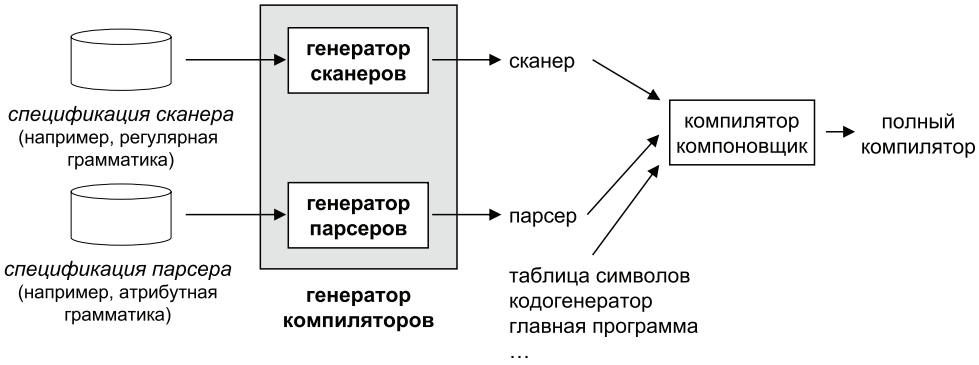


Рис. 7.1 ❖ Функционирование и компоненты генератора компиляторов

Генератор сканеров генерирует сканер в виде класса исходного кода по спецификации сканера (например, по регулярной грамматике терминальных символов). *Генератор парсеров* генерирует парсер по спецификации парсера (например, по атрибутной грамматике) – тоже в виде класса исходного кода. Программист все равно должен добавить написанные вручную классы (например, для обработки таблицы символов и для генерирования кода).

Затем классы компилируются и компонуются, и в результате получается полный компилятор. В большинстве случаев генераторы сканеров и парсеров объединены в одном инструменте, который тогда называется *генератором компиляторов*, хотя и генерирует только части компилятора.

Есть множество таких генераторов, большинство которых бесплатны. Из самых почтенных назовем генератор сканеров *Lex* и генератор парсеров *Yacc* [LMB92], которые порождают части компилятора на языке C, хотя существуют также их версии для других языков. Самыми известными на сегодняшний день генераторами компиляторов являются *ANTLR* [Parr13] и *JavaCC* [Cope20], которые генерируют одновременно сканер и парсер (JavaCC – только на Java, ANTLR – на разных языках).

Coco/R

В этой книге мы будем использовать генератор компиляторов *Coco/R* [Coco], который умеет генерировать сканеры и парсеры на разных языках. Акроним «Coco» расшифровывается как «compiler compiler» (компилятор компиляторов, другое название генератора компиляторов). Суффикс «R» означает, что он генерирует парсеры рекурсивного спуска (но есть также другие версии Coco, которые порождают таблично управляемые парсеры). Coco/R способен обрабатывать грамматики типа LL(*), т. е. грамматики для нисходящего разбора с произвольным числом опережающих символов. На рис. 7.2 показаны части, генерируемые Coco/R.

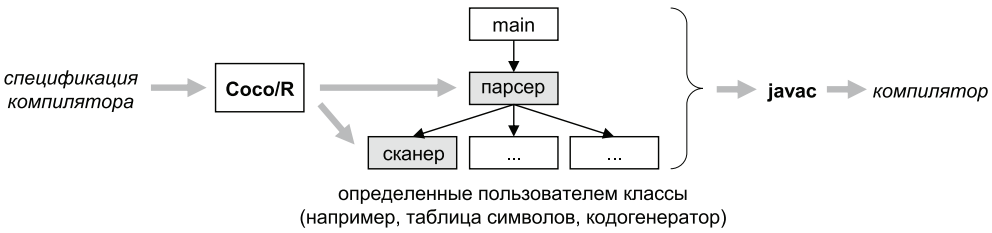


Рис. 7.2 ❖ Части компилятора, генерируемые Coco/R

Coco/R генерирует сканер и парсер на языке Java[1] по спецификации компилятора. Сканер реализован в виде конечного автомата и генерируется по

[1] Существуют также версии для C#, C++, VB.NET, Delphi, Modula-2 и других языков. См. *https:// ssw.jku.at/Coco/*.

регулярной грамматике терминальных символов. Парсер генерируется по атрибутной грамматике и реализован методом рекурсивного спуска.

Сканер и парсер образуют ядро сгенерированного компилятора. Чтобы получить полный компилятор, пользователь должен добавить дополнительные классы, например обработчик таблицы символов и кодогенератор.

Coco/R можно скачать бесплатно в виде исходного кода на условиях лицензии GPL с сайта *https://ssw.jku.at/Coco/*.

Пример спецификации компилятора

Чтобы получить представление о том, как выглядит спецификация компилятора для Coco/R, начнем с простого примера. Мы хотим написать «компилятор», который разбирает константное выражение (например, `2 * (3 + 5)`), вычисляет его значение и печатает его. Спецификация такого компилятора приведена ниже, и у читателей этой книги не возникнет трудностей с ее пониманием.

```
COMPILER Calc
CHARACTERS
  digit = '0' .. '9'.
TOKENS
  number = digit {digit}.
COMMENTS FROM "/*" TO "*/" NESTED
PRODUCTIONS
  Calc                          (. int x; .)
   = "CALC" Expr <out x>        (. System.out.println(x); .) .
  Expr <out int x>              (. int y; .)
  = Term <out x>
    { '+' Term <out y>          (. x = x + y; .)
 } .
  Term <out int x>              (. int y; .)
  = Factor <out x>
    { '*' Factor <out y>        (. x = x * y; .)
  } .
  Factor <out int x>
  = number                      (. x = Integer.parseInt(t.val); .)
  | '(' Expr <out x> ')' .
 END Calc .
```

Первая часть определяет наборы литер, терминальные символы и комментарии языка. Вторая часть – атрибутная грамматика языка, подлежащего трансляции, понять ее нетрудно.

По этой спецификации Coco/R генерирует сканер и парсер. Для нашего простого примера ни таблица символов, ни кодогенератор не нужны, нужна лишь небольшая главная программа, которая в простейшем случае могла бы выглядеть так:

```
class Calc {
   public static void main (String[] arg) {
      Scanner scanner = new Scanner(arg[0]);
```

```
      Parser parser = new Parser(scanner);
      parser.Parse(); System.out.println(parser.errors.count + " errors detected");
   }
}
```

Главная программа создает сканер и передает ему имя входного файла (`arg[0]`). Затем она создает парсер, соединяет его со сканером и запускает (`parser.Parse()`). Теперь парсер анализирует вход (например, `CALC 2 * (3 + 5)`), вычисляет значение выражения (`16`) и печатает его. Заметим, что сканеры и парсеры, генерируемые Coco/R, являются объектами, а не классами со статическими методами, как в предыдущих главах.

Из этого примера уже должно быть понятно, как использовать Coco/R. Тем не менее мы рассмотрим более пристально отдельные части спецификации компилятора, не вдаваясь, впрочем, глубоко в детали. Более полную документацию можно найти в руководстве [Coco].

Структура спецификации компилятора

Спецификация компилятора в Coco/R имеет следующую структуру:

```
CompilerSpecification
= [ImportClauses]
  "COMPILER" ident
  [GlobalFieldsAndMethods]
  ScannerSpecification
  ParserSpecification
  "END" ident "." .
```

Она начинается ключевым словом `COMPILER`, за которым следует имя, обозначающее заодно начальный символ грамматики. Спецификация заканчивается ключевым словом `END` и тем же именем. Строке `COMPILER` могут предшествовать предложения импорта на Java, например:

```
import java.util.ArrayList;
import java.io.*;
```

После строки `COMPILER` могут быть объявлены произвольные поля и методы, которые станут полями и методами сгенерированного парсера, например:

```
int sum;
void add (int x) {
  sum = sum + x;
}
```

К этим полям и методам можно обращаться из семантических действий атрибутной грамматики. Они глобальны для грамматики в целом, а не локальны для конкретной продукции.

Наиболее важные части спецификации компилятора – описание сканера и парсера, мы объясним их в следующих разделах.

7.1. Спецификация сканера

Спецификация сканера описывает лексическую структуру компилируемого языка. Она определяет терминальные символы, а также наборы литер, из которых терминальные символы состоят. Наконец, она определяет комментарии и другие подлежащие игнорированию литеры. Ниже отдельные части спецификации сканера рассмотрены более подробно.

7.1.1. Наборы литер

Терминальные символы состоят из литер (например, букв и цифр). Однако мы должны определить, какие литеры будут считаться буквами и цифрами. Иными словами, необходимо определить используемые наборы литер. Ниже на примерах показано, как это делается:

```
CHARACTERS
  digit = "0123456789".         // множество цифр
  letter = 'A' .. 'Z'.          // множество заглавных букв
  hexDigit = digit + "ABCDEF". // множество шестнадцатеричных цифр
  eol = '\n'.                   // знак конца строки
  noDigit = ANY - digit.        // множество литер, не являющихся цифрами
```

Набор литер можно определить либо строкой (например, `"0123456789"`), либо диапазоном (например, `'A' .. 'Z'`), либо одиночной литерой (например, `'\n'`). Наборы литер можно также комбинировать с помощью знаков `+` и `-` (например, `digit + "ABCDEF"`). Интересная особенность – ключевое слово `ANY`, означающее «все литеры»; его можно использовать для определения дополнительных наборов. Например, `ANY - digit` обозначает все литеры, не являющиеся цифрами.

Coco/R поддерживает набор литер Юникод (UTF-8), а также допускает обычные управляющие последовательности, такие как `'\n'` (переход на новую строку) или `'\u03c0'` (число π).

7.1.2. Терминальные символы

Терминальные символы могут быть *литералами* или *терминальными классами*. Такие литералы, как `"if"`, `"while"`, `'+'` или `">="`, объявлять необязательно, а можно просто использовать в продукциях атрибутной грамматики. Терминальные классы, например `ident` или `number`, которые могут существовать в нескольких экземплярах, должны объявляться в секции `TOKENS`, например:

```
TOKENS
  ident = letter {letter | digit | '_'}.
  number = digit {digit}
```

```
        | "0x" hexDigit hexDigit hexDigit hexDigit.
  float = digit {digit} '.' digit {digit} ['E" ['+' | '-'] digit {digit}].
```

Каждый терминальный класс объявляется с помощью регулярной продукции РБНФ, в которой в правой части могут встречаться только литералы или имена наборов литер.

Заметим, что обе альтернативы `number` могут начинаться одной и той же литерой. Первая альтернатива начинается набором `digit` (который содержит `'0'`), а вторая – литерой `'0'`. Этот конфликт автоматически разрешается Coco/R. Сначала создается недетерминированный конечный автомат, а затем он преобразуется в детерминированный автомат, в котором для каждого символа в каждом состоянии имеется только один переход.

Coco/R также позволяет сгенерировать только парсер, а сканер написать вручную. В этом случае терминальные символы объявляются только своими именами (см. руководство пользователя [Coco]).

7.1.3. Прагмы

С понятием прагмы мы еще не встречались. Это терминальные символы, которые распознаются и семантически обрабатываются сканером, но не передаются парсеру. Это можно использовать, например, для обработки параметров компилятора, которые могут встретиться в любой точке исходного кода. Например, параметр `$ABC` можно было бы использовать для задания некоторых флагов компилятора. Прагмы не являются частью синтаксиса языка и потому не передаются парсеру так же, как комментарии. Тем не менее обрабатывать их необходимо. Вот пример объявления прагмы:

```
PRAGMAS
  option = '$' {letter}. (. for (char ch: la.val.toCharArray()) {
                               if (ch == 'A') ... установить флаг компилятора A ...
                               else if (ch == 'B') ... установить флаг компилятора B ...
                               ...
                          } .)
```

Прагмы, как и терминальные символы, описываются продукцией РБНФ, за которой следует семантическое действие для обработки прагмы. В нашем примере анализируется строка прагмы `la.val` (например, `"$ABC"`), а входящие в нее буквы используются для задания флагов компилятора.

Помимо параметров компилятора, прагмы можно использовать для обработки команд препроцессора (например, `#ifdef`), документирующих комментариев (например, как в *javadoc* [JavaDoc]) или знаков конца строки. Обычно знаки конца строки трактуются как пробелы и игнорируются. Однако существуют языки, в которых окончания строк имеют специальный смысл. В таких языках их можно объявить как прагмы и обрабатывать в семантическом действии.

7.1.4. Комментарии

Комментарии не являются частью синтаксиса языка. Они содержат пояснения для читателя, но игнорируются компилятором. Поэтому комментарии объявляются не как терминальные символы, а в отдельной секции. Кроме того, вложенные комментарии, в отличие от терминальных символов, вообще нельзя описать регулярной грамматикой.

В объявлении указываются строки, которыми комментарий может начинаться и заканчиваться. Если комментарии могут быть вложенными, то в конце объявления должно присутствовать слово `NESTED`. В спецификации сканера могут быть указаны разные виды комментариев, например:

```
COMMENTS FROM "/*" TO "*/" NESTED
COMMENTS FROM "//" TO "\r\n"
```

7.1.5. Игнорируемые символы

В большинстве языков имеются символы, которые могут встречаться между терминальными символами, но игнорируются компилятором (*пустое место*). Пробелы всегда рассматриваются как пустое место. Для других символов, например знаков конца строки или табуляторов, это нужно указывать явно. Директива

```
IGNORE '\r' + '\n' + '\t'
```

говорит, что сгенерированный компилятор должен игнорировать окончания строк (`'\r'` и `'\n'`) и табуляторы (`'\t'`) в дополнение к пробелам.

7.1.6. Чувствительность к регистру

По умолчанию Coco/R генерирует компиляторы, в которых регистр имеет значение (т. е. имена `Foo` и `foo` считаются различными). Если вам это не нужно, то можно добавить директиву

```
IGNORECASE
```

в начало спецификации сканера (до определения наборов символов). В результате все буквы в терминальных символах принимаются вне зависимости от регистра. То есть ключевые слова `while`, `While` и `WHILE` считаются одинаковыми, равно как константы `3.14e0` и `3.14E0`. Однако значение лексемы `t.val` таких терминальных символов, как `ident` или `string`, сохраняется в том регистре, в каком было написано.

7.1.7. Интерфейс сгенерированного сканера

По спецификации сканера Coco/R генерирует класс `Scanner` с таким интерфейсом:

```
public class Scanner {
   public Scanner (String fileName) { ... }
   public Scanner (InputStream s) { ... }
   public Token Scan() { ... }
   public Token Peek() { ... }
   public void ResetPeek() { ... }
}
```

Первый конструктор принимает имя входного файла. Предполагается, что сканер читает исходный текст программы из этого файла. Второй конструктор принимает уже открытый входной поток типа `InputStream`.

Самым важным методом является `Scan()`[1]. Он вызывается парсером в цикле и при каждом вызове возвращает следующую распознанную лексему. Метод `Peek()` можно использовать, если нужно заглянуть вперед в исходном коде, но не возвращать прочитанные из входного потока лексемы. Это полезно для разрешения LL(1)-конфликтов путем заглядывания вперед более чем на один символ (см. раздел 7.4). Метод `ResetPeek()` восстанавливает позицию, чтобы можно было несколько раз заглянуть вперед из одной и той же позиции с разными целями. Лексемы, возвращаемые методами `Scan()` и `Peek()`, имеют следующий тип:

```
public class Token {
   public int    kind; // вид лексемы (т. е. ее код)
   public String val;  // значение лексемы
   public int    pos;  // позиция лексемы в исходном коде (начиная с 0)
   public int    col;  // столбец лексемы (начиная с 1)
   public int    line; // строка лексемы (начиная с 1)
}
```

Поле `val` содержит значение лексемы (для имен – текст имени, для чисел – строку цифр и т. д.). Поля `col` и `line` определяют номера столбца и строки лексемы и используются для выдачи сообщений об ошибках. Поле `pos` определяет позицию лексемы в тексте относительно начала входного потока.

Подчеркнем, что для чисел `val` содержит не числовое значение, а строку цифр, которую можно преобразовать в число при необходимости.

7.2. Спецификация парсера

В роли спецификации парсера выступает атрибутная грамматика компилируемого языка. Она состоит из продукций, семантических действий и атри-

[1] `Scan()` соответствует методу `next()` из главы 2.

бутов, по которым Coco/R генерирует парсер рекурсивного спуска. Ниже рассмотрены отдельные части спецификации парсера.

7.2.1. Продукции

Продукции образуют контекстно-свободную РБНФ-грамматику. Их можно объявлять в любом порядке, но для каждого нетерминального символа должна быть ровно одна продукция. В частности, должна существовать продукция для начального символа грамматики, имя которого было задано после ключевого слова COMPILER. Ниже приведен пример (части, набранные серым шрифтом, относятся к спецификации сканера).

```
COMPILER Expr
CHARACTERS
  Ident  = ...
  number = ...
  ...
PRODUCTIONS
  Expr  = Term {AddOp Term}. // продукция начального символа
  Term  = Factor {MulOp Factor}.
  Factor = ident | number | '(' Expr ')' | '-' Factor.
  AddOp  = '+' | '-'.
  MulOp  = '*' | '/' | '%'.
END Expr.
```

7.2.2. Семантические действия

Семантические действия – это произвольные фрагменты кода на Java, которые записываются между операторными скобками (. и .) и выполняются сгенерированным парсером в том месте, где встречаются в грамматике. Например:

```
IdentList      (. int n; .)
= ident        (. n = 1; .)
  { ',' ident  (. n++; .)
}              (. System.out.println(n); .) .
```

Семантические действия копируются в сгенерированный парсер без проверки со стороны Coco/R. Любые синтаксические и семантические ошибки в Java-коде обнаруживаются только на этапе компиляции сгенерированного парсера. Такие ошибки нужно исправлять в атрибутной грамматике, а не в коде сгенерированного парсера, чтобы исправления не потерялись при перегенерировании парсера.

Любая продукция может объявлять собственные локальные переменные (например, переменную n в примере выше). Глобальные объявления, равно как и все предложения импорта, задаются после (соответственно, до) строки с ключевым словом COMPILER, например:

```
// импорт
import java.io.*;
COMPILER Sample
  // глобальные объявления
  FileWriter w;
  void open (String path) {
     w = new FileWriter(path);
  }
CHARACTERS
...
TOKENS
...
PRODUCTIONS
  Sample = ...        (. open("in.txt"); .) .
...
END Sample.
```

Глобальные объявления становятся полями и методами сгенерированного парсера и не проверяются Coco/R. К ним можно обращаться из семантических действий грамматики.

7.2.3. Атрибуты

У **терминальных символов** нет явных атрибутов, но к их значениям можно обращаться из семантических действий посредством следующих глобальных переменных парсера:

```
public Token t;   // последняя распознанная лексема
public Token la;  // опережающая лексема (еще не распознана)
```

Например, значение лексемы `number` можно получить следующим образом:

```
Factor = number (. int numVal = Integer.parseInt(t.val); ) .
```

`t.val` содержит строку цифр ранее распознанной лексемы `number`, которую можно преобразовать в число методом `Integer.parseInt()`.

Нетерминальные символы могут иметь любое число входных атрибутов, но не более одного выходного. Атрибуты заключаются в угловые скобки (`<...>`) после соответствующего нетерминального символа. Различаются формальные и фактические атрибуты.

Формальные атрибуты, если они необходимы, задаются в левой части продукции и объявляются вместе с типом. Выходному атрибуту предшествует ключевое слово `out`, и он должен быть первым в списке атрибутов:

```
A <int n, char c> = ... .    // формальные атрибуты n и c
B <out int x, int y> = ... . // формальный выходной атрибут x, формальный входной атрибут y
```

Фактические атрибуты задаются, когда нетерминальный символ встречается в правой части продукции. В этом случае тип не задается, но выходным атрибутам по-прежнему должно предшествовать ключевое слово `out`:

```
... A <y, 'a'> ...   // фактические входные атрибуты y и 'a'
... B <out z, 3> ... // фактический выходной атрибут z, фактический входной атрибут 3
```

Фактические входные атрибуты могут быть константами, переменными или выражениями. Фактические выходные атрибуты должны быть переменными, потому что в продукции нетерминального символа им присваиваются значения. Конечно, количество и типы формальных и фактических атрибутов должны совпадать.

7.2.4. Трансляция в методы парсера

По каждой продукции атрибутной грамматики Coco/R генерирует метод парсера. Атрибуты становятся параметрами или возвращаемыми значениями функции, а семантические действия вставляются в надлежащие места. Например, по продукции

```
Expr <out int n>        (. int n1; .)
= Term <out n>
  { '+' Term <out n1>   (. n = n + n1; .)
  } .
```

Coco/R генерирует следующий метод парсера (атрибуты и семантические действия напечатаны серым шрифтом, метод `Get()` соответствует методу `scan()` из главы 3):

```
int Expr() {
   int n; int n1;
   n = Term();
   while (la.kind == 3) {
      Get();
      n1 = Term(); n = n + n1;
   }
   return n;
}
```

Сгенерированный метод сознательно сделан малопонятным (например, коды лексемы представлены числами, а не именованными константами), чтобы у программиста не возникало соблазна отредактировать его. Ошибки в семантических действиях должны быть исправлены в атрибутной грамматике, а не в сгенерированном по ней коде парсера.

7.2.5. Символ ANY

Символ `ANY` уже встречался нам в спецификации сканера, где он означает множество всех литер. Однако `ANY` можно использовать и в спецификации парсера, где он означает множество всех терминальных символов, не являющихся альтернативой данному `ANY`.

В таком качестве ANY можно использовать для реализации *частичного разбора*, т. е. такой формы синтаксического анализа, при которой проверяются только части синтаксиса, а прочие части покрываются ANY. Простым примером может служить парсер, который подсчитывает, сколько раз имя типа int встречается в программе, тогда как все остальные примитивные типы игнорируются. Это можно выразить следующим образом:

```
Type
= "int"      (. intCounter++; .)
| ANY .
```

В этом контексте ANY означает все терминальные символы, кроме ключевого слова int, потому что только int является альтернативой ANY. Чуть более интересный пример – парсер, который анализирует атрибутную грамматику и определяет длины семантических действий. Здесь мы покажем только продукцию SemAction.

```
SemAction <out int len>
= "(."        (. int beg = t.pos + 2; .)
    {ANY}
    ".)"        (. len = t.pos - beg; .) .
```

Здесь ANY означает все терминальные символы, кроме ".)", потому что только ".)" является (неявной) альтернативой данному ANY. Все символы внутри семантического действия охватываются ANY и пропускаются, что позволяет обойтись без точного синтаксического анализа предложений Java в семантических действиях.

Наконец, рассмотрим еще более сложный пример: парсер, который подсчитывает число предложений Java-программы, оканчивающихся точкой с запятой. Это делается в продукции Block, которая обрабатывает предложения и учитывает их в счетчике:

```
Block <out int statements>   (. int n; .)
= "{"                        (. statements = 0; .)
  { ";"                      (. statements++; .)
  | Block <out n>            (. statements += n; .)
  | ANY
  }
  "}" .
```

Альтернативами данному ANY являются ";", терминальные начальные символы Block (т. е. "{") и терминальные последующие символы повторения (т. е. "}"). Таким образом, ANY сопоставляется со всеми символами, кроме ";", "{" и "}". Обрабатываются и подсчитываются только точки с запятой, все прочие символы в предложениях охватываются ANY (*частичный разбор*). Вложенные блоки возвращают число содержащихся в них точек с запятой, которые тоже учитываются при подсчете.

7.2.6. Генерирование сканера и парсера

Coco/R генерирует сканер и парсер в виде Java-классов. Однако эти классы создаются не с нуля, генерируются только те их части, которые имеют отношение к делу (автомат сканера и методы парсера). Эти части вставляются в *скелетные файлы* (frame file) (рис. 7.3).

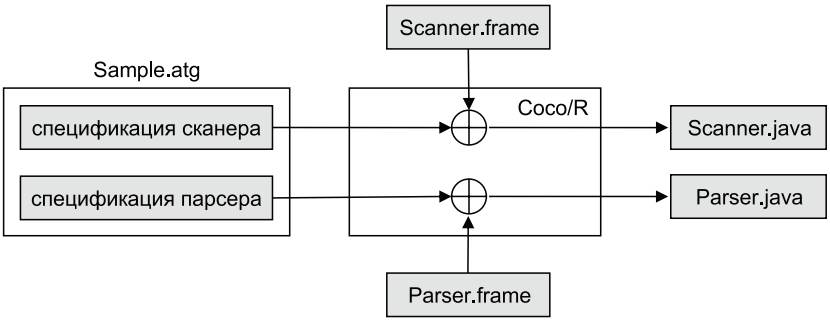


Рис. 7.3 ❖ Генерирование сканера и парсера из скелетных файлов

Скелетные файлы содержат маркеры `"-->..."`, вместо которых подставляются части, сгенерированные Coco/R. Ниже приведен фрагмент скелетного файла `Scanner.frame`:

```
public class Scanner {
   static final int EOL = '\n';
   static final int eofSym = 0;
-->declarations
   ...
   static {
      start = new StartStates(); literals = new HashMap();
-->initializations
   }
}
```

Coco/R вставляет сгенерированные объявления вместо маркера `"-->declarations"`, предложения инициализации вместо маркера `"-->initializations"` и т. д.

Скелетные файлы – это обычные текстовые файлы, которые можно редактировать и таким образом до определенной степени настраивать сгенерированные сканер и парсер под себя. Но надо внимательно следить за тем, чтобы не нанести ущерб функциональности сканера и парсера. В общем случае модифицировать скелетные файлы необязательно. Они должны находиться в одном каталоге со спецификацией компилятора, которая обычно имеет расширение `*.atg`.

7.2.7. Интерфейс сгенерированного парсера

Имея спецификацию парсера и скелетный файл `Parser.frame`, Coco/R генерирует класс `Parser` со следующим интерфейсом:

```
public class Parser {
   public Scanner scanner;                 // сканер, соединенный с этим парсером
   public Errors errors;                   // поток сообщений об ошибках
   public Token t;                         // последняя распознанная лексема
   public Token la;                        // опережающая лексема (еще не распознана)
   public Parser (Scanner scanner) { ... } // конструктор; соединяет парсер со сканером
   public void Parse() { ... }             // точки входа; отсюда начинается разбор
   public void SemErr (String msg) { ... } // для сообщений об ошибках
}
```

Чтобы запустить парсер, его необходимо соединить со сканером, а затем вызвать метод `Parse()`. Метод `main()` компилятора должен выглядеть примерно так:

```
public static void main (String[] arg) {
   Scanner scanner = new Scanner(arg[0]); // arg[0] - имя входного файла
   Parser parser = new Parser(scanner);
   parser.Parse(); System.out.println(parser.errors.count + " errors detected");
}
```

Ошибки диагностируются и подсчитываются с помощью класса `Errors`, который также генерируется (см. следующий раздел). Количество обнаруженных в процессе компиляции ошибок можно получить методом `parser.errors.count`.

7.3. Обработка ошибок

Обработка ошибок в сгенерированных компиляторах включает вывод сообщений об ошибках и восстановление после синтаксических ошибок. Оба процесса в высокой степени автоматизированы, но нуждаются в некотором вмешательстве со стороны проектировщика компилятора, чтобы улучшить тексты сообщений и активировать восстановление после ошибок.

7.3.1. Сообщения о синтаксических ошибках

Что касается синтаксических ошибок, парсер автоматически генерирует сообщения, включающие номера строки и столбца. Если ожидаемый *терминальный символ* не распознан, то генерируется такое сообщение:

```
production:      S = a b c.
input:           a × c
error message:   -- line ... col ...: b expected
```

Если ни одна из альтернатив в *списке альтернатив* не подходит, то парсер сообщает, что нетерминальный символ, которому принадлежит этот список, недопустим:

```
production:      S = a (b | c | d) e.
input:           a × e
error message:   -- line ... col ..: invalid S
```

Чтобы сделать такие сообщения об ошибках более конкретными, список альтернатив можно превратить в отдельный нетерминальный символ:

```
production:      S = a T e.
                   T = b | c | d.
input:           a × e
error message:   -- line ... col ..: invalid T
```

Если для `T` выбрано осмысленное имя, то будет напечатано достаточно хорошее сообщение об ошибке.

7.3.2. Восстановление после синтаксических ошибок

После обнаружения синтаксической ошибки входной поток необходимо синхронизировать с грамматикой, чтобы парсер мог продолжить работу и найти дополнительные ошибки. Парсеры, генерируемые Coco/R, применяют технику *специальных якорей* для восстановления после ошибок (см. раздел 3.4.3): синхронизация производится только в нескольких местах, где ожидаются особо «безопасные» терминальные символы, которые не встречаются больше нигде в грамматике. Такими местами являются начала предложений (где ожидаются такие ключевые слова, как `if` или `while`) и начала объявлений (где ожидаются такие ключевые слова, как `void` или `public`). Однако Coco/R не может определить такие места автоматически и пометить их в грамматике, потому что они зависят от языка. Для решения этой задачи используется ключевое слово `SYNC`, как показано в следующем примере.

```
Statement
= SYNC
  ( Designator ('=' Expr | '(' ActPars ')') SYNC ';'
  | "if" '(' Expr ')' Statement ["else" Statement]
  | "while" '(' Expr ')' Statement
  | ...
  ) .
```

Если обнаружена синтаксическая ошибка, то парсер сообщает о ней, а затем продолжает работать, пока не дойдет до точки синхронизации, обо-

значенной SYNC (сообщения о наведенных ошибках подавляются). В точке синхронизации парсер пропускает терминальные символы, пока не найдет ожидаемый:

```
while (la.kind is not expected here) {
  Get(); // получить следующую лексему от сканера
}
```

7.3.3. Сообщения о семантических ошибках

Семантические ошибки обнаруживаются в семантических действиях, для сообщения о них следует вызывать метод SemErr(). Например:

```
Expr <out Struct type>    (. Struct type1; .)
= Term <out type>
  { '+' Term <out type1>  (. if (type != type1) SemErr("incompatible types"); .)
  } .
```

SemErr() – метод сгенерированного парсера, который дополняет сообщение об ошибке номерами строки и столбца последней распознанной лексемы и передает управление методу SemErr() класса Errors:

```
public void SemErr (String msg) {
   errors.SemErr(t.line, t.col, msg);
}
```

В процессе восстановления после ошибки терминальные символы могут быть пропущены, а некоторые семантические действия могут не выполняться. Таким образом, вы должны понимать, что некоторые переменные могут иметь не те значения, что ожидались. Этому можно противопоставить защитное программирование, например включив для отдельных значений проверки на правдоподобие.

7.3.4. Класс Errors

Для управления сообщениями об ошибках Coco/R также генерирует класс Errors со следующим интерфейсом:

```
public class Errors {
   public int count = 0;                          // число обнаруженных ошибок
   public PrintStream errorStream = System.out; // поток сообщений об ошибках
   public String errMsgFormat = "-- line {0} col {1}: {2}"; // 0 = строка, 1 = столбец,
                                                            // 2 = сообщение
   public void SynErr (int line, int col, int n) { ... }
   public void SemErr (int line, int col, String msg) { ... }
}
```

Парсер сообщает о синтаксических ошибках с помощью метода `SynErr()`, а о семантических – с помощью метода `SemErr()`. Оба метода выводят сообщение в поток `errorStream` и увеличивают счетчик ошибок `count`.

Выходным потоком по умолчанию является консоль, но полю `errorStream` можно присвоить и другое значение, например чтобы выводить сообщения об ошибках в файл. Изменяя переменную `errMsgFormat`, вы можете настроить формат сообщений об ошибках (например, чтобы выводить их не на английском языке).

7.4. LL(1)-конфликты

Быть может, самое главное преимущество Coco/R заключается в том, что он проверяет корректность грамматики: что все нетерминальные символы объявлены и между продукциями нет циклических зависимостей. В частности, проверяется наличие LL(1)-конфликтов, и если они имеются, то выдается сообщение с предложением устранить их путем преобразования грамматики.

Если вы брали грамматику из интернета или из описания языка, то, скорее всего, она не обладает свойством LL(1). Нахождение и разрешение LL(1)-конфликтов в нетривиальных грамматиках – сложное и чреватое ошибками дело. Coco/R помогает в этом и гарантирует, что грамматика пригодна для разбора методом рекурсивного спуска. Например, для грамматики

```
PRODUCTIONS
  Sample     = {Statement} .
  Statement  = Qualident '=' number ';'
             | Call
             | "if" '(' ident ')' Statement ["else" Statement] .
  Call       = ident '(' ')' ';' .
  Qualident  = [ident '.'] ident .
```

Coco/R сообщает о следующих LL(1)-конфликтах:

```
LL1 warning in Statement: ident is start of several alternatives
LL1 warning in Statement: "else" is start & successor of deletable structure
LL1 warning in Qualident: ident is start & successor of deletable structure
```

Первый конфликт проистекает из того, что первые две альтернативы `Statement` начинаются с `Qualident` и `Call`, а они, в свою очередь, начинаются с `ident`. Таким образом, обе альтернативы могут начинаться с `ident`, и парсер не знает, какую выбрать. Следовательно, конфликт необходимо разрешить, преобразовав грамматику.

Второй конфликт – хорошо известная проблема *висячего else* (см. раздел 3.3). Ключевое слово `else` может быть как началом факультативной ветви

else, так следовать за факультативным элементом (т. е. Statement). Но, как мы видели, этот конфликт можно игнорировать. Парсер всегда выбирает первую подходящую альтернативу, и в данном случае это факультативная ветвь else. А это и есть желаемое поведение в предложениях if.

Третий конфликт имеет место в продукции Qualident, которая содержит факультативный элемент. И начальным символом этого элемента, и символом, следующим за ним, является ident, поэтому парсер не может решить, входить в него или пропустить. Этот конфликт также необходимо устранить с помощью преобразования.

Эта грамматика не особенно сложна, поэтому LL(1)-конфликты, скорее всего, были бы найдены и без помощи Coco/R. Но в больших и сложных грамматиках поиск таких конфликтов бывает весьма трудным и долгим. Обычно LL(1)-конфликты можно устранить, преобразовав грамматику, это необходимо для первого и третьего конфликтов в нашем примере. Второй конфликт можно проигнорировать (в данном случае это всего лишь предупреждение). Преобразованная грамматика выглядит так:

```
Sample    = {Statement} .
Statement = ident ( ['.' ident] '=' number ';'
                  | '(' ')' ';'
                  )
          | "if" '(' ident ')' Statement ["else" Statement] .
```

Здесь конфликты разрешены посредством факторизации. Символ ident вынесен из Qualident и Call, и только после этого альтернативы разделяются. Но такое преобразование не всегда возможно, а даже если возможно, как в данном случае, оно не делает грамматику более простой для понимания. Поэтому Coco/R предлагает и другие способы разрешения LL(1)-конфликтов.

7.4.1. Разрешение конфликтов путем заглядывания вперед на несколько символов

Следующая грамматика

```
 A = ident {',' ident} ':' ...
   | ident {',' ident} ';' .
```

не обладает свойством LL(1), но это легко исправить посредством факторизации:

```
A = ident {',' ident} ( ':' ... | ';' ) .
```

Однако если в альтернативах нужно выполнить разные семантические действия, например:

```
A = ident (. x = 1; .) {',' ident (. x++; .) } ':' ...
  | ident (. foo(); .) {',' ident (. bar(); .) } ';' .
```

то с факторизацией ничего не получится. Поэтому Coco/R предлагает, как вариант, поместить так называемый *конфликтный арбитр* перед первой из конфликтующих альтернатив. Конфликтный арбитр имеет вид:

```
IF (BooleanExpr)
```

Альтернатива выбирается, только если булево выражение принимает значение `true`. Тогда нашу грамматику можно переписать следующим образом:

```
A = IF (followedByColon())
   ident (. x = 1; .) {',' ident (. x++; .) } ':' ...
  | ident (. foo(); .) {',' ident (. bar(); .) } ';' .
```

Метод `followedByColon()` разрешает конфликт путем заглядывания вперед в поток лексем, чтобы узнать, имеется ли `':'` после списка идентификаторов. В этом случае он возвращает `true`, иначе `false`. Если метод вернул `true`, то парсер заходит в альтернативу, аннотированную арбитром, в противном случае продолжает обрабатывать следующую альтернативу.

Мы реализуем метод `followedByColon()` в глобальных объявлениях спецификации компилятора и воспользуемся методом сканера `Peek()` для заглядывания вперед. При каждом вызове `Peek()` возвращает следующую лексему, не удаляя эти лексемы из входного потока, чтобы их можно было вернуть позже, в процессе регулярного сканирования. Таким образом, `Peek()` позволяет заглянуть сколь угодно далеко вперед.

```
boolean followedByColon() {
   Token x = la; // начать с опережающей лексемы
   while (x.kind == _ident || x.kind == _comma) { // читать, пока не кончатся идентификаторы
                                                  // и запятые
      x = scanner.Peek();
   }
   return x.kind == _colon;
}
```

Имена `_ident`, `_comma` и `_colon` автоматически генерируются Coco/R для всех объявленных лексем. Но чтобы это работало, мы должны явно объявить символы `','` и `':'` как лексемы в дополнение к их использованию в качестве литералов в грамматике:

```
TOKENS
  ident = letter {letter | digit}.
  comma = ',' .
  colon = ':' .
  ...
```

Тогда Coco/R генерирует для них объявления констант:

```
  static final int
     _ident = 1,
     _comma = 2,
     _colon = 3,
     ...
```

7.4.2. Разрешение конфликтов с помощью семантической информации

Конфликтные арбитры могут разрешать LL(1)-конфликты не только путем заглядывания вперед, но и с помощью семантической информации. Рассмотрим следующую грамматику:

```
Factor = '(' ident ')' Factor // приведение типа
       | '(' Expr ')'         // вложенное выражение
       | ident
       | number.
```

Первая альтернатива описывает приведение типа, вторая – вложенное выражение. Обе начинаются литерой `'('`, поэтому грамматика не обладает свойством LL(1). Но она не обладает и свойством LL(2), потому что `Expr` может начинаться с `ident`, так что заглядывание вперед на 2 лексемы тоже не поможет. И даже если мы заглянем вперед на 3 лексемы, все равно конфликт разрешить не удастся, потому что `'(' ident ')'` может быть как приведением типа, так и вложенным выражением.

Ни заглядывание вперед, ни преобразование грамматики здесь не выручат, но мы можем снова поместить конфликтный арбитр перед первой альтернативой, поручив ему проверить, обозначает ли `ident` тип. Если да, то это должно быть приведение типа, в противном случае – вложенное выражение.

```
Factor = IF (isCast())
         '(' ident ')' Factor // приведение типа
       | '(' Expr ')'         // вложенное выражение
       | ident
       | number.
```

Вот как мы реализуем метод арбитра:

```
boolean isCast() { // la содержит _lpar, ident или number
   Token next = scanner. Peek();
   if (la.kind == _lpar && next.kind == _ident) { // если имеем  '(' ident
       Obj obj = Tab.find(next.val);              // искать имя в таблице символов
       return obj.kind == Obj.Type;               // вернуть true, если имя обозначает тип
   } else return false;
}
```

С помощью конфликтного арбитра можно разрешать LL(1)-конфликты посредством заглядывания вперед на несколько символов или с помощью семантической информации. Поэтому Coco/R может обрабатывать LL(*)-грамматики, т. е. такие, для которых различить альтернативы можно путем заглядывания вперед (сколь угодно далеко) или воспользовавшись семантической информацией.

7.5. Примеры

Теперь рассмотрим несколько примеров, показывающих, как Coco/R используется на практике, при этом мы сознательно выбираем также примеры, не имеющие отношения к конструированию компиляторов в строгом смысле слова. Ведь эта книга призвана продемонстрировать, что представленные в ней методы полезны также и для других приложений.

7.5.1. Чтение двоичного дерева

Начнем с небольшого, но не тривиального примера. Мы хотим написать программу, которая читает текстовое представление двоичного дерева в скобочной нотации и строит двоичное дерево в памяти. Строго говоря, это не компилятор, а программа, которая обрабатывает синтаксически структурированный вход и транслирует его в какое-то другое представление. Но применить методы конструирования компиляторов мы можем.

На рис. 7.4 показано текстовое представление входа (в скобочной нотации) и соответствующее ему двоичное дерево.

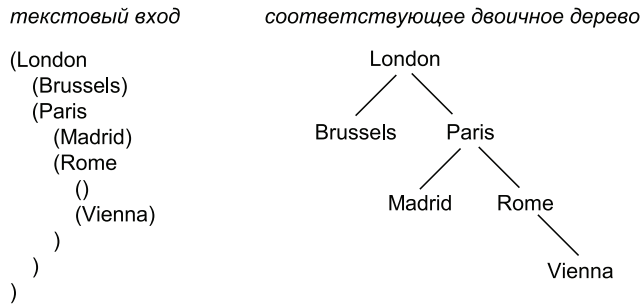


Рис. 7.4 ❖ Двоичное дерево в скобочной нотации

Чтобы вход можно было обработать с помощью атрибутной грамматики, нужно сначала придумать описывающую его контекстно-свободную грамматику. Для этого полезно подумать о структурах, которые могут встречаться в тексте входа. Поддерево может принимать одну из трех форм.

```
Subtree
= '(' ')'                          // пустое поддерево
| '(' ident ')'                    // поддерево с одним листовым узлом
| '(' ident Subtree Subtree ')' . // поддерево с корнем и двумя поддеревьями
```

Эти три варианта можно объединить с помощью факторизации и получить в результате грамматику, которая описывает вход и обладает свойством LL(1):

```
Subtree
= '(' [ ident [ Subtree Subtree ] ] ')' .
```

Нетерминальный символ Subtree должен возвращать в качестве выходного атрибута корень двоичного дерева с узлами следующего типа:

```
class Node {
   String name;
   Node left, right;
   Node (String s) { name = s; }
}
```

Семантические действия должны собирать новые деревья из поддеревьев. И наконец, чтобы проверить правильность построенного дерева, его следует вывести в скобочной нотации методом print(). Спецификация компилятора, хранящаяся в файле TreeReader.atg, имеет вид:

```
COMPILER TreeReader
  class Node { ... } // см. выше
  static void print (Node n, int indent) { ... } // см. ниже
CHARACTERS
  letter = 'A' .. 'Z' + 'a' .. 'z'.
TOKENS
  ident = letter {letter}.
IGNORE '\t' + '\r' + '\n'
PRODUCTIONS
TreeReader                (. Node n;.)
= Subtree <out n>         (. print(n, 0); .).
Subtree <out Node n>
= '('                     (. n = null; .)
 [  ident                 (. n = new Node(t.val); .)
   [   Subtree <out n.left>
       Subtree <out n.right>
   ]
 ]
 ')' .
END TreeReader.
```

Терминальными символами являются ident, а также литералы '(' и ')', которые объявлять не обязательно, поскольку они могут использоваться в атрибутной грамматике непосредственно. ident объявляется как последовательность букв. Знаки конца строки и табуляторы следует игнорировать.

Атрибутная грамматика состоит из продукций – для начального символа TreeReader и для нетерминального символа Subtree. Subtree возвращает (возможно, пустое) поддерево n и рекурсивно вызывается для построения левого и правого поддеревьев, связанных полями n.left и n.right. Вызов Subtree из TreeReader возвращает все дерево целиком, которое печатается методом print(). Параметр indent используется для управления отступами поддеревьев в выходе.

```
static void print (Node n, int indent) {
   for (int i = 0; i < indent; i++) System.out.print(' '); // отступ
   System.out.print('(');                                  // открывающая скобка
   if (n != null) {
      System.out.print(n.name);                            // имя узла
      if (n.left != null || n.right != null) {
         System.out.println();
         print(n.left, indent + 2);              // левое поддерево с увеличенным отступом
         print(n.right, indent + 2);             // правое поддерево с увеличенным отступом
         for (int i = 0; i < indent; i++) System.out.print(' ');
      }
   }
   System.out.println(')'); // closing bracket
 }
```

Наконец, нужна главная программа, запускающая парсер:

```
class TreeReader {
   public static void main (String[] arg) {
      Scanner.scanner = new Scanner(arg[0]);
      Parser parser = new Parser(scanner);
      parser.Parse();
      System.out.println(parser.errors.count + " errors detected");
   }
}
```

Теперь спецификация компилятора обрабатывается Coco/R:

```
java -jar Coco.jar TreeReader.atg
```

Coco/R генерирует файлы `Scanner.java` и `Parser.java`, которые должны быть откомпилированы компилятором Java наряду с главной программой:

```
javac Scanner.java Parser.java TreeReader.java
```

Теперь наш «компилятор» можно вызвать для входного файла `input.txt`:

```
java TreeReader input.txt
```

`input.txt` содержит текстовое представление двоичного дерева в форме, показанной на рис. 7.4 слева. Этот файл разбирается, двоичное дерево строится в памяти, а затем печатается для контроля.

7.5.2. Генератор анкет

В нашем втором примере демонстрируется *предметно-ориентированный язык*, т. е. самоопределяемый язык для некоторой предметной области. Конкретно мы хотим спроектировать язык для создания анкет. Результатом будет анкета, отображаемая в веб-браузере, которая могла бы выглядеть, как показано на рис. 7.5.

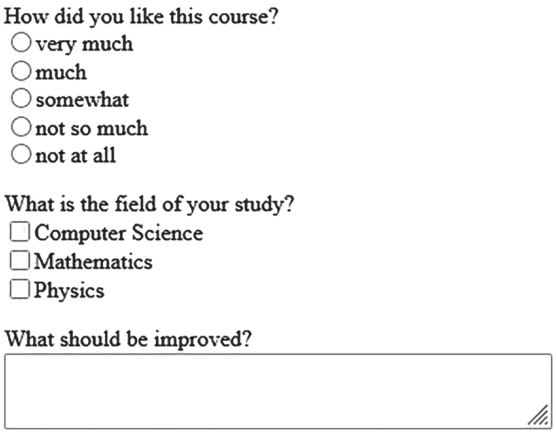


Рис. 7.5 ❖ Веб-анкета

Разумеется, мы могли бы написать интерфейс пользователя прямо на языке разметки гипертекста (HTML), но это было бы утомительно и чревато ошибками. Вместо этого мы спроектируем небольшой язык, который можно будет использовать для задания содержимого и структуры анкеты. Затем, обработав спецификацию анкеты, мы автоматически получим пользовательский интерфейс для нее.

Описание анкеты на нашем предметно-ориентированном языке могло бы выглядеть так:

```
RADIO "How did you like this course?"
  ("very much", "much", "somewhat", "not so much", "not at all")
CHECKBOX "What is the field of your study?"
  ("Computer Science", "Mathematics", "Physics")
TEXTBOX "What should be improved?"
```

Как же транслировать этот вход в интерактивную анкету на сайте? Сначала нужно описать вход контекстно-свободной грамматикой. Затем нужно решить, какие атрибуты должны предоставлять символы и какие методы вызывать в семантических действиях для генерирования их эквивалента на HTML. Наконец, нужно написать атрибутную грамматику и обработать ее с помощью Coco/R.

Вход, очевидно, состоит из нескольких вопросов, которые могут принимать одну из трех форм: (`RADIO`, `CHECKBOX`, `TEXTBOX`). Первые два включают список строковых значений. Поэтому контекстно-свободная грамматика имеет вид:

```
Queryform = {Query}.
Query     = "RADIO" Caption Values
          | "CHECKBOX" Caption Values
          | "TEXTBOX" Caption.
Caption   = string.
Values    = '(' string {',' string} ')'.
```

Нетерминальный символ `Caption` должен возвращать строку в выходном атрибуте, а нетерминальный символ `Values` – список строк. Для вывода HTML-кода мы определяем следующие методы, которые реализованы в классе `HtmlGenerator`:

- `printHeader()` создает HTML-заголовок;
- `printFooter()` создает HTML-хвостовик;
- `printRadio(caption, values)` создает последовательность переключателей;
- `printCheckbox(caption, values)` создает последовательность флажков;
- `printTextbox(caption)` создает текстовое поле.

Терминальными символами грамматики являются `string`, а также литералы `'('`, `')'` и `','`, которые необязательно объявлять как лексемы. В итоге атрибутная грамматика записывается следующим образом:

```
import java.util.ArrayList;
COMPILER QueryForm
  HtmlGenerator html; // для генерирования HTML-кода
CHARACTERS
  noQuote = ANY - '"'.
TOKENS
  string = '"' {noQuote} '"'.
COMMENTS FROM "//" TO "\r\n"
IGNORE '\t' + '\r' + '\n'
PROUDUCTIONS
QueryForm =                                 (. html.printHeader(); .)
{ Query }                                   (. html.printFooter(); .) .
//-----------------------------------------------------------------------------------
Query                                       (. String caption; ArrayList values; .)
= "RADIO" Caption <out caption> Values <out values>
                                            (. html.printRadio(caption, values); .)
| "CHECKBOX" Caption <out caption> Values <out values>
                                            (. html.printCheckbox(caption, values); .)
| "TEXTBOX" Caption <out caption>
                                            (. html.printTextbox(caption); .) .
//-----------------------------------------------------------------------------------
Caption <out String s> = StringVal<out s>.
//-----------------------------------------------------------------------------------
Values <out ArrayList values>               (. String s; .)
= '(' StringVal <out s>                     (. values = new ArrayList(); values.add(s); .)
  { ',' StringVal <out s> (. values.add(s); .)
  }
  ')'.
//-----------------------------------------------------------------------------------
 StringVal <out String s>
= string                                    (. s = t.val.substring(1, t.val.length()-1); .)
.
END QueryForm.
```

Грамматика определяет дополнительный нетерминальный символ `StringVal`, который возвращает значение строки (без кавычек) в качестве атрибута.

Атрибуты, предоставленные продукциями Caption и Values, передаются методам генератора HTML, который порождает соответствующий HTML-код. Класс HtmlGenerator выглядит следующим образом:

```
import java.io.*;
import java.util.ArrayList;
class HtmlGenerator {
   PrintStream s;
   int itemNo = 0; // номер текущего вопроса
   public HtmlGenerator(String fileName) throws FileNotFoundException {
      s = new PrintStream(fileName);
   }
   public void printHeader() {
      s.println("<html>");
      s.println("<head><title>Query Form</title></head>");
      s.println("<body>");
      s.println(" <form>");
   }

   public void printFooter() {
      s.println(" </form>");
      s.println("</body>");
      s.println("</html>");
      s.close();
   }

   public void printRadio(String caption, ArrayList values) {
      s.println(caption + "<br>");
      for (Object val: values) {
         s.print("<input type='radio' name='Q" + itemNo + "' ");
         s.print("value='" + val + "'>" + val + "<br>");
         s.println();
     }
      itemNo++; s.println("<br>");
   }

   public void printCheckbox(String caption, ArrayList values) {
      s.println(caption + "<br>");
      for (Object val: values) {
         s.print("<input type='checkbox' name='Q" + itemNo + "' ");
         s.print("value='" + val + "'>" + val + "<br>"); s.println();
      }
      itemNo++; s.println("<br>");
   }

   public void printTextbox(String caption) {
      s.println(caption + "<br>");
      s.println("<textarea name='Q" + itemNo + "' cols='50' rows='3'></textarea><br>");
      itemNo++; s.println("<br>");
   }
}
```

Например, метод printRadio() выводит следующий HTML-код для показанного выше входа:

```
How did you like this course?<br>
<input type='radio' name='Q0' value='very much'>very much<br>
<input type='radio' name='Q0' value='much'>much<br>
<input type='radio' name='Q0' value='somewhat'>somewhat<br>
<input type='radio' name='Q0' value='not so much'>not so much<br>
<input type='radio' name='Q0' value='not at all'>not at all<br>
<br>
```

Наконец, нам нужна главная программа, которая читает аргументы командной строки, создает и инициализирует сканер и парсер, а затем запускает парсер. Она также создает объект класса HtmlGenerator и присваивает его глобальной переменной parser.html, так чтобы атрибутная грамматика могла получить к нему доступ. Главная программа могла бы выглядеть так:

```
import java.io.*;
class MakeQueryForm {
   public static void main(String[] args) {
      String inFileName = args[0];
      String outFileName = args[1];
      Scanner scanner = new Scanner(inFileName);
      Parser parser = new Parser(scanner);
      try {
         parser.html = new HtmlGenerator(outFileName);
         parser.Parse();
         System.out.println(parser.errors.count + " errors detected");
      } catch (FileNotFoundException e) {
         System.out.println("-- cannot create file " + outFileName);
      }
   }
 }
```

Снова подадим спецификацию компилятора на вход Coco/R:

```
java -jar Coco.jar QueryForm.atg
```

и получим на выходе файлы Scanner.java и Parser.java, которые откомпилируем вместе с файлами HtmlGenerator.java и MakeQueryForm.java:

```
javac Scanner.java Parser.java HtmlGenerator.java MakeQueryForm.java
```

В предположении, что описание анкеты находится в файле input.txt, а выход должен быть записан в файл output.html, наш генератор анкет вызывается так:

```
java MakeQueryForm input.txt output.html
```

7.5.3. Абстрактные синтаксические деревья

Последний пример уже напрямую связан с конструированием компиляторов. Мы хотим написать компилятор, который транслирует программы на простом языке *Taste* в абстрактные синтаксические деревья (АСД, *англ.* AST). Абстрактные синтаксические деревья (см. раздел 1.5) – широко распространенное промежуточное представление программ, которое позволяет произвести оптимизацию еще до генерирования машинного кода. Но здесь нас будет интересовать только генерирование АСД без оптимизации и генерирования кода.

В некоторых компиляторах используется специальная нотация для генерирования абстрактных синтаксических деревьев. Однако здесь мы хотим показать, что для этой цели вполне пригодны обыкновенные атрибутные грамматики. Напомним, что АСД – это дерево, листья которых являются операндами, а внутренние узлы – операторами. На рис. 7.6 показано присваивание `x = 3 * x + 1`, которое выглядит как АСД; в нем оператор присваивания `'='` также рассматривается как оператор.

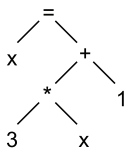


Рис. 7.6 ❖ АСД
для присваивания `x = 3 * x + 1`

Основная идея

Основная идея генерирования АСД заключается в том, чтобы всякий нетерминальный символ предоставлял поддерево в выходном атрибуте, которое затем можно объединить с другими поддеревьями для образования нового АСД. Примером может служить следующая продукция:

```
Expr <out Node e>                  (. Node e1, e2; .)
= Term <out e1> '+' Term <out e2>  (. e = new BinExpr(e1, Operator.PLUS, e2); .).
```

Результирующее АСД показано на рис. 7.7.

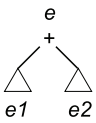


Рис. 7.7 ❖ АСД,
сгенерированное для продукции `Expr`

Узлами АСД являются объекты таких классов, как `Expr` или `BinExpr`, производные от общего базового класса `Node`.

```
abstract class Node {}              // базовый класс всех узлов

abstract class Expr extends Node {} // базовый класс узлов, образующих (под)выражения

class BinExpr extends Expr {        // класс, описывающий бинарные выражения
   Operator op;
   Expr left;                       // левое подвыражение
   Expr right;                      // правое подвыражение
   BinExpr (Expr e1, Operator op, Expr e2) { this.op = op; left = e1; right = e2; }
 }
 enum Operator {ADD, SUB, MUL, DIV, ...}
```

Далее мы сначала опишем демонстрационный язык *Taste*, а затем покажем, как строятся абстрактные синтаксические деревья для выражений, предложений, объявлений и процедур.

Демонстрационный язык Taste

В нашем демонстрационном языке *Taste* имеется несколько простых видов предложений, глобальные переменные и процедуры без параметров, но с локальными переменными. Единственные типы данных – `int` и `bool`. Язык сознательно сделан очень простым. Но у читателей не должно возникнуть трудностей с включением в него дополнительных конструкций. Контекстно-свободная грамматика *Taste* имеет вид:

```
Taste    = "program" ident "{" { VarDecl | ProcDecl } "}".
VarDecl  = Type ident { "," ident } ";".
Type     = "int" | "bool".
ProcDecl = "void" ident "(" ")" Block.
Block    = "{" { Stat | VarDecl } "}".
Stat     = ident ( "=" Expr | "(" ")" ) ";"
         | "if" "(" Expr ")" Stat [ "else" Stat ]
         | "while" "(" Expr ")" Stat
         | "read" ident ";"
         | "write" Expr ";"
         | Block.
Expr     = SimExpr [ RelOp SimExpr ].
SimExpr  =  Term { AddOp Term }.
Term     = Factor { MulOp Factor }.
Factor   = ident | number | "-" Factor | "true" | "false".
RelOp    = "==" | "<" | ">".
AddOp    = "+" | "-".
MulOp    = "*" | "/".
```

Абстрактные синтаксические деревья для выражений

В АСД для выражений мы различаем *бинарные выражения*, *унарные выражения* и *листовые узлы* (рис. 7.8).

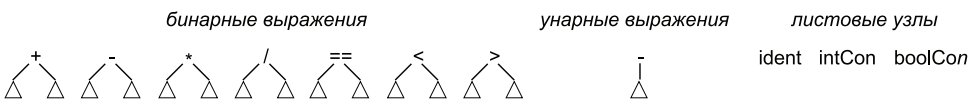


Рис. 7.8 ❖ АСД для бинарных выражений, унарных выражений и листовых узлов

Как мы уже видели, каждый узел АСД представляется объектом класса, производного от Node. Следовательно, нам нужны классы для бинарных выражений, унарных выражений и листовых узлов, а также тип перечисления для различных операторов.

```
abstract class Expr extends Node {}
   class BinExpr extends Expr {
   Operator op;
   Expr left, right;
   BinExpr (Expr e1, Operator op, Expr e2) { this.op = op; left = e1; right = e2; }
}
class UnaryExpr extends Expr {
   Operator op;
   Expr e;
   UnaryExpr (Operator op, Expr e) { this.op = op; this.e = e; }
}
class Ident extends Expr {
   Obj obj;
   Ident (Obj obj) { this.obj = obj; }
}
class IntCon extends Expr {
   int val;
   IntCon (int val) { this.val = val; }
}
class BoolCon extends Expr {
   boolean val;
   BoolCon (boolean val) { this.val = val; }
}
enum Operator {EQU, LSS, GTR, ADD, SUB, MUL, DIV}
```

Теперь мы можем написать атрибутную грамматику для построения АСД выражений. Нетерминальные символы предоставляют частичные АСД, которые собираются в новые АСД с помощью конструкторов описанных выше классов.

```
Expr <out Expr e>          (. Operator op; Expr e2; .)
= SimExpr <out e>
  [ RelOp <out op>
    SimExpr <out e2>       (. e = new BinExpr(e, op, e2); .)
  ].
SimExpr <out Expr e>       (. Operator op; Expr e2; .)
= Term <out e>
  { AddOp <out op>
```

```
    Term <out e2>          (. e = new BinExpr(e, op, e2); .)
  }.
Term <out Expr e>          (. Operator op; Expr e2; .)
= Factor <out e>
  { MulOp <out op>
    Factor <out e2>        (. e = new BinExpr(e, op, e2); .)
  }.
Factor <out Expr e>        (. String name; .)
= Ident <out name>         (. e = new Ident(curProc.find(name)); .)
| number                   (. e = new IntCon(Integer.parseInt(t.val)); .)
| "-" Factor <out e>       (. e = new UnaryExpr(Operator.SUB, e); .)
| "true"                   (. e = new BoolCon(true); .)
| "false"                  (. e = new BoolCon(false); .).
Ident <out String name>
= ident (. name = t.val; .).
AddOp <out Operator op>
= "+"                      (. op = Operator.ADD; .)
| "-"                      (. op = Operator.SUB; .).
 MulOp <out Operator op>
 = "*"                     (. op = Operator.MUL; .)
 | "/"                     (. op = Operator.DIV; .).
 RelOp <out Operator op>
 = "=="                    (. op = Operator.EQU; .)
 | "<"                     (. op = Operator.LSS; .)
 | ">"                     (. op = Operator.GTR; .).
```

В принципе, эта грамматика должна быть понятна читателю без пространных объяснений. `Factor` возвращает лист дерева, а `Expr`, `SimExpr` и `Term` собирают из поддеревьев новые деревья. Вызов `curProc.find(name)` в продукции `Factor` производит поиск имени в области видимости текущей процедуры и возвращает объект типа `Obj`, о котором мы подробнее расскажем ниже. На рис. 7.9 показано АСД для выражения `-3 * x + y`.

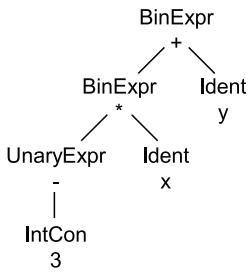


Рис. 7.9 ❖ АСД для выражения `-3 * x + y`

Абстрактные синтаксические деревья для выражений

Для каждого вида предложений имеется узел АСД, представляющий корень поддерева этого предложения. На рис. 7.10 показаны различные виды предложений и их поддеревьев.

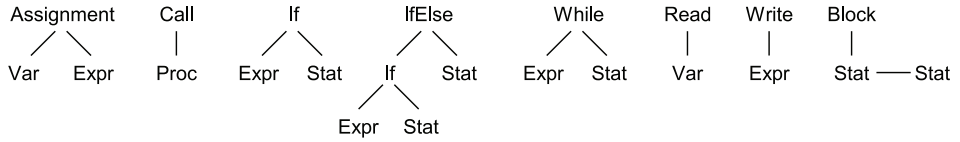


Рис. 7.10 ❖ АСД для предложений

Необходимые нам классы узлов объявляются следующим образом (классы для `Var` и `Proc` будут описаны в следующем разделе):

```
abstract class Stat extends Node {}

class Assignment extends Stat {
   Var left;
   Expr right;
   Assignment (Var v, Expr e) { left = v; right = e; }
}
class Call extends Stat {
   Proc proc;
   Call (Proc p) { proc = p; }
}
class If extends Stat {
   Expr cond;
   Stat stat;
   If (Expr e, Stat s) { cond = e; stat = s; }
}
class IfElse extends Stat {
   Stat ifPart;
   Stat elsePart;
   IfElse (Stat i, Stat e) { ifPart = i; elsePart = e; }
}
class While extends Stat {
   Expr cond;
   Stat stat;
   While (Expr e, Stat s) { cond = e; stat = s; }
}
class Read extends Stat {
   Var var;
   Read (var v) { var = v; }
}
class Write extends Stat {
   Expr e;
   Write (Expr e) { this.e = e; }
}
class Block extends Stat {
   List<Stat> stats = new ArrayList<Stat>();
   void add (Stat s) { stats.add(s); }
}
```

АСД для предложений строит следующая атрибутная грамматика:

```
Block <out block b>                (. Stat s; .)
= "{"                              (. b = new Block(); .)
  { Stat <out s>                   (. b.add(s); .)
  | VarDecl
  }
  "}" .

Stat <out Stat s>                  (. String name; Expr e; Stat s2; Block b; .)
=                                  (. s = null; .)
  ( Ident <out name>               (. Obj obj = curProc.find(name); .)
    ( "=" Expr <out e> ";"         (. s = new Assignment((Var)obj, e); .)
    | "(" ")" ";"                  (. s = new Call((Proc)obj); .)
    )
  | "if" "(" Expr <out e> ")"
    Stat <out s>                   (. s = new If(e, s); .)
    [ "else" Stat <out s2>         (. s = new IfElse(s, s2); .)
    ]
  | "while" "(" Expr <out e> ")"
    Stat <out s>                   (. s = new While(e, s); .)
  | "read" Ident <out name> ";"    (. s = new Read((Var)curProc.find(name)); .)
  | "write" Expr <out e> ";"       (. s = new Write(e); .)
  | Block <out b>                  (. s = b; .)
  ) .
```

Для следующего фрагмента программы на *Taste*

```
{ if (x > y) max = x; else max = y;
  while (max > 0) {
     z = max / 10;
     write max – 10 * z;
     max = z;
  }
}
```

генерируется АСД, показанное на рис. 7.11.

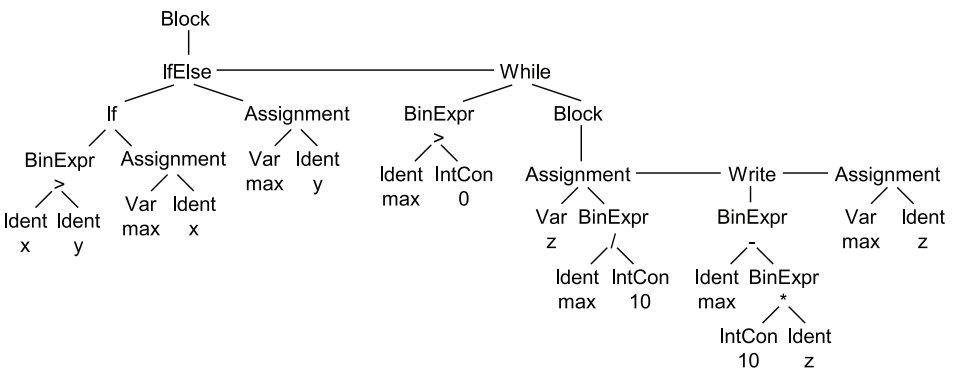


Рис. 7.11 ❖ АСД для фрагмента программы на Taste

Абстрактные синтаксические деревья для объявлений и процедур

Объявленные имена обычно хранятся в таблице символов (см. главу 5). Но в нашем примере объявления включены в АСД в виде специальных узлов. В языке Taste есть два типа объявлений: переменных и процедур. Поэтому мы объявим классы `Var` и `Proc`, производные от общего класса `Obj`. В узлах типа `Proc` хранится список локальных переменных. Можно сказать, что они представляют область видимости таблицы символов с методами `insert()` для вставки и `find()` для поиска имен. Для объектов хранится также тип (`INT` или `BOOL`). Для процедур используется тип `VOID`.

```
class Obj extends Node {  // любой объявленный объект, имеющий имя
   String name;           // имя этого объекта
   Type type;             // тип этого объекта (для процедур VOID)
   Obj (String s, Type t) { name = s; type = t; }
}
class Var extends Obj {   // переменная
   int adr;               // адрес в памяти
   Var (String name, Type type) { super(name, type); }
}
class Proc extends Obj {  // процедура (также используется для программы)
   List<Obj> locals;      // объекты, объявленные в этой процедуре
   int nVars = 0;         // число переменных в этой процедуре == следующий свободный адрес
   Block block;           // блок этой процедуры (null для программы, блока не имеющей)
   Proc program;          // ссылка на узел Proc главной программы или null
   Proc (String name, Proc program) {
      super(name, Type.VOID);
      locals = new ArrayList<Obj>();
      this.program = program;
   }
   void insert (Obj obj) {
      for (Obj x: locals) {
         if (x.name.equals(obj.name)) SemErr(obj.name + " declared twice");
      }
      locals.add(obj);
      if (obj instanceof Var) ((Var)obj).adr = nVars++;
   }
   Obj find (String name) {
      for (Obj x: locals) { if (x.name.equals(name)) return x; }
      if (program != null) {
         for (Obj x: program.locals) { if (x.name.equals(name)) return x; }
      }
      SemErr(name + " undeclared"); // name not found
      return new Obj("_undef", Type.INT); // error object
   }
}
enum Type { VOID, INT, BOOL }
```

Следующая атрибутная грамматика обрабатывает объявления и помещает объявленные имена в таблицу символов:

```
Taste                          (. String name; .)
= "program" Ident <out name>   (. curProc = new Proc(name, null); .)
  "{"
  { VarDecl | ProcDecl }
  "}" .
VarDecl                        (. String name; Type type; .)
= Type <out type>
  Ident <out name>             (. curProc.insert(new Var(name, type)); .)
  { "," Ident <out name>       (. curProc.insert(new Var(name, type)); .)
  } ";" .
Type <out Type type>
= "int"                        (. type = Type.INT; .)
| "bool"                       (. type = Type.BOOL; .) .
ProcDecl                       (. String name; .)
= "void" Ident <out name>      (. Proc program = curProc;
                                  curProc = new Proc(name, program);
                                  program.insert(curProc); .)
  "(" ")"
 Block <out curProc.block>     (. curProc = program; .) .
```

curProc – глобальная переменная, в которой хранится текущая процедура. Для фрагмента программы

```
program Sample {
   int x;
   bool y;
   void foo() { int a, b; ... }
   void bar() { int c, d; ... }
}
```

генерируется АСД, показанное на рис. 7.12.

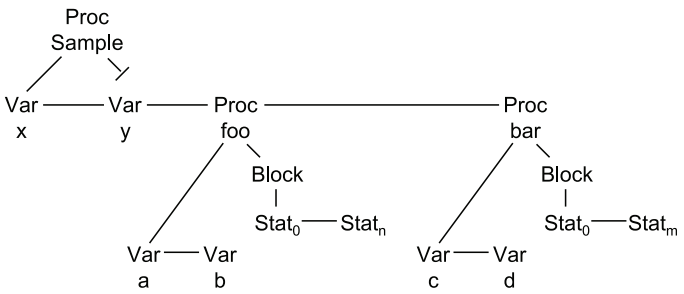


Рис. 7.12 ❖ АСД для программы Sample

На рис. 7.12 показана иерархия классов узлов АСД.

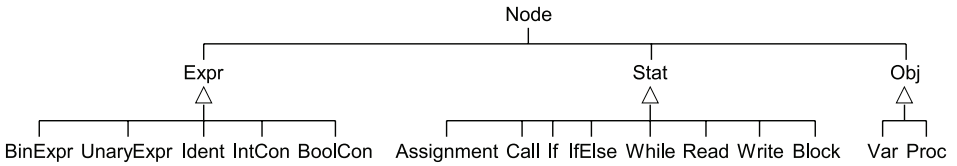


Рис. 7.13 ❖ Классы узлов АСД

Мы рассмотрели все языковые конструкции *Taste* и показали, как транслировать их в абстрактные синтаксические деревья. Приведенные выше продукции можно просто вставить в спецификацию компилятора для Coco/R. Но пока еще не хватает описания сканера. Нам нужно объявить только два терминальных символа: `ident` и `number`, все остальные являются литералами. Поэтому спецификация компилятора выглядит так:

```
COMPILER Taste
  Proc curProc; // текущая программная единица (процедура или главная программа)
CHARACTERS
  letter   = 'A' .. 'Z' + 'a' .. 'z'.
  digit    = '0' .. '9'.
TOKENS
  ident    = letter {letter | digit}.
  number   = digit {digit}.
COMMENTS FROM "//" to "\r\n"
IGNORE '\t' + '\r' + 'n'
PRODUCTIONS
  ... // описанные выше продукции
END Taste.
```

Классы узлов АСД можно либо объявить прямо в глобальных семантических объявлениях, либо сохранить в отдельных файлах. Нам также понадобится главная программа, похожая на главную программу из предыдущих примеров.

```
class Taste {
   public static void main (String[] arg) {
      Scanner scanner = new Scanner(arg[0]);
      Parser parser = new Parser(scanner);
      parser.Parse();
      System.out.println(parser.errors.count + " error(s) detected");
   }
}
```

Наконец, спецификацию компилятора следует подать на вход Coco/R, а сгенерированные сканер и парсер скомпилировать с Java-файлами классов узлов АСД:

```
java -jar Coco.jar Taste.atg javac Scanner.java Parser.java Taste.java ...
```

После этого компилятор *Taste* можно вызвать следующим образом:

```
java Taste inputFile.tas
```

Если хотите испытать компилятор Taste в деле и продолжить эксперименты с ним, то можете скачать необходимые файлы с сопроводительного сайта книги.

Резюме

Примеры, приведенные в этой главе, показали, что Coco/R – полезный инструмент для генерирования компиляторов и компилятороподобных инструментов по компактной спецификации. Coco/R и методы, изложенные в этой книге, можно использовать всюду, где имеется синтаксически структурированный вход, который можно описать грамматикой. Вот несколько примеров таких приложений (помимо собственно компиляторов):

- компиляторы предметно-ориентированных языков;
- средства статического анализа программ;
- средства вычисления метрик по исходному коду;
- средства инструментального оснащения исходного кода;
- средства анализа журналов;
- средства обработки временных рядов измерений.

Coco/R основана на атрибутных грамматиках, обладающих свойством LL(*), т. е. таких, что альтернативы можно различить путем заглядывания вперед неограниченно далеко или посредством анализа семантической информации. Это делает Coco/R полезным инструментом для многих приложений, в т. ч. выходящих за рамки собственно конструирования компиляторов.

7.6. Упражнения

Для задач к этой главе нужно будет использовать Coco/R. Поэтому скачайте файлы `Coco.jar`, `Scanner.frame` и `Parser.frame` (для Java) с сайта *https://ssw.jku.at/coco/* и сохраните их в том же каталоге, где находится спецификация компилятора для задачи.

1. *Вычисление пути*. Пусть дан текстовый файл, содержащий последовательность точек с координатами в формате (`n`, `m`), где `n` и `m` – целые числа. Напишите атрибутную грамматику, которая будет обрабатывать этот файл с целью вычислить и напечатать длину пути от первой точки до последней. Используйте Coco/R для генерирования сканера и парсера и напишите главную программу, которая запускает обработку.
2. *Интерпретатор булевых выражений*. В упражнении 7 из главы 4 мы специфицировали скриптовый язык булевых выражений и описали его обработку с помощью атрибутной грамматики. Теперь воспользуйтесь Coco/R, чтобы реализовать интерпретатор, который выполняет программы на

этом языке. Никакого кода генерировать не нужно, команды должны выполняться прямо в процессе разбора.

В качестве спецификации парсера можете использовать атрибутную грамматику, разработанную в упражнении 7 из главы 4, переписав ее в нотации Coco/R. В спецификации сканера нужно объявить только терминальный символ `ident`. Таблицей символов, содержащей имена и значения переменных, следует управлять как хеш-таблицей, которая объявляется в секции глобальных объявлений спецификации компилятора. Напишите главную программу, которая создает сканер и парсер (как в предыдущих разделах) и запускает парсер (а стало быть, и интерпретатор). Имя файла скриптовой программы должно передаваться главной программе в аргументе командной строки.

Воспользуйтесь Coco/R для генерирования сканера и парсера, а затем откомпилируйте их вместе с главной программой для получения исполняемого интерпретатора.

3. *Обработка телефонного справочника*. Предположим, что люди и их телефонные номера хранятся в текстовом файле следующего формата:

```
Doe, John Maxwell
  home +44 (20) 24684567,
  work (020) 234567
Miller, Ann
  23456789
...
```

Каждая запись начинается фамилией, за которой следует запятая и одно или несколько имен. Далее идет список телефонных номеров через запятую, а номера могут быть записаны в одном из следующих форматов:

```
+44 (20) 24680  Код страны, код региона без начального 0, номер
(020) 234567    Код региона в скобках с начальным нулем, номер
23456789        Номер без кода страны и кода региона
```

Каждому телефонному номеру может предшествовать ключевое слово `"home"` или `"work"`. Если оно отсутствует, предполагается `"home"`. Если отсутствует код страны, то используется `"+44"`, а если отсутствует код области, то предполагается `"020"`.

   (a) Опишите структуру такого текстового файла контекстно-свободной грамматикой.
   (b) Преобразуйте ее в атрибутную грамматику, так чтобы имена и телефонные номера считывались и записывались в подходящую структуру данных с отдельными полями для кода страны, кода региона, номера и описателя `home/work`. Выведите эту структуру данных для контроля. С помощью Coco/R создайте программу, которая будет читать входной файл, строить структуру данных и печатать ее.

4. *Анализ сложности*. Чтобы охарактеризовать сложность программы, можно вычислить метрики по ее исходному коду и тем самым оценить, насколько

программа трудна для понимания и сопровождения. Существуют разные метрики сложности, но мы здесь рассмотрим такую, которая вычисляет сложность методов MicroJava по сложности составляющих их предложений. Для этого определим базовую сложность для каждого типа предложений:

- присваивание: 1;
- вызов метода: 2;
- ++ или --: 1;
- предложение `if`: 1 + 2 * (сложность вложенных предложений);
- предложение `while`: 1 + 2 * (сложность вложенных предложений);
- предложение `break`: 5;
- предложение `return`: 1;
- предложение `read`: 1;
- предложение `print`: 1.

Сложность метода равна сумме сложностей его предложений. Поскольку вложенность увеличивает сложность, сложность вложенных предложений умножается на 2. Например, полная сложность предложения

```
if (x > 0) val = 1; else val = 2;
```

равна 5: базовая сложность каждого присваивания равна 1, но она умножается на 2, так что в итоге получается 4. Базовая сложность самого предложения `if` равна 1, она прибавляется к итоговой сложности, что дает 5. Если бы предложение `if` было еще и вложенным, то его полная сложность также была бы умножена на 2 и т. д.

Воспользовавшись Coco/R, напишите программу, которая вычисляет и печатает сложность всех методов в программе на MicroJava. Возьмите грамматику MicroJava из приложения A в качестве спецификации парсера и преобразуйте ее в атрибутную грамматику, которая вычисляет сложность методов по базовым сложностям предложений. Вызовы функций в выражениях можете игнорировать.

5. *Извлечение документирующих комментариев.* Для Java имеется инструмент *javadoc* [JavaDoc], который извлекает документирующие комментарии из исходного кода программы. Мы хотим написать похожий инструмент для MicroJava. Документирующий комментарий начинается `/**` и заканчивается `*/`. Внутри может быть любой текст, но вложенные комментарии запрещены. В MicroJava должна быть возможность поместить документирующие комментарии перед объявлением констант, переменных, классов и методов. Комментарии могут содержать *теги*, из которых мы здесь реализуем только два – в комментариях, предшествующих объявлениям методов.

   `@param ident ...` *`текст`* `... @returns ...` *`текст`* `...`

   Тег `@param` описывает параметр с именем `ident`, а тег `@returns` – возвращаемое методом значение. Текст, объясняющий параметр или возвращаемое значение, продолжается до следующего тега или до конца комментария.

Воспользовавшись Coco/R, реализуйте инструмент, который извлекает все документирующие комментарии из программы на MicroJava и выводит их. Для каждого прокомментированного элемента программы (константы, переменной, класса или метода) укажите его имя, а также документирующий его комментарий со всеми тегами и их компонентами.

В своей грамматике вы должны специфицировать только имеющие отношение к делу части программы на MicroJava. Прочие части можно пропустить с помощью {ANY} (*частичный разбор*, см. описание символа ANY в разделе 7.2).

Глава 8

Восходящий синтаксический анализ

В главе 3 мы обсуждали *нисходящий* синтаксический анализ методом рекурсивного спуска. Этот метод эффективен, и для большинства языков его достаточно (возможно, после преобразования грамматики или с помощью заглядывания на несколько лексем вперед). Это также единственный метод разбора, который можно реализовать вручную (без применения инструментов).

В этой главе мы рассмотрим альтернативный метод синтаксического анализа, который работает *снизу вверх* и хорош тем, что позволяет без преобразования обработать более широкий класс грамматик. Но есть и недостатки: инструментам (генераторам парсеров) обычно необходимо генерировать необходимые для разбора таблицы, а механизм семантической обработки более сложен и не такой мощный.

8.1. Как работает восходящий парсер

В отличие от нисходящего, восходящий парсер строит синтаксическое дерево программы снизу вверх. Заметим, что грамматика должна быть выражена в виде чистой БНФ, а не РБНФ. Поясним на примере. Следующая грамматика

```
S = X Y | S X Y.
X = a | a a b.
Y = b | b b a.
```

не обладает свойством LL(1), потому что первая продукция леворекурсивна, а в остальных альтернативы начинаются с одинаковых терминальных символов (см. раздел 3.3). Чтобы эту грамматику можно было обработать методом

рекурсивного спуска, ее необходимо преобразовать. Восходящий же парсер способен работать с ней без предварительного преобразования. Чтобы проанализировать вход

```
a b b a a b
```

синтаксическое дерево строится снизу вверх, используя *свертку* вместо вывода, как показано на рис. 8.1.

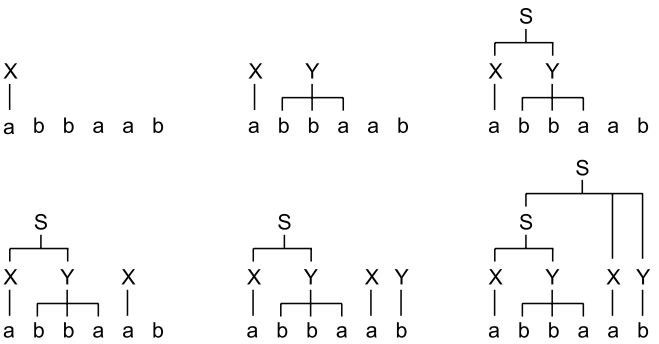


Рис. 8.1 ❖ Построение синтаксического дерева снизу вверх

Сначала `a` сворачивается в `X`, затем `b b a` – в `Y`, а потом `X Y` – в `S`. Далее следующая `a` сворачивается в `X`, затем `b` – в `Y` и, наконец, `S X Y` – в `S`. Таким образом, вход распознан, и синтаксическое дерево построено снизу вверх.

Но как парсер решает, какие части входа сворачивать? Он следует за действиями автомата с магазинной памятью (АМП), который мы обсуждали в разделе 3.1. Для входа

```
a b b a a b #
```

(где `#` – символ конца файла) парсер выполняет следующие действия, используя стек для хранения уже обработанных частей входа:

```
стек      вход            действие
          a b b a a b #   перенос
      a   b b a a b #     свертка a в X
      X   b b a a b #     перенос
    X b   b a a b #       перенос (не сворачивать b, иначе парсер упрется в тупик)
  X b b   a a b #         перенос
X b b a   a b #           свертка b b a в Y
    X Y   a b #           свертка X Y в S
      S   a b #           перенос
    S a   b #             свертка a в X
    S X   b #             перенос
  S X b   #               свертка b в Y
  S X Y   #               свертка S X Y в S
      S   #               вход распознан (стек содержит начальный символ; вход пуст)
```

Действие `shift` читает следующий входной символ и переносит его в стек, а действие `reduce` сворачивает части в конце стека (так называемое *окончание*) в нетерминальный символ. Какие части следует рассматривать как окончание, определяется таблицей разбора (см. раздел 8.3). В четвертой строке можно было бы свернуть `b` в `Y`, но это завело бы парсер в тупик. И на этот раз таблица разбора, которую мы обсудим ниже, говорит парсеру, что в этой ситуации следует переносить, а не сворачивать.

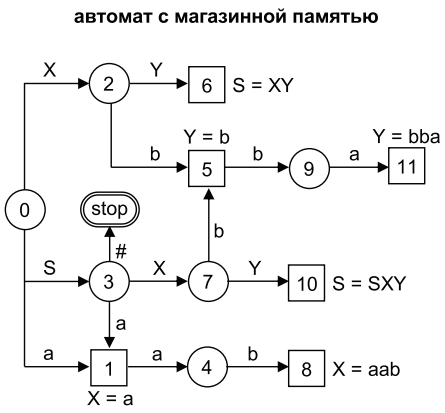
Как видим, восходящий парсер обрабатывает вход, многократно выполняя действия переноса `shift` и свертки `reduce`. Поэтому такие парсеры называются *парсерами типа перенос–свертка*. Есть еще одно действие, которое выполняется, когда вход полностью распознан (accept – приемка), а также действие, выполняемое в случае ошибки (error). Итого имеется четыре действия:

- *перенос*– читает следующий входной символ и переносит его в стек;
- *свертка* – сворачивает конец стека в нетерминальный символ;
- *приемка* – вход успешно распознан (только для `S . #`);
- *ошибка* – никакие другие действия невозможны (начинается обработка ошибки).

А как выглядит таблица разбора? Рассмотрим приведенный выше пример грамматики:

```
1 S = X Y.
2 S = S X Y.
3 X = a.
4 X = a a b.
5 Y = b.
6 Y = b b a.
```

Как показано в разделе 3.1, по этой грамматике можно построить автомат с магазинной памятью (АМП), а уже из этого автомата вывести таблицу переходов состояний (рис. 8.2). Затем эта таблица используется парсером для синтаксического анализа.



переходы состояний (таблица разбора)

	T-символы			*HT-символы*		
	a	b	#	S	X	Y
0	s1	–	–	s3	s2	–
1	s4	r3	–	–	–	–
2	–	s5	–	–	–	s6
3	s1	–	acc	–	s7	–
4	–	s8	–	–	–	–
5	r5	s9	r5	–	–	–
6	r1	–	r1	–	–	–
7	–	s5	–	–	–	s10
8	–	r4	–	–	–	–
9	s11	–	–	–	–	–
10	r2	–	r2	–	–	–
11	r6	–	r6	–	–	–

Рис. 8.2 ❖ АМП и результирующая таблица разбора

Таблица содержит действия, выполняемые в каждом состоянии (строка) для каждого символа (столбец). Возможны следующие виды действий.

sn	*shift n*	*потребить следующий входной символ и перейти в состояние n*
rp	*reduce p*	*свернуть согласно продукции* p
acc	*accept*	*вывод успешно распознан*
-	*error*	*текущий символ невозможно распознать*

Посмотрим, как парсер использует эту таблицу для распознавания сентенции a b b a a b #. Для этого он хранит *посещенные состояния* в стеке (*необработанные символы*, как было указано в кратком описании выше). Парсер начинает работу в состоянии 0 с еще не обработанным входом:

```
стек    вход          действие
  0     a b b a a b #  s1
```

Он ищет текущее состояние 0 и следующий входной символ a в таблице и находит там действие s1 (*shift 1*). Поэтому символ a потребляется, и парсер переходит в состояние 1, которое переносится в стек:

```
 0 1     b b a a b #     r3 (X = a)
```

При чтении символа b в состоянии 1 должно быть выполнено действие r3, т. е. парсер должен выполнить свертку согласно продукции 3 (X = a). Для этого он выталкивает из стека столько состояний, какова длина правой части продукции (т. е. 1 состояние), а затем вставляет нетерминальный символ в левую часть продукции в начало входа:

```
   0     X  b b a a b #   s2
```

При чтении символа X в состоянии 0 должно быть выполнено действие s2, т. е. X потребляется, и парсер переходит в состояние 2 (после свертки всегда имеется перенос по распознанному нетерминальному символу):

```
 0 2     b b a a b #   ...
```

И таким образом парсер, управляемый таблицей разбора, обрабатывает весь вход. Полностью анализ выглядит так:

```
      стек    вход          действие
        0     a b b a a b #  s1
      0 1     b b a a b #    r3 (X = a)
        0  X  b b a a b #    s2
      0 2     b b a a b #    s5
    0 2 5     b a a b #      s9
  0 2 5 9     a a b #        s11
0 2 5 9 11    a b #          r6 (Y = bba)
      0 2 Y   a b #          s6
    0 2 6     a b #          r1 (S = XY)
```

```
      0 S  a b #          s3
    0 3    a b #          s1
  0 3 1    b #            r3 (X = a)
    0 3 X  b #            s7
  0 3 7    b #            s5
0 3 7 5    #              r5 (Y = b)
  0 3 7 Y  #              s10
0 3 7 10   #              r2 (S = SXY)
      0 S  #              s3
    0 3    #              acc
```

Последним действием является *приемка* (acc), а это значит, что вход распознан. Протрассировать действия можно с помощью АМП на рис. 8.2, чтобы понять, как работает парсер, направляемый автоматом. После каждой свертки он строит часть синтаксического дерева (см. рис. 8.1), хотя это всего лишь воображаемый процесс (как и в случае рекурсивного спуска) – в действительности никакой структуры данных, которую можно было бы назвать синтаксическим деревом, не создается. Она «скрыта» за свертками парсера.

Теперь обратимся к парсеру как к программе. Она управляется тремя таблицами: таблица action, содержащая действия парсера (см. рис. 8.2), таблица length, в которой кодируется длина правой части для каждой продукции, и таблица leftSide, содержащая нетерминальный символ в левой части каждой продукции. Продукции и нетерминальные символы нумеруются последовательно. Ради экономии памяти действия парсера хранятся в элементах типа short, первый байт которых содержит вид действия (shift, reduce, accept, error), а второй – операнд действия.

```
void parse() {
   short[][] action = { {...}, {...}, ...}; // таблица переходов состояний
   byte[]    length = { ... };              // длины продукций
   byte[]    leftSide = { ... };            // нетерминальный символ в левой части
                                            // каждой продукции
   int       state;                         // текущее состояние
   int       sym;                           // следующий входной символ
   int       op, n, a;
   clearStack();
   state = 0; sym = next(); // получить следующий символ от сканера
   for (;;) {
      push(state);
      a = action[state][sym];
      op = a / 256; n = a % 256;
      switch (op) {
         case shift:          // shift n
            state = n; sym = next();
            break;
         case reduce:         // reduce n
            for (int i = 0; i < length[n]; i++) pop();
```

```
            a = action[top()][leftSide[n]]; n = a % 256; // shift n
            state = n;
            break;
         case accept:
            return;
         case error:
            System.exit(1); // обработка ошибки, см. ниже
      }
   }
}
```

Основной частью парсера является цикл, в котором по текущему состоянию (`state`) и следующему входному символу (`sym`) определяется следующее действие парсера, исходя из таблицы `action`. Прочитанный из таблицы элемент разбивается на вид действия `op` (первый байт) и операнд `n` (второй байт). Далее в предложении `switch` обрабатываются четыре вида действий, и цикл начинается сначала. Выход из цикла происходит, когда действие равно `accept` или `error`.

Парсер запоминает посещенные состояния в стеке (`push(state)`). Когда встречается действие `reduce`, он производит частичный возврат и продолжает после свертки нетерминального символа. Для этого он выталкивает из стека столько состояний, какова длина сворачиваемой продукции (`length[n]`), а затем использует состояние на вершине стека (`top()`) и нетерминальный символ, в который была свернута продукция (`leftSide[n]`), чтобы определить, в какое состояние перейти.

Таким образом, восходящий парсер – таблично управляемая программа, одинаковая для всех грамматик. Специфика конкретной грамматики заключена в таблицах, которыми парсер параметризован.

8.2. LR-грамматики

Восходящие парсеры еще называют *LR-парсерами*. Название происходит от слова «распознавание слева (**l**eft) направо с помощью правостороннего (**r**ight-canonical) вывода». Хотя такие парсеры работают не с выводом, а со сверткой, правосторонний вывод, при котором сначала выводится самый правый нетерминальный символ

S ⇒ S X **Y** ⇒ S **X** b ⇒ **S** a b ⇒ X **Y** a b ⇒ **X** b b a a b ⇒ a b b a a b

соответствует с точностью до направления левосторонней свертке, при которой сначала сворачивается самое левое окончание. Так что название, хотя и не самое удачное, неверным все же не является.

В зависимости о того, насколько далеко вперед заглядывает парсер в процессе разбора, различают LR(0)- и LR(1)-грамматики.

8.2.1. LR(0)-грамматики

В этом случае парсер не использует опережающий символ для свертки – он сворачивает вне зависимости от следующего входного символа. Грамматика называется LR(0), если

- не существует состояния свертки, в котором возможен также перенос;
- в каждом состоянии свертки возможна свертка согласно единственной продукции.

Наш пример не является LR(0)-грамматикой, потому что АМП на рис. 8.2 имеет два состояния (1 и 5), в которых возможны как свертка, так и перенос. Поэтому без опережающего символа различить `reduce` и `shift` невозможно. Практическое значение LR(0)-грамматик невелико.

8.2.2. LR(1)-грамматики

В этом случае парсер использует опережающий символ, чтобы различать действия `reduce` и `shift`. Грамматика называется LR(1), если в каждом состоянии можно с помощью только одного опережающего символа решить:

- какое действие – `shift` или `reduce` – следует выполнить;
- в какой нетерминальный символ нужно свернуть.

Наш пример является LR(1)-грамматикой, потому что АМП на рис. 8.2 может использовать опережающий символ в состояниях 1 и 5, чтобы решить, нужно ли выполнять `shift` или `reduce`. Если опережающим символом в состоянии 1 является `a`, то производится перенос с переходом в состояние 4, иначе свертка. Если опережающим символом в состоянии 5 является `b`, то производится перенос с переходом в состояние 9, иначе свертка.

8.2.3. LALR(1)-грамматики

LR(1)-грамматики весьма мощные, но приводят к большим таблицам разбора. Поэтому на практике применяется вариант этой идеи, получивший название LALR(1)-грамматики (LR(1)-грамматики с заглядыванием вперед). Название не вполне удачное, потому что и LR(1)-грамматики заглядывают вперед, но этот термин прижился.

LALR(1)-грамматики образуют подмножество LR(1)-грамматик. Они лишь ненамного менее мощные, но зато таблицы разбора гораздо меньше, потому что состояния с одними и теми же действиями, но, возможно, разными последующими символами объединяются. Рассмотрим пример небольшой грамматики:

```
E = v | E "+" v | "(" E ")".
```

На рис. 8.3 слева показан АМП, основанный на подходе LR(1), а справа – основанный на подходе LALR(1), когда некоторые состояния объединены.

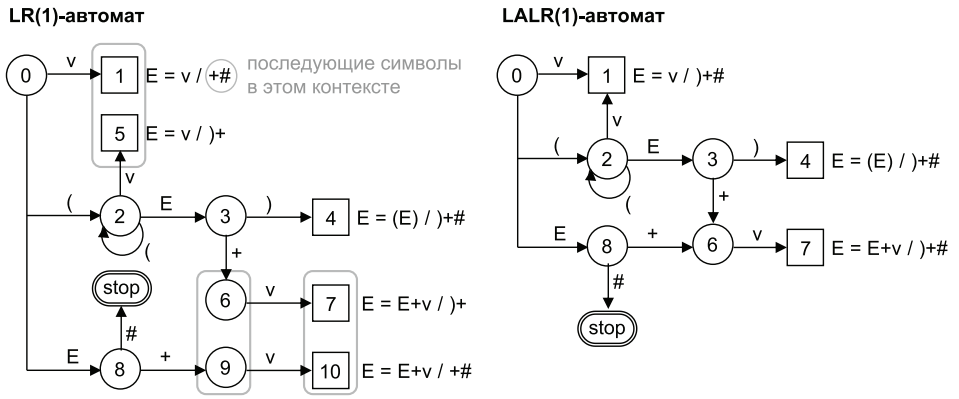


Рис. 8.3 ❖ АМП, основанные на подходах LR(1) и LALR(1)

Прежде всего мы видим, что в состоянии 1 левого автомата свертка производится согласно продукции `E = v`. Однако также видно, что эта свертка может иметь место, только если далее во входном потоке находится один из символов "+" или "#" (что обозначается / +#). В состоянии 5 также имеется свертка согласно продукции `E = v`, но только если далее следует ")" или "+". Поэтому оба состояния можно объединить, образовав состояние 1 правого автомата, для которого множество последующих символов является объединением двух предыдущих множеств, т. е. ")", "+" и "#". Аналогично можно объединить состояния 6 и 9, а равно состояния 7 и 10 левого автомата. Если это приводит к автомату, в котором с помощью единственного опережающего символа все еще можно решить, какое действие – `shift` или `reduce` – следует выполнять в каждом состоянии, то мы имеем LALR(1)-грамматику. Именно такой случай налицо здесь. Три состояния удалось устранить посредством объединения. Поэтому таблица разбора уменьшилась.

Но можно ли просто объединить множества последующих символов, когда объединяются состояния свертки? Что, если автомат справа, прочитав символ `v` в состоянии 0, переходит из него в состояние 1, а затем видит символ ")"? Сворачивая согласно продукции `E = v`, автомат возвращается в состояние 0, а оттуда, прочитав `E`, в состояние 8, где допустимы только символы "+" и "#". Если далее следует символ ")", то парсер сообщает об ошибке. Поэтому ошибка не потерялась, она просто диагностируется не сразу во время свертки, а позже, когда выполняется следующий перенос по терминальному символу.

8.2.4. Сильные стороны восходящего синтаксического анализа

- LALR(1)-грамматики мощнее LL(1)-грамматик, используемых для разбора методом рекурсивного спуска. Они допускают леворекурсивные продукции, а также альтернативы, начинающиеся одним и тем же терминальным символом. Поэтому такие грамматики можно подвергать восходящему синтаксическому анализу без предварительного преобразования.
- Код LALR(1)-парсеров компактнее, чем код парсеров рекурсивного спуска. Однако таблицы потребляют много памяти.
- Показанный выше восходящий парсер является универсальной программой, которая параметризуется таблицами.
- Таблично управляемые восходящие парсеры допускают улучшенную обработку ошибок, в чем мы убедимся в разделе 8.6.

8.2.5. Сильные стороны восходящего синтаксического анализа

- Таблицы LALR(1) трудно строить вручную. В разделе 8.3 мы увидим, как это делается для небольших грамматик, но при увеличении размера необходимы инструментальные средства (т. е. генераторы парсеров).
- LALR(1)-парсеры немного медленнее парсеров рекурсивного спуска из-за необходимости доступа к таблицам.
- Семантическая обработка сложнее и менее мощная, чем в парсерах рекурсивного спуска (см. раздел 8.5).
- Действия восходящих парсеров труднее трассировать и проверять. Парсер проходит последовательность состояний, но их трудно отобразить на исходную грамматику.
- LALR(1)-парсеры требуют записи грамматики в форме БНФ, а такую запись читать труднее, чем РБНФ-грамматики, применяемые в методе рекурсивного спуска.

По этим причинам восходящий синтаксический анализ имеет смысл применять только к действительно сложным языкам, грамматику которых трудно преобразовать в форму LL(*). Для языков поменьше, например Java или MicroJava (а также для самоопределяемых языков команд), рекурсивный спуск гораздо проще и к тому же может быть реализован вручную, что является дополнительным преимуществом.

8.3. Генерирование LR-таблиц

В разделе 8.2 мы видели, как парсер использует таблицу переходов состояний на рис. 8.2 для анализа входной фразы. Но как создать такую таблицу? Именно этот вопрос мы сейчас и рассмотрим. Таблица разбора выводится из грамматики, но делать это вручную очень утомительно и практически возможно только для совсем небольших грамматик. Для грамматик побольше нужны генераторы парсеров.

Предположим, что парсеру нужно разобрать программу согласно следующей грамматике:

```
S = a X b
X = c
```

Давайте промоделируем анализ, в ходе которого парсер перемещается по продукциям грамматики. Точка обозначает текущую позицию разбора:

```
S = . a X b
S = a . X b
X = . c     / b
X = c .     / b
S = a X . b
S = a X b .
```

Парсер начинает работу с первой продукции, точка находится в ее начале. После распознавания `a` точка сдвигается за `a`, и парсер оказывается перед `X`. Находясь перед нетерминальным символом, он также находится в начале его продукции, в данном случае – перед `c`. Символом, следующим за продукцией `X`, в этом контексте является `b` (что представляется как `/ b`). Теперь парсер распознает `c` и оказывается в конце продукции `X`, а значит, также после `X` в продукции более высокого уровня. Наконец, он распознает `b`, и на этом анализ завершается.

Сделаем такой мгновенный снимок анализа и назовем его *LR-элементом*. На рис. 8.4 показана общая структура LR-элемента (α и β – строки терминальных или нетерминальных символов, γ обозначает множество терминальных символов).

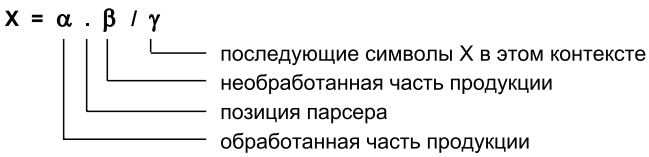


Рис. 8.4 ❖ Структура LR-элемента

Существует два типа LR-элементов: такие, для которых точка находится в конце продукции (*элементы свертки*), и такие, для которых она находится

не в конце (*элементы переноса*). Символ, следующий сразу за точкой, называется *управляющим символом*. В элементе переноса

```
X = a . a b / c
```

управляющий символ равен a, а в элементе свертки

```
X = a a b . / c
```

управляющий символ равен c (в принципе, это может быть любой символ из множества последующих).

В любой момент времени парсер находится в некотором *состоянии*, где он может работать над несколькими элементами параллельно т. е. над всеми возможными продукциями, например:

```
S = X . Y c / #
Y = . b / c
Y = . b b a / c
```

Поэтому мы называем *состоянием разбора* множество элементов, над которыми парсер работает в данный момент. Поскольку в первом элементе парсер находится перед нетерминальным символом Y, он одновременно работает над всеми продукциями Y, для которых точка находится в начале, а их множества последующих символов являются множествами символов, следующих за Y в этом контексте (здесь c). Таким образом, эти элементы также являются членами состояния.

Данный пример уже демонстрирует возможности LR-анализа. Нисходящий парсер в каждый момент работает только с одной продукцией, тогда как восходящий может работать с несколькими продукциями параллельно. На рис. 8.5 показано, как восходящий парсер может анализировать две продукции, начинающиеся одинаковыми терминальными символами, параллельно, пока наконец не будет принужден выбрать одну из них.

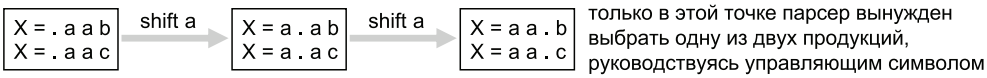


Рис. 8.5 ❖ Восходящий парсер работает над двумя продукциями параллельно

8.3.1. Ядро, замыкание и последующее состояние

Множество всех элементов состояния, которые не начинаются точкой, называется *ядром* состояния. В примере выше множество

```
S = X . Y c / #
```

образует ядро. По ядру можно вычислить *замыкание* состояния: для всех элементов, в которых точка находится перед нетерминальным символом,

продукции этого нетерминального символа добавляются в состояние как новые элементы. Если грамматика имеет вид

```
S = X Y c
X = a
X = a a b
Y = b
Y = b b a
```

то замыкание вышеупомянутого ядра можно построить, добавив продукции Y с точкой в начале и множества следующих за ними элементов (в данном случае c):

```
S = X . Y c  / #
Y = . b      / c
Y = . b b a  / c
```

Псевдокод алгоритма формирования замыкания состояния имеет вид (First(βγ) обозначает терминальные начальные символы β; если β пусто, то это множество последующих символов γ):

```
foreach (item in state) {
   if (элемент вида X = α . Y β / γ) {
      добавить все продукции Y = . ω / First(βγ) в состояние в качестве новых элементов;
   }
}
```

Замыкание состояния описывает все элементы, над которыми парсер работает, находясь в этом состоянии. Построив замыкание, мы можем для каждого управляющего символа sym определить последующее состояние *Succ(state, sym)*, в которое мы переходим из state, прочитав sym (рис. 8.6).

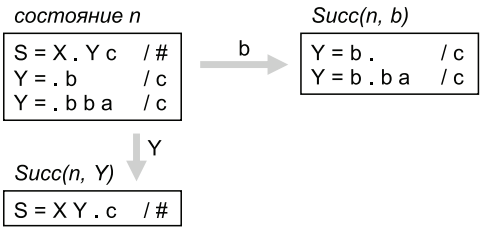


Рис. 8.6 ❖ Вычисление последующих состояний и их множеств элементов

Управляющими символами в состоянии n являются Y и b (символы после точки). Последующее состояние Succ(n, b) содержит все элементы n с управляющим символом b, для которых точка пересекла b. Последующее состояние Succ(n, Y) содержит все элементы n с управляющим символом Y, для которых точка пересекла Y.

Затем продукции, принадлежащие последующим состояниям, снова анализируются параллельно. Заметим, что сформированное таким образом последующее состояние представляет только ядро, по которому, возможно, еще придется вычислять замыкание, но в данном примере такой необходимости не возникает, потому что ни в одном из элементов нет точки перед нетерминальным символом.

8.3.2. Алгоритм генерирования таблиц

Теперь рассмотрим алгоритм генерирования LALR(1)-таблицы, который затем применим к нашему примеру. Ниже приведен его псевдокод:

```
расширить грамматику псевдопродукцией S' = S #
создать состояние 0 с ядром S' = . S# // исключительное ядро; см. комментарий ниже
while (не все состояния посещены) {
   s = следующее непосещенное состояние;
   построить замыкание s;
   for (всех элементов s) {
      switch (вид элемента) {
         case S' = S . #:     сгенерировать acc #;
                              break;
         case X = α . y β / γ : создать вспомогательное состояние s1 = Succ(s, y);
                              if (∃ s2: kernel(s1) == kernel(s2)) {
                                 объединить множества последующих символов s1 в s2;
                                 сгенерировать shift y, s2;
                              } else {
                                 добавить s1 в качестве нового состояния
                                 сгенерировать shift y, s1;
                              }
                              break;
         case X = α . / γ :    сгенерировать reduce γ, (X = α);
                              break;
      }
   }
   // все неопределенные переходы – ошибочные действия
}
```

Этот алгоритм сначала расширяет грамматику путем добавления псевдопродукции `S' = S #`, где `S` – оригинальный начальный символ, а `#` – символ `eof`. Затем создается состояние 0 с ядром `S' = . S #` (это исключение, обычно ядро содержит только элементы, не начинающиеся точкой).

Теперь начинается цикл построения таблицы, в котором посещаются все состояния, создаются новые состояния и генерируются действия во всех состояниях. Для каждого посещенного состояния сначала по его ядру строится замыкание. Затем исследуются все элементы состояния, каковых может быть три вида:

- элемент `S' = S . #` – тот, для которого был распознан оригинальный начальный символ `S`, а кроме него, вход содержит только символ конца файла `#`. Поэтому в текущем состоянии генерируется действие `acc`;

- в элементах переноса типа `X = α . y β / γ` точка находится перед символом `y` (который может быть терминальным или нетерминальным). Поэтому будет существовать действие `shift` по `y`. Алгоритм сначала создает вспомогательное состояние `s1`, используя `Succ(s, y)`. Затем он проверяет, существует ли уже другое состояние `s2` с таким же ядром, как у `s1` (различия в множествах последующих символов игнорируются). Если да, то символы, следующие за элементами в `s1`, добавляются к множеству символов, следующих за теми же элементами в `s2`, и генерируется действие `shift y, s2`. Если нет, то `s1` добавляется как новое состояние и генерируется действие `shift y, s1`;
- в элементах свертки типа `X = α . / γ` точка находится в конце продукции. Поэтому генерируется действие `reduce` с последующим множеством γ.

Рассмотрим пример. Грамматика (включая псевдопродукцию `S'`), для которой мы хотим сгенерировать таблицу, имеет вид:

```
0  S' = S #
1  S = X Y
2  S = S X Y
3  X = a
4  X = a a b
5  Y = b
6  Y = b b a
```

Мы начинаем с состояния 0, которое первоначально содержит только следующее ядро:

```
S' = . S #
```

Чтобы построить замыкание этого состояния, мы должны сначала добавить как элементы все продукции `S` с точкой в начале и последующим символом #:

```
S' = . S #
S = . X Y    / #
S = . S X Y  / #
```

В добавленных элементах точка снова находится перед нетерминальным символом, поэтому нам нужно добавить продукции `X` и `S` как новые элементы. Последующими символами `X` являются терминальные начальные символы `Y`, т. е. `b`. Последующими символами `S` в третьем элементе являются терминальные начальные символы `X`, т. е. `a`. Таким образом, мы добавляем эти новые последующие символы `S` в множество последующих символов уже существующих элементов `S` (что дает нам / #a):

```
S' = . S #
S = . X Y / #a
S = . S X Y / #a
X = . a / b
X = . a a b / b
```

Теперь замыкание вычислено полностью, потому что больше нет элементов с точкой перед нетерминальным символом. Далее мы создадим последующие состояния, в которые попадаем при чтении управляющих символов a, X и S. Увидев управляющий символ a, мы переходим в новое состояние 1, содержащее элементы

```
X = a .       / b
X = a . a b   / b
```

В случае управляющего символа X мы переходим в новое состояние 2, содержащее элемент ядра

```
S = X . Y     / #a
```

В случае управляющего символа S мы переходим в новое состояние 3, содержащее элементы ядра

```
S' = S . #
S = S . X Y   / #a
```

Элементами новых состояний являются только ядро, которое впоследствии должно быть расширено до замыкания, когда эти состояния будут посещаться (это относится к состояниям 2 и 3, для которых перед нетерминальным символом находится точка). В состоянии 0 генерируются действия

```
shift a, 1  // перенос по a в состояние 1
shift X, 2  // перенос по X в состояние 2
shift S, 3  // перенос по S в состояние 3
```

Ниже показаны состояния, созданные этим алгоритмом, – с элементами и сгенерированными действиями (элементы ядра выделены полужирным шрифтом):

```
0   S' = . S #                  shift   a    1
    S = . X Y       / #a        shift   X    2
    S = . S X Y     / #a        shift   S    3
    X = . a         / b
    X = . a a b     / b
1   X = a .         / b         red     b    3 (X = a)
    X = a . a b     / b         shift   a    4
2   S = X . Y       / #a        shift   b    5
    Y = . b         / #a        shift   Y    6
    Y = . b b a     / #a
3   S' = S . #                  acc     #
    S = S . X Y     / #a        shift   a    1 (!)
    X = . a         / b         shift   X    7
    X = . a a b     / b
4   X = a a . b     / b         shift   b    8
5   Y = b .         / #a        red     #,a  5 (Y = b)
    Y = b . b a     / #a        shift   b    9
6   S = X Y .       / #a        red     #,a  1 (S = X Y)
```

```
7   S = S X . Y    / #a          shift   b    5 (!)
    Y = . b        / #a          shift   Y    10
    Y = . b b a    / #a
8   X = a a b .    / b           red     b    4 (X = a a b)
9   Y = b b . a    / #a          shift   a    11
10  S = S X Y .    / #a          red     #,a  2 (S = S X Y)
11  Y = b b a .    / #a          red     #,a  6 (Y = b b a)
```

Заметим, что в состоянии 3 имеет место перенос по a в уже существующее состояние 1 (нового состояния не создается). Аналогично в состоянии 7 имеет место перенос по b в уже существующее состояние 5. Состояния и сгенерированные действия дают таблицу разбора, показанную на рис. 8.7.

	a	b	#	S	X	Y
0	s1	–	–	s3	s2	–
1	s4	r3	–	–	–	–
2	–	s5	–	–	–	s6
3	s1	–	acc	–	s7	–
4	–	s8	–	–	–	–
5	r5	s9	r5	–	–	–
6	r1	–	r1	–	–	–
7	–	s5	–	–	–	s10
8	–	r4	–	–	–	–
9	s11	–	–	–	–	–
10	r2	–	r2	–	–	–
11	r6	–	r6	–	–	–

Рис. 8.7 ❖ Таблица разбора для демонстрационной грамматики

Записи таблицы, для которых не определено действие, являются ошибочными. Например, парсер сообщит об ошибке, если увидит символ b в состоянии 0. После этого он начнет процедуру восстановления после ошибки, которую мы будем рассматривать в разделе 8.6.

Как видим, эта таблица сильно разрежена, что становится еще заметнее для больших грамматик с большим количеством символов. Поэтому иногда таблицу хранят в виде списка, в котором для каждого состояния указаны управляющий символ и действие.

```
    T-действия      НТ-действия
0   a  s1           S   s3
    *  error        X   s2
1   a  s4
    *  r3
2   b  s5           Y   s6
    *  error
3   a  s1           X   s7
    #  acc
    *  error
4   b  s8
    *  error
```

```
5   b  s9
    *  r5
6   *  r1
7   b  s5          Y   s10
    *  error
8   *  r4
9   a  s11
    *  error
10  *  r2
11  *  r6
```

Если символ a встречается в состоянии 0, то выполняется действие s1 (перенос в состояние 1), в противном случае (обозначено звездочкой) диагностируется ошибка. Действия состояния последовательно проверяются для соответствующего управляющего символа. Последнее действие всегда является «действием по умолчанию» и обозначается звездочкой. Как видим, его можно использовать также для свертки. Если свертка производится по некорректному управляющему символу, то парсер запомнит ошибку и сообщит о ней при следующем переносе.

Поскольку часть таблицы для нетерминальных символов разрежена еще сильнее, чем для терминальных, используется два отдельных списка – для терминальных и нетерминальных символов. В нетерминальной части нет ошибочных действий, потому что после свертки всегда имеется перенос по свернутому нетерминальному символу.

Список эффективнее таблицы, но и доступ к нему медленнее, потому что действия приходится просматривать последовательно. Поэтому в последующих разделах мы будем использовать табличную форму.

8.3.3. LR(1)-конфликты

Мы видели, что в процессе нисходящего синтаксического анализа могут возникать LL(1)-конфликты, которые обычно следует устранять (раздел 3.3). Аналогично в процессе восходящего синтаксического анализа могут возникать LR(1)-конфликты, делающие грамматику непригодной для восходящего разбора. Существует два типа таких конфликтов.

- *Конфликт типа перенос–свертка* имеет место, если парсер (ориентируясь на опережающий символ) не может решить, что выполнять: перенос или свертку:

  ```
  shift a, ...
  reduce a, ...
  ```

- *Конфликт типа свертка–свертка* имеет место, если парсер (ориентируясь на опережающий символ) не может решить, согласно какой продукции выполнить свертку:

  ```
  reduce a, p1
  reduce a, p2
  ```

Если такие конфликты возникают, то их необходимо устранить, прежде чем грамматику можно будет подвергнуть восходящему разбору. К сожалению, устранять LR(1)-конфликты гораздо труднее, чем LL(1)-конфликты, и никакой общей процедуры тут не придумаешь. Вам придется понять, что вызывает конфликт, а затем преобразовать грамматику, так чтобы этот конфликт убрать. Тут не обойтись без интуиции.

LALR(1)-таблицы более подвержены конфликтам, чем LR(1)-таблицы, потому что множества состояний и последующих символов объединяются, тогда как в LR(1)-таблицах они хранятся отдельно. Однако объединение может привести только к конфликтам типа свертка–свертка, поскольку конфликт типа перенос–свертка существовал бы уже в LR(1)-таблице.

8.4. Сжатие LR-таблиц

Для сколько-нибудь реалистичного языка программирования таблица разбора может быть очень велика. Например, если в грамматике 80 терминальных и 200 нетерминальных символов, а таблица разбора имеет 2500 состояний, то в ней будет 280 × 2500 = 700 000 элементов. Если каждый элемент занимает четыре байта, то для таблицы потребуется 2,8 мегабайта. Хотя для современных компьютеров такой объем памяти не составляет проблемы, существуют методы, позволяющие сжать таблицы с коэффициентом, достигающим 90 %. Мы рассмотрим два таких метода.

8.4.1. Объединение действий переноса и свертки

Иногда действие `shift` приводит к состоянию, в котором возможна только свертка. Например, в состоянии 4 на рис. 8.8 перенос по символу `b` ведет в состояние 8, в котором никакие дальнейшие переносы невозможны, а допустима только свертка согласно продукции 4 (ошибочные записи игнорируются, потому что возможная ошибка после преждевременной свертки обнаруживается при следующем переносе).

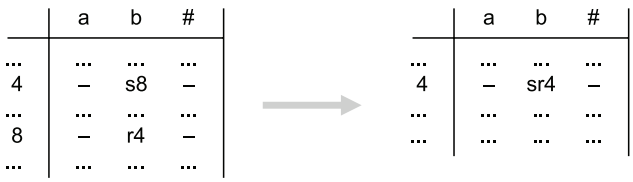


Рис. 8.8 ❖ Объединение действий переноса и свертки

Поэтому действие `s8` можно объединить с действием `r4`, образовав действие `sr4` (перенос, а затем свертка согласно продукции 4). После этого состояние 8 оказывается недостижимым, поэтому всю строку таблицы, соот-

ветствующую состоянию 8, можно удалить. Объединение действий переноса и свертки не замедляет парсер, а наоборот – делает его быстрее, поскольку вместо двух действий нужно интерпретировать только одно. Однако парсер (см. раздел 8.1) необходимо модифицировать, потому что теперь появилось новое действие `shiftred`:

```
switch (op) {
   ...
   case shiftred: // shiftred n
      sym = next();
      do {
         for (int i = 0; i < length[n] - 1; i++) pop(); // reduce, но выталкивается на одно
                                                        // состояние меньше
         a = action[top()][leftSide[n]];
         op = a / 256; n = a % 256;
      } while (op == shiftred);
      // op == shift
      state = n; break;
   case reduce: // reduce n
      for (int i = 0; i < length[n]; i++) pop(); // reduce
      a = action[top()][leftSide[n]];
      op = a / 256; n = a % 256;
      while (op == shiftred) {
         for (int i = 0; i < length[n] - 1; i++) pop(); // reduce, но выталкивается на одно
                                                        // состояние меньше
         a = action[top()][leftSide[n]];
         op = a / 256; n = a % 256;
      }
      // op == shift
      state = n; break;
}
```

Действие `shiftred n` читает следующий символ, но не выполняет переноса; вместо этого оно немедленно выполняет свертку согласно продукции `n`, но выталкивает из стека на одно состояние меньше, чем длина продукции, т. к. последний перенос не был выполнен. После этого `shiftred` могло бы возникнуть еще раз со свернутым нетерминальным символом (`leftside[n]`), по этой причине и выполняется цикл до тех пор, пока не встретится свертка в нетерминальный символ, по которому можно было бы выполнить перенос.

То же самое относится к действию `reduce`. Здесь тоже могло бы иметь место действие `shiftred` со свернутым нетерминальным символом, и потому выполняется цикл, пока не встретится `shift`.

8.4.2. Объединение строк

Как мы видели, таблица разбора разрежена. Поэтому имеет смысл объединять строки, действия в которых не конфликтуют между собой. В результате таблица сжимается. Например, на рис. 8.9 строки 0, 3 и 4 не конфликтуют (они содержат одинаковые действия или ошибочные записи), а потому могут быть объединены.

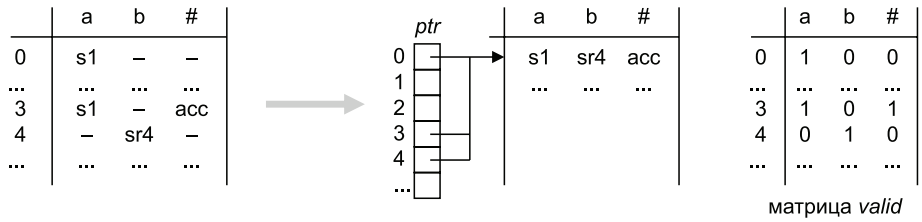


Рис. 8.9 ❖ Объединение строк

В результате объединения некоторые строки удаляются из таблицы. Однако нам необходим указательный массив `ptr`, чтобы отобразить строки 0, 3 и 4 на строку 0. Кроме того, нужна битовая матрица `valid`, сообщающая, какие действия допустимы в каждом из оригинальных состояний. Если мы хотим, чтобы парсер (см. раздел 8.1) работал с такой объединенной таблицей, то придется внести в него кое-какие изменения (напечатаны полужирным шрифтом).

```
BitSet[] valid;
short[] ptr;
...
a = action[ptr[state]][sym];
op = a / 256; n = a % 256;
if (!valid[state].get(sym)) op = error;
switch (op) {
...
```

При поиске в таблице необходимо принимать во внимание косвенную адресацию через `ptr`. Кроме того, нужно проверять, допустим ли вообще символ `sym` в состоянии `state`. Это немного замедляет разбор.

Рекомендуется объединять части таблицы для терминальных и нетерминальных символов порознь, поскольку часть для нетерминальных символов гораздо более разрежена, что обеспечивает большую степень сжатия.

В принципе, такую же технику можно было бы применить для объединения столбцов, но это замедлило бы разбор еще сильнее. Сегодня, когда память дешева и ее может быть много, лучше поискать компромисс между размером таблицы и скоростью работы парсера.

8.4.3. Пример

В качестве примера рассмотрим таблицу разбора из раздела 8.3 и сначала объединим действия `shift` и `reduce` в действия `shiftred` (рис. 8.10).

Действия `s6`, `s8`, `s10` и `s11` можно объединить в `sr1`, `sr4`, `sr2` и `sr6`, устранив тем самым строки 6, 8, 10 и 11. В оригинальной таблице было 12 строк по 6 столбцов по 2 байта, т. е. всего 144 байта, а в новой осталось только 8 строк по 6 столбцов по 2 байта, т. е. 96 байт – сокращение на 33 % без принесения в жертву скорости разбора.

	a	b	#	S	X	Y
0	s1	–	–	s3	s2	–
1	s4	r3	–	–	–	–
2	–	s5	–	–	–	(s6)
3	s1	–	acc	–	s7	–
4	–	(s8)	–	–	–	–
5	r5	s9	r5	–	–	–
6	r1	–	r1	–	–	–
7	–	s5	–	–	–	(s10)
8	–	r4	–	–	–	–
9	(s11)	–	–	–	–	–
10	r2	–	r2	–	–	–
11	r6	–	r6	–	–	–

	a	b	#	S	X	Y
0	s1	–	–	s3	s2	–
1	s4	r3	–	–	–	–
2	–	s5	–	–	–	**sr1**
3	s1	–	acc	–	s7	–
4	–	**sr4**	–	–	–	–
5	r5	s9	r5	–	–	–
7	–	s5	–	–	–	**sr2**
9	**sr6**	–	–	–	–	–

Рис. 8.10 ❖ Пример: объединение действий `shift` и `reduce`

Теперь перейдем к объединению строк. В части с терминальными символами можно объединить строки 0, 3 и 4, а также строки 2, 7 и 9, а в части с нетерминальными символами – строки 0 и 2, а также строки 3 и 7 (рис. 8.11).

T-действия

	a	b	#
0,3,4	s1	sr4	acc
1	s4	r3	–
2,7,9	sr6	s5	–
5	r5	s9	r5

NT-действия

	S	X	Y
0,2	s3	s2	sr1
3,7	–	s7	sr2

valid

	a	b	#
0	1	0	0
1	1	1	0
2	0	1	0
3	1	0	1
4	0	1	0
5	1	1	1
7	0	1	0
9	1	0	0

Рис. 8.11 ❖ Пример: объединение строк

Теперь нам нужно 6 строк по 3 столбца по 2 байта (36 байт) для действий, 2 × 8 строк по 2 байта (32 байта) для таблиц `ptr` и 8 битовых массивов по одному байту (8 байт) для матрицы `valid` (в предположении, что каждую строку можно сохранить как битовый массив в одном байте) – всего 76 байт. Это сокращение на 47 % по сравнению с оригинальной таблицей, которая занимала 144 байта. Для больших грамматик, в которых таблицы обычно гораздо сильнее разрежены, экономия может достигать 90 %.

8.5. Семантическая обработка

До сих пор мы говорили только о синтаксическом анализе. Однако компилятор должен не только анализировать, но и транслировать программу, а для этого нужны семантические действия и атрибуты. Как включить их в восходящий парсер?

8.5.1. Семантические действия

Как мы видели, восходящий парсер может анализировать несколько продукций параллельно. В крайних случаях он знает, какая продукция правильная, только когда имеется свертка. Поэтому семантические действия можно выполнять лишь во время сверток. Если бы семантическое действие было разрешено выполнять в середине продукции

```
X = a (. семантическое действие .) b.
```

то нам пришлось бы вставить там пустой нетерминальный символ и выполнить семантическое действие во время свертки этого символа:

```
X = a Y b.
Y = (. семантическое действие .).
```

Однако это могло бы нарушить свойство LR(1). В следующем примере

```
X = a b c.
X = a (. семантическое действие .) b d.
```

контекстно-свободная грамматика обладает свойством LR(1). Но если преобразовать ее к виду

```
X = a b c.
X = a Y b d.
Y = (. семантическое действие .).
```

то возникает конфликт типа перенос–свертка:

```
i X = a . b c / #    shift  b   i+1
  X = a . Y b d / #  red    b   p (Y = )
  Y = . / b          shift  Y   i+2
```

Видя опережающий символ b, парсер не может решить, что делать: перенос или свертку (и попутно выполнить семантическое действие). Следовательно, он больше не может обрабатывать обе продукции X параллельно, потому что во второй продукции должно быть выполнено семантическое действие. Таким образом, семантическое действие в середине продукции лишило грамматику свойства LR(1). По этой причине семантические действия в общем случае могут выполняться только в конце продукции, когда стало ясно, какая из нескольких параллельно анализируемых продукций правильна.

8.5.2. Атрибуты

Входные атрибуты приводят к таким же проблемам, как семантические действия. В атрибутной грамматике

```
X = a Y<↓v> b.
Y<↓w> = ... .
```

передачу атрибута v в w можно рассматривать как присваивание в семантическом действии:

```
X = a (. w = v; .) Y b.
Y = ... .
```

что опять-таки нарушило бы свойство LR(1). Поэтому в восходящем синтаксическом анализе входные атрибуты обычно не используются.

С другой стороны, *выходные атрибуты* не составляют проблемы. Однако обычно предполагается, что у нетерминального символа есть только один выходной атрибут, и этот атрибут оставляется в специальном *стеке атрибутов*, откуда его можно извлечь и обработать, например:

```
X<↑x> = A<↑a> B<↑b> C<↑c> (. c = popAttr(); b = popAttr(); a = popAttr();
                             x = f(a, b, c);
                             pushAttr(x); .) .
```

Каждый из нетерминальных символов A, B и C оставляет свой атрибут в стеке атрибутов. В продукции X эти атрибуты извлекаются из стека с помощью метода popAttr(), обрабатываются, после чего результирующее значение x помещается назад в стек атрибутов как выходной атрибут X методом pushAttr(x). Все это можно проделать в семантическом действии в конце продукции, которая выполняется, когда A B C сворачивается в X.

Рассмотрим конкретный пример такого вычисления атрибута, а именно вычисление простых константных выражений, описываемых следующей атрибутной грамматикой:

```
Expr<↑x>   = Term<↑x>                (. pushAttr(popAttr()); .) .
Expr<↑x>   = Expr<↑x> "+" Term<↑y>   (. pushAttr(popAttr() + popAttr()); .) .
Term<↑x>   = Factor<↑x>              (. pushAttr(popAttr()); .) .
Term<↑x>   = Term<↑x> "*" Factor<↑x> (. pushAttr(popAttr() * popAttr()); .) .
Factor<↑x> = number                  (. pushAttr(t.numVal); .) .
Factor<↑x> = "(" Expr<↑x> ")"        (. pushAttr(popAttr()); .) .
```

Все семантические действия находятся в конце продукций. В первой продукции атрибут, предоставленный Term, может быть просто использован в качестве выходного атрибута Expr и, стало быть, оставлен в стеке атрибутов. Похожая ситуация имеет место для третьей и последней продукций.

Остальные семантические действия нумеруются последовательно, что приводит к следующей грамматике:

```
Expr   = Term .
Expr   = Expr "+" Term   (. 1 .) .
Term   = Factor .
Term   = Term "*" Factor (. 2 .) .
Factor = number          (. 3 .) .
Factor = "(" Expr ")" .
```

Все семантические действия собираются в метод semAction(), параметризованный номером подлежащего выполнению действия. Затем действия идентифицируются своим номером в предложении switch:

```
void semAction (int n) {
   switch (n) {
      case 1: pushAttr(popAttr() + popAttr()); break;
      case 2: pushAttr(popAttr() * popAttr()); break;
      case 3: pushAttr(t.numVal); break;
   }
}
```

Наконец, мы должны убедиться, что семантические действия выполняются в нужном месте. Для этого создадим таблицу `sem`, в которой для каждой продукции указывается семантическое действие, которое должно быть выполнено при свертке этой продукции. Если никакого действия выполнять не нужно, то соответствующий элемент таблицы равен 0. На рис. 8.12 показаны таблицы `leftSide`, `length` и `sem` для описанной выше грамматики.

	leftSide	length	sem
0	Expr	1	0
1	Expr	3	1
2	Term	1	0
3	Term	3	2
4	Factor	1	3
5	Factor	3	0

Рис. 8.12 ❖ Таблицы разбора `leftSide`, `length` и `sem`

Нам необходимо изменить парсер (см. раздел 8.1), так чтобы для продукции выполнялось семантическое действие, если оно указано в таблице `sem`.

```
switch (op) {
   ...
   case reduce: // reduce n
      if (sem[n] != 0) semAction(sem[n]);
      for (int i = 0; i < length[n], i++) pop();
      a = action[top()][leftSide[n]]; n = a % 256; // shift n
      state = n; break;
   }
```

Аналогичное изменение необходимо, если парсер поддерживает также действия `shiftred`.

8.5.3. Оценка

Как видим, в восходящих парсерах семантическая обработка более громоздка и не столь мощна, как в парсерах рекурсивного спуска. Можно использовать только выходные атрибуты, а семантические действия возможны лишь в конце продукций. Кроме того, все семантические действия объединяются в глобальный метод `semAction()` и, значит, могут работать только с глобальными переменными, которые нужно сохранять перед рекурсивными нетерминальными символами и восстанавливать после них.

Поэтому семантическая обработка в восходящих парсерах часто сводится к построению (реального) синтаксического дерева, которое затем можно обойти для проверки контекстных условий и выполнения трансляции. Построить синтаксическое дерево с помощью семантических действий в конце продукций легко. Каждый нетерминальный символ предоставляет поддерево в качестве своего выходного атрибута. Затем семантические действия строят новое дерево из поддеревьев, например:

```
A<↑tree3> = B<↑tree1> C<↑tree2> (. tree2 = popAttr(); tree1 = popAttr();
                                   tree3 = new NodeA(tree1, tree2);
                                   pushAttr(tree3); .) .
```

8.6. Обработка ошибок в LR-грамматиках

Синтаксическая ошибка возникает, когда следующий входной символ недопустим в текущем состоянии. Парсер должен сообщить об ошибке и попытаться восстановиться, т. е. синхронизировать свое состояние с остатком входа, чтобы продолжить разбор. Это проще сделать для восходящего синтаксического анализа, нежели для рекурсивного спуска, потому что грамматика неявно закодирована в таблицах разбора, который можно проанализировать в случае ошибки и найти точку синхронизации.

Если произошла синтаксическая ошибка, то стек содержит последовательность состояний $s_0 \ldots s_n$, а остаток входа имеет вид $t_0 \ldots t_m$ #:

$s_0 \ldots s_n \; . \; t_0 \ldots \; t_m$ #

Наша цель – синхронизировать стек с входом, так чтобы состояние s_i в конце стека и следующий входной символ t_j были таковы, что t_j допустим в состоянии s_i; тогда анализ можно будет продолжить:

s0...$\mathbf{s_i}$. $\mathbf{t_j}$... t_m #

Для этого состояния заталкиваются и (или) выталкиваются, а лексемы вставляются и (или) удаляются до тех пор, пока не будет достигнута нужная конфигурация (рис. 8.13). Детали объясняются в следующем разделе.

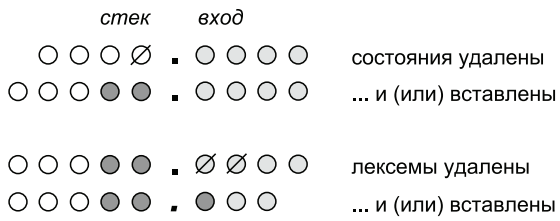


Рис. 8.13 ❖ Вставка/удаление состояния, вставка/удаление лексем

8.6.1. Алгоритм восстановления после ошибки

Алгоритм восстановления после синтаксических ошибок [Röhr80] состоит из трех шагов.

1. **Поиск пути эвакуации**
   Вход $t_0 \ldots t_m$ заменяется виртуальным (фиктивным) входом $v_0 \ldots v_k$, который выводит парсер из состояния ошибки s_n в конечное состояние как можно быстрее (т. е. по «пути эвакуации»).

   $s_0 \ldots sn \; . \; \sout{t_0 \ldots t_m \; \#}$
   $\qquad\qquad\quad v_0 \ldots v_k \; \#$

   На пути эвакуации собираются все лексемы, допустимые в посещаемых состояниях; они называются *якорями*.

2. **Удалить непригодные лексемы**
   Из оригинального входа $t_0 \ldots t_m$ лексемы удаляются, пока не встретится лексема t_j, принадлежащая множеству якорей.

   $s_0 \ldots s_n \; . \; \sout{t_0 \ldots t_{j-1}} \; t_j \ldots t_m \; \#$

3. **Вставка недостающих лексем**
   Теперь парсер снова проводится по пути эвакуации, т. е. из ошибочного состояния s_n с виртуальным входом $v_0 \ldots v_k$ до тех пор, пока не встретится состояние s_i, в котором якорь t_j допустим. Такое состояние должно существовать, потому что якоря собирались вдоль пути эвакуации. Это дает следующую конфигурацию, в которой синтаксический анализ можно продолжить:

   $s_0 \ldots s_i \; . \; t_j \ldots t_m \; \#$

   На этом шаге все потребленные лексемы виртуального входа $v_0 \ldots v_k$ вставляются в остаток входа перед t_j. Это соответствует «исправлению» ошибочного входа. Если бы вставленные лексемы были там с самого начала, то ошибки бы не возникло.

8.6.2. Пример

Прежде чем переходить к реализации алгоритма восстановления, посмотрим на примере, как он работает. Предположим, что парсер представлен АМП, изображенным на рис. 8.14, и должен разобрать вход `a b b #`.

АМП начинает работу в состоянии 1, читает символ `a` и переходит в состояние 2, затем читает символ `b` и переходит в состояние 3. Но следующий входной символ `b` в состояние 3 недопустим, т. е. мы имеем синтаксическую ошибку. Начинается восстановление.

АМП должен найти кратчайший путь эвакуации из состояния 3 в конечное состояние `stop`, но не вправе проходить ребра в обратном направлении. Однако он может перейти в состояние 4 по фиктивному входному символу `c`,

там выполнить свертку a b c в X и таким образом вернуться на три ребра назад в состояние 1, откуда есть переход в состояние 5 по символу X.

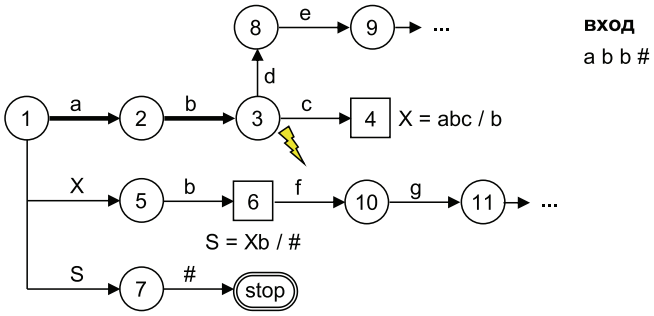


Рис. 8.14 ❖ Обработка ошибок с использованием АМП

Из состояния 5 он может перейти в состояние 6 по фиктивному входному символу b, там свернуть X b в S и таким образом вернуться на два ребра назад в состояние 1, откуда есть переход в состояние 7 по символу S. Из состояния 7 он может достичь конечного состояния по фиктивному входному символу #. Таким образом, виртуальный вход, который выводит парсер из ошибочного состояния 3 в конечное состояние, имеет вид c b #. Вдоль пути эвакуации парсер посещает состояния, показанные на рис. 8.15, и собирает допустимые в них символы – якоря.

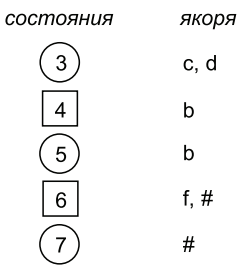


Рис. 8.15 ❖ Сбор якорей вдоль пути эвакуации

Таким образом, множеством якорей, т. е. символов, прочитав которые, парсер может продолжить работу в некотором состоянии на маршруте эвакуации, является множество b, c, d, f, #.

Следующий шаг – удалять символы из остатка входа b # до тех пор, пока не встретится якорь. В нашем примере удалять ничего не нужно, потому что b – уже якорь.

На третьем шаге парсер снова проходит по пути эвакуации из ошибочного состояния 3 с найденным выше виртуальным входом c b #, пока не достигнет состояния, в котором якорь b допустим. В состоянии 3 b еще не допустим, поэтому парсер переходит в состояние 4 по фиктивному входному символу

c и вставляет c во вход в качестве «исправления». Но в состоянии 4 якорь b допустим, и, следовательно, восстановление завершено. Исправленный вход имеет вид

```
a b c b #
```

При таком входе синтаксической ошибки не произошло бы. Замечательно, что этот алгоритм не только обеспечивает хорошее восстановление после ошибки, но даже исправляет ошибочный вход. Совпадает ли это исправление с тем, чего хотел программист, – другой вопрос, но по крайней мере вход получается допустимым.

Однако что напечатать в сообщении об ошибке? Проще всего описать выполненное исправление. Например, если из остатка входа были удалены символы a, b и c, то будет напечатано такое сообщение об ошибке:

```
строка ... столбец ...: "a b c" удалены
```

Если перед остатком входа были вставлены символы x и y, то будет напечатано:

```
строка ... столбец ...: "x y" вставлены
```

Наконец, если символы a, b и c были удалены, а символы x и y вставлены, то сообщение об ошибке будет иметь вид:

```
строка ... столбец ...: "a b c" заменены на "x y"
```

8.6.3. Направляющие символы для поиска пути эвакуации

Как парсер ищет путь эвакуации? Иначе говоря, откуда он знает, что нужно перейти из состояния 3 в состояние 4 и там выполнить свертку? Для этого парсер использует *направляющие символы*, которые указывают ему путь эвакуации в каждом состоянии. На рис. 8.16 показана таблица разбора для грамматики из раздела 8.3. В столбце «направление» показаны направляющие символы в каждом состоянии.

Позже мы покажем, как вычислить эти направляющие символы, но сначала посмотрим, как парсер использует эту таблицу – на этот раз для анализа ошибочного ввода

```
a a a b #
```

Анализ начинается с еще правильной части ввода:

```
 стек   вход       действие
    0   a a a b #  s1
  0 1   a a b #    s4
0 1 4   a b #      -- ошибка!
```

грамматика		a	b	#	S	X	Y	направление
0 S' = S #	0	**s1**	–	–	s3	s2	–	a
1 S = X Y	1	s4	**r3**	–	–	–	–	b
2 S = S X Y	2	–	**s5**	–	–	–	s6	b
3 X = a	3	s1	–	**acc**	–	s7	–	#
4 X = a a b	4	–	**s8**	–	–	–	–	b
5 Y = b	5	r5	s9	**r5**	–	–	–	#
6 Y = b b a	6	r1	–	**r1**	–	–	–	#
	7	–	**s5**	–	–	–	s10	b
	8	–	**r4**	–	–	–	–	b
	9	**s11**	–	–	–	–	–	a
	10	r2	–	**r2**	–	–	–	#
	11	r6	–	**r6**	–	–	–	#

Рис. 8.16 ❖ Таблица разбора с направляющими символами

В состоянии 4 имеет место синтаксическая ошибка, потому что символ `a` в нем недопустим. Начинается восстановление. С помощью направляющих символов парсер переходит из ошибочного состояния 4 по пути эвакуации в конечное состояние. Попутно собираются якоря – символы, допустимые в посещенных состояниях.

```
   стек    направление действие якоря
  0 1 4    b           s8       b
0 1 4 8    b           r4, s2   b
    0 2    b           s5       b
  0 2 5    #           r5, s6   a, b, #
  0 2 6    #           r1, s3   a, #
    0 3    #           acc a,   #
```

В состоянии 4 направляющим символом является `b`, а действие в таблице для `b` – это `s8`. Кроме того, `b` – единственный терминальный символ, допустимый в состоянии 4, поэтому он добавляется в множество якорей. Парсер переходит в состояние 8, где направляющим символом снова является `b`. Соответствующее действие – `r4`, т. е. парсер производит свертку согласно продукции 4 (`X = a a b`). При этом из стека выталкивается 3 состояния, и парсер возвращается в состояние 0, откуда по символу `X` перейдет в состояние 2 (`s2`). В состоянии 8 единственный допустимый терминальный символ – это `b`, поэтому `b` добавляется в множество якорей. И таким образом путешествие по пути эвакуации продолжается, пока в состоянии 3 не будет прочитан символ `#` и, стало быть, достигнуто конечное состояние. Попутно были собраны якоря `a`, `b` и `#`.

Теперь из остатка входа `a b #` символы должны удаляться, пока не встретится якорь, но т. к. `a` уже является якорем, то удалять нечего.

На последнем шаге парсер снова проходит по пути эвакуации из ошибочного состояния 4, используя направляющие символы, пока не достигнет состояния, в котором якорь `a` допустим. При каждом действии `shift` по направляющему символу этот символ вставляется во вход в качестве исправления.

```
   стек    направление действие вставлено
  0 1 4    b           s8       b // только переносы по направляющему символу приводят
                                   // к вставке
0 1 4 8    b           r4, s2
    0 2    b           s5       b
  0 2 5
```

В состоянии 5 якорь a допустим, и анализ можно продолжить:

```
    стек   вход     действие
   0 2 5   a b #    r5, s6
   0 2 6   a b #    r1, s3
     0 3   a b #    s1
   0 3 1   b #      r3, s7
   0 3 7   b #      s5
 0 3 7 5   #        r5, s10
0 3 7 10   #        r2, s3
     0 3   #        acc
```

Анализ успешно завершен. Исправленный вход имеет вид

```
a a b b a b #
```

а сообщение об ошибке –

```
строка ... столбец ...: "b b" вставлены
```

8.6.4. Нахождение направляющих символов

Как вычислить направляющие символы для каждого состояния? Это очень просто и является побочным продуктом построения таблицы разбора. Но сначала мы должны переупорядочить продукции грамматики, так чтобы первой продукцией каждого нетерминального символа была кратчайшая. Кроме того, она не должна быть леворекурсивной. Ниже показана наша демонстрационная грамматика с правильным порядком продукций:

```
0   S' = S #
1   S = X Y
2   S = S X Y
3   X = a
4   X = a a b
5   Y = b
6   Y = b b a
```

Теперь по этой грамматике мы можем создать LALR(1)-таблицу. Одна при построении замыкания состояния требуется осторожность: новые элементы следует вставлять сразу после элемента, инициировавшего их вставку. Например, видя состояние

```
S' = . S #
S = . X Y     / #
S = . S X Y   / #
```

при добавлении продукций X в качестве новых элементов, мы должны вставить их сразу же после элемента, в котором точка находится перед X:

```
S' = . S #
S = . X Y    / #
X = . a      / b
X = . a a b  / b
S = . S X Y  / #
```

Если теперь мы применим алгоритм построения таблицы, то действие по первому терминальному символу в каждом состоянии будет определять направляющий символ (самый правый столбец).

```
0   S' = . S #          shift  a    1             a
    S = . X Y    / #a   shift  X    2
    X = . a      / b    shift  S    3
    X = . a a b  / b
    S = . S X Y  / #a
1   X = a .      / b    red    b    3 (X = a)     b
    X = a . a b  / b    shift  a    4
2   S = X . Y    / #a   shift  b    5             b
    Y = . b      / #a   shift  Y    6
    Y = . b b a  / #a
3   S' = S . #          acc    #                  #
    S = S . X Y  / #a   shift  a    1 (!)
    X = . a      / b    shift  X    7
    X = . a a b  / b
4   X = a a . b  / b    shift  b    8             b
5   Y = b .      / #a   red    #,a  5 (Y = b)     #
    Y = b . b a  / #a   shift  b    9
6   X = X Y .    / #a   red    #,a  1 (S = X Y)   #
7   S = S X . Y  / #a   shift  b    5 (!)         b
    Y = . b      / #a   shift  Y    10
    Y = . b b a  / #a
8   X = a a b .  / b    red    b    4 (X = a a b) b
9   Y = b b . a  / #a   shift  a    11 a
10  S = S X Y .  / #a   red    #,a  2 (S = S X Y) #
11  Y = b b a .  / #a   red    #,a  6 (Y = b b a) #
```

Если действием по первому терминальному символу в некотором состоянии является reduce с несколькими управляющими символами (как в состоянии 5), то в качестве направляющего символа может быть выбран любой из них.

8.6.5. Оценка

Описанный выше метод обработки ошибок не тривиален, но является поразительно мощным. Он не замедляет разбор корректного входа, а начинает работать, только когда возникает синтаксическая ошибка. После этого

для восстановления приходится приложить некоторые усилия, но в случае ошибки они оправданы.

Восстановление гарантированно завершается. Поскольку последний переход на пути эвакуации производится по символу конца файла #, то этот символ всегда является якорем. В худшем случае весь остаток входа вплоть до # придется пропустить, но синхронизация успешно завершится на символе #.

Описанная здесь техника не только приводит к восстановлению после синтаксических ошибок, но и «исправляет» такие ошибки, хотя исправление не всегда совпадает с намерением программиста. Однако по крайней мере исправленный вход позволяет осуществить непротиворечивую семантическую обработку, что в случае рекурсивного спуска бывает не всегда.

И последнее, но оттого не менее важное: этот метод обработки ошибок дает хорошие сообщения об ошибках, которые просто описывают исправление ошибочного входа, и в большинстве случаев такое объяснение всех устраивает.

В общем и целом механизм обработки ошибок при восходящем анализе более мощный, чем при рекурсивном спуске, благодаря тому что грамматика представлена в табличной форме и может быть проанализирована в случае ошибки для сбора якорей и содержательного восстановления.

8.7. Упражнения

1. *Построение таблицы (1).* Для следующей грамматики создайте таблицу разбора для восходящего синтаксического анализа.

```
A = A B c.
A = a.
B = b.
```

   Является ли эта грамматика LR(0), LR(1) или LALR(1)? Объясните свой ответ.

2. *Построение таблицы (2).* Для следующей грамматики создайте таблицу разбора для восходящего синтаксического анализа и вычислите направляющие символы для восстановления после ошибки.

```
S = a.
S = S X.
X = c.
X = b X b.
```

   Является ли эта грамматика LR(0), LR(1) или LALR(1)? Объясните свой ответ.

3. *Построение таблицы (3).* Следующая грамматика обладает свойством LR(1), но не LALR(1):

```
S = d X b.
S = d Y a.
S = X a.
S = Y b.
X = c.
Y = c.
```

Создайте таблицы разбора для LR(1)- и LALR(1)-анализов и покажите, в каком месте возникает конфликт. Кроме того, нарисуйте АМП для LR(1)- и LALR(1)-анализов и объясните, почему конфликт имеется в LALR(1)-таблице, но отсутствует в LR(1)-таблице.

4. *Обработка ошибок (1).* Пусть дана грамматика на рис. 8.16 с соответствующими таблицей разбора и направляющими символами. Продемонстрируйте анализ ошибочного входа

   ```
   a b b b a #
   ```

   включающий восстановление после ошибки, по образцу в разделе 8.6.

5. *Обработка ошибок (2).* Покажите, что механизм восстановления после ошибок работает и тогда, когда действия `shift` и `reduce` объединены в действия `shiftred`. Как и в предыдущем примере, используйте грамматику на рис. 8.16 и ее сжатую форму на рис. 8.10, в которой действия `shift` и `reduce` объединены в действия `shiftred`. Воспользовавшись этим фактом, продемонстрируйте анализ ошибочного входа

   ```
   b a b #
   ```

   включающий восстановление после ошибки.

Приложение A

Язык MicroJava

В этом приложении описаны лексическая структура, синтаксис и семантика языка MicroJava, для которого в этой книге написан компилятор. В частности, специфицированы контекстные условия для MicroJava, т. е. требования, которые компилятор должен проверять, чтобы гарантировать семантическую правильность программы.

Лексическая структура

Классы литер

```
letter      = 'a' .. 'z' | 'A' .. 'Z'
digit       = '0' .. '9'
char        = ANY – '\''
```

Классы терминальных символов

```
ident       = letter {letter | digit | '_'}
number      = digit {digit}
charCon     = "'" char "'" // может также содержать '\r', '\n', '\t'
```

Ключевые слова

```
program class
if else while read print return break
void final new
```

Операторы

```
+   -   *   /   %   ++   --   ==   !=   >   >=   <   <=
&&   ||   (   )   [   ]   {   }   =   ;   ,
```

Комментарии

```
// … до конца строки
```

Пустые символы

```
пробелы, '\r', '\n', '\t'
```

Синтаксис

```
Program    = "program" ident {ConstDecl | VarDecl | ClassDecl} "{" {MethodDecl} "}".
ConstDecl  = "final" Type ident "=" (number | charCon) ";".
VarDecl    = Type ident {"," ident } ";".
ClassDecl  = "class" ident "{" {VarDecl} "}".
MethodDecl = (Type | "void") ident "(" [FormPars] ")" {VarDecl} Block.
FormPars   = Type ident {"," Type ident}.
Type       = ident ["[" "]"].
```

```
Block      = "{" {Statement} "}".
Statement  = Designator ("=" Expr | ActPars | "++" | "--") ";"
           | "if" "(" Condition ")" Statement ["else" Statement]
           | "while" "(" Condition ")" Statement
           | "break" ";"
           | "return" [Expr] ";"
           | "read" "(" Designator ")" ";"
           | "print" "(" Expr ["," number] ")" ";"
           | Block
           | ";".
ActPars    = "(" [ Expr {"," Expr} ] ")".
Condition  = CondTerm {"||" CondTerm}.
CondTerm   = CondFactor {"&&" CondFactor}.
CondFactor = Expr RelOp Expr.
RelOp      = "==" | "!=" | ">" | ">=" | "<" | "<=".
Expr       = ["-"] Term {AddOp Term}.
Term       = Factor {MulOp Factor}.
Factor     = Designator [ActPars]
           | number
           | charCon
           | "new" ident [ "[" Expr "]" ]
           | "(" Expr ")".
Designator = ident {"." ident | "[" Expr "]"}.
AddOp      = "+" | "-".
MulOp      = "*" | "/" | "%".
```

Семантика

Все последующие термины, для которых имеется определение или которые рассматриваются как определение, подчеркнуты с целью отметить их специальное значение.

Ссылочные типы

Массивы и классы являются ссылочными типами.

Типы констант

- Типом числовой константы является `int`.
- Типом литерной константы является `char`.

Равенство типов

Два типа считаются равными:

- если они обозначаются одним и тем же именем типа или
- если оба типа являются массивами и типы их элементов равны.

Совместимость типов

Два типа совместимы:

- если они равны или
- если один из них является ссылочным, а другой – типом `null`.

Совместимость по присваиванию

Тип *src* совместим по присваиванию с типом *dst*:

- если *src* и *dst* равны или
- если *dst* – ссылочный тип, а *src* – тип `null`.

Предопределенные имена

`int`	стандартный тип
`char`	стандартный тип
`null`	значение `null` переменной класса или массива
`chr`	стандартный метод; `chr(i)` преобразует выражение `i` типа `int` в значение типа `char`
`ord`	стандартный метод; `ord(ch)` преобразует значение `ch` типа `char` в значение типа `int`
`len`	стандартный метод; `len(a)` возвращает число элементов в массиве `a`

Область видимости

Область видимости – область программы, метода или класса. Она простирается от точки после имени программы, метода или класса до закрывающей фигурной скобки. Вложенные области видимости из нее исключаются. Существует (искусственная) самая внешняя область видимости (универсальная), которая содержит главную программу и все предопределенные имена. Объявление имени во внутренней области видимости скрывает все объявления этого имени в объемлющей области видимости.

Примечание

- Косвенная рекурсия не допускается, потому что каждое имя должно быть объявлено до его использования, что в случае косвенной рекурсии невозможно.
- Предопределенное имя (например, `int` или `char`) разрешается переопределять новым объявлением того же имени во внутренней области видимости (хотя это не рекомендуется).

Контекстные условия

Общие контекстные условия

- Каждое имя должно быть объявлено до его использования.
- Никакое имя не может быть объявлено более одного раза в одной и той же области видимости.
- Программа должна содержать метод `main()`, который должен быть объявлен как `void` и не должен иметь параметров.

Контекстные условия для стандартных методов

`chr(e)`	`e` должно быть выражением типа `int`
`ord(c)`	`c` должно иметь тип `char`
`len(a)`	`a` должно быть массивом

Контекстные условия для продукций

Ниже описаны контекстные условия, которые должны проверяться для различных продукций грамматики MicroJava. Заметим, что существуют продукции, для которых нет контекстных условий.

```
Program = "program" ident {ConstDecl | VarDecl | ClassDecl} "{" {MethodDecl} "}".
ConstDecl = "final" Type ident "=" (number | charCon) ";".
```

- Тип `number` или `charCon` должен быть равен типу `Type`.

```
VarDecl = Type ident {"," ident} ";". ClassDecl = "class" ident "{" {VarDecl} "}".
MethodDecl = (Type | "void") ident "(" [FormPars] ")" {VarDecl} Block.
```

- `Type` должен быть `int` или `char`.
- Метод `main()` не должен иметь параметров и должен быть объявлен как `void`.
- Функциональные методы должны возвращать управление посредством предложения `return` (проверяется во время выполнения).

```
FormPars = Type ident {"," Type ident}. Type = ident ["[" "]"].
```

- `ident` должен обозначать тип.

```
Block = "{" {Statement} "}". Statement = Designator "= "Expr" ;".
```

- `Designator` должен обозначать переменную, элемент массива или поле объекта.
- Тип `Expr` должен быть совместим по присваиванию с типом `Designator`.

```
Statement = Designator ("++" | "--") ";".
```

- `Designator` должен обозначать переменную, элемент массива или поле объекта.
- `Designator` должен быть типа `int`.

```
Statement = Designator ActPars ";".
```

- `Designator` должен обозначать метод.

```
Statement = "return" [Expr] ";".
```

- Тип `Expr` должен быть совместим по присваиванию с функциональным типом текущего метода.
- Если текущий метод является функциональным, то опускать `Expr` нельзя.
- Если `Expr` опущено, то текущий метод должен быть объявлен как `void`.

```
Statement = "break" ";".
```

- Предложение `break` не должно встречаться вне цикла.

```
Statement = "read" "(" Designator ")" ";".
```

- `Designator` должен обозначать переменную, элемент массива или поле объекта.
- `Designator` должен быть типа `int` или `char`.

```
Statement = "print" "(" Expr ["," number] ")" ";".
```

- `Expr` должно быть типа `int` или `char`.

```
Statement = "if" "(" Condition ")" Statement ["else" Statement]
          | "while" "(" Condition ")" Statement
          | Block
          | ";".
ActPars = "(" [ Expr {"," Expr} ] ")".
```

- Количество фактических и формальных параметров должно быть одинаково.
- Тип каждого фактического параметра должен быть совместим по присваиванию с типом соответствующего формального параметра.

```
Condition = CondTerm {"||" CondTerm}.
CondTerm = CondFactor {"&&" CondFactor}.
CondFactor = Expr RelOp Expr.
```

- Типы обоих выражений должны быть совместимы.
- Классы и массивы достаточно проверять только на равенство или неравенство.

```
Expr = Term. Expr = "-" Term.
```

- `Term` должен быть типа `int`.

```
Expr0 = Expr1 AddOp Term.
```

- $\texttt{Expr}_1$ и `Term` должны быть типа `int`.

```
Term = Factor.
Term0 = Term1 MulOp Factor.
```

- $\texttt{Term}_1$ и `Factor` должны быть типа `int`.

```
Factor = Designator | number | charCon | "(" Expr ")".
Factor = Designator ActPars.
```

- `Designator` должен обозначать метод.
- Тип метода, обозначаемого `Designator`, не должен быть `void`.

```
Factor = "new" ident.
```

- `ident` должен обозначать класс.

```
Factor = "new" ident "[" Expr "]".
```

- `ident` должен обозначать тип.
- Тип `Expr` должен быть `int`.

```
Designator = ident.
Designator0 = Designator1 "." ident.
```

- Типом $\texttt{Designator}_1$ должен быть класс.
- `ident` должен обозначать поле $\texttt{Designator}_1$.

```
Designator0 = Designator1 "[" Expr "]".
```

- Типом $Designator_1$ должен быть массив.
- Типом Expr должен быть int.

```
RelOp = "==" | "!=" | ">" | ">=" | "<" | "<=".
AddOp = "+" | "-".
MulOp = "*" | "/" | "%".
```

Ограничения реализации

- Не должно быть больше 128 локальных переменных.
- Не должно быть больше 32 768 глобальных переменных.
- В классе не должно быть больше 32 768 полей.

Приложение B

Компилятор MicroJava

В этом приложении описана архитектура компилятора MicroJava и перечислены интерфейсы его классов.

Общее описание

В этой книге описываются методы реализации компилятора для языка программирования MicroJava. Читатель может скачать с сопроводительного сайта [Download] и изучить полный исходный код этого компилятора. В этом приложении приводятся общий обзор его архитектуры и интерфейсы его классов.

Компилятор MicroJava состоит из 12 Java-классов, сгруппированных в три пакета (рис. B.1).

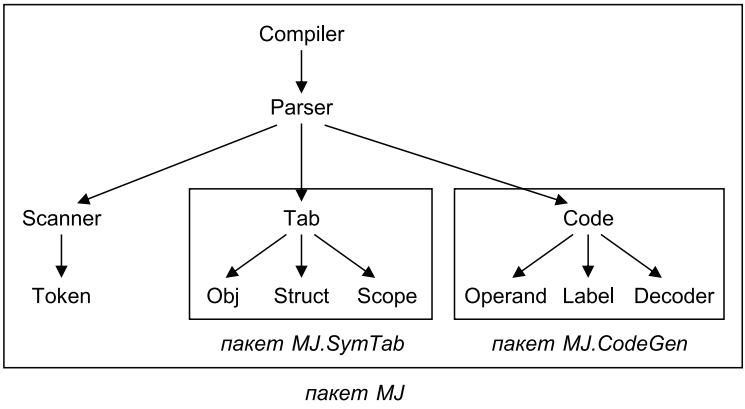


Рис. B.1 ❖ Архитектура компилятора MicroJava

`Tab` – это таблица символов с классами для записей объектов (`Obj`), записей структур (`Struct`) и записей областей видимости (`Scope`). `Code` – кодогенератор с классами для операндов генерирования кода (`Operand`), меток перехода (`La-`

bel) и декодирования сгенерированного байт-кода (Decoder). Парсер управляет компиляцией и вызывает сканер, который поставляет терминальные символы в виде объектов класса Token. Парсер еще вызывает методы таблицы символов, которые осуществляют вставки и поиск имен в областях видимости, а также методы кодогенератора, которые порождают целевой код.

Интерфейсы классов компилятора

Для справки ниже описаны интерфейсы классов компилятора. Реализацию см. в исходном коде.

Compiler

Класс Compiler – главный класс компилятора MicroJava. Его метод main() читает имя исходного файла, подлежащего компиляции, из командной строки, инициализирует им сканер и вызывает парсер. Компилятор запускается командой

```
java MJ.Compiler sourceFileName.mj
```

Если ошибок не найдено, то компилятор создает файл *sourceFileName*.obj, который может быть выполнен ВМ MicroJava (имеется на сопроводительном сайте):

```
java MJ.Run sourceFileName.obj [-debug]
```

Если задан параметр -debug, то интерпретацию байт-кода и содержимое стека выражений можно протрассировать.

Parser

Парсер управляет компиляцией и вызывает методы сканера, таблицы символов и кодогенератора. Его интерфейс имеет вид:

```
public class Parser {
   public static int errors;                    // счетчик ошибок
   public static void error (String msg) {...} // печатает сообщение об ошибке, включающее
                                                // номера строки и столбца
   public static void parse() {...}             // начинает разбор
}
```

Scanner

Сканер читает исходную программу и предоставляет парсеру ее терминальные символы в виде лексем. Комментарии и пустые символы отфильтровываются. Интерфейс сканера имеет вид:

```
public class Scanner {
   public static void init (Reader r) {...}  // инициализирует сканер потоком ввода
   public static Token next() {...}          // возвращает следующую лексему из потка ввода
}
```

Token

Лексемы предоставляются сканером и описывают вид, значение и позицию терминальных символов в исходной программе.

```
public class Token {
   public int kind;    // код лексемы
   public int line;    // номер строки
   public int col;     // номер столбца
   public String val;  // значение лексемы
   public int numVal;  // числовое значение лексемы (для number и charCon)
}
```

Для компилятора MicroJava определены следующие коды лексем:

```
static final int
   //--- лексема ошибки
   none = 0,
   //--- классы лексем
   ident = 1,     number = 2,      charCon = 3,
   //--- операторы и специальные литеры
   plus = 4,      minus = 5,       times = 6,      slash = 7,      rem = 8,
   eql = 9,       neq = 10,        lss = 11,       leq = 12,       gtr = 13,
   geq = 14,      and = 15,        or = 16,        assign = 17,    pplus = 18,
   mminus = 19,   semicolon = 20,  comma = 21,     period = 22,    lpar = 23,
   rpar = 24,     lbrack = 25,     rbrack = 26,    lbrace = 27,    rbrace = 28,
   //--- ключевые слова
   break_ = 29,   class_ = 30,     else_ = 31,     final_ = 32,    if_ = 33,
   new_ = 34,     print_ = 35,     program_ = 36,   read_ = 37,    return_ = 38,
   void_ = 39,    while_ = 40,
   //--- лексема конца файла
   eof = 41;
```

Tab

Класс `Tab` реализует таблицу символов в компиляторе MicroJava. Метод `insert()` включает объявленное имя в текущую область видимости, `find()` ищет имя во всех открытых областях видимости, а `findField()` ищет имя поля в классе.

```
public class Tab {
   public static Scope curScope;           // текущая область видимости
   public static int curLevel;             // уровень вложенности текущей области видимости
   public static Struct intType, charType, nullType, noType; // предопределенные типы
   public static Obj chrObj, ordObj, lenObj, noObj;          // предопределенные объекты
   public static void openScope() {...}    // открывает новую область видимости
   public static void closeScope() {...}   // закрывает текущую область видимости
   public static Obj insert (int kind, String name, Struct type) {...}
   public static Obj find (String name) {...}
   public static Obj findField (String name, Struct type) {...}
}
```

Obj

В классе `Obj` хранится информация об объявленном имени.

```
public class Obj {
   public static final int Con = 0, Var = 1, Type = 2, Meth = 3, Prog = 4; // виды объектов
   public int kind;       // Con, Var, Type, Meth, Prog
   public String name;    // имя объекта
   public Struct type;    // тип объекта
   public Obj next;       // следующий объект в этой области видимости
   public int val;        // для Con: значение константы
   public int adr;        // для Var, Meth: адрес
   public int level;      // для Var: уровень объявления
   public int nPars;      // для Meth: число параметров
   public Obj locals;     // для Meth: параметры и локальные объекты
   public Obj (int kind, String name, Struct type) {...}
}
```

Struct

В классе `Struct` хранится информация о виде и структуре типа. Для каждого типа имеется только одна запись типа `Struct` во всей таблице символов.

```
public class Struct {
   public static final int None = 0, Int = 1, Char = 2, Arr = 3, Class = 4; // виды структур
   public int kind;          // None, Int, Char, Arr, Class
   public Struct elemType;   // для Arr: тип элемента
   public int nFields;       // для Class: число полей
   public Obj fields;        // для Class: список полей
   public Struct (int kind) {...}
   public Struct (int kind, Struct elemType) {...}
   public boolean isRefType() {...}                  // true для массивов и классов
   public boolean equals (Struct other) {...}        // true, если "this" и other равны
   public boolean compatibleWith (Struct other) {...} // true, если "this" сравним с other
   public boolean assignableTo (Struct dest) {...}   // true, если "this" допускает
                                                     // присваивание dest
}
```

Scope

Класс `Scope` определяет начальную запись области видимости, к которой присоединяются записи объектов этой области.

```
public class Scope {
   public Scope outer;     // на следующую объемлющую область видимости
   public Obj locals;      // на локальные объекты этой области видимости
   public int nVars;       // число переменных в этой области видимости
}
```

Code

Класс `Code` реализует кодогенератор компилятора MicroJava. В нем определены команды для генерирования команд и переходов.

```
public class Code {
  public static final int // instruction codes
     load = 1,       load0 = 2,       load1 = 3,       load2 = 4,       load3 = 5,
     store = 6,      store0 = 7,      store1 = 8,      store2 = 9,      store3 = 10,
     getstatic = 11, putstatic = 12,  getfield = 13,   putfield = 14,   const0 = 15,
     const1 = 16,    const2 = 17,     const3 = 18,     const4 = 19,     const5 = 20,
     const_m1 = 21,  const_ = 22,     add = 23,        sub = 24,        mul = 25,
     div = 26,       rem = 27,        neg = 28,        shl = 29,        shr = 30,
     new_ = 31,      newarray = 32,   aload = 33,      astore = 34,     baload = 35,
     bastore = 36,   arraylength = 37, pop = 38,       jmp = 39,        jeq = 40,
     jne = 41,       jlt = 42,        jle = 43,        jgt = 44,        jge = 45,
     call = 46,      return_ = 47,    enter = 48,      exit = 49,       read = 50,
     print = 51,     bread = 52,      bprint = 53,     trap = 54;
  public static final int eq = 0, ne = 1, lt = 2,
                          le = 3, gt = 4, ge = 5; // операторы сравнения
  public static int[] inverse = {ne, eq, ge, gt, le, lt};
  public static int pc;       // следующий свободный адрес в буфере кода
  public static int mainPc;   // адрес метода main() method (устанавливается парсером)
  public static int dataSize; // длина области глобальных данных в словах
                              // (устанавливается парсером)
  public static void put (int x) {...}  // добавляет 1 байт в буфер кода
  public static void put2 (int x) {...} // добавляет 2 байта в буфер кода
  public static void put2 (int pos, int x) {...} // корректирует 2 байта в позиции pos
                                                 // буфера кода
  public static void put4 (int x) {...} // добавляет 4 байта в буфер кода
  public static void load (Operand x) {...}     // загружает x в EStack
  public static void assignTo (Operand x) {...} // присваивает x элемент на вершине EStack
  public static void callMethod (Operand m) {...}    // вызывает метод m
  public static void inc (Operand x, int val) {...}  // увеличивает x на val (+/- 1)
  public static void jump (Label lab) {...}          // безусловный переход
  public static void tJump (int op, Label lab) {...} // переход, если true в зависимости
                                                     // от op
  public static void fJump (int op, Label lab) {...} // переход, если false в зависимости
                                                     // от op
  public static void write (OutputStream s) {...}  // записывает буфер кода в объектный
                                                   // файл
}
```

Operand

Кодогенератор использует дескрипторы операндов описанного ниже типа для хранения вида и местоположения операндов в процессе генерирования кода.

```
public class Operand {
   public static final int // виды операндов
      Con = 0, Local = 1, Static = 2, Stack = 3, Fld = 4, Elem = 5, Meth = 6, Cond = 7;
   public int kind;      // Con, Local, Static, Stack, Fld, Elem, Meth, Cond
   public Struct type;   // тип операнда
   public int val;       // для Con: значение
   public int adr;       // для Local, Static, Fld, Meth: адрес
   public Obj obj;       // для Meth: объект метода
   public int op;        // для Cond: последний оператор сравнения
```

```
   public Label tLabel;  // для Cond: цель перехода, если true
   public Label fLabel;  // для Cond: цель перехода, если false
   public Operand (Obj obj) {...}
   public Operand (int val) {...}
   public Operand (int kind, int val, Struct type) {...}
}
```

Label

Класс `Label` управляет метками переходов. Адреса в командах перехода на еще не определенную метку автоматически корректируются в момент определения метки.

```
public class Label {
   public Label() {...}       // создает еще не определенную метку
   public void here() {...}   // определяет метку в текущей позиции pc
   public void putAdr() {...} // записывает длину перехода на эту метку в код
}
```

Decoder

Для проверки правильности сгенерированного байт-кода можно использовать декодер, который выводит команды в понятной форме.

```
public class Decoder {
   public static void decode (byte[] code, int beg, int end) {...}
}
```

Метод `decode()` декодирует содержимое буфера кода в диапазоне `[beg .. end]` и выводит его на консоль. Он вызывается кодогенератором в конце генерирования кода.

Имеется также автономная программа `Decode`, которую можно использовать для декодирования объектного файла MicroJava. Она вызывается следующим образом:

```
java MJ.Decode fileName.obj
```

Вывод направляется на консоль.

Литература

[AGH00] Arnold, K.; Gosling, J.; Holmes, D.: *The Java Programming Language*. Addison- Wesley, 2000.

[ALSU06] Aho, A. V.; Lam, M.; Sethi, R.; Ullman, J. D.: *Compilers: Principles, Techniques, and Tools*. Pearson, 2nd ed., 2006.

[Appe02] Appel, A. W.: *Modern Compiler Implementation in Java*. 2nd ed., Cambridge University Press, 2002.

[Back56] Backus, J. et al.: *Programmer's Reference Manual Fortran*. IBM, 1956. *http://bitsav-ers.informatik.uni-stuttgart.de/pdf/ibm/704/704_Fortran-ProgRefMan_Oct56.pdf*.

[Coco] *The Compiler Generator Coco/R – User Manual*. *https://ssw.jku.at/Coco/*.

[Coop22] Cooper, K. D.: *Engineering a Compiler*. 3rd ed., Morgan Kaufmann, 2022.

[Cope20] Copeland, T.: *Generating Parsers with JavaCC*, 2020. *https://javacc.github.io/javacc/*.

[Download] Дополнительные материалы к книге (слайды, исходный код, примеры решений). *https://ssw.jku.at/CompilerBook/*.

[FCL09] Fisher, C. N.; Cytron, R. K.; LeBlanc, R. J.: *Crafting a Compiler*. Pearson Education, 2009.

[GTWW77] Goguen, J. A.; Thatcher, J. W.; Wagner, E. W.; Wright, J. B.: *Initial Algebra Semantics and Continuous Algebras*. Journal of the ACM, 24 (1): 68–95, 1977.

[HOPL-I] Wexelblat, R. L.: *History of Programming Languages*. Academic Press, 2014, First ACM SIGPLAN conference on History of Programming Languages, 1978.

[HOPL-II] Second ACM SIGPLAN conference on History of Programming Languages, 1993. *https://dl.acm.org/doi/proceedings/10.1145/154766*.

[HOPL-III] Third ACM SIGPLAN conference on History of Programming Languages, 2007. *https://dl.acm.org/doi/proceedings/10.1145/1238844*.

[JavaDoc] *Java documentation tool*. *https://www.oracle.com/technical-resources/articles/java/javadoc- tool.html*.

[JW77] Jensen, K.; Wirth, N.: Pascal User Manual and Report. Springer, 1975.

[Knut68] Knuth, D. E.: Semantics of Context-free Languages. Mathematical Systems Theory, vol. 2, no. 2, 1968, 127–145.

[LMB92] Levine, J. R.; Mason, T.; Brown, D.: lex & yacc. O'Reilly, 1992.

[Much97] Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997.

[Naur64] Naur, P.: Revised Report on the Algorithmic Language Algol60. *http://algol60.org/reports/algol60_rr.pdf*.

[Oder08] Odersky, M.: Programming in Scala. artima, 2008.

[Röhr80] Röhrich, J.: Methods for the Automatic Construction of Error-correcting Parsers. Acta Informatica 13, 115–139, 1980.

[Schm86] Schmidt, D. A.: Denotational Semantics: A Methodology for Language Development. William C. Brown Publishers, 1986.

[Stro85] Stroustrup, B.: The C++ Programming Language. Addison-Wesley, 1985.

[Parr13] Parr, T.: The Definitive ANTLR 4 Reference. Pragmatic Book shelf, 2013. see also: *https://github.com/antlr/antlr4/blob/master/doc/index.md*.

[Wirt77] Wirth, N.: What can we do about the unnecessary diversity of notation for syntactic definitions?, Communications of the ACM, vol. 20, no. 11, November 1977, 822–823.

Предметный указатель

К

Л

М

Н

О

П

Р

С

Т

У

Книги издательства «ДМК Пресс»
можно купить оптом и в розницу на складе издательства по адресу:
Москва, ул. Электродная, д. 2, стр. 12, офис 7, тел. **+7 (499) 322-19-38**,
а также заказать на сайте **www.dmkpress.com**
с доставкой в любой регион РФ

Ханспетер Мёссенбёк

Конструирование компиляторов

Главный редактор *Мовчан Д. А.*
Зам. главного редактора *Яценков В. С.*
editor@dmkpress.com
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 22,43. Тираж 100 экз.

Веб-сайт издательства: **www.dmkpress.com**

«Синтаксический анализ и другие методы, описанные в книге, полезен не только для обработки языков программирования, его можно применить и ко многим другим задачам, включающим систематическую обработку наборов данных или текстов, и к постоянно расширяющемуся множеству других структурированных входов».

Заслуженный профессор, д-р Никлаус Вирт

В книге рассматриваются основы конструирования компиляторов — от лексического и синтаксического анализа до семантической обработки и генерирования кода. В качестве сквозного примера описан и реализован компилятор простого Java-подобного языка программирования (MicroJava). Также приведены процессы трансляции с использованием атрибутных грамматик и применение генератора компиляторов для автоматического порождения основных частей компилятора. В части синтаксического анализа основное внимание уделено нисходящему разбору методом рекурсивного списка.

На сайте издательства www.dmkpress.com размещен полный комплект слайдов к вводному курсу по компиляторам, решения более 70 упражнений, полный код компилятора MicroJava, а также исходный и исполняемый код генератора компиляторов Coco/R.

Издание ориентировано на студентов и преподавателей а также на программистов-практиков, которые хотят применять базовые методы компиляции в повседневной работе.

Ханспитер Мёссенбёк – профессор, доктор наук, руководитель Института системного программного обеспечения, руководитель Комитета по учебным программам в области компьютерных наук.

Научные интересы включают языки программирования, построение компиляторов и автоматическую разработку программного обеспечения.

ДМК издательство
www.дмк.рф

ISBN 978-5-93700-391-1