Сергей Свердлов

Конструирование компиляторов

Учебное пособие

LAP LAMBERT Academic Publishing

В книге подробно рассматривается разработка компилятора языка программирования высокого уровня. Обсуждаются все этапы реализации от спецификации языка до формирования машинного кода. Приводится исходный код компилятора на нескольких языках программирования. Даются необходимые для создания компилятора сведения по теории формальных языков и грамматик. Изложение сопровождается многочисленными примерами. Учебное пособие адресуется студентам вузов, специализирующимся по компьютерным технологиям и всем, кто интересуется программированием.

Сергей Залманович Свердлов — профессор кафедры прикладной математики Вологодского государственного университета.

978-3-659-71665-2

Сергей Свердлов

Конструирование компиляторов

Учебное пособие

LAP LAMBERT Academic Publishing

Impressum / Выходные данные
Bibliografische Information der Deutschen Nationalbibliothek: Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Verlag / Издатель:
LAP LAMBERT Academic Publishing
ist ein Imprint der / является торговой маркой
OmniScriptum GmbH & Co. KG
Heinrich-Böcking-Str. 6-8, 66121 Saarbrücken, Deutschland / Германия
Email / электронная почта: info@lap-publishing.com

Herstellung: siehe letzte Seite /
Напечатано: см. последнюю страницу
ISBN: 978-3-659-71665-2

ОГЛАВЛЕНИЕ

ГЛАВА 1. ЯЗЫКИ ПРОГРАММИРОВАНИЯ И ИХ РЕАЛИЗАЦИЯ

С начала 50-х годов XX века создаются и развиваются языки программирования. Чтобы язык программирования можно было реально применять — писать программы на этом языке и исполнять их на компьютере, язык должен быть *реализован*.

Язык и его реализация

Реализацией языка программирования называют создание комплекса программ, обеспечивающих работу на этом языке. Такой набор программ называется *системой программирования*.

Основу каждой системы программирования составляет *транслятор*. Это программа, переводящая текст на языке программирования в форму, пригодную для исполнения (на другой язык). Такой формой обычно являются машинные команды, которые могут непосредственно исполняться компьютером. Совокупность машинных команд данного компьютера (процессора) образует его *систему команд* или *машинный язык*. Программу, которую обрабатывает транслятор, называют *исходной программой*, а язык, на котором записывается исходная программа — *входным языком* этого транслятора.

Компилятор, интерпретатор, конвертор

Различают несколько видов трансляторов: компиляторы, интерпретаторы, конверторы (рис. 1.1).

Компилятор, обрабатывая исходную программу, создает эквивалентную программу на машинном языке, которая называется также *объектной программой* или *объектным кодом*. Объектный код, как правило, записывается в файл, но не обязательно представляет собой готовую к исполнению программу. Для программ, состоящих из многих модулей, может образовываться много объектных файлов. Объектные файлы объединяются в ис-

полняемый модуль с помощью специальной *программы-компоновщика*, которая входит в состав системы программирования. Возможен также вариант, когда модули не объединяются заранее в единую программу, а загружаются в память при выполнении программы по мере необходимости.

Интерпретатор, распознавая, как и компилятор, исходную программу, не формирует машинный код в явном виде. Для каждой операции, которая может потребоваться при исполнении исходной программы, в программе-интерпретаторе заранее заготовлена машинная команда или команды. «Узнав» очередную операцию в исходной программе, интерпретатор выполняет соответствующие команды, потом — следующие, и так всю программу. Интерпретатор — это переводчик и исполнитель исходной программы.

Различие между интерпретаторами и компиляторами можно проиллюстрировать такой аналогией. Чтобы пользоваться каким-либо англоязычным документом, мы можем поступить поразному. Можно один раз перевести документ на русский, записать этот перевод и потом пользоваться им сколько угодно раз. Это ситуация, аналогичная компиляции. Заметьте, что после выполнения перевода переводчик (компилятор) больше не нужен.

Интерпретация же подобна чтению и переводу «с листа». Перевод не записывают, но всякий раз, когда нужно воспользоваться документом, его переводят заново, каждый раз при этом используя переводчика[1].

[1] Одно из основных значений английского interpreter — устный (синхронный) переводчик.

Рис. 1.1. Схема работы компилятора и интерпретатора

Нетрудно понять преимущества и недостатки компиляторов и интерпретаторов.

Компилятор обеспечивает получение быстрой программы на машинном языке, время работы которой намного меньше времени, которое будет затрачено на исполнение той же программы интерпретатором. Однако компиляция, являясь отдельным этапом обработки программы, может потребовать заметного времени, снижая оперативность работы. Откомпилированная программа представляет собой самостоятельный продукт, для использования которого компилятор не требуется. Программа в машинном коде, полученная компиляцией, лучше защищена от внесения несанкционированных искажений, раскрытия лежащих в ее основе алгоритмов.

Интерпретатор исполняет программу медленно, поскольку должен распознавать конструкции исходной программы каждый раз, когда они исполняются. Если какие-то действия расположены в цикле, то распознавание одних и тех же конструкций происходит многократно. Зато интерпретатор может начать испол-

нение программы сразу же, не затрачивая времени на компиляцию, — получается оперативней. Интерпретатор, как правило, проще компилятора, но его присутствие требуется при каждом запуске программы. Поскольку интерпретатор исполняет программу по ее исходному тексту, программа оказывается незащищенной от постороннего вмешательства.

Но даже при использовании компилятора получающаяся машинная программа работает, как правило, медленнее и занимает в памяти больше места, чем такая же программа, написанная вручную на машинном языке или языке ассемблера квалифицированным программистом. Компилятор использует набор шаблонных приемов для преобразования программы в машинный код, а программист может действовать нешаблонно, отыскивая оптимальные решения. Разработчики тратят немало усилий, стремясь улучшить качество машинного кода, порождаемого компиляторами.

В действительности различие между интерпретаторами и компиляторами может быть не столь явным. Некоторые интерпретаторы выполняют предварительную трансляцию исходной программы в промежуточную форму, удобную для последующей интерпретации. Некоторые компиляторы могут не создавать файла объектного кода. Например, Turbo Pascal может компилировать программу в память, не записывая машинный код в файл, и тут же ее запускать. А поскольку компилирует Turbo Pascal быстро, то такое его поведение вполне соответствует работе интерпретатора.

Существуют трансляторы, переводящие программу не в машинный код, а на другой язык программирования. Такие трансляторы иногда называют *конверторами*. Например, в качестве первого шага при реализации нового языка часто разрабатывают конвертор этого языка в язык Си. Дело в том, что Си — один из самых распространенных и хорошо стандартизованных языков.

Обычно ориентируются на версию ANSI Си — стандарт языка, принятый Американским Национальным Институтом Стандартов (American National Standards Institute, ANSI). Компиляторы ANSI Си есть практически в любой системе.

Кросскомпиляторы (cross-compilers) генерируют код для машины, отличной от той, на которой они работают.

Пошаговые компиляторы (incremental compilers) воспринимают исходный текст программы в виде последовательности задаваемых пользователем *шагов* (шагом может быть описание, группа описаний, оператор, группа операторов, заголовок процедуры и др.); допускается ввод, модификация и компиляция программы по шагам, а также отладка программ в терминах шагов.

Динамические компиляторы (Just-in-Time — JIT compilers), получившие в последнее время широкое распространение, транслируют промежуточное представление программы в объектный код во время исполнения программы.

Двоичные компиляторы (binary compilers) переводят объектный код одной машины (платформы) в объектный код другой. Такая разновидность компиляторов используется при разработке новых аппаратных архитектур, для переноса больших системных и прикладных программ на новые платформы, в том числе — операционных систем, графических библиотек и др.

Транслятор — это большая и сложная программа. Размер программы-транслятора составляет от нескольких тысяч до сотен тысяч строк исходного кода. Вместе с тем разработка транслятора для не слишком сложного языка — задача вполне посильная для одного человека или небольшого коллектива. Методы разработки трансляторов и будут рассмотрены в этой книге.

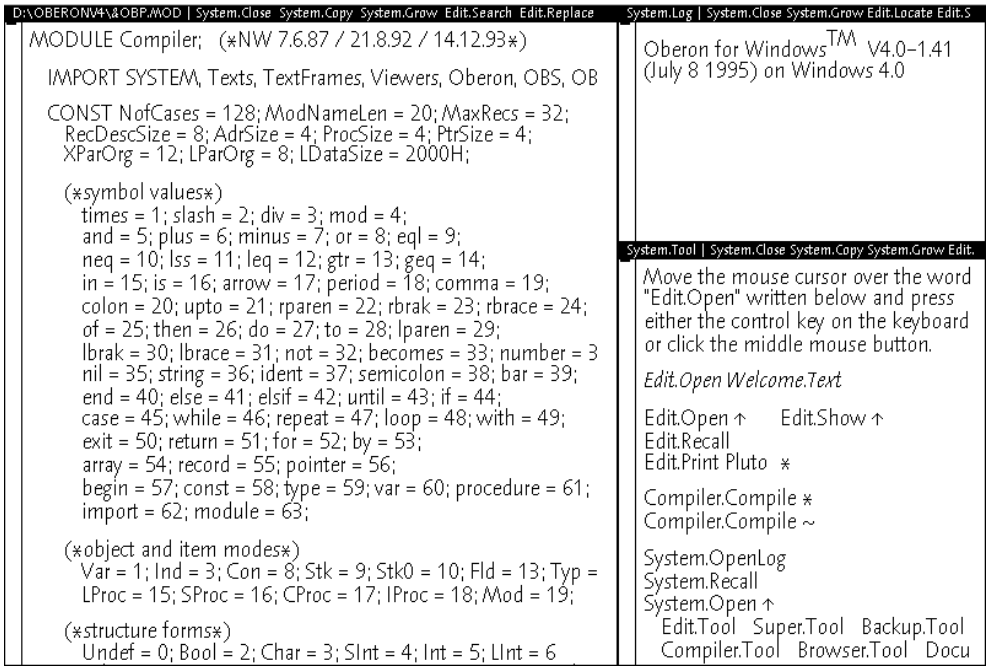


Рис. 1.2. Компилятор языка Оберон, написанный Никлаусом Виртом, с его «автографом» (NW). Фрагмент исходного текста основного модуля показан в левом окне Оберон-системы, частью которой является компилятор. Компилятор написан на языке Оберон и может компилировать сам себя

Транслятор связан в общем случае с тремя языками. Во-первых, это входной язык, с которого выполняется перевод. Во-вторых, целевой (объектный) язык, на который выполняется перевод. И, наконец, третий язык — это язык, на котором написан сам транслятор, — *инструментальный язык*. Трансляторы удобно изображать в виде Т-диаграмм[2] (рис. 1.3), которые предложил Х. Брэтман (H. Bratman) в 1961 году. Слева на такой диаграмме записывается исходный язык, справа — объектный, снизу — инструментальный.

[2] Кроме того, что диаграмма имеет форму буквы «Т», примечательно, что с этой буквы начинается и само слово «транслятор», как русское, так и английское.

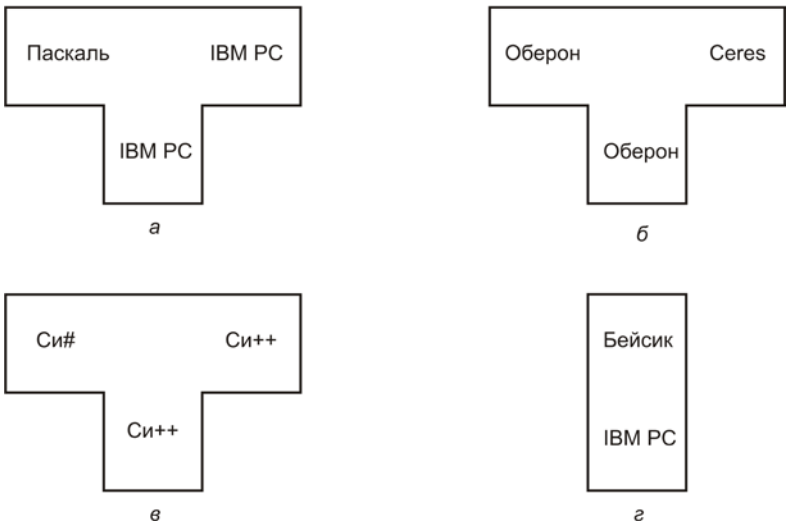


Рис. 1.3. Диаграммы трансляторов

Первые трансляторы, появившиеся в 1950-е годы, программировались на машинном языке или на языке низкого уровня — языке ассемблера (автокоде, как принято было говорить в то время). Сейчас для разработки трансляторов используются языки высокого уровня.

Изображая Т-диаграммы, мы можем не знать, на каком языке написан транслятор. В этом случае, а также когда недоступен исходный текст транслятора (т.е. мы располагаем только его исполняемой откомпилированной версией), на Т-диаграмме в качестве инструментального записывают машинный язык или обозначение того компьютера, на котором этот транслятор работает. Можно считать, что на рис. 1.3а изображен компилятор Turbo Pascal, Delphi или Free Pascal, транслирующий с языка Паскаль в машинный код IBM PC-совместимого компьютера и существующий в виде программы в машинном коде IBM PC. На рис. 1.3б показана Т-диаграмма компилятора с языка Оберон (см. рис. 1.2), транслирующего в машинный код компьютера Ceres и написанного на языке Оберон. Рисунок 1.3в соответствует конвертору. Си# — язык программирования, созданный в корпорации Microsoft. Первая реализация Си# вполне могла

быть выполнена по такой схеме. В дальнейшем мы еще обратимся к Т-диаграммам и узнаем, как они могут сочетаться подобно костям домино, иллюстрируя процесс получения одних трансляторов с помощью других.

Интерпретатор, кстати, можно изобразить в виде I-диаграммы (Interpreter — интерпретатор). Сверху записываем название входного, а внизу — инструментального языка. Пример диаграммы для интерпретатора Бейсика, работающего на IBM PC, можно видеть на рис. 1.3г.

Метаязыки

Говоря о Т-диаграммах, мы забыли еще об одном, четвертом языке, который неизбежно присутствует в разговоре о трансляторах и языках программирования. Это язык, например, русский, на котором мы ведем сам разговор. Язык, используемый для описания других языков, и называется *метаязыком.* В дальнейшем мы рассмотрим формализованные метаязыки, а пока в этой роли будем применять родной, естественный язык.

Понятно, что бессмысленно обсуждать сколько-нибудь содержательно сразу многие языки программирования, если не знаешь и одного, не имеешь хотя бы небольшого опыта программирования. Я рассчитываю, что у вас такой опыт есть. Я даже предполагаю, что вы, скорее всего, знакомы с языком Паскаль или, может быть, Си. Эти языки в нашем обзоре тоже будут в известном смысле играть роль метаязыков. Рассматривая другие языки, мы будем сравнивать имеющиеся в них средства с теми, что есть в Паскале или Си.

ГЛАВА 2. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ТРАНСЛЯЦИИ

Языки программирования — искусственно созданные формальные системы, которые могут изучаться математическими методами. Теория формальных языков и грамматик — это обширная область математики, примыкающая к алгебре, математической логике и теории автоматов. Цель этой главы — познакомиться с понятиями и результатами теории формальных языков и грамматик в той мере, как это требуется для понимания принципов конструирования трансляторов.

Формальные языки и грамматики

Основные термины и определения

Введем в обиход основные понятия, которые будут использованы в определении формального языка.

Алфавит — *конечное непустое множество символов*.

Термин *символ* следует понимать здесь в самом широком смысле. Это может быть буква, цифра или знак препинания. Но символом можно считать и любой другой знак, рассматриваемый как нечто неделимое — служебное слово языка программирования, иероглиф и т. д. Будем обозначать алфавиты буквой Σ (сигма).

Примеры алфавитов:

$\Sigma_1 = \{0, 1\}$

$\Sigma_2 = \{a, b, c\}$

$\Sigma_3 = \{A, B, C, \ldots, Z, a, b, c, \ldots, z,$

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, –, *, ..., **div**, ..., **program** }

Алфавит — это множество, поэтому при перечислении его элементов использованы фигурные скобки, как это принято в мате-

матике. Алфавит Σ_1 содержит два символа, алфавит Σ_2 — три. Под Σ_3 подразумевается алфавит языка Паскаль.

Цепочка *над алфавитом* Σ *— произвольная конечная последовательность символов из* Σ.

Примеры цепочек над алфавитом Σ_2:

$$\alpha = abbca$$

$$\beta = ab$$

$$\gamma = ba$$

$$\delta = c$$

Цепочки будем обозначать греческими буквами.

Пустая цепочка — *цепочка, не содержащая символов* (*содержащая ноль символов*). *Обозначается буквой* ε.

Если α и β — цепочки, то запись $\alpha\beta$ означает их *конкатенацию* (склеивание), то есть $\alpha\beta$ — это цепочка, образованная приписыванием к цепочке α цепочки β справа.

Если α — цепочка, то α^n означает цепочку, образованную n-кратным повторением цепочки α:

$$\alpha^n = \underbrace{\alpha\alpha\ldots\alpha\alpha}_{n\ раз}.$$

В частном случае, если a — символ, то $a^n = \underbrace{aa\ldots aa}_{n\ раз}$.

Будем обозначать Σ^* — (бесконечное) множество всех цепочек над алфавитом Σ, включая пустую цепочку; Σ^+ — множество всех цепочек над алфавитом Σ, не включая пустой цепочки. Например, если $\Sigma_1 = \{0, 1\}$, то Σ_1^* представляет собой множество всех цепочек, которые могут быть составлены из символов 0 и 1. В это множество входят пустая цепочка, все цепочки, состоящие

из одного символа, все цепочки, состоящие из двух символов и т. д.: $\Sigma_1^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$.

$\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$, где $\cup$ — знак операции объединения множеств.

Теперь можно дать и определение формального языка.

Языком *над алфавитом* Σ *называется произвольное множество цепочек, составленных из символов* Σ.

Будем обозначать язык над алфавитом (с алфавитом) Σ — $L(\Sigma)$ или просто L, если алфавит ясен из контекста.

Таким образом, речь идет о том, что язык — это некоторое, тем или иным образом определенное, подмножество множества всех цепочек, которые могут быть построены из символов данного алфавита. $L(\Sigma) \subseteq \Sigma^*$. На рисунке 2.1 наглядно показано соотношение множеств Σ^* и $L(\Sigma)$.

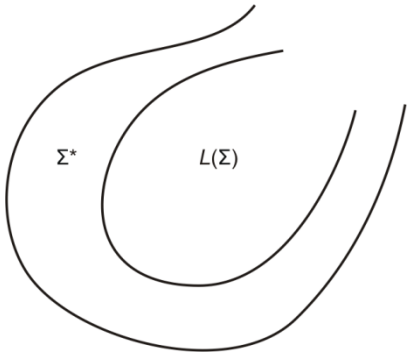


Рис. 2.1 Язык — подмножество множества всех цепочек над алфавитом Σ

Принадлежащие языку цепочки называют также *предложениями* языка.

Еще раз отметим, что множество цепочек Σ^* всегда бесконечно, в то время как множество цепочек, образующих язык, может быть и конечным. Практический интерес представляют, конечно, языки, содержащие бесконечное множество цепочек; к числу таких языков относятся и языки программирования.

Примеры языков

Пример 1. Определим язык $L_1 = \{a^n b^n \mid n \geq 0\}$, используя принятую в теории множеств нотацию, как множество всех цепочек, содержащих вначале некоторое количество символов a, а затем такое же количество символов b. Заметим, что L_1 включает и пустую цепочку, поскольку n может равняться нулю.

Записанное выше правило, определяющее язык L_1, разделяет все цепочки над алфавитом $\{a, b\}$, то есть состоящие из символов a и b, на принадлежащие L_1 и не принадлежащие ему.

Примеры цепочек, принадлежащих языку:

$\varepsilon \in L_1$ — пустая цепочка принадлежит L_1;

$ab \in L_1$ — цепочка из одной буквы a, за которой следует b;

$aaabbb \in L_1$.

Цепочки, не принадлежащие языку L_1:

$aaab \notin L_1$ — неодинаковое количество символов a и b;

$abba \notin L_1$ — порядок следования символов не соответствует определению L_1.

Пример 2. Язык $L_2 = \{a^n b^n c^n \mid n \geq 0\}$ — множество всех цепочек, содержащих вначале некоторое (возможно нулевое) количество символов a, затем такое же количество символов b, затем — столько же букв c. Например, $aaabbbccc \in L_2$, в то время как $aaabbc \notin L_2$.

Далеко не всегда удается определить язык, особенно если речь идет о языках, представляющих практический интерес, используя нотацию, примененную при определении L_1 и L_2. Значительная часть последующего материала будет посвящена рассмотрению порождающих грамматик, позволяющих компактно и однозначно определить обширный класс формальных языков. Пока

же, в следующих примерах, дадим словесное описание некоторых представляющих интерес языков.

Пример 3. Рассмотрим язык правильных скобочных выражений, составленных только из круглых скобок, известный также как *язык Дика*. Обозначим его L_3. Алфавит языка Дика — это множество из двух символов — открывающей «(» и закрывающей «)» скобок: $\Sigma_3 = \{ (,) \}$. Цепочки, содержащие правильно расставленные скобки принадлежат языку Дика, все остальные последовательности открывающих и закрывающих круглых скобок — нет. Например: $(())()() \in L_3$; $)(()))(\notin L_3$.

Пример 4. Язык L_4 — множество всех цепочек, содержащих одинаковое количество символов a и b. Несмотря на простое «устройство», задать язык L_4 формулой, подобной формулам для L_1 или L_2, оказывается затруднительно. Можно заметить, что рассмотренный ранее язык L_1 является подмножеством языка L_4: $L_1 \subseteq L_4$, поскольку любая цепочка, принадлежащая L_1, принадлежит и языку L_4. Но не наоборот. Так,

$$aabb \in L_1,\ aabb \in L_4,\ abba \in L_4, \text{ но } abba \notin L_1.$$

Пример 5. В качестве языка L_5 рассмотрим множество всех правильных арифметических выражений языка Паскаль, составленных из символов алфавита $\Sigma_5 = \{a, b, c, +, –, *, /, (,) \}$. Например, $a*(b+c) \in L_5$, но $c++ \notin L_5$.

Порождающие грамматики (грамматики Н. Хомского[3])

Порождающие грамматики — это простой и мощный механизм, позволяющий задавать обширный класс языков, содержащих бесконечное множество цепочек. С помощью порождающих грамматик мы сможем, в частности, определить языки L_3, L_4 и L_5, для задания которых раньше ограничивались словесными формулировками. Порождающие грамматики используются и при описании синтаксиса языков программирования.

Порождающей грамматикой называется четверка:

$$G = (T, N, P, S), \text{ где}$$

T — конечное *множество терминальных* (*основных*) *символов* — основной алфавит. Элементами множества T являются символы, из которых в конечном итоге и состоят цепочки языка, порождаемого данной грамматикой. Терминальный (от лат. terminus — предел, конец) и означает «конечный, концевой». T — это не что иное, как алфавит языка, порождаемого грамматикой. В дальнейшем, если не оговорено особо, терминальные символы или просто *терминалы* будут обозначаться малыми буквами латинского алфавита: a, b, c и т. д.

N — конечное *множество нетерминальных* (*вспомогательных*) *символов* — вспомогательный алфавит. Нетерминальные символы, по-другому *нетерминалы,* — это понятия грамматики (языка), которые используются при его описании. Нетерминалы бу-

[3] Ноам Хомский (Noam Chomsky, р. 1928) — американский лингвист и политический активист. Начиная с 1957 года, опубликовал ряд работ, заложивших основы математической лингвистики. Выступал против войны США во Вьетнаме, автор нескольких десятков книг на политические темы. Участник движения антиглобалистов. В отечественной прессе его фамилия иногда звучит как Чомски, однако в научных публикациях на русском языке, начиная с 1962 года, принято написание Хомский.

дем обозначать заглавными латинскими буквами: *A, B, C, D, E* и т. д.

P — конечное *множество правил вывода*, называемых также *продукциями*. Каждое правило множества *P* имеет вид:

$$\alpha \to \beta,$$

где α и β — цепочки терминальных и нетерминальных символов. Цепочка α не должна быть пустой, цепочка β может быть пуста: $\alpha \in (T \cup N)^+$; $\beta \in (T \cup N)^*$. Правило $\alpha \to \beta$ определяет возможность *подстановки* β вместо α в процессе вывода (порождения) цепочек языка.

S (*S*∈*N*) — *начальный символ грамматики* — один из множества нетерминальных символов, *начальный (стартовый) нетерминал*. Начальный нетерминал — это понятие, соответствующее правильному предложению языка. Например, начальный нетерминал грамматики выражений обозначает «выражение», а начальный нетерминал грамматики языка Паскаль — «Программа».

Примеры грамматик. Порождение предложений языка

Пример 1. Рассмотрим грамматику

$$G_1 = (\ \{a, b\}, \{S\}, \{S \to aSb, S \to \varepsilon\}, S\ ).$$

Здесь все элементы четверки записаны явно. Множество терминальных символов $T = \{a, b\}$; множество нетерминалов содержит один элемент: $N = \{S\}$, а множество правил — два: $P = \{S \to aSb, S \to \varepsilon\}$; роль начального нетерминала исполняет *S*.

Грамматика может использоваться для порождения (вывода) цепочек — предложений языка. Процесс порождения начинается с начального нетерминала, в нашем примере это *S*. Если среди правил есть такое, в левой части которого записана цепочка *S*, то начальный нетерминал может быть заменен правой частью лю-

бого из таких правил. Оба правила грамматики G_1 содержат в левой части S. Применим подстановку, заданную первым правилом, заменив S на aSb:

$$S \underset{(1)}{\Rightarrow} aSb .$$

К получившейся цепочке aSb снова, если удастся, можно применить одно из правил грамматики. Если в цепочке есть подцепочка, совпадающая с левой частью хотя бы одного из правил, то эту подцепочку можно заменить правой частью любого из таких правил. В цепочке aSb есть подцепочка S, совпадающая с левой частью обоих правил грамматики G_1. Мы вправе применить любое из этих правил. Используем снова правило (1) для продолжения вывода:

$$S \underset{(1)}{\Rightarrow} aSb \underset{(1)}{\Rightarrow} aaSbb .$$

Теперь к получившейся цепочке применим правило (2) ($S \to \varepsilon$), заменив S пустой цепочкой. Получим такую последовательность подстановок (саму букву ε в последней цепочке записывать, конечно, не нужно):

$$S \underset{(1)}{\Rightarrow} aSb \underset{(1)}{\Rightarrow} aaSbb \underset{(2)}{\Rightarrow} aabb .$$

Очевидно, что к получившейся цепочке ни одно из правил грамматики G_1 больше не применимо. Процесс порождения завершен. Нетрудно заметить, что с помощью грамматики G_1 можно породить любую цепочку языка $L_1 = \{a^n b^n \mid n \geq 0\}$, применив к начальному нетерминалу правило (1) ($S \to aSb$) n раз, а затем один раз правило (2). В то же время, грамматика G_1 не порождает ни одной цепочки терминальных символов, не принадлежащей языку L_1. То есть множество терминальных цепочек, порождаемых грамматикой G_1, совпадает с языком L_1. Другими словами, грамматика G_1 порождает язык L_1:

$$L(G_1) = L_1 .$$

Обычно при записи грамматики не выписывают четверку ее элементов явно. При соблюдении соглашений об обозначениях терминалов и нетерминалов достаточно записать только правила. Правила с одинаковой левой частью можно объединять в одно, отделяя альтернативные правые части вертикальной чертой. Первым записывается правило, содержащее в левой части начальный нетерминал. С учетом этого грамматика G_1 может быть записана так:

$$G_1: \quad S \to aSb \mid \varepsilon .$$

Пример 2. Рассмотрим грамматику G_2 (цифры справа — номера правил)

G_2: $S \to aSBc$ (1)

$S \to abc$ (2)

$cB \to Bc$ (3)

$bB \to bb$ (4)

$S \to \varepsilon$ (5)

Проведем вывод цепочек из начального нетерминала грамматики G_2. Под стрелкой, обозначающей подстановку, будем указывать, как и раньше, номер примененного правила. Итак:

$$S \underset{(1)}{\Rightarrow} aSBc \underset{(2)}{\Rightarrow} aabcBc \underset{(3)}{\Rightarrow} aabBcc \underset{(4)}{\Rightarrow} aabbcc$$

Еще одна серия подстановок:

$$S \underset{(1)}{\Rightarrow} aSBc \underset{(1)}{\Rightarrow} aaSBcBc \underset{(2)}{\Rightarrow} aaabcBcBc \underset{(3)}{\Rightarrow} aaabBccBc \underset{(3)}{\Rightarrow} aaabBcBcc \underset{(3)}{\Rightarrow}$$

$$aaabBBccc \underset{(4)}{\Rightarrow} aaabbBccc \underset{(4)}{\Rightarrow} aaabbbccc$$

Можно убедиться, что грамматика G_2 порождает цепочки терминалов вида $a^n b^n c^n$ и никакие другие. Количество повторений символов в результирующей цепочке определяется тем, на каком шаге применяется правило (2). Наличие правила (5) позво-

ляет получить пустую цепочку, если применить это правило первым, в то время как попытка использования этого правила на последующих шагах не позволит вывести цепочку терминалов. Грамматика G_2 порождает множество терминальных цепочек, совпадающее с языком $L_2 = \{a^n b^n c^n \mid n \geq 0\}$:

$$L(G_2) = L_2 .$$

Пример 3. Грамматика, порождающая язык правильных скобочных выражений (язык Дика).

G_3: $S \to (S)$ (1)

$S \to SS$ (2)

$S \to \varepsilon$ (3)

Нетрудно понять логику построения правил этой грамматики. Смысл первого правила таков: заключив в скобки правильное скобочное выражение S, мы снова получим правильное скобочное выражение. Второе правило означает, что два правильных скобочных выражения, записанные одно за другим, дают новое правильное выражение. Наконец, по правилу (3) пустая цепочка считается правильным выражением. Если бы мы решили, что не следует разрешать пустые выражения, правило (3) можно было бы заменить на $S \to ()$.

Пример 4. Грамматика простых арифметических выражений. Единственным нетерминалом этой грамматики (он же начальный) будет *Выражение:*

G_4: *Выражение* → *Выражение* + *Выражение* |
Выражение – *Выражение* |
Выражение * *Выражение* |
Выражение / *Выражение* | *a* | *b* | *c* | (*Выражение*) .

Такая грамматика порождает цепочки терминалов, являющиеся правильными арифметическими выражениями. Символы *a, b* и *c* в таких выражениях обозначают операнды, а «+», «–», «*», и

«/» — знаки операций. Разрешаются круглые скобки (в том числе вложенные). Примеры правильных выражений: (*a*+*b*)/(*b***c*), (*a*). Все операции двуместные, унарные плюс и минус не предусмотрены, поэтому, например, цепочка –*a* не принадлежит языку, порождаемому этой грамматикой.

Выражение — одно из основных понятий языков программирования. В дальнейшем грамматикам выражений будет уделено немалое внимание. Чтобы запись этих грамматик была короче, заменим нетерминал *Выражение* на *E* (от *expression* — выражение), вернувшись тем самым к принятым раньше соглашениям об именовании нетерминалов. Тогда грамматика, которую обозначим G_5, запишется следующим образом:

$$G_5: \quad E \to E + E \mid E - E \mid E * E \mid E / E \mid a \mid b \mid c \mid (E) .$$

Очевидно, что она порождает тот же язык, что и грамматика G_4. А именно:

$$L(G_5) = L_5 .$$

Еще несколько определений

В предыдущих примерах мы уже пользовались такими терминами как «вывод», «язык, порождаемый грамматикой». Теперь дадим формальные определения для этих и ряда других понятий.

Пусть имеются грамматика $G = (T, N, P, S)$ и цепочка α_1, составленная из терминалов и нетерминалов грамматики G, причем α_1 представима в виде $\alpha_1 = \gamma_1\alpha\gamma_2$, где γ_1, γ_2 — цепочки терминалов и нетерминалов грамматики G, α — непустая цепочка терминалов и нетерминалов грамматики G: $\gamma_1, \gamma_2 \in (T \cup N)^*$; $\alpha \in (T \cup N)+$. Пусть также среди множества правил P грамматики G есть правило $\alpha \to \beta$. Тогда подцепочка α цепочки α_1 может быть заменена цепочкой β, в результате чего будет получена цепочка $\alpha_2 = \gamma_1\beta\gamma_2$. В этом случае говорят, что цепочка α_2 *непосредст-*

венно выводится (порождается) из цепочки α_1 в грамматике G, что записывается следующим образом:

$$\alpha_1 \underset{G}{\Rightarrow} \alpha_2 \text{ или просто } \alpha_1 \Rightarrow \alpha_2,$$

если используемая грамматика очевидна.

Если имеется последовательность цепочек $\alpha_1, \alpha_2, \ldots, \alpha_n$ $(n > 1)$, таких что

$$\alpha_1 \underset{G}{\Rightarrow} \alpha_2 \ldots \underset{G}{\Rightarrow} \alpha_n, \qquad (*)$$

то говорят, что цепочка α_n *нетривиально выводится* из α_1 в грамматике G (выводится за один или более шагов), что обозначается так:

$$\alpha_1 \overset{+}{\underset{G}{\Rightarrow}} \alpha_n \text{ или просто } \alpha_1 \overset{+}{\Rightarrow} \alpha_n.$$

Последовательность цепочек $\alpha_1, \alpha_2, \ldots, \alpha_n$ в этом случае называется **выводом** цепочки α_n из α_1 в грамматике G.

В дальнейшем выводом будем называть также запись, подобную (*). Используется также запись

$$\alpha_1 \overset{*}{\underset{G}{\Rightarrow}} \alpha_n \text{ или } \alpha_1 \overset{*}{\Rightarrow} \alpha_n,$$

означающая, что цепочка α_n *выводится* из α_1 в грамматике G (выводится *за ноль или более* шагов), что следует понимать так, что, либо α_n совпадает с α_1, либо $\alpha_1 \overset{+}{\Rightarrow} \alpha_n$.

Сентенциальной формой грамматики G называется цепочка, выводимая из начального нетерминала грамматики G.

Цепочка α — есть сентенциальная форма грамматики G, если $S \overset{*}{\underset{G}{\Rightarrow}} \alpha$.

Сентенцией (от *sentence* — предложение) грамматики G называется сентенциальная форма, состоящая только из терминальных символов.

Язык, порождаемый грамматикой, есть множество всех её сентенций.

Можно сказать, что сентенции грамматики — это предложения порождаемого ею языка.

Дерево вывода

В последующем рассмотрении предполагается, что мы имеем дело с грамматиками, все правила которых в своей левой части содержат единственный нетерминал. Именно такие грамматики, называемые контекстно-свободными, представляют для нас наибольший практический интерес.

Рассмотрим грамматику

$$
\begin{aligned}
G_6\text{:}\quad & S \to AB && (1)\\
& A \to aA \mid a && (2)\\
& B \to bB \mid b && (3)
\end{aligned}
$$

Выполним вывод цепочек в этой грамматике. Вначале используем правило (1):

$$S \Rightarrow AB\,.$$

Будем сопровождать процесс вывода построением дерева (рисунок 2.2). Корнем дерева будет вершина, соответствующая начальному нетерминалу S. Дочерними вершинами корня будут вершины A и B, соответствующие правой части первого примененного правила (рис. 2.2а). Вершины A и B следуют в дереве слева направо в том же порядке, что и в правиле (1): слева — A, справа — B.

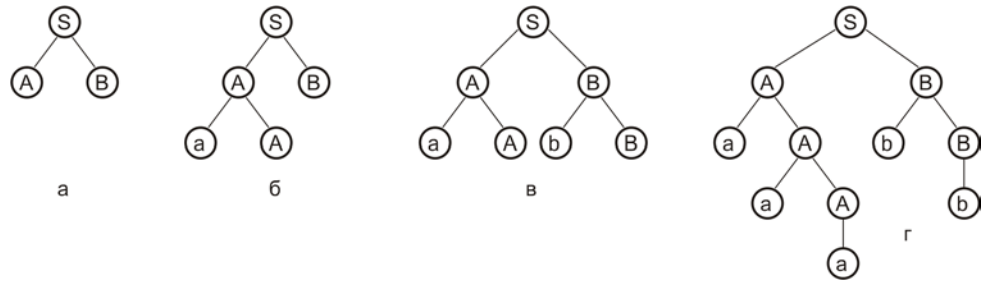


Рис. 2.2. Построение дерева вывода

Продолжая вывод, мы можем выбрать, как правило для нетерминала A — правило (2), так и правило для B — правило (3). Используем вначале первую часть правила (2) ($A \to aA$):

$$S \Rightarrow AB \Rightarrow aAB$$

и продолжим построение дерева (рис. 2.2б). Теперь выполним подстановку вместо нетерминала B цепочки bB по правилу (3):

$$S \Rightarrow AB \Rightarrow aAB \Rightarrow aAbB\ ,$$

добавив к имеющейся вершине B дочерние вершины b и B (рис. 2.2в). Еще раз применим правило $A \to aA$ и, чтоб получить цепочку, состоящую только из терминалов, выполним подстановки по правилам $A \to a$ и $B \to b$. После этого вывод приобретет следующий вид:

$$S \Rightarrow AB \Rightarrow aAB \Rightarrow aAbB \Rightarrow aaAbB \Rightarrow aaabB \Rightarrow aaabb\ ,$$

а получившееся дерево показано на рис. 2.2г.

ВОПРОС

Какой язык порождает грамматика G_6?

Получившееся дерево называется *деревом вывода, деревом разбора* или *синтаксическим деревом*[4]. Его корень — начальный

[4] К сожалению, в литературе на русском языке существует путаница в терминах. Одни авторы (и их переводчики) [Грис, 1975] считают «дерево разбора» и «синтаксическое дерево» синонимами, другие [Ахо, 2001] придают понятию «синтак-

символ грамматики, внутренние вершины — нетерминалы, листья дерева (концевые вершины) — терминалы. Обход листьев дерева слева направо дает цепочку терминалов, выведенную из начального символа грамматики (сентенцию)[5].

Нетрудно заметить, что построенное нами дерево будет соответствовать и другим выводам цепочки *aaabb* в грамматике G_6. Вот один из них:

$$S \Rightarrow AB \Rightarrow AbB \Rightarrow aAbB \Rightarrow aAbb \Rightarrow aaAbb \Rightarrow aaabb\ .$$

Рассмотрение дерева вместо вывода позволяет игнорировать порядок применения правил, если он не важен.

Задача разбора

Задача разбора состоит в восстановлении дерева вывода для заданной сентенции.

Разбор — это построение вывода для заранее заданной цепочки. Другими словами, разбор — это тот же вывод, прослеженный в обратном порядке. Последовательность сентенциальных форм, приводящая к цепочке терминалов (сентенции, предложению языка, порождаемого грамматикой), определяет структуру этой цепочки. Дерево вывода представляет структуру цепочки нагляднее и независимо от последовательности применения правил.

Результатом решения задачи разбора в случае, если удалось восстановить дерево для заданной терминальной цепочки, является выявление структуры этой цепочки. Построенное дерево назы-

---

сическое дерево» иной смысл. Я буду придерживаться первого варианта, поскольку термины «разбор» и «синтаксический анализ» означают одно и то же, и было бы странно их различать, говоря о деревьях.

[5] Могут рассматриваться и деревья вывода, листьями которых являются как терминалы, так и нетерминалы. В этом случае обход листьев дает сентенциальную форму грамматики. Однако такие деревья нам не потребуются.

вается *деревом разбора*. Успешное восстановление дерева разбора для заданной цепочки означает, что эта цепочка есть правильное предложение языка, порождаемого грамматикой. Наоборот, если для некоторой цепочки терминалов дерево разбора в данной грамматике построить невозможно, это значит, что цепочка не принадлежит порождаемому грамматикой языку.

Для чего надо решать задачу разбора

Разбор (по-английски — parsing) называют также распознаванием или синтаксическим анализом. Синтаксический анализ имеет две цели — выяснение принадлежности цепочки языку и выявление ее структуры.

Работа любого транслятора основана на распознавании структуры предложений транслируемого языка. Синтаксический анализ — обязательная фаза в работе компиляторов и интерпретаторов языков программирования. Синтаксический анализатор — это часть транслятора, составляющая его основу.

Только распознавая структуру входной программы, определяя наличие или отсутствие отдельных ее частей — описаний, операторов, выражений и подвыражений — транслятор может выполнить работу по переводу программы на другой язык. Часто трансляторы в явном виде строят дерево программы, которое представляется внутренними динамическими структурами данных транслятора, а затем используется при формировании эквивалентной выходной программы. Если в ходе распознавания дерево вывода не строится, оно присутствует неявно, отражаясь в последовательности выполняемых синтаксическим анализатором действий.

Рассмотрению способов решения задачи разбора — синтаксического анализа будет посвящена большая часть этой главы.

Домино Де Ремера

Де Ремер (De Remer F. L.) предложил наглядную интерпретацию задачи разбора, представив ее как игру в своеобразное домино. Играющий располагает «костями» домино нескольких типов. Типов столько, сколько правил в грамматике. Каждое правило дает один тип пластинки. Типы домино для грамматики G_6 показаны на рис. 2.3. Считается, что «костяшек» каждого типа имеется сколько необходимо.

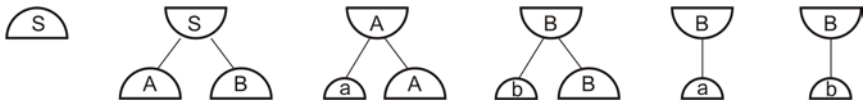


Рис. 2.3. Домино Де Ремера для грамматики G_6

Верхняя часть каждого домино соответствует левой части правила грамматики, нижняя — правой. Верхняя и нижние пластинки соединены «резиновыми» нитями. Пластинки можно приставлять друг к другу плоскими сторонами полукруга, если на них записаны одинаковые символы. Фигуры домино нельзя переворачивать, и нельзя менять порядок следования символов (перекрещивать нити).

В начале игры в верхней части поля помещается полукруг, обращенный выпуклостью вверх, в котором записан начальный нетерминал грамматики. В нижней части игрового поля в полукругах, обращенных плоской частью вверх, размещаются терминальные символы распознаваемой цепочки. На рис. 2.4 показана начальная конфигурация игры для цепочки *aaabb*.

Рис. 2.4. Начало игры в домино Де Ремера

Цель состоит в том, чтобы соединить с помощью имеющихся фигур символы терминальной цепочки и начальный нетерминал. Полученная конфигурация домино для цепочки *aaabb* и грамматики G_6 (набор домино на рис. 2.3) показана на рис. 2.5.

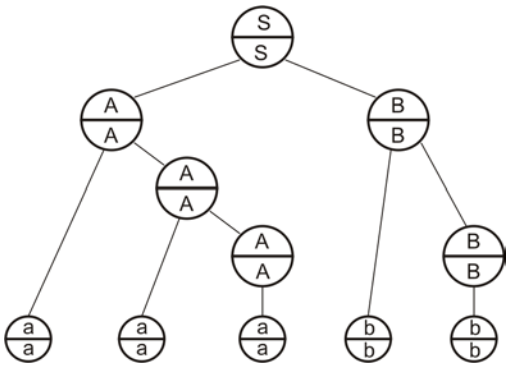


Рис. 2.5. Дерево разбора, построенное из домино Де Ремера

Разновидности алгоритмов разбора

Имея в виду интерпретацию Де Ремера, можно представить себе и различные подходы к решению задачи разбора. Если подбор костей удается осуществлять так, что однажды поставленную кость никогда не придется убирать, мы имеем дело с детерминированным алгоритмом разбора — выбор применяемого правила грамматики всегда однозначен. Если принятые решения о выборе типа домино приходится отменять — алгоритм работает с возвратами, он недетерминирован. Детерминированные алгоритмы эффективней и, конечно, всегда существует стремление

найти и использовать такой алгоритм для синтаксического анализа.

Если дерево строится сверху вниз от начального нетерминала в сторону терминальной цепочки, алгоритм относится к классу нисходящих; от цепочки в сторону корня дерева — восходящий (рис. 2.6). Можно вначале подбирать домино для левых символов терминальной цепочки, а можно вначале для правых — соответственно говорят о левосторонних или правосторонних алгоритмах разбора.

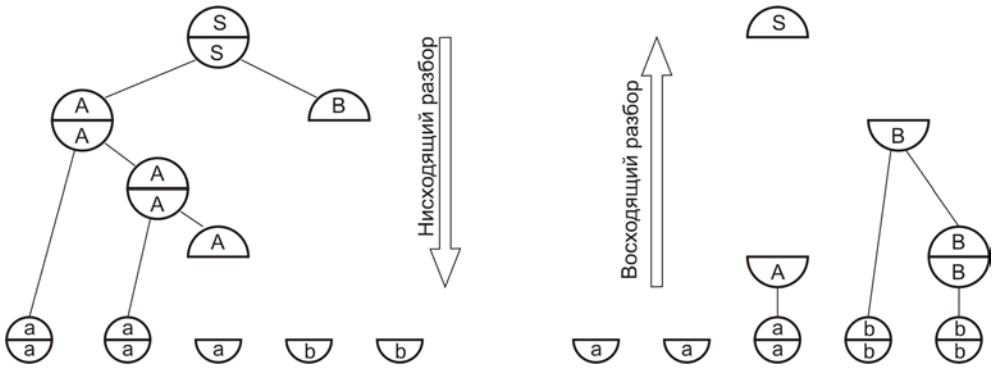


Рис. 2.6. Начало нисходящего и восходящего разбора в грамматике G_6

Эквивалентность и однозначность грамматик

Возьмем для примера грамматику арифметических выражений

$$G_5: \quad E \to E + E \mid E - E \mid E * E \mid E / E \mid a \mid b \mid c \mid (E) .$$

Рассмотрим деревья вывода терминальных цепочек в этой грамматике. На рис. 2.7 показаны два различных дерева разбора выражения *a*+*b***c*. Возможность построить эти деревья убеждает в том, что цепочка *a*+*b***c* действительно принадлежит языку арифметических выражений.

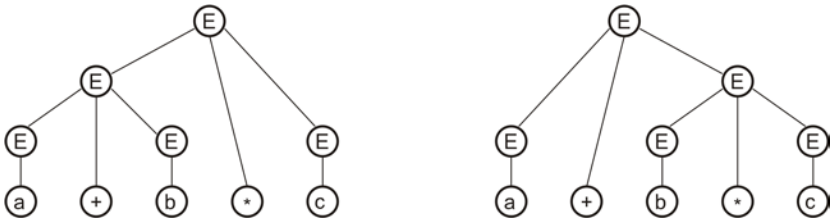


Рис. 2.7. Деревья разбора цепочки *a+b*c* в грамматике G_5

Но два разных дерева дают две разные трактовки структуры этой цепочки. Дерево слева объединяет *a+b* в одно подвыражение, которое затем участвует в роли операнда в операции умножения. Такая трактовка не соответствует общепринятому приоритету арифметических операций — умножение должно выполняться раньше сложения. Дерево справа представляет структуру, соответствующую правильному порядку выполнения операций.

Грамматика G называется однозначной, если любой сентенции $x \in L(G)$ *соответствует единственное дерево вывода.*

Грамматика G_5 неоднозначна, и это, безусловно, ее серьезный недостаток, который не позволяет применить ее на практике для определения языка выражений, поскольку эта грамматика не позволяет однозначным образом выявить структуру выражения.

Попробуем построить однозначную грамматику выражений. Для этого используем еще один нетерминал, обозначив его *X*:

$$G_7: \quad E \to X \mid E + X \mid E - X \mid E * X \mid E / X$$

$$X \to a \mid b \mid c \mid (E)$$

Содержательно *X* можно понимать как операнд выражения. Теперь для цепочки *a+b*c* можно построить единственное дерево разбора. Оно показано на рис. 2.8 слева. Можно убедиться, что в грамматике G_7 единственное дерево вывода соответствует любому правильному выражению. Грамматика G_7 однозначна.

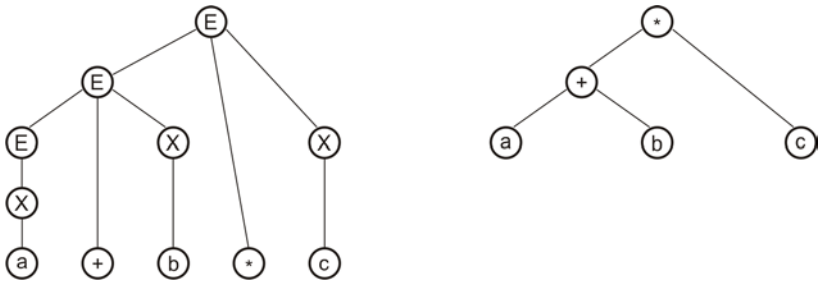


Рис. 2.8. Дерево разбора выражения *a+b*c* в грамматике G_7

Справа на рисунке показано редуцированное (упрощенное) дерево того же выражения в грамматике G_7. Такие деревья могут служить для представления структуры выражений в трансляторе. Будем называть их *семантическими деревьями*[6]. Семантическое дерево может быть получено из дерева разбора устранением нетерминальных вершин и помещением знаков операций во внутренние вершины, в то время как операнды остаются листьями дерева.

Несмотря на однозначность, G_7 непригодна для использования в трансляторе, поскольку приписывает выражениям неподходящую структуру. Уже на примере цепочки *a+b*c* (см. рис. 2.8) видно, что операции и операнды связываются неправильно. Нетрудно заметить, что G_7 предполагает выполнение операций без учета их приоритета в порядке слева направо.

Исправить положение можно, определив нетерминалы для двух категорий подвыражений — слагаемых и множителей. После этого выражение представляется как последовательность сла-

[6] Как уже говорилось, существует несогласованность в использовании терминов, относящихся к синтаксическим и семантическим деревьям. В этой книге мы будем, следуя [Рейуорд-Смит, 1988], придерживаться названия «семантическое дерево». В то же время в издании [Ахо, 2001] в этом случае говорится о синтаксическом дереве, что противоречит терминологии классической монографии [Грис, 1975] и, по моему мнению, создает путаницу, поскольку трудно видеть разницу между синтаксическим деревом и деревом разбора.

гаемых, разделенных знаками плюс и минус. В свою очередь слагаемые образуются из элементарных операндов — множителей, соединенных знаками умножения и деления. Слагаемые обозначим T (от term — элемент), множители — M (от multiplier).

$$G_8: \quad E \to T \mid E + T \mid E - T$$
$$T \to M \mid T * M \mid T / M$$
$$M \to a \mid b \mid c \mid (E)$$

Дерево вывода цепочки $a+b*c$ в грамматике G_8 показано на рис. 2.9. Эта грамматика однозначна и приписывает арифметическим выражениям структуру, соответствующую правильному порядку выполнения операций.

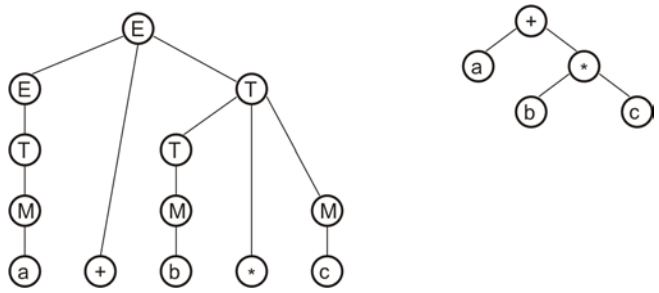


Рис. 2.9. Дерево выражения $a+b*c$ в грамматике G_8

Все три рассмотренные выше грамматики выражений (G_5, G_7, G_8), хотя и различаются и обладают разными свойствами, задают один и тот же язык.

Грамматики называются **эквивалентными**, *если порождают один и тот же язык.*

Грамматики G_5, G_7, G_8 эквивалентны, поскольку $L(G_5) = L(G_7) = L(G_8)$.

Иерархия грамматик Н. Хомского

Н. Хомский предложил разделение порождающих грамматик на 4 типа в зависимости от вида их правил.

Тип 0. *Произвольные грамматики.* На вид их правил не накладывается каких-либо ограничений. Правила имеют вид:

$$\alpha \to \beta,$$

где α и β — цепочки терминалов и нетерминалов. Цепочка α не должна быть пустой.

Тип 1. *Контекстно-зависимые грамматики.* Правила таких грамматик имеют вид:

$$\alpha A \beta \to \alpha \gamma \beta,$$

где α, β, γ — цепочки терминалов и нетерминалов; A — нетерминальный символ. Такой вид правил означает, что нетерминал A может быть заменен цепочкой γ только в контексте, образуемом цепочками α и β.

Тип 2. Контекстно-свободные грамматики. Их правила имеют вид:

$$A \to \gamma,$$

где, A — нетерминал; γ — цепочка терминалов и нетерминалов. Характерная особенность — в левой части правил всегда один нетерминальный символ.

Тип 3. *Автоматные грамматики.* Все правила автоматных грамматик имеют одну из трех форм:

$$A \to aB$$

$$A \to a$$

$$A \to \varepsilon,$$

где A, B — нетерминалы; a — терминал; ε — пустая цепочка. Автоматные грамматики называют также *регулярными*.

Как можно видеть, грамматики типа 1 являются частным случаем грамматик типа 0, грамматики типа 2 — частный случай кон-

текстно-зависимых грамматик, автоматные — частный случай контекстно-свободных. То есть, грамматика типа 3 является и грамматикой типа 2, и типа 1, и типа 0. Однако в дальнейшем, если не оговорено особо, будет иметься в виду что, например, контекстно-свободной называется грамматика, не являющаяся автоматной.

Языки, порождаемые грамматиками типа 0–3, называются соответственно языками без ограничений, контекстно-зависимыми, контекстно-свободными и автоматными (регулярными) языками. Но контекстно-свободным языком называют язык, для которого существует порождающая его контекстно-свободная, но не автоматная грамматика. Такой же подход применяется к контекстно-зависимым языкам и языкам без ограничений.

Примеры грамматик различных типов

Рассмотрим грамматику G_2, порождающую язык $L_2 = \{a^n b^n c^n \mid n \geq 0\}$.

G_2: $S \to aSBc$ (тип 2)

$S \to abc$ (тип 2)

$cB \to Bc$ (тип 0)

$bB \to bb$ (тип 1)

$S \to \varepsilon$ (тип 3)

Справа около каждого правила помечен тип грамматики, к которой оно может быть отнесено. Типом грамматики естественно считать минимальный из типов ее правил. Следовательно, грамматика G_2 — это грамматика типа 0 — произвольная. Утверждается, однако, что может быть построена контекстно-зависимая грамматика (типа 1), порождающая тот же язык, что и G_2. Проверку этого утверждения предоставлю читателям.

Примером контекстно-свободной грамматики может служить грамматика арифметических выражений. С помощью контекстно-свободных грамматик задается и синтаксис языков программирования. Грамматики этого класса будут подробно обсуждаться и в дальнейшем, сейчас же возьмем конкретный пример.

G_9: $N \to a$ (1)

$N \to Na$ (2)

$N \to Nb$ (3)

Это грамматика типа 2, поскольку правила (2) и (3) относятся именно к этому типу. Рассмотрим язык $L_9 = L(G_9)$, порождаемый этой грамматикой. Цепочка a принадлежит L_9 по правилу (1). Если к правильному предложению N языка приписать справа символ a, то снова получится правильное предложение (по правилу (2)). Аналогично, приписывание к N символа b снова дает предложение языка L_9. Принадлежащие языку L_9 цепочки начинаются символом a, за которым могут следовать a и b в произвольном порядке. Если под a понимать любую латинскую букву, а b воспринимать как цифру, то G_9 можно считать грамматикой идентификаторов. Она порождает последовательности букв и цифр, начинающиеся с буквы, которые используются в языках программирования в роли имен переменных, типов и т. д.

Опираясь на такую содержательную трактовку G_9 и L_9, попытаемся сконструировать автоматную грамматику, порождающую язык идентификаторов.

Первое правило грамматики G_9 может быть сохранено. Оно, во-первых, соответствует одному из допустимых видов правил автоматных грамматик, во-вторых, определяет, что идентификатор, состоящий из одного символа может быть только буквой.

$N \to a$ (1)

Нетерминал N — это начальный символ нашей грамматики. Он и обозначает само понятие «идентификатор». Как синоним термина «идентификатор» будем использовать также слово «имя». Можно считать, что название N происходит от Name — имя. Обозначим B часть идентификатора, которая может следовать за первой буквой. Тогда можно записать правило (2):

$$N \to aB \qquad (2)$$

Запишем правила для B. «Хвост» может быть буквой или цифрой:

$$B \to a \qquad (3)$$

$$B \to b \qquad (4)$$

Если B — это «хвост» идентификатора, то, записав его за буквой или цифрой, снова получим правильный «хвост»:

$$B \to aB \qquad (5)$$

$$B \to bB \qquad (6)$$

Обозначим сконструированную грамматику G_{10}. Она автоматная, поскольку все ее правила удовлетворяют ограничениям автоматных грамматик.

G_{10}:

$$N \to a \qquad (1)$$

$$N \to aB \qquad (2)$$

$$B \to a \qquad (3)$$

$$B \to b \qquad (4)$$

$$B \to aB \qquad (5)$$

$$B \to bB \qquad (6)$$

Грамматика G_{10} эквивалентна G_9, поскольку порождает тот же язык: $L(G_{10}) = L(G_9)$. Этот язык — язык идентификаторов — следует считать автоматным. Нетрудно увидеть возможности упро-

щения грамматики G_{10}. Отложим, однако, на некоторое время такое упрощение.

Автоматные грамматики и языки

Рассмотрим автоматные грамматики и языки подробнее, имея целью построение алгоритмов распознавания этого класса языков.

Граф автоматной грамматики

Для каждой автоматной грамматики можно построить направленный граф по следующим правилам:

1. Каждому нетерминальному символу грамматики ставится в соответствие вершина графа, которая помечается этим символом.

2. При наличии правил вида

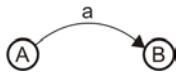
$$A \to a$$

добавляется дополнительная вершина, которая помечается символом K.

3. Каждое правило вида

$$A \to aB$$

порождает дугу графа, ведущую из вершины A в вершину B.

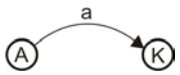


Дуга помечается символом a.

4. Каждое правило вида

$$A \to a$$

порождает дугу графа, ведущую из вершины A в вершину K.



Дуга помечается символом a.

5. Вершина, соответствующая начальному нетерминалу, помечается стрелкой.

6. Вершина K и вершины, соответствующие нетерминалам, для которых есть правило

$$A \to \varepsilon,$$

помечаются как конечные. Мы будем изображать их двойным кружком.

Построим граф автоматной грамматики G_{10} (рис. 2.10). Двум нетерминалам этой грамматики будут соответствовать вершины N и B (п. 1). Поскольку в грамматике есть несколько правил, в правой части которых записан единственный терминал, добавим вершину K (п. 2).

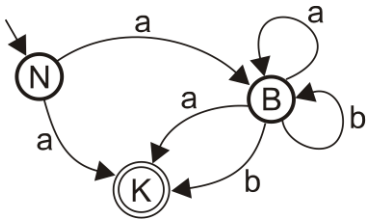


Рис. 2.10. Граф автоматной грамматики G_{10}

Соединим вершины дугами, как это предписывается п. 3 и п. 4. Вершину N пометим стрелкой как начальную (п. 5).

Граф автоматной грамматики может использоваться для порождения цепочек языка. Любой путь из начальной вершины графа в одну из конечных вершин порождает цепочку терминалов, соответствующую проходимым дугам. Эта цепочка принадлежит

языку, порождаемому грамматикой. И, наоборот, для любой сентенции грамматики можно найти путь, ведущий из начальной вершины в одну из конечных и проходящий по дугам, помеченным символами этой сентенции.

Грамматика G_{10} порождает язык идентификаторов. Нетрудно убедиться, что для любого идентификатора найдется путь из вершины N в вершину K, а любой путь из N в K соответствует правильному идентификатору.

Граф автоматной грамматики идентичен диаграмме переходов конечного автомата — абстрактного устройства, являющегося моделью определенного класса реальных автоматических устройств и объектом изучения теории автоматов.

Конечные автоматы

Конечным автоматом (КА) называется пятерка:

$$A = (N, T, P, S, F),$$

где N — конечное множество состояний автомата.

T — входной алфавит — конечное множество символов.

P — функция переходов автомата (в общем случае неоднозначная), отображающая множество пар состояние–входной символ в множество состояний[7].

S — начальное состояние. $S \in N$.

F — множество конечных (финитных) состояний. $F \subseteq N$.

Конечный автомат действует следующим образом. Вначале он находится в состоянии S. На вход КА поступают символы, принадлежащие входному алфавиту. Последовательность входных

[7] Вместо того чтоб считать функцию переходов неоднозначной, можно было бы говорить, что ее значениями для данной пары символ–состояние являются не отдельные состояния, а множества состояний.

символов образует входную цепочку. Находясь в некотором состоянии и получив на вход очередной символ, автомат переходит в следующее состояние, определяемое значением функции переходов для данной пары символ–состояние, и считывает очередной символ. В общем случае функция переходов может определять переход в несколько состояний для данной пары символ–состояние. В этом случае говорят о недетерминированным конечном автомате (НКА). Автомат останавливается, когда заканчиваются символы на его входе.

Если, прочитав входную цепочку α, автомат остановился в некотором состоянии B, говорят, что цепочка α перевела автомат из начального состояния в состояние B. Если B — одно из конечных состояний ($B \in F$), то говорят, что автомат принимает (допускает) цепочку α.

Множество всех цепочек, переводящих конечный автомат A из начального в одно из конечных состояний (*множество цепочек, принимаемых КА*), *образует язык* $L(A)$, *принимаемый* (*допускаемый*) *КА.*

Язык, порождаемый автоматной грамматикой G, совпадает с языком, принимаемым соответствующим конечным автоматом A.

$$L(G) = L(A)$$

Как мы уже видели, КА может задаваться с помощью диаграммы переходов. Например, граф автоматной грамматики G_{10}, показанный на рис. 2.10, может считаться диаграммой переходов автомата A_{10}. При этом $L(G_{10}) = L(A_{10})$.

Преобразование недетерминированного конечного автомата (НКА) в детерминированный конечный автомат (ДКА)

То обстоятельство, что при переходе от автоматной грамматики к КА мы получаем в общем случае НКА, затрудняет его использование в роли распознавателя автоматного языка. Недетерминированность автомата выражается в том, что для некоторых вершин его диаграммы переходов имеется несколько дуг, выходящих из этих вершин и помеченных одним и тем же символом. Так, автомат, изображенный на рис. 2.10, является недетерминированным. Из вершины *N* исходят две дуги, помеченные символом *a*, из вершины *B* — по две дуги, помеченных символами *a* и *b*.

Было бы крайне желательно иметь возможность строить для автоматной грамматики детерминированный конечный автомат.

Теорема Клини. *Для каждого НКА можно построить ДКА, допускающий тот же язык.*

Рассмотрим алгоритм построения ДКА, эквивалентного данному НКА. Для иллюстрации алгоритма будем применять его к НКА A_{10} (см. рис. 2.10), принимающему язык идентификаторов.

1. Пусть исходный НКА имеет *k* состояний. Для построения ДКА возьмем $2^k - 1$ состояний, каждое из которых соответствует одному элементу множества всех подмножеств состояний исходного автомата, кроме пустого множества.

Автомат A_{10} имеет три (k=3) состояния: *N*, *B* и *K*. У нового автомата будет $2^3 - 1 = 7$ состояний. Они соответствуют таким множествам состояний исходного НКА: {*N*}, {*B*}, {*K*}, {*N*, *B*}, {*B*, *K*}, {*N*, *K*}, {*N*, *B*, *K*}. Будем обозначать состояния нового автомата просто последовательностями букв: *N*, *B*, *K*, *NB*, *BK*, *NK*, *NBK*.

2. Из каждого состояния *S* нового автомата направим *не более чем один переход, помеченный данным символом,* в такое состояние, которое соответствует множеству состояний НКА, в которые есть переходы по этому символу хотя бы из одного состояния НКА, образующего *S*.

У НКА A_{10} из состояния *N* есть переход по символу *a* в состояния *B* и *K* (см. рис. 2.10). Следовательно, из состояния *N* нового автомата дугу, помеченную символом *a*, направляем в состояние *BK*.

Рассматривая состояние *NB* нового автомата, выясняем, что переходы из состояния *N* по символу *a* в исходном автомате есть в состояния *B* и *K*, из состояния *B* исходного автомата — также в состояния *B* и *K*. Направляем дугу, помеченную *a*, из состояния *NB* в состояние *BK* (рис. 2.11). Перехода по символу *b* из состояния *N* в исходном автомате нет. Из состояния *B* исходного автомата есть переходы, помеченные *b*, в состояния *B* и *K*. Направляем дугу *b* из состояния *NB* в состояние *BK*.

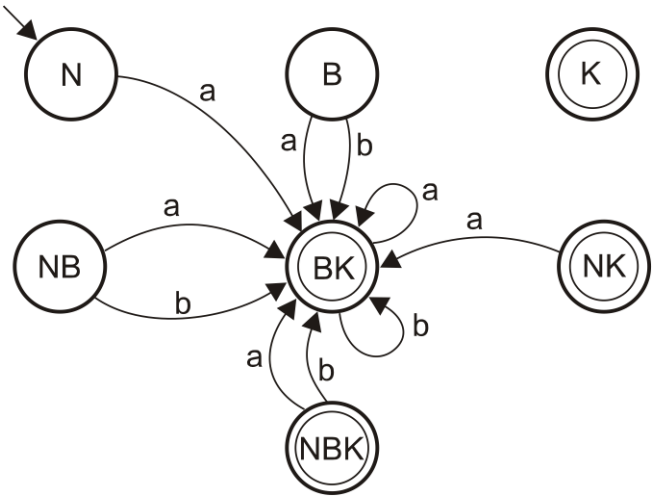


Рис. 2.11. Детерминированный конечный автомат A_{11}, эквивалентный недетерминированному автомату A_{10}

Аналогично формируем переходы из других состояний нового автомата A_{11}, который будет эквивалентен A_{10}. В нашем примере

оказывается, что все дуги, помеченные как символом *a*, так и символом *b* ведут в состояние *BK*.

3. В качестве начального состояния ДКА отметим состояние, имеющее то же обозначение, что и начальное состояние исходного НКА. Как конечные отметим все состояния ДКА, в которые входит хотя бы одно из конечных состояний исходного НКА.

В нашем примере начальным будет состояние *N*. Конечными состояниями ДКА будут все состояния, включающие состояние *K* исходного автомата, то есть *K, BK, NK, NBK* (см. рис. 2.11).

Получившийся автомат является детерминированным (из любого состояния исходит не более одной дуги, помеченной данным символом) и принимает тот же язык, что и исходный недетерминированный автомат.

Детерминированный автомат A_{11} имеет больше состояний, чем исходный НКА A_{10}. Нетрудно, однако, увидеть возможности упрощения получившегося ДКА. Большинство его состояний (*B, K, NB, NK, NBK*) недостижимы из начального состояния, поэтому могут быть отброшены. Получающийся после этого ДКА A_{12} показан на рис. 2.12. Обозначение состояния *BK* упрощено, оно снова названо просто *B*. Этот автомат не только детерминирован, но и проще исходного НКА A_{10}.

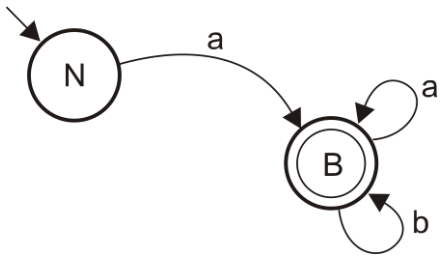


Рис. 2.12. Минимальный ДКА A_{12}, распознающий идентификаторы

По диаграмме переходов получившегося автомата можно снова записать автоматную грамматику, порождающую язык иденти-

фикаторов, эквивалентную грамматике G_{10}, но содержащую меньше правил:

$$G_{12}: \quad N \to aB$$

$$B \to aB \mid bB \mid \varepsilon$$

Кстати, смысл нетерминала B в новой грамматике сохранился — это «хвост», завершающий идентификатор.

Детерминированный конечный автомат можно рассматривать как распознаватель автоматного языка — устройство, с помощью которого просто и эффективно решается задача разбора для автоматной грамматики. В связи с этим, наряду с автоматами принимающими (допускающими) некоторый язык, будем говорить об автоматах, *распознающих* язык.

Для любого автоматного языка можно построить детерминированный конечный автомат, распознающий этот язык.

Задача получения возможно более простого ДКА также имеет общее решение.

Для любого автоматного языка можно построить единственный ДКА, распознающий этот язык и имеющий минимально возможное число состояний.

С алгоритмом построения ДКА с минимальным числом состояний можно познакомиться в книгах [Ахо, 2001], [Карпов, 2002], [Хопкрофт, 2002].

Таблица переходов детерминированного конечного автомата

Наряду с представлением графом, функция переходов ДКА может быть задана таблицей, что, безусловно, больше подходит для программной реализации конечного автомата (табл. 2.1). Рассмотрим таблицу переходов ДКА A_{12}, распознающего язык идентификаторов.

Таблица 2.1. Таблица переходов конечного автомата A_{12}

Состояние	Символ	
	a	***b***
N	*B*	*E*
B	*B*	*B*
E	*E*	*E*

В таблице записано состояние, в которое переходит автомат, находясь в состоянии, соответствующем данной строке таблицы и, получив входной символ, обозначенный в соответствующем столбце. Конечное состояние автомата ***B*** помечено в таблице жирным шрифтом.

Наряду с состояниями *N* и *B* предусмотрено дополнительное состояние *E* — состояние ошибки. Это сделано для того, чтобы функция переходов была определена для всех возможных пар символ–состояние. Иначе переход из состояния *N* при поступлении на вход символа *b* был бы не определен. Попав в состояние *E*, автомат остается в нем. Состояние *E* не является конечным. На практике при программной реализации, кроме символов входного алфавита, потребуется, скорее всего, определить реакции автомата и на любые другие символы, которые, очевидно, должны переводить автомат в состояние *E*.

Программная реализация автоматного распознавателя

В листинге 2.1 приведен эскиз программы, моделирующей работу детерминированного конечного автомата. Эта программа и является универсальным распознавателем (синтаксическим анализатором) автоматных языков. В ней есть лишь некоторые условности: предполагается, что состояния обозначаются латинскими буквами (*S* — начальное состояние), а входной алфавит — малые латинские буквы. Не конкретизированы также способы считывания символов и проверки их наличия на входе,

а также то, как автомат реагирует на принятие или непринятие входной цепочки — эти части программы записаны по-русски.

Листинг 2.1. Универсальный распознаватель автоматных языков

```
type
   tCondition = (S, A, B, C, …, E, …); { Состояния }
   tAlpha     = 'a'..'z';              { Алфавит   }
   tJump      = array [tCondition, tAlpha] of tCondition;
      { Таблица переходов }
   tFinish    = set of tCondition;
      { Тип множества конечных состояний }
var
   Cond : tCondition;      { Текущее состояние  }
   Ch   : tAlpha;          { Входной символ     }
   P    : tJump;           { Функция переходов  }
   Fin  : tFinish;         { Конечные состояния }
…
{ Здесь задаются значения P и Fin }
…
Cond := S;
while Есть символы do begin
   Читать(Ch);
   Cond := P[Cond, Ch]
end;
if Cond in Fin then
   Цепочка принята
else
   Цепочка не принята
```

Дерево разбора в автоматной грамматике

Говоря о задаче синтаксического анализа, мы сводили ее к построению дерева разбора. Между тем, в предыдущих разделах на роль распознавателя автоматных языков предложен конечный автомат, который дерево не строит. Нет ли здесь противоречия? Нет. Дерево разбора цепочки в автоматной грамматике может быть однозначно построено, если известна последовательность переходов конечно-автоматного распознавателя. То есть распознающий автомат не только дает ответ на вопрос о принадлежности цепочки языку, но и позволяет выявить структуру цепочки. Структура при этом представлена последовательностью переходов автомата.

Рассмотрим, какой вид имеет дерево разбора терминальной цепочки в автоматной грамматике. Из трех видов правил автоматной грамматики правила вида $A \to a$ и $A \to \varepsilon$ могут быть использованы в процессе порождения цепочки ровно один раз, после чего процесс порождения заканчивается. Все остальные подстановки выполняются по правилам вида $A \to aB$. Каждая такая подстановка приводит к появлению в дереве новой внутренней вершины, помеченной нетерминалом. Её левая дочерняя вершина помечается терминалом (рис. 2.13).

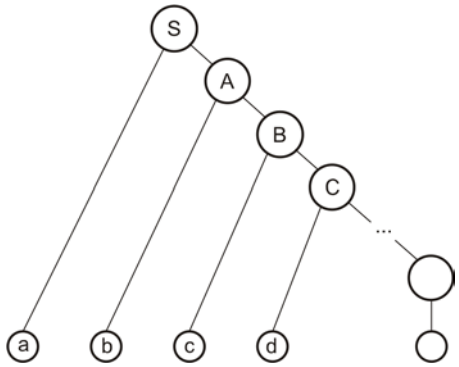


Рис. 2.13. Дерево разбора в автоматной грамматике

Если для грамматики типа 3 построен ДКА, то входная цепочка однозначно определяет последовательность проходимых автоматом состояний и дерево вывода.

Пример автоматного языка

Рассмотрим язык целых чисел со знаком. Примеры правильно записанных чисел:

```
177  +22   -1   0   02
```

Построим конечный автомат, который распознает этот язык. Зададим этот автомат с помощью диаграммы переходов. Это диаграмма будет служить также и формальным определением самого языка.

Начальное состояние автомата обозначим *S* (рис. 2.14). Находясь в этом состоянии, автомат ожидает символ, с которого может начинаться запись числа. Это знаки «+», «–» и цифры. Соответственно, из состояния *S* должны исходить дуги, помеченные этими символами. Десять дуг, помеченных цифрами от 0 до 9, заменим одной, пометив ее символом *ц*. После того как принят знак числа, автомат должен перейти в состояние, в котором он ожидает первую цифру. Обозначим такое состояние *A*. Таким образом, переходы по символам «+» и «–» ведут из состояния *S* в состояние *A*.

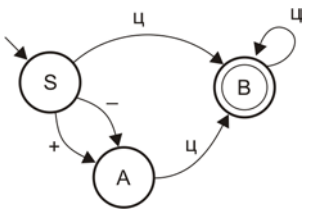


Рис. 2.14. Конечный автомат, распознающий целые числа со знаком

Если, находясь в начальном состоянии *S*, автомат получил цифру, он должен перейти в состояние (обозначим его *B*), в котором могут быть приняты последующие цифры, если они есть. Из состояния *A* при получении цифры автомат также переходит в состояние *B*. Состояние *B* следует пометить как конечное, поскольку переход в это состояние означает, что на вход автомата поступила правильная запись целого числа. Дуга, помеченная символом *ц*, ведущая из состояния *B* в него же, позволяет автомату принять вторую и последующие цифры числа, если они есть.

По диаграмме переходов можно записать и грамматику, порождающую язык целых чисел со знаком. Каждой дуге соответствует правило. Конечные состояния порождают правила с пустой цепочкой в правой части.

$$S \to +A \mid -A \mid цB$$

$$A \to цB$$

$$B \to цB \mid \varepsilon$$

На практике, как уже говорилось, приходится учитывать возможность поступления на вход автомата не только символов входного алфавита, но и любых других символов. В этой ситуации можно предполагать, что из любого состояния исходит дуга, ведущая в состояние ошибки *E* (рис. 2.15). Кроме того, удобно считать, что входная цепочка всегда завершается специальным символом «конец текста», который обозначают ⊥.

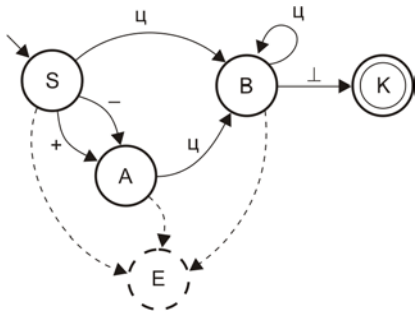


Рис. 2.15. Конечный автомат с состоянием ошибки и дополнительным конечным состоянием

При использовании символа ⊥ к автомату следует добавить состояние *K*, которое будет единственным конечным, и в которое из «бывших» конечных состояний будут направлены дуги, помеченные ⊥.

Не составило бы труда заполнить для полученного автомата таблицу переходов и использовать универсальный распознаватель (см. листинг 2.1), моделирующий поведение ДКА. Но здесь мы рассмотрим другую возможность.

Пользуясь диаграммой переходов (см. рис. 2.14 и рис. 2.15), напишем программу, которая ведет себя подобно конечному автомату, но не моделирует его поведение напрямую.

В начале работы, то есть, находясь в исходном состоянии, программа считывает первый символ входной цепочки и проверяет его допустимость:

```
Читать(Символ);
if not (Символ in ['+', '-', '0'.. '9']) then
   Ошибка
```

Будем считать, что обращение к процедуре `Ошибка` останавливает работу распознавателя. Далее (по-прежнему оставаясь в начальном состоянии) проверяем, какой именно из допустимых символов поступил на вход. Если это знак, читаем следующий символ, переходя к состоянию *A* и предусматривая действия, которые автомат выполняет, находясь в этом состоянии:

```
else if Символ in ['+', '-'] then begin
   Читать(Символ);
   { Состояние A }
   if Символ in ['0'..'9'] then
      Читать(Символ)
   else
      Ошибка
   end
else { Символ - цифра (в состоянии S) }
   Читать(Символ);
{ Состояние B }
```

Если в состоянии *S* поступил символ цифры, считывается следующий входной символ и происходит переход к состоянию *B*. После выполнения приведенного выше фрагмента программа или останавливается по причине ошибки, или приходит в состояние, аналогичное состоянию *B* конечного автомата. Находясь в состоянии *B*, автомат должен принимать все поступившие на вход цифры, оставаясь при этом в состоянии *B*:

```
{ Состояние B }
while Символ in ['0'.. '9'] do
   Читать(Символ);
```

Выход из цикла происходит, если очередной считанный символ — не цифра. Если это символ ⊥ — «конец текста», то входная цепочка закончена и автомат переходит в состояние *K* (см.

рис. 2.15), принимая входную цепочку. Если цикл прекращен по причине поступления символа, отличного от цифры и ⊥, автомат переходит в состояние ошибки:

```
if Символ = ⊥ then
   Цепочка принята
else
   Ошибка;
```

Как видим, распознаватель автоматного языка, каким и является наша программа, можно написать, не прибегая к прямому моделированию поведения конечного автомата с использованием таблицы переходов. Такой подход может иметь свои преимущества. Технология программирования распознавателя оказывается довольно простой: программа пишется по диаграмме переходов ДКА, которая исполняет роль схемы алгоритма.

Синтаксические диаграммы автоматного языка

В построенной программе-распознавателе состояния конечного автомата не фигурировали явно. Они просто соответствовали некоторым точкам программы. В противоположность состояниям, символы, которыми помечены дуги диаграммы переходов, явно используются в распознавателе.

Устраним с диаграммы переходов обозначения состояний (заглавные буквы и кружки). Те места диаграммы, где были состояния автомата, превращаются в точки ветвления и соединения дуг. Терминальные символы, отмечающие переходы-дуги, наоборот разместим в кружках на этих дугах. Результат такого преобразования для диаграммы переходов автомата, распознающего целые числа, или, что то же самое, для графа автоматной грамматики, порождающей целые числа, показан на рис. 2.16а. Полученный граф носит название *синтаксической диаграммы* автоматного языка или автоматной грамматики.

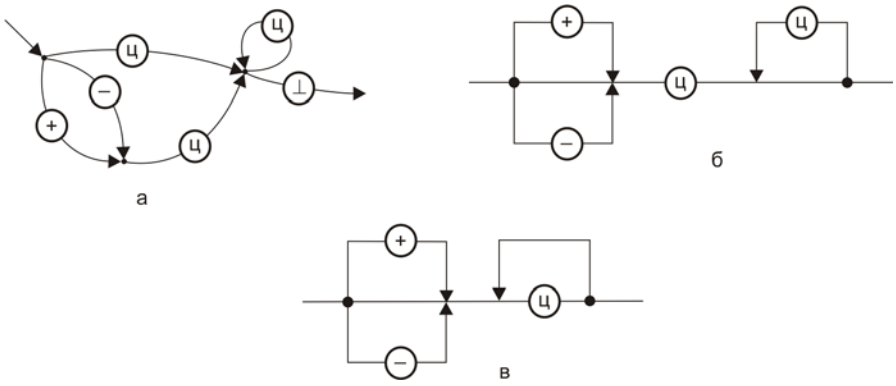


Рис. 2.16. Синтаксические диаграммы грамматики целых чисел

Синтаксическая диаграмма может использоваться для тех же целей, что грамматика и граф переходов автомата, то есть для порождения и для распознавания предложений языка. Любой путь от входа диаграммы (соответствует начальному состоянию автомата) к её выходу (конечному состоянию *K*) порождает цепочку символов, являющуюся правильным предложением языка (сентенцией грамматики). Решение же задачи распознавания сводится к поиску такого пути от входа к выходу диаграммы, который соответствует заданной цепочке.

По сравнению с порождающей грамматикой и конечным автоматом синтаксические диаграммы гораздо наглядней и лучше подходят для спецификации языка при его конструировании. Очень удобна диаграмма в роли схемы алгоритма при написании синтаксического анализатора.

Используя другую манеру начертания дуг, переместив символ *ц* на дугу, ведущую в бывшее состояние *B* и отказавшись от явного изображения конца текста, получим более простую диаграмму, задающую синтаксис целых со знаком (рис. 2.16б). Упрощение диаграммы можно продолжить и дальше (рис. 2.16в), сохраняя ее эквивалентность исходной (рис. 2.16а). Однако, при программировании анализатора на Паскале эта последняя диаграмма не удобней предыдущей.

Перепишем анализатор целых чисел, пользуясь синтаксической диаграммой, показанной на рис. 2.16б, как схемой алгоритма (листинг 2.2). Чтобы избавиться от условностей будем считать, что программа имеет доступ к глобальной переменной `Ch`, которая хранит текущий символ. Чтение следующего входного символа выполняет процедура `NextCh`, которая помещает прочитанное значение в переменную `Ch`. Считается, что константа `EOT` (от End Of Text) обозначает конец текста. Реакция на ошибку возложена на процедуру `Error`, которая выдает сообщение об ошибке и останавливает работу программы-распознавателя. В случае принятия входной цепочки никакого специального сообщения не предусматривается.

Листинг 2.2. Распознаватель целых со знаком

```
NextCh; { Прочитать первый символ }

if Ch in ['+', '-'] then
   NextCh;

if Ch in ['0'..'9'] then
   NextCh
else
   Error;

while Ch in ['0'..'9'] do
   NextCh;

if Ch <> EOT then
   Error;
```

Следует отметить ряд важных черт получившейся программы. Она состоит из нескольких частей, разделенных в листинге пустыми строками. Первая и последняя части, как нетрудно понять, должны присутствовать всегда: перед началом анализа надо получить первый символ, а по завершении — убедиться, что в момент, соответствующий выходу из диаграммы, входная цепочка исчерпана.

Три других части строго соответствуют структуре синтаксической диаграммы (см. рис. 2.16б). На диаграмме выделяются три последовательно соединенных участка, и программа содержит три последовательно записанных и выполняемых фрагмента. Первый из них (`if-then`) проверяет наличие (необязательного) знака. Второй (`if-then-else`) — наличие обязательной цифры. Цикл `while` (который, как известно, может не выполниться ни разу) соответствует циклу на диаграмме, задающему последовательность из нуля или более цифр.

Программа-распознаватель может быть написана по синтаксической диаграмме автоматной грамматики с использованием формальных приемов.

Регулярные выражения и регулярные множества

Регулярные выражения — альтернативный, отличный от порождающих грамматик и синтаксических диаграмм и имеющий свои преимущества, способ задания языка. Регулярное выражение обозначает (порождает) множество цепочек, которое называют *регулярным множеством*. Множество цепочек, соответствующее регулярному выражению R, будем обозначать $R^\wedge$.

Регулярные выражения над алфавитом Σ образуются по следующим правилам:

1. Отдельный символ алфавита $a \in \Sigma$ является регулярным выражением. Обозначаемое таким выражением множество цепочек есть $\{a\}$, то есть состоит из одной цепочки a.
2. Пустая цепочка ε есть регулярное выражение. Обозначает регулярное множество $\{\varepsilon\}$.
3. Если R и Q — регулярные выражения над алфавитом Σ, то запись RQ (конкатенация) также является регулярным выражением. Множество, обозначаемое RQ, состоит из всех цепочек, образованных конкатенацией двух цепочек, так, что первая

цепочка пары порождается выражением R, а вторая — выражением Q. Формально это может быть записано так: $(RQ)^\wedge = \{\alpha\beta \mid \alpha \in R^\wedge, \beta \in Q^\wedge\}$.

4. Если R и Q — регулярные выражения над алфавитом Σ, то запись $R \mid Q$ (читается «R или Q») также является регулярным выражением и обозначает регулярное множество $R^\wedge \cup Q^\wedge$, то есть множество всех цепочек, порождаемых как выражением R, так и выражением Q.

5. Если R — регулярное выражение над алфавитом Σ, то запись R^* (итерация R) также является регулярным выражением, и обозначает множество всех цепочек, полученных повторением цепочек, порождаемых R, ноль или более раз.

6. Если R — регулярное выражение над алфавитом Σ, то (R) (R в скобках) также является регулярным выражением, которое обозначает то же множество, что и R.

Предполагается определенный приоритет операций, с помощью которых образуются регулярные выражения. Наивысший приоритет имеет итерация (знак «*»), далее — конкатенация, далее — «или» (знак «|»). Скобки используются для изменения порядка операций.

Пример 1. С помощью регулярного выражения можно задать правила записи целых чисел со знаком:

$$(+ \mid - \mid \varepsilon)\ ц ц^*,$$

где *ц* обозначает любую цифру от 0 до 9.

Если это не вызывает разночтения, символ ε можно не записывать. Повторение *один или более раз* иногда обозначают знаком «$^+$». $R^+ = RR^*$. Другая форма выражения, определяющего целые:

$$(+ \mid - \mid)\ ц+.$$

Нетрудно, впрочем, записать выражение, обозначающее множество всех целых со знаком, не прибегая к условному обозначению цифр с помощью «*ц*»:

$$(+ \mid - \mid) \; (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^{+}$$

Пример 2. Регулярное выражение, задающее множество идентификаторов:

$$б \; (б \mid ц)^{*}.$$

где *б* — буква; *ц* — цифра.

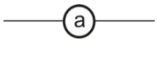
Эквивалентность регулярных выражений и автоматных грамматик

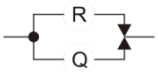
Автоматные языки являются регулярными множествами. Регулярные множества являются автоматными языками.

Сказанное означает, что для любой автоматной грамматики можно записать такое регулярное выражение, что обозначаемое этим выражением множество цепочек совпадает с языком, порождаемым грамматикой. И, наоборот, для любого регулярного выражения можно найти автоматную грамматику, порождающую то же множество цепочек, что и регулярное выражение.

Будем считать, что автоматный язык задается синтаксической диаграммой. Можно установить взаимно однозначное соответствие между конструкциями, из которых строятся регулярные выражения (правила 1–6) и фрагментами, из которых состоят синтаксические диаграммы автоматных грамматик. Это соответствие показано в таблице 2.2.

Таблица 2.2. Эквивалентность регулярных выражений и автоматных грамматик

Номер правила	Фрагмент выражения	Участок диаграммы
1	*a*	

Номер правила	Фрагмент выражения	Участок диаграммы
2	ε	
3	RQ	
4	$R \mid Q$	
5	R^*	
6	(R)	

Используя такое соответствие, по выражению можно построить диаграмму, а по диаграмме — регулярное выражение. Уточнение деталей таких построений [Свердлов, 2008], которым мы не будем здесь заниматься, и доказывает справедливость сформулированного выше утверждения об эквивалентности.

Уместно напомнить, что автоматные грамматики называют также *регулярными*.

Для чего нужны регулярные выражения

Автоматные грамматики, регулярные выражения и синтаксические диаграммы являются эквивалентными способами задания автоматных языков. По сравнению с грамматиками синтаксические диаграммы обладают большей наглядностью. Регулярные же выражения имеют то важное достоинство, что представляют собой строки символов, которые могут быть легко обработаны с помощью компьютерных программ.

Обработка регулярного выражения, выступающего в роли исходных данных для некоторой программы, может иметь целью его анализ, преобразование и даже… создание распознавателя автоматного языка, порождаемого этим выражением. Последнее представляет безусловный интерес, поскольку открывает возможность автоматизации построения синтаксических анализато-

ров. Работа подобной программы может происходить по одной из схем, показанных на рисунке 2.17.

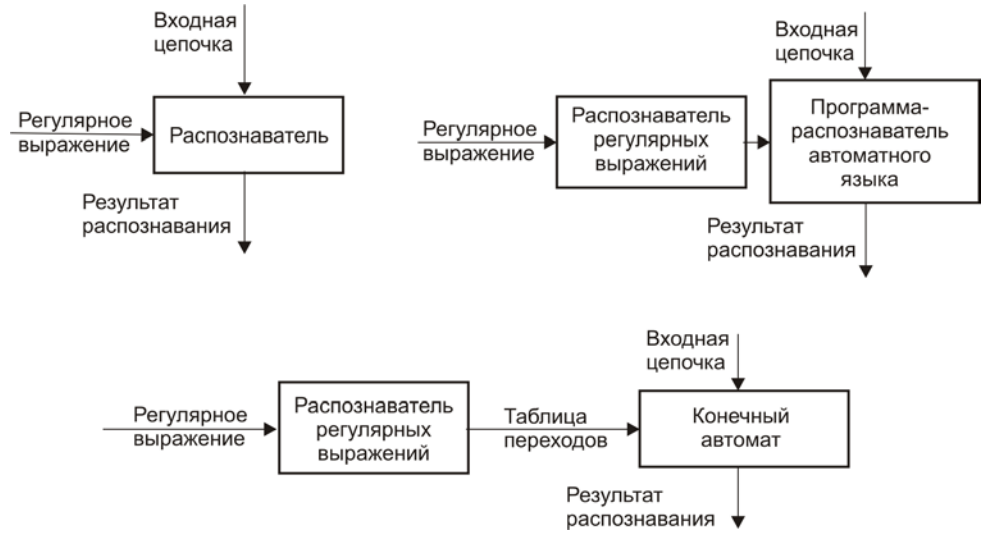


Рис. 2.17. Использование регулярных выражений

Регулярные выражения наглядней порождающих грамматик. Это обусловлено тем, что предусмотрено явное обозначение повторения (знак итерации «*»). В нотации грамматик итерация задается с помощью рекурсии. Сравните, например, грамматики G_{10} и G_{12}, задающие язык идентификаторов, с эквивалентным регулярным выражением из примера 2.

Регулярные выражения как языки

Регулярное выражение над алфавитом Σ — это цепочка символов в расширенном алфавите $\Sigma \cup \{ |, ^*, (,) \}$. Множество всех таких цепочек-выражений образует язык.

Возникает естественный вопрос, к языкам какого типа по классификации Н. Хомского этот язык принадлежит. К автоматным? Тогда, быть может, правила записи регулярных выражений можно задать регулярным выражением? Нет, нельзя. Синтаксис регулярных выражений может быть определен только контекст-

но-свободной, но не автоматной грамматикой. Вот эта грамматика:

$$R \to a \mid RR \mid R^* \mid R\,''|''\,R \mid (\,R\,) \mid \varepsilon$$

В этой записи есть ряд условностей: *a* обозначает любой символ алфавита Σ, запись ″|″, представляет знак «|», используемый в регулярных выражениях и совпадающий с аналогичным знаком, применяемым при записи грамматик.

Приведенная грамматика не отражает принятый для регулярных выражений порядок операций. Грамматика, трактующая структуру регулярного выражения в соответствии с приоритетами операций, может быть записана так:

$$R \to T \mid R\,''|''\,T$$

$$T \to M \mid RM$$

$$M \to a \mid M^* \mid (\,R\,) \mid \varepsilon$$

Расширенная нотация для регулярных выражений

Регулярные выражения — это строки символов, и тем они интересны как средство задания автоматных языков. Но использование надстрочных знаков «*» и «$^+$» несколько затрудняет запись выражений и их считывание компьютерной программой. Получили распространение другие варианты обозначений. Повторение ноль или более раз обозначают фигурными скобками:

$$R^* = \{R\}.$$

Используются также квадратные скобки, обозначающие необязательность заключенного в них выражения:

$$[\,R\,] = (\,R \mid \varepsilon\,).$$

Знаки «*» и «$^+$» в этом случае уже не используются. Соглашения о способах записи символов, с помощью которых строятся сами выражения (скобки, знак «|»), в случае, если они также входят в

терминальный алфавит, могут быть разными. Можно заключать такие метасимволы в кавычки «″». При необходимости записать саму кавычку ее заключают в апострофы «′», а апостроф, если нужно, записывается в кавычках.

По этим правилам регулярные выражения, обозначающие множество целых со знаком и множество идентификаторов, будут выглядеть так:

$$[+ \mid -]\ ц\ \{ц\}$$

$$б\ \{б \mid ц\}.$$

На этом мы заканчиваем рассмотрение автоматных грамматик, в ходе которого удалось построить простые и эффективные методы распознавания автоматных языков.

С помощью автоматных грамматик определяется синтаксис простейших элементов языков программирования: идентификаторов, чисел, других констант, знаков операций и разделителей.

Контекстно-свободные (КС) грамматики и языки

К классу контекстно-свободных относятся грамматики, у которых не накладывается никаких ограничений на вид правых частей их правил, а левая часть каждого правила — единственный нетерминал. С помощью КС-грамматик задают синтаксис языков программирования.

Однозначность КС-грамматики

Как уже формулировалось выше, однозначной называется грамматика, в которой каждой сентенции соответствует единственное дерево вывода. Однако, как мы видели, дерево вывода не всегда строится явно в ходе решения задачи разбора. Поэтому имеет смысл сформулировать определение однозначности без привлечения понятия «дерево вывода».

Левосторонние и правосторонние выводы в КС-граматике

Рассмотрим (однозначную) грамматику G_8, задающую синтаксис арифметических выражений.

$$G_8: \quad E \to T \mid E + T \mid E - T$$
$$T \to M \mid T * M \mid T / M$$
$$M \to a \mid b \mid c \mid (E)$$

Построим два различных вывода цепочки $a + b*c$ в этой грамматике. В первом случае, если сентенциальная форма содержит более одного нетерминала, будем выполнять подстановку (замену нетерминала правой частью одного из правил) для самого левого нетерминала этой сентенциальной формы:

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow M + T \Rightarrow a + T \Rightarrow a + T*M \Rightarrow a + M*M \Rightarrow$$
$$a + b*M \Rightarrow a + b*c$$

Такой вывод называется **левосторонним**. Аналогично, вывод, в ходе которого замене всегда подвергается самый правый нетерминал сентенциальной формы, называется **правосторонним**. Для цепочки $a + b*c$ в грамматике G_8 он будет таким:

$$E \Rightarrow E + T \Rightarrow E + T*M \Rightarrow E + T*c \Rightarrow E + M*c \Rightarrow E + b*c \Rightarrow$$
$$T + b*c \Rightarrow M + b*c \Rightarrow a + b*c$$

Нетрудно убедиться, что обе эти последовательности подстановок соответствуют одному и тому же дереву вывода, хотя и разному порядку его построения. Для цепочки $a + b*c$ в грамматике G_8 и левосторонний и правосторонний вывод строятся однозначно. Это же справедливо и для любой другой цепочки, порождаемой G_8.

КС-грамматика называется однозначной, если для каждого предложения языка, порождаемого этой грамматикой, существует единственный левосторонний вывод.

Алгоритмы распознавания КС-языков

Существует алгоритм, позволяющий для любой КС-грамматики, проверить принадлежность произвольной цепочки терминальных символов языку, порождаемому этой грамматикой, и получить вывод этой цепочки.

Наличие такого алгоритма и принцип его устройства следуют из того простого соображения, что, если имеется конечная цепочка терминалов, то для проверки ее принадлежности языку достаточно построить все возможные сентенциальные формы грамматики, имеющие длину, совпадающую с длиной этой цепочки. Количество этих сентенциальных форм конечно. Такого рода алгоритм действует по принципу полного перебора. Объем перебора может быть сокращен, если процесс порождения сентенциальных форм будет организован так, что станет возможным определить тупики в ходе перебора еще до получения сентенциальной формы нужной длины.

Переборные алгоритмы работают с возвратами и вследствие своей неэффективности мало пригодны для практического использования. Для организации перебора с возвратами используют стек — структуру данных, подчиняющуюся дисциплине «последним пришел — первым ушел» (LIFO — Last In First Out).

В то же время для достаточно обширных подклассов КС-грамматик, подчиняющихся некоторым дополнительным ограничениям, существуют эффективные алгоритмы распознавания, которые мы будем в дальнейшем рассматривать и применять.

Для синтаксического анализа КС-языков используются как нисходящие (строящие дерево разбора от корня к листьям), так и восходящие алгоритмы.

Распознающий автомат для КС-языков

Для автоматных языков роль распознавателя может выполнять детерминированный конечный автомат. Существует ли универсальный автоматный распознаватель КС-языков? Да, существует.

Для произвольной КС-грамматики может быть построен недетерминированный автомат с магазинной памятью, принимающий язык, порождаемый этой грамматикой.

Автомат с магазинной памятью (магазинный автомат, МП-автомат) подобен конечному автомату, оснащенному дополнительным запоминающим устройством со стековой[8] дисциплиной «последним пришел — первым ушёл». Переходы МП-автомата определяются не только входным символом и текущим состоянием, но и значением вершины стека — элемента, поступившего в магазин последним.

Недетерминированный МП-автомат — это не что иное, как устройство, реализующее перебор с возвратами. В этом смысле он эквивалентен обсуждавшемуся выше общему алгоритму распознавания КС-языков и так же неэффективен. Для КС-грамматик, подчиняющихся определенным ограничениям, могут быть построены эффективные детерминированные магазинные автоматы, которые могут использоваться на практике для трансляции языков программирования.

Самовложение в КС-грамматиках

Если в грамматике *G* есть нетерминал *A*, для которого

[8] В литературе на русском языке стек часто называют «магазином» по аналогии с магазином автоматического оружия: патрон, заряженный в магазин последним, выстреливается первым.

$$A \overset{+}{\underset{G}{\Rightarrow}} \alpha_1 A \alpha_2,$$

то есть из A нетривиально выводится цепочка $\alpha_1 A \alpha_2$, где α_1, α_2 — непустые цепочки терминалов и нетерминалов, то говорят, что такая грамматика содержит самовложение.

Например, грамматика арифметических выражений G_8 содержит самовложение, поскольку из ее начального нетерминала E выводится цепочка (E).

$$E \Rightarrow T \Rightarrow M \Rightarrow (E).$$

Содержит самовложение и грамматика регулярных выражений, поскольку: $R \overset{+}{\Rightarrow} (R)$.

Самовложение — характерный признак КС-грамматик.

КС-грамматика, не содержащая самовложения, эквивалентна автоматной грамматике.

Языки арифметических и регулярных выражений являются контекстно-свободными и не могут быть заданы автоматными грамматиками.

Синтаксические диаграммы КС-языков

Синтаксические диаграммы КС-языка могут быть построены по его грамматике на основании следующих правил:

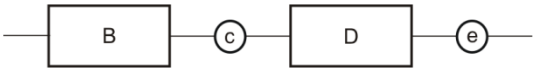
1. Для каждого нетерминала грамматики строится отдельная диаграмма, обозначенная названием этого нетерминала.
2. Нетерминалы из правых частей правил изображаются на диаграммах прямоугольниками, внутри которых записывается название нетерминала. Терминальные символы изображаются в кружках или овалах.
3. Для каждой правой части правила строится ветвь, представляющая собой последовательно соединенные прямоугольники

и круги (овалы), следующие в том же порядке слева направо, что и соответствующие нетерминалы и терминалы правой части правила.

4. Ветви, соответствующие альтернативным правым частям правил для одного нетерминала, соединяются параллельно и образуют диаграмму для данного нетерминала.

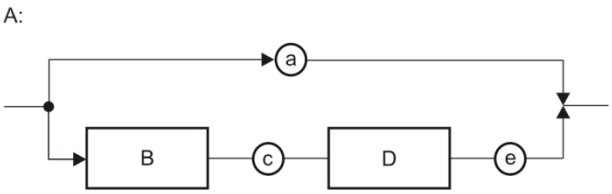
Рассмотрим примеры построения диаграмм. Пусть в некоторой грамматике имеется правило $A \to a$, тогда на диаграмме для нетерминала A будет ветвь

Правило $A \to BcDe$ порождает ветвь

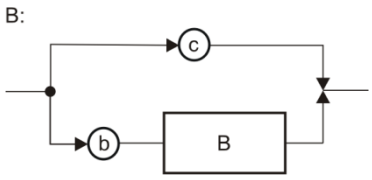


Если других правил для нетерминала A в грамматике нет, то диаграмма для этого нетерминала получается параллельным соединением ветвей. Правила для A удобней объединить в одно с альтернативными правыми частями:

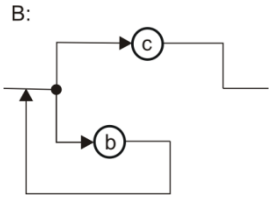
$$A \to a \mid BcDe.$$



Продолжим пример. Поскольку в правилах для A фигурируют нетерминалы B и D, то в грамматике должны быть правила, в которых B и D записаны в левой части. Пусть правило для B имеет вид: $B \to c \mid bB$. Тогда строится такая диаграмма:

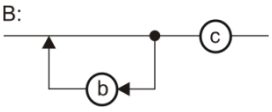


Можно, однако, заметить, что правила для *B* удовлетворяют ограничениям автоматных грамматик. А синтаксические диаграммы автоматных грамматик не должны содержать нетерминалов. Противоречия нет. Диаграмма для *B* может быть преобразована. Поскольку прохождение прямоугольного блока, обозначающего *B*, равносильно (порождает такую же цепочку терминалов) повторному входу в диаграмму, вход в блок *B* можно заменить повторным входом в диаграмму.



Такое преобразование, устранившее с диаграммы нетерминальный блок *B*, стало возможным благодаря тому, что нетерминал *B* был самым правым символом в одной из альтернативных правых частей правил для *B*. В результате преобразования концевая (правая) рекурсия заменена циклом.

Выполнив элементарное преобразование, можно нарисовать диаграмму нетерминала *B* в традиционном виде:

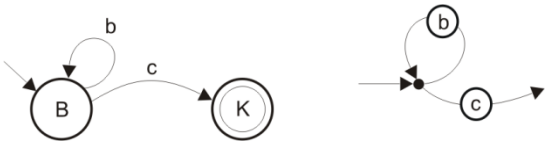


Замена правой рекурсии циклом всегда возможна (и желательна) при построении синтаксических диаграмм КС-грамматики[9]. Ин-

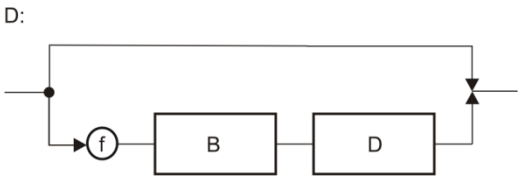
[9] Замена концевой рекурсии циклом возможна и в программах. Если в некоторой процедуре последним выполняется рекурсивный вызов этой процедуры, то он может быть заменен переходом к началу процедуры, то есть циклом.

тересно заметить, что сами правила 1–4 не предусматривают циклов, в то время как на практике циклы на диаграммах имеются почти всегда.

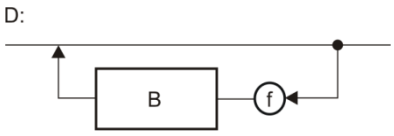
Такую же диаграмму для B можно было получить, построив фрагмент конечного автомата, а затем устранив из него состояния.



Завершая пример, зададим правило для нетерминала D: $D \rightarrow fBD \mid \varepsilon$ и построим диаграмму.



Наличие пустой цепочки в одной из альтернативных правых частей правила приводит к появлению на диаграмме параллельной ветви, в которой нет символов. Получившаяся диаграмма может быть, однако, снова упрощена:



Определение языка с помощью синтаксических диаграмм

В действительности синтаксические диаграммы строятся, как правило, не по имеющейся грамматике, а служат самостоятельным средством проектирования языков, в том числе и языков программирования. При этом язык определяется совокупностью диаграмм, первая из которых соответствует начальному нетерминалу грамматики.

Определять синтаксис в виде совокупности диаграмм, на которых имеются нетерминальные блоки, можно не только для контекстно-свободных, но и для автоматных языков. Только из-за отсутствия самовложения диаграммы автоматного языка всегда можно объединить в одну, не содержащую нетерминалов. Для этого достаточно «подставить» в диаграмму начального нетерминала другие диаграммы вместо соответствующих прямоугольных блоков. Для КС-языка такая подстановка невозможна из-за самовложения.

Язык многочленов

Для примера построим синтаксические диаграммы, задающие правила записи (синтаксис) многочленов от x с постоянными целочисленными коэффициентами, то есть определяющие язык многочленов.

Примеры таких многочленов:

$$5x^3 + x^2 - 12x + 10$$

$$-x$$

$$199$$

Последний пример может вызвать возражение, поскольку не содержит переменной x. Условимся, однако, и такую запись считать правильным многочленом нулевой степени. Чтобы запись многочленов могла быть обработана компьютерной программой (транслятором или вычислителем многочленов), предусмотрим возможность записи символов «в строку» без надстрочных показателей степени. Возведение в степень будем обозначать, как это принято в языке Бейсик и некоторых диалектах Алгола, с помощью знака «^». Тогда первый пример многочлена запишется так:

```
5x^3 + x^2 – 12x + 10
```

Построим синтаксические диаграммы, определяющие правила записи многочленов. Первой будет диаграмма для начального

нетерминала, который в нашем случае есть не что иное как «Многочлен». Многочлен состоит из отдельных слагаемых, между которыми записываются знаки операций. Перед первым слагаемым также можно записать знак. Слагаемых должно быть не меньше одного. С учетом этого получается диаграмма, показанная на рисунке 2.18.

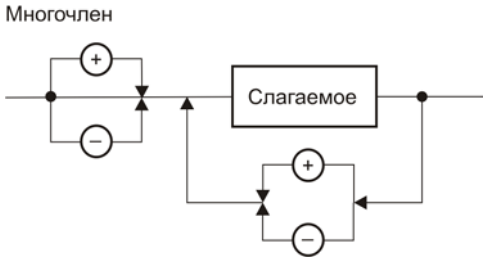


Рис. 2.18. Синтаксическая диаграмма многочлена

Теперь надо построить диаграмму для нетерминала «Слагаемое», который мы ввели в грамматику многочленов. Вначале изобразим ветвь диаграммы, соответствующую полному варианту слагаемого, когда присутствуют все его элементы: коэффициент, буква x, знак возведения в степень и сама степень (рис. 2.19а).

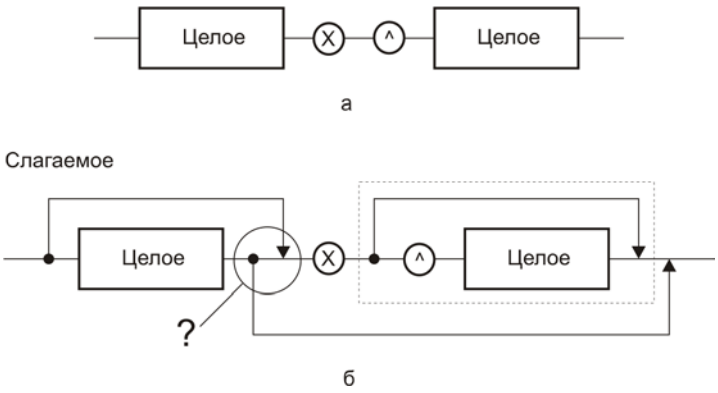


Рис. 2.19. Синтаксическая диаграмма слагаемого

Затем проведем «обходные» ветви, позволяющие предусмотреть такие варианты слагаемого, когда нет коэффициента (предполагается равным единице), буквы *x* и последующей степени, или

только степени (см. рис. 2.19б). При этом не должно появиться такого пути на диаграмме, пройдя по которому мы минуем как коэффициент, так и *x*.

Коэффициент перед слагаемым и показатель степени записываются как целые числа без знака. Соответствующий нетерминал назван «Целое». В дальнейшем мы всегда будем считать (если не оговорено иное), что «целое» означает целое без знака. Целое без знака есть последовательность, состоящая из одной или более цифр (рис. 2.20а). Диаграмма для нетерминала «Цифра» показана на рис. 2.20б.

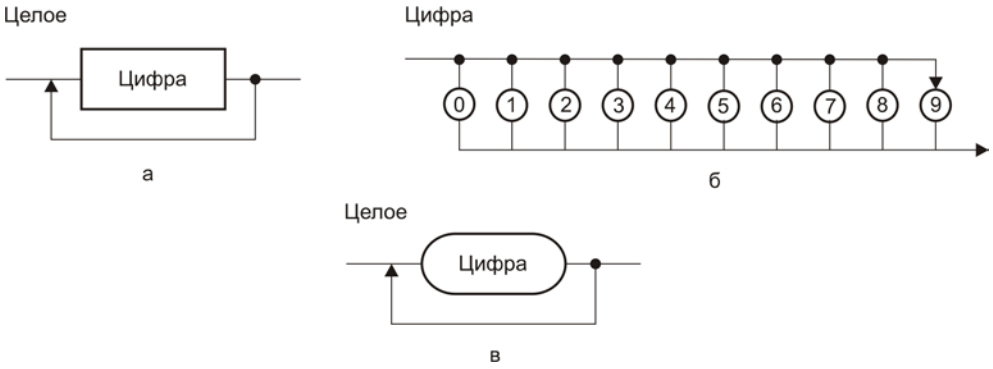


Рис. 2.20. Синтаксические диаграммы для целого и для цифры

Поскольку арабские цифры используются в самых разнообразных языках, было бы неудобно каждый раз приводить диаграмму, подобную изображенной на рис. 2.20б. В дальнейшем будем вместо нетерминального блока «Цифра» использовать на диаграммах овал (рис. 2.20в), считая что «цифра» — это «почти терминальный символ».

Нетрудно понять (хотя бы по отсутствию самовложения), что язык многочленов — автоматный. Все диаграммы можно было бы объединить в одну, не содержащую нетерминальных блоков. Однако делать этого мы не будем. Во-первых, несколько несложных диаграмм воспринимаются проще, чем одна громоздкая. Во-вторых, использование промежуточных понятий, таких

как, например, «Целое», позволяет избежать дублирования: нетерминал «Целое» встречается на диаграмме слагаемого дважды. В-третьих, представив автоматный язык с помощью КС-диаграмм, мы на простом примере рассмотрим методы распознавания КС-языков, не потеряв при этом общности подхода.

Синтаксический анализ КС-языков методом рекурсивного спуска

Рекурсивный спуск — это эффективный и простой нисходящий алгоритм распознавания. Он состоит в следующем.

Для каждого нетерминала грамматики (понятия, конструкции языка), то есть для каждой синтаксической диаграммы, записывается отдельная распознающая процедура. При этом соблюдаются следующие соглашения:

1. Перед началом работы процедуры текущим является первый символ анализируемого понятия (см. рис. 2.21).
2. В процессе работы процедура считывает все символы входной цепочки, относящиеся к данному нетерминалу (выводимые из данного нетерминала) или сообщает об ошибке. Если правила для данного нетерминала содержат в правых частях другие нетерминалы (синтаксическая диаграмма данного нетерминала содержит другие нетерминалы), то процедура обращается к распознающим процедурам этих нетерминалов для анализа соответствующих частей входной цепочки.
3. По окончании работы процедуры текущим становится первый символ, следующий во входной цепочке за данной конструкцией языка (символами, выводимыми из данного нетерминала).

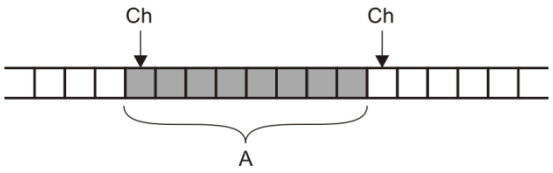


Рис. 2.21. Текущий символ в начале и конце работы распознающей процедуры нетерминала *A*

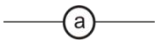
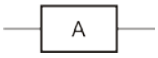
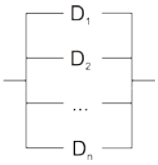
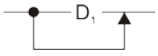
Распознавание начинается вызовом распознающей процедуры начального нетерминала. При этом текущим символом, как это следует из п. 1, должен быть первый символ входной цепочки. По завершении работы начальной процедуры текущим должен быть символ «конец текста». Таким образом, анализ методом рекурсивного спуска всегда строится по следующей схеме:

```
NextCh;          {Чтение первого символа в переменную Ch}
S;       {Вызов распознающей процедуры нач. нетерминала}
if Ch <> EOT then {Проверка исчерпания входной цепочки}
   Error;
```

Название «рекурсивный спуск» обусловлено тем, что при наличии в грамматике самовложения вызовы распознающих процедур будут рекурсивными. Процесс распознавания развивается от начального нетерминала (корень дерева разбора) через вызов процедур для промежуточных нетерминалов (внутренние вершины дерева) к анализу отдельных терминальных символов (листья дерева). Это нисходящий разбор.

Каждая распознающая процедура строится по соответствующей синтаксической диаграмме, которая играет роль схемы алгоритма. Соответствие участков диаграмм и фрагментов распознающих процедур показано в табл. 2.3. В таблице участки диаграмм обозначаются D, D_1, D_2, …, D_n. Соответствующие этим участкам фрагменты программы-распознавателя (распознающих процедур) обозначены $P(D)$, $P(D_1)$, $P(D_2)$, …, $P(D_n)$.

Таблица 2.3. Правила построения распознавателя по синтаксической диаграмме

Диаграмма D	Распознаватель P(D)	Примечание
	`if Ch = 'a' then NextCh else Error;`	Анализ отдельного терминального символа
	`A;`	Анализ нетерминала: вызов распознающей процедуры нетерминала *A*
	`P(D1); P(D2);`	Анализ последовательно соединенных участков диаграммы
	`if Ch in first(D1) then P(D1) else if Ch in first(D2) then P(D2) ... else if Ch in first(Dn) then P(Dn) else Error;`	Ветвление. $first(D_1)$, $first(D_2)$, …, $first(D_n)$ — множества направляющих символов ветвей D_1, D_2, …, D_n. Множество направляющих символов ветви D_i образуют терминальные символы, которые могут встретиться первыми при движении по ветви D_i.
	`if Ch in first(D1) then P(D1);`	Ветвление с пустой альтернативой (необязательное вхождение D_1)
	`while Ch in first(D1) do P(D1);`	Повторение D_1 ноль или более раз

Принцип работы анализатора, который строится по предлагаемым схемам, состоит в том, что, анализируя очередной символ входной цепочки, распознаватель выбирает путь движения по синтаксической диаграмме, соответствующий этой цепочке.

Пример: анализатор многочленов

Пользуясь методом рекурсивного спуска, запрограммируем синтаксический анализатор многочленов, правила записи которых определены диаграммами на рис. 2.18–2.20. Будем считать, что анализатор должен просто отвечать на вопрос, является ли введенная пользователем строка правильно записанным многочленом.

Основная программа анализатора

Вначале необходимо подготовить текст для чтения анализатором. Было бы неправильно делать так, чтобы в основной программе и распознающих процедурах отражались особенности представления входной цепочки, то есть программировать таким образом, чтобы основные части анализатора зависели от того, считываются ли символы из файла, вводятся с терминала, извлекаются из окна редактора текста или получаются как-либо по-другому. Эта специфика будет скрыта в нескольких процедурах, которые только и будут зависеть от конкретного представления входной цепочки. Предусмотрим, что подготовка входного текста к чтению выполняется процедурой `ResetText`:

```
begin
   ResetText;
```

Далее вызываем распознающую процедуру начального нетерминала, которым в нашей задаче является «Многочлен»:

```
   Polynom;
```

Процедура `Polynom` считывает символы входной цепочки, проверяя, соответствует ли их порядок синтаксису многочленов. Если необходимо, она обращается к другим распознающим процеду-

рам. В случае, когда входная цепочка действительно содержит многочлен, процедура Polynom оставляет текущим (содержащимся в переменной Ch) символ, следующий за многочленом. Если при анализе многочлена обнаруживается ошибка, вызывается процедура Error, останавливающая работу всего анализатора.

Для завершения анализа остается проверить, не содержится ли за правильно записанным многочленом во входной цепочке лишних символов, то есть, следует ли за многочленом символ «конец текста»:

```
    if Ch <> EOT then
       Error('Ожидается конец текста')
    else
       WriteLn('Правильно');
    WriteLn;
end.
```

Константы, переменные и вспомогательные процедуры

Теперь можно записать начало программы с описаниями констант и переменных, использованными в основном блоке. Кроме глобальной переменной Ch, обозначающей текущий символ, предусмотрим глобальную переменную Pos, которая будет хранить номер этого символа во входной цепочке.

```
program ParsePoly;
const
   EOT = chr(0);   { Признак "конец текста" }
var
   Ch  : char;     { Очередной символ   }
   Pos : integer;  { Номер символа      }
```

Взаимодействие с входным текстом будут выполнять процедуры ResetText — готовит входную цепочку к считыванию распознавателем, и NextCh — читает очередной символ входной цепочки, помещая его в переменную Ch.

Использование глобальных переменных в процедурах, вообще-то, плохая практика. Хотя мы еще не начали обращаться к `Ch` и `Pos` в процедурах распознавателя, но вот-вот сделаем это. Оправданием служит специфика задачи. Синтаксический анализатор и компилятор — своеобразные программы, в которых используется не слишком много переменных. А в нашем анализаторе переменных будет и вовсе две — только что определенные `Ch` и `Pos`. Их передача в процедуры через параметры загромоздила бы программу, в то время как обращение за очередным символом, например, к процедуре `NextCh` все равно выглядело бы всегда одинаково: `NextCh(Ch, Pos)`.

Предусмотрим чтение исходных данных (записи многочлена) из стандартного входного файла (по умолчанию — ввод с клавиатуры). Сообщения распознавателя будут выводиться в стандартный выходной файл (т.е. на экран). В этом случае процедура, подготавливающая текст, запишется так:

```
{ Подготовить текст }
procedure ResetText;
begin
   WriteLn('Введите многочлен от X с целыми коэфф-тами');
   Pos := 0;
   NextCh; { Чтение первого символа }
end;
```

При чтении очередного символа будем игнорировать пробелы, считая их незначащими. Это позволит при записи исходного многочлена применять пробелы, например, для отделения одного слагаемого от другого.

```
procedure NextCh;
{ Читать следующий символ }
begin
   repeat
      Pos := Pos+1;
      if not eoln then
         Read(Ch)
      else begin
         ReadLn;
         Ch := EOT;
```

```
      end;
    until Ch <> ' ';
  end;
```

Такой пропуск пробелов делает их допустимыми в любом месте записи многочлена. К примеру, такая строка должна теперь считаться правильной:

```
1 2 3 x ^ 4 + 5 6 7 x + 8 9
```

Это не соответствует соглашениям современных языков программирования. Более совершенное решение вопроса о пробелах будет рассмотрено в следующей главе, пока же примем простейший подход, который все же предпочтительней полного запрета пробелов.

Процедура, реагирующая на ошибку, тоже зависит от соглашений по вводу и выводу. Как мы уже условились, она будет выдавать сообщение на экран:

```
procedure Error(Message: string); { Ошибка  }
   { Message - сообщение об ошибке }
begin
   WriteLn('^': Pos);
   WriteLn('Синтаксическая ошибка: ', Message);
   Halt; { Прекращение работы анализатора }
end;
```

Вывод знака `'^'` в позиции `Pos` позволяет указать стрелкой на символ, вызвавший ошибку. Диалог с анализатором может быть таким:

```
Введите многочлен от X с целыми коэфф-тами
2x + 1.2
      ^
Синтаксическая ошибка: Ожидается конец текста
```

Реакция программы в этом примере может показаться непонятной, хотя она совершенно корректна. Получение такого сообщения означает: «Если бы на этом месте запись многочлена закончилась, было бы синтаксически правильно. Но в указанном месте стоит неподходящий символ». Понятно, что распознаватель

не может знать наших желаний и содержательно реагировать на попытку записать вещественное число вместо целого.

Распознающие процедуры

Для каждого нетерминала грамматики многочленов, то есть для каждой синтаксической диаграммы (рис. 2.18 – 2.20) записываем одну распознающую процедуру. Первой — процедуру для нетерминала «Многочлен» — начального нетерминала грамматики (рис. 2.18). Трудность, однако, состоит в том, что диаграмма на рис. 2.18 неудобна для программирования — она содержит цикл с выходом из середины, в то время как в Паскале такого цикла нет. Заменим диаграмму эквивалентной, содержащей цикл с предусловием (с выходом в начале).

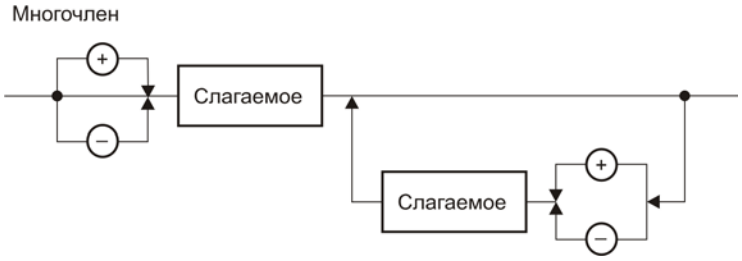


Рис. 2.22. Преобразованная диаграмма «Многочлен»

Теперь не составляет труда записать распознающую процедуру, структура которой в точности повторяет структуру диаграммы: на диаграмме три последовательно соединенных участка — в процедуре три оператора, выполняемых один за другим: **if**, вызов процедуры Addend, цикл **while**.

```
procedure Polynom; { Многочлен }
begin
   if Ch in ['+', '-'] then
      NextCh;
   Addend; { Слагаемое }
   while Ch in ['+', '-'] do begin
      NextCh;
      Addend;
   end;
end;
```

Следующий нетерминал — «Слагаемое». Однако диаграмма, показанная на рис. 2.19, не разделяется на типовые фрагменты, что затрудняет программирование распознавателя. Неструктурированность обусловлена фрагментом, который помечен на рисунке знаком «?». Преобразуем диаграмму в эквивалентную, но состоящую только из совокупности типовых структур (рис. 2.23). Для этого изобразим отдельно две ветви: одна соответствует слагаемому, начинающемуся с числа, другая — с буквы *x*. Фрагмент, выделенный на исходной диаграмме пунктирной рамкой, преобразуем в нетерминал «Степень».

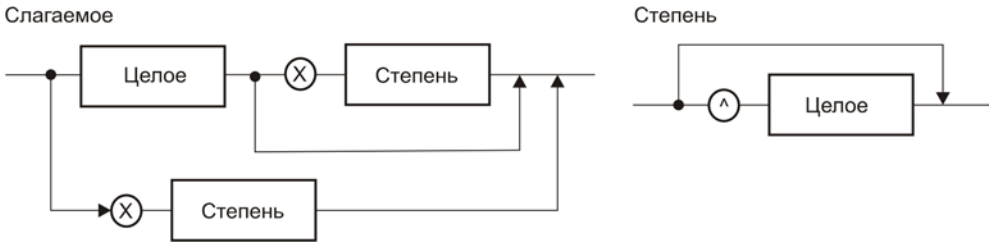


Рис. 2.23. Синтаксические диаграммы «Слагаемое» и «Степень»

Программирование распознающих процедур `Addend` (слагаемое) и `Power` (степень) теперь выполняется легко: диаграммы служат схемами алгоритмов.

```
procedure Addend; { Слагаемое }
begin
   if Ch = 'x' then begin
      NextCh;
      Power;  { Степень }
      end
   else begin
      Number; { Целое }
      if Ch = 'x' then begin
         NextCh;
         Power;
      end;
   end;
end;

procedure Power; { Степень }
begin
   if Ch = '^' then begin
```

```
      NextCh;
      Number;
   end;
end;
```

Полезно обратить внимание на дисциплину вызова процедуры `NextCh`. Следующий символ считывается, когда опознан текущий.

Последняя распознающая процедура — для «Целого». И в этом случае тоже можно преобразовать исходную диаграмму (см. рис. 2.20), отделив блок, соответствующий первой цифре (обязательной), от остальных блоков (рис. 2.24).

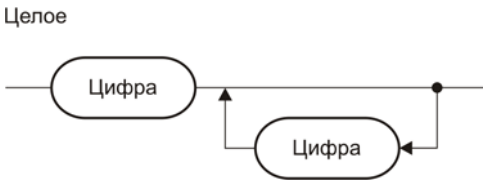


Рис. 2.24. Синтаксическая диаграмма «Целое»

Предусматривать отдельную распознающую процедуру для «почти терминального символа» «цифра» неразумно. Проще и наглядней непосредственно проверять принадлежность очередного знака множеству цифр.

```
procedure Number; { Целое }
begin
   if Ch in ['0'..'9'] then
      NextCh
   else
      Error('Число начинается не с цифры');
   while Ch in ['0'..'9'] do
      NextCh;
end;
```

Распознаватель готов. Осталось только расположить в программе написанные процедуры в правильном порядке — описание процедуры поместить перед ее вызовом. Поскольку рекурсии в этом примере нет, сделать это легко.

На рассмотренном примере мы продемонстрировали, что синтаксический анализатор методом рекурсивного спуска можно написать «почти так же быстро, как мы вообще можем писать» [Хантер, 1984].

Требование детерминированного распознавания

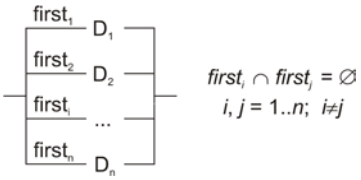
Уже в ходе предыдущего рассмотрения можно было заметить, что рекурсивный спуск позволяет построить анализатор не для любой КС-грамматики. Ограничения возникают при анализе направляющих символов отдельных ветвей синтаксической диаграммы.

Множество направляющих символов first(D) ветви D синтаксической диаграммы образуют терминальные символы, которые могут встретиться первыми при движении по диаграмме вдоль этой ветви.

Движение по диаграмме предполагает, что при достижении прямоугольного блока, изображающего нетерминал, оно продолжается по диаграмме этого нетерминала с последующим возвратом на исходную диаграмму.

Порядок действий анализатора, работающего по методу рекурсивного спуска, и определяется анализом направляющих символов отдельных ветвей синтаксической диаграммы. Чтобы алгоритм работал без возвратов, выбор направления движения по диаграмме выполнялся однозначно должно соблюдаться требование детерминированного распознавания:

В каждом разветвлении синтаксической диаграммы множества направляющих символов отдельных ветвей не должны попарно пересекаться.



В рассмотренной выше грамматике многочленов требование детерминированного распознавания всегда соблюдается.

Возьмем, например, диаграмму, показанную на рис. 2.22. На ней две точки ветвления. Первая — на входе в диаграмму, вторая — на выходе. В первом разветвлении три ветви. Множества направляющих символов этих ветвей: `['+']` — верхняя ветвь; `['0'..'9', 'X']` — средняя ветвь; `['-']` — нижняя ветвь. Как видно, эти множества не пересекаются.

В правой точке происходит ветвление на два направления. Одно ведет на выход из диаграммы, множество направляющих символов этой ветви состоит из символа «конец текста»: `[⊥]`. Другая ветвь соответствует очередному витку цикла, ее множество направляющих символов равно `['+', '-']`. Пересечения множеств снова нет.

Нетрудно убедиться, что не пересекаются и множества направляющих символов отдельных ветвей диаграмм, показанных на рисунках 2.23 и 2.24.

LL-грамматики

LL(*k*)-*грамматикой называется КС-грамматика, в которой выбор правила в ходе левостороннего вывода однозначно определяется не более чем k очередными символами входной цепочки, считываемой слева направо.*

Название «*LL*» происходит от двух слов «left» (левый), встречающихся в описании хода распознавания в *LL*-грамматике: левосторонний вывод при чтении слева. Самыми удобными для распознавания, конечно же, являются *LL*(1) грамматики, в кото-

рых выбор направления распознавания однозначно определяется очередным входным символом.

Сформулированное выше требование детерминированного распознавания при рекурсивном спуске есть не что иное, как необходимость того, чтобы используемая грамматика относилась к классу *LL*(1).

Рекурсивный спуск — это детерминированный метод нисходящего разбора КС-языков, порождаемых LL(1)*-грамматиками.*

Левая и правая рекурсия

Рассмотрение грамматик с левой и правой рекурсией позволит нам получить некоторые признаки, помогающие определить наличие или отсутствие *LL*(1) свойства у КС-грамматик.

Если в грамматике G существует нетерминал A, для которого $A \overset{+}{\underset{G}{\Rightarrow}} A\alpha$, где α — непустая цепочка, грамматика содержит левую рекурсию.

Грамматика, содержащая левую рекурсию, не может быть LL(1) *грамматикой.*

Если в грамматике G существует нетерминал A, для которого $A \overset{+}{\underset{G}{\Rightarrow}} \alpha A$, где α — непустая цепочка, грамматика содержит правую рекурсию.

Леворекурсивная грамматика всегда может быть преобразована в эквивалентную праворекурсивную.

Синтаксический анализ арифметических выражений

Рассмотрим арифметические выражения, синтаксис которых задается грамматикой

G_8: $E \to T \mid E + T \mid E - T$

$T \to M \mid T * M \mid T / M$

$$M \to a \mid b \mid c \mid (E)$$

Эта КС-грамматика, построенная нами раньше, обладает рядом достоинств. Она однозначна и ассоциирует операнды в соответствии с общепринятым порядком выполнения операций, когда вначале выполняются умножение и деление, затем — сложение и вычитание. Однако нетрудно видеть, что G_8 леворекурсивна. Действительно, для нетерминала E справедливо: $E \Rightarrow E + T$. Наличие левой рекурсии препятствует использованию рекурсивного спуска.

G_8 может быть заменена эквивалентной (порождающей тот же язык) праворекурсивной грамматикой:

G_{13}: $E \to T \mid T + E \mid T - E$

$T \to M \mid M * T \mid M / T$

$M \to a \mid b \mid c \mid (E)$

Хотя левой рекурсии больше нет, грамматика G_{13} не является $LL(1)$-грамматикой. Чтоб убедиться в этом, достаточно обратиться к правилам для нетерминала E:

$$E \to T \mid T + E \mid T - E$$

Напомню, что эту строку следует рассматривать как сокращенную запись трех альтернативных правил для нетерминала E. Очевидно, что выбор одного из этих трех правил на основе анализа одного входного символа невозможен, поскольку правая часть каждого правила начинается одним и тем же нетерминалом T. На рис. 2.25 показана синтаксическая диаграмма нетерминала E грамматики G_{13}, рассмотрение которой также убеждает, что требование детерминированного распознавания не выполнено: множества направляющих символов всех трех ветвей совпадают: $first_1 = first_2 = first_3 = \{ a, b, c, (\}$.

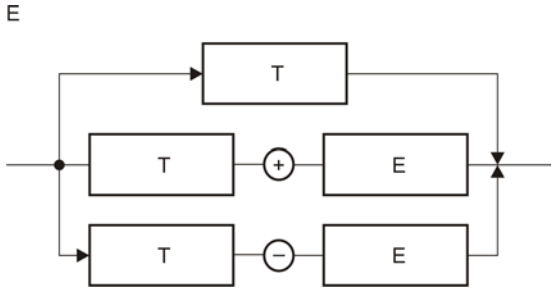


Рис. 2.25. Диаграмма нетерминала *E* грамматики G_{13}

Грамматика G_{13} и соответствующие синтаксические диаграммы могут быть преобразованы так, чтобы использование рекурсивного спуска стало возможным. Особенно легко увидеть возможность преобразования диаграмм. Действительно, достаточно вынести блок *T* из трех параллельных ветвей и поместить его в общую ветвь, как требование детерминированного распознавания для диаграммы нетерминала *E* будет соблюдено (рис. 2.26).

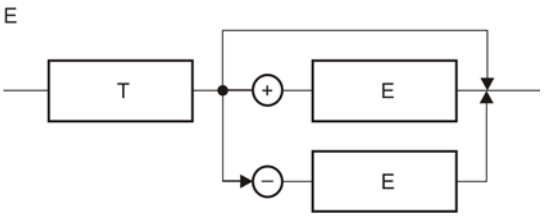


Рис. 2.26. Преобразованная диаграмма нетерминала *E* грамматики выражений

Аналогичное изменение может быть выполнено и для диаграммы слагаемого (нетерминал *T*).

Менее очевиден способ преобразования грамматики. Потребуется два дополнительных нетерминала. Обозначим их *A* и *B*. Тогда новая грамматика может быть записана так:

G_{14}: $E \to T\,A$

$A \to \varepsilon \mid +\,E \mid -\,E$

$T \to M\,B$

$B \to \varepsilon \mid *\,T \mid /\,T$

$$M \to a \mid b \mid c \mid (E)$$

Содержательно A и B можно трактовать как «выражение без первого слагаемого» и «слагаемого без первого множителя» соответственно.

G_{14} — это $LL(1)$-грамматика для арифметических выражений. Но и она не вполне может нас устроить. Дело в том, что G_{14}, как и G_{13} (но не G_8), связывает операнды нескольких идущих подряд операций одного приоритета неподходящим образом, группируя их справа налево. На рис. 2.27*а* показано дерево вывода выражения $a - b - c$ в грамматике G_{14}.

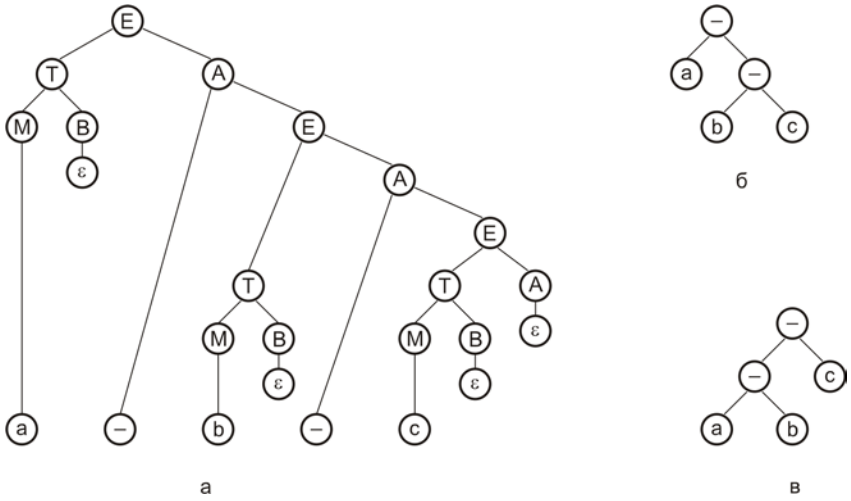


Рис. 2.27. Дерево выражения $a - b - c$

Устранив из этого синтаксического дерева нетерминалы (и ε) и перенося знаки операций во внутренние вершины, получим семантическое дерево выражения (рис. 2.27б), которое, увы, не соответствует правильному порядку его вычисления: оно подразумевает группировку операндов $a - (b - c)$, в то время как $a - b - c = (a - b) - c$. Правильное семантическое дерево можно видеть на рис. 2.27в.

Модернизируем G_{14} с целью обеспечить правильную ассоциацию операций и операндов. Получаем грамматику G_{15}:

G_{15}: $E \to T\,A$

$A \to \varepsilon \mid + TA \mid - TA$

$T \to M\,B$

$B \to \varepsilon \mid * MB \mid / MB$

$M \to a \mid b \mid c \mid (\,E\,)$

Эта однозначная праворекурсивная $LL(1)$-грамматика приписывает выражению правильную структуру. По ней без труда может быть написан синтаксический анализатор, работающий по алгоритму рекурсивного спуска. Интересно отметить, что программа-распознаватель, написанная по этой грамматике, будет рекурсивной (что, как известно, эквивалентно итерации), но не будет содержать циклов. Недостатками такой грамматики является некоторая ее громоздкость и наличие нетерминалов A и B с неочевидным смыслом.

Удовольствие написать распознаватель по грамматике G_{15} предоставлю читателям. Запрограммировать его вы сможете так же быстро, как быстро умеете писать.

А теперь мы воспользуемся теми выразительными средствами, которые дают синтаксические диаграммы. Это в первую очередь возможность использования цикла вместо рекурсии. Вообще, можно заметить, что некоторые трудности, возникающие на практике при использовании грамматик Хомского, обусловлены тем, что с их помощью приходится выражать повторение через рекурсию. Так, о выражении, представляющем собой просто последовательность слагаемых, мы вынуждены думать как о рекурсивной конструкции: выражение — это первое слагаемое, за которым после знака снова записано выражение (правая рекурсия, неподходящая группировка слагаемых: первое слагаемое складывается со всем остальным выражением). Или: если к выражению прибавить или от выражения отнять слагаемое, то сно-

ва получится выражение (левая рекурсия, правильная группировка слагаемых). Эти проблемы могут быть устранены добавлением в нотацию формальных грамматик явного обозначения для повторения, что и будет сделано при рассмотрении грамматик языков программирования. Синтаксические диаграммы также позволяют обойтись без рекурсии там, где она служит всего лишь для задания повторений.

Рассмотрим диаграмму нетерминала *E* (см. рис. 2.26). Налицо рекурсия: на диаграмме *E* присутствует нетерминальный блок *E*. Но это правая, концевая рекурсия. Обращение к блоку *E* можно заменить переходом на вход диаграммы (рис. 2.28а).

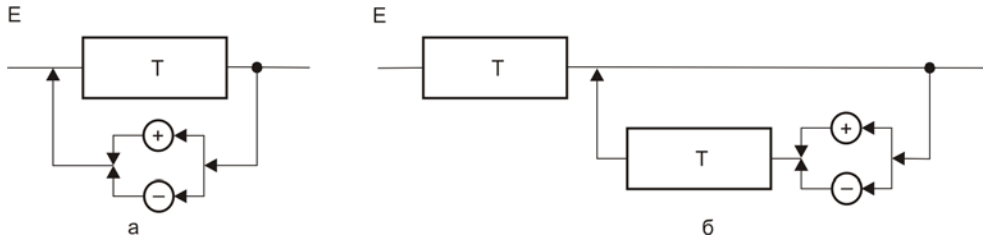


Рис. 2.28. Синтаксическая диаграмма выражения, не содержащая рекурсии

Такую диаграмму для выражения можно было построить и без выписывания и преобразования грамматики. Суть изображенного проста: выражение — это последовательность одного или более слагаемых, между которыми записываются знаки «+» или «–». Поскольку обработка слагаемых в ходе синтаксического анализа будет происходить последовательно в цикле, не составит труда организовать трансляцию так, чтобы это соответствовало выполнению операций слева направо.

Чтобы программировать распознаватель на Паскале было удобней, преобразуем диаграмму в эквивалентную, содержащую цикл с предусловием (рис. 2.28б). Аналогично строится диаграмма слагаемого (нетерминал *T*) (рис. 2.29а). Диаграмма множителя соответствует правилам для нетерминала *M* в грамматиках G_8, G_{13}, G_{14}, G_{15} (рис. 2.29б).

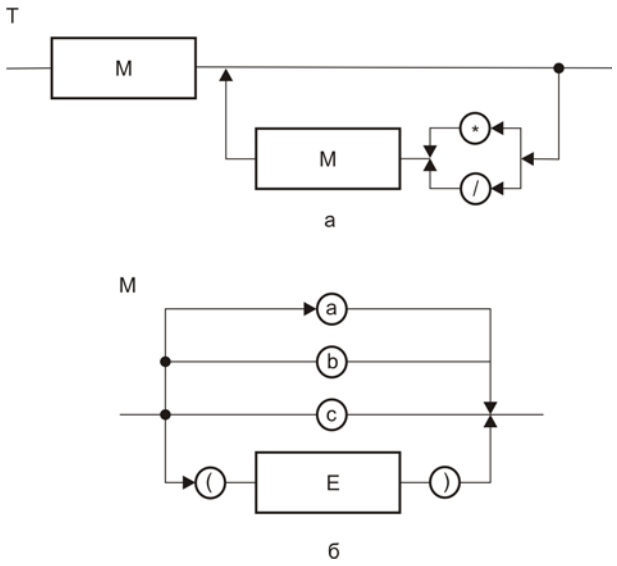


Рис. 2.29. Синтаксические диаграммы слагаемого и множителя

Теперь можно программировать анализатор. Запишем только распознающие процедуры. Константы, переменные и вспомогательные процедуры предполагаются такими же, как в распознавателе многочленов.

Начать естественно с процедуры, соответствующей начальному нетерминалу, то есть с процедуры *E*:

```
procedure E;
begin
   T;
   while Ch in ['+', '-'] do begin
      NextCh;
      T;
   end;
end;
```

Далее записываем распознаватель слагаемого:

```
procedure T;
begin
   M;
   while Ch in ['*', '/'] do begin
      NextCh;
      M;
   end;
end;
```

И, наконец, распознающую процедуру для нетерминала *M* — множителя:

```
procedure M;
begin
   if Ch in ['a', 'b', 'c'] then
      NextCh
   else if Ch = '(' then begin
      NextCh;
      E;
      if Ch = ')' then
         NextCh
      else
         Error('Ожидается ")"');
      end
   else
      Error('Ожидается a, b, c или "("');
end;
```

Остается только один вопрос: как разместить три эти процедуры в программе? По общему правилу языка Паскаль любой идентификатор может быть использован только после его описания. Процедура `E` вызывает процедуру `T`, поэтому должна располагаться после нее. Процедура `T` вызывает `M`, поэтому `M` надо разместить перед `T`. Получается такой порядок: `M`; `T`; `E`. Но из процедуры `M` вызывается `E`, в то время как описание `E` мы разместили после `M`. Выходом из этой ситуации, которая возникла из-за наличия в нашей программе косвенной рекурсии, является использование директивы **`forward`** для опережающего описания процедуры `E`. Структура программы будет такой:

```
procedure E; forward;
procedure M;
...
procedure T;
...
procedure E;
...
```

Есть и другое, весьма изящное решение проблемы: использование вложенных процедур. Такой распознаватель представлен в листинге 2.3.

Листинг 2.3. Распознаватель арифметических выражений

```
procedure E;

   procedure T;

      procedure M;
      begin
         if Ch in ['a', 'b', 'c'] then
            NextCh
         else if Ch = '(' then begin
            NextCh;
            E;
            if Ch = ')' then
               NextCh
            else
               Error('Ожидается ")"');
            end
         else
            Error('Ожидается a, b, c или "("');
      end {M};

   begin
      M;
      while Ch in ['*', '/'] do begin
         NextCh;
         M;
      end;
   end {T};

begin
   T;
   while Ch in ['+', '-'] do begin
      NextCh;
      T;
   end;
end {E};
```

Включение действий в синтаксис

Синтаксический анализатор отвечает на вопрос о принадлежности входной цепочки языку. В ходе распознавания выявляется структура входного текста. Эта структура может быть представлена явно, например, в виде дерева, или неявно — последовательностью действий, совершенных распознавателем.

На основе распознавания структуры входного текста строится и его содержательная обработка, трансляция. Синтаксический анализатор служит основой, остовом транслятора, предоставляя возможность выполнить необходимые действия по смысловой (семантической) обработке в нужные моменты в соответствии со структурой входной цепочки.

Семантические процедуры

Встраиваемые в распознаватель действия, предназначенные для выполнения смысловой обработки входного текста, будем называть *семантическими процедурами*.

Слово «процедура» употребляется здесь в широком смысле, как определённая последовательность действий. В программе-распознавателе это может быть не обязательно процедура-подпрограмма, но и просто один или несколько операторов.

Рассмотрим использование семантических процедур на простом примере.

Пусть речь идет о распознавателе целых чисел без знака, который был использован в программе анализаторе многочленов. Теперь в его задачу будет входить получение значения числа, представленного последовательностью цифр.

Обозначив семантические процедуры P_1 и P_2, разместим на синтаксической диаграмме «Целое» (рис. 2.30) соответствующие им треугольные значки в тех местах, при прохождении которых во время анализа эти процедуры должны выполняться.

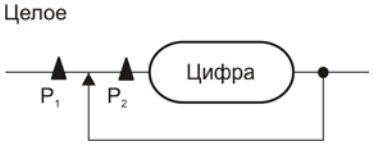


Рис. 2.30. Семантические процедуры на диаграмме целого без знака

Процедура P_1 будет выполняться в начале обработки и присвоит искомому числу исходное нулевое значение. Переменная y — это формируемое значение числа.

```
P1: y := 0;
```

Задача P_2 — добавлять к числу справа очередную цифру. Для этого «старое» значение y умножим на 10 (основание десятичной системы счисления) и добавим значение прочитанной цифры. Условимся, что семантическая процедура, значок которой размещен перед обозначением терминального символа, выполняется в тот момент, когда этот символ является текущим (содержится в переменной Ch). Надо побеспокоиться и о том, чтобы при вычислениях не произошло переполнения — то есть не получилось, число, превышающее максимально допустимое целое.

```
P2: d := ord(Ch) – ord('0');
    if y <= (maxint – d) div 10 then
       y := 10*y + d
    else
         Error('Слишком большое число');
```

Вы, конечно, понимаете, что выполнять такую проверку

```
if 10*y + d <= maxint then …
```

было бы, как минимум, наивно — переполнение случится еще до того, как дело дойдет до сравнения.

Теперь можно написать распознающую процедуру, выполняющую не только синтаксический анализ, но и получение числового значения целого, которое будет выходным параметром этой процедуры.

```
procedure Number(var y : integer); { Целое }
var
   d : integer;
begin
   y := 0;                                    { P1 }
   if not( Ch in ['0'..'9'] ) then
      Error('Число начинается не с цифры');
   repeat
      d := ord(Ch) - ord('0');                { P2 }
      if y <= (maxint - d) div 10 then        { P2 }
```

```
            y := 10*y + d                          { P2 }
         else                                      { P2 }
            Error('Слишком большое число');        { P2 }
         NextCh;
      until not( Ch in ['0'..'9'] );
   end;
```

Напомню, что процедура `Error` останавливает работу всего анализатора, поэтому не надо беспокоиться, что при возникновении ошибки в цикле работа этого цикла разладится. В дальнейшем мы обсудим другие варианты обработки ошибок, но пока такое соглашение позволяет не загромождать транслятор дополнительными проверками. Можно еще отметить, что ошибка «Слишком большое число» уже не синтаксическая. Она связана с содержательной обработкой, и вправе называться семантической.

Пример: умножение многочленов

Рассмотрим задачу, которая была предложена на одной из олимпиад по программированию.

ЗАДАЧА

> Составить программу, вводящую в символьной форме два многочлена от *x* с целыми коэффициентами и выводящую их произведение в символьной форме в порядке убывания степеней. Суммарная степень многочленов не превышает 255.

Вот пример работы такой программы:

```
Перемножение многочленов
------------------------
1-й многочлен:
3x^2+3x-5
2-й многочлен:
x^3 - 2x
Произведение равно:
3x^5 + 3x^4 - 11x^3 - 6x^2 + 10x
```

Программа печатает запрос, в ответ на который можно ввести запись первого, а затем второго многочлена в привычной, используемой в математике форме. Обработав полученные дан-

ные, программа печатает результат — произведение многочленов в таком же естественном виде.

Проектирование

Приступим к решению поставленной задачи. Сформулируем общий план:

1. Напечатать заголовок.
2. Ввести текст 1-го многочлена, проанализировать его запись и преобразовать в удобную для дальнейших вычислений форму.
3. Для выполнения действий с многочленами (в том числе перемножения) удобно хранить их в программе в виде массива коэффициентов, количество которых соответствует степени многочлена, а индекс каждого коэффициента равен степени соответствующего слагаемого.
4. Ввести текст 2-го многочлена, проанализировать его запись и преобразовать в удобную для дальнейших вычислений форму.
5. Перемножить многочлены.
6. Имея представление обоих многочленов-сомножителей во внутреннем формате и выполняя необходимые вычисления с их коэффициентами, получаем коэффициенты многочлена-произведения.
7. Напечатать результат.

Запишем начало программы, в котором в соответствии с уже принятыми решениями определим необходимые константы, типы данных и переменные.

```
program MultPoly; { Перемножение многочленов }
const
   nmax = 255;          { Максимальная степень }
type
   tPoly = record { Тип многочленов }
      n: integer;                    { Степень }
      a: array [0..Nmax] of integer; { Коэффициенты }
   end;
```

```
var
   P1, P2, Q: tPoly;   { Сомножители и произведение }
```

Обратите внимание, что многочлены отнесены к типу `tPoly`, который представляет собой запись, содержащую, кроме упоминавшегося массива коэффициентов, величину *n* — фактическую степень.

Теперь, пользуясь предварительно составленным планом, запишем основную программу.

```
begin
   WriteLn('Перемножение многочленов');
   WriteLn('------------------------');
   WriteLn;
   WriteLn('1-й многочлен');
   GetPoly(P1);
   WriteLn('2-й многочлен');
   GetPoly(P2);
   { Перемножение }
      MultPoly(P1, P2, Q);
   WriteLn;
   WriteLn('Произведение:');
   { Печать результата }
      WritePoly(Q);
   WriteLn;
end.
```

Умножение и вывод

Детализацию начнем с процедуры, которая выполняет перемножение. Её входными параметрами являются многочлены-сомножители, выходным — многочлен-произведение. И входные, и выходной параметры являются многочленами, представленными в виде совокупности массива коэффициентов и величины, задающей степень многочлена, то есть относятся к типу `tPoly`. Обозначив сомножители `X` и `Y`, а результат — `Z`, запишем заголовок процедуры.

```
procedure MultPoly(X, Y: tPoly; var Z: tPoly);
```

Основная идея вычисления коэффициентов многочлена-произведения состоит в том, что при попарном перемножении слагаемых первого и второго многочлена получающееся произ-

ведение участвует (в качестве одного из слагаемых) в формировании того слагаемого многочлена-результата, степень которого равна сумме степеней сомножителей. То есть при перемножении *i*-го члена многочлена `X` и *j*-го члена многочлена `Y` получается величина, которая должна быть добавлена к *i*+*j*-му слагаемому `Z`:

```
Z.a[i+j] := Z.a[i+j] + X.a[i]*Y.a[j];
```

Такое вычисление нужно выполнить для всех сочетаний *i* и *j*, не забыв присвоить нулевые начальные значения коэффициентам `Z` и побеспокоиться об определении степени многочлена `Z`. Получаем:

```
{ Умножение многочленов. Z = X*Y }
procedure MultPoly(X, Y: tPoly; var Z: tPoly);
var
   i, j: integer;
begin
   ClearPoly(Z);   { "Обнуление" Z }
   for i := 0 to X.n do
      for j := 0 to Y.n do
         Z.a[i+j] := Z.a[i+j] + X.a[i]*Y.a[j];
   { Определение степени многочлена Z }
      Z.n := nmax;
      while ( Z.n>0 ) and ( Z.a[Z.n]=0 ) do
         Z.n := Z.n-1;
end;
```

Процедура `ClearPoly`, выполняющая «обнуление» многочлена, выглядит так:

```
procedure ClearPoly(var P : tPoly);
var
   i : integer;
begin
   for i := 0 to nmax do
      P.a[i] := 0;
   P.n := 0;
end;
```

Теперь займемся печатью многочлена. Слагаемые должны выводиться в порядке убывания степеней. Слагаемому может предшествовать знак. Коэффициент (при ненулевой степени) печатается, если он не равен 0 или 1. Буква `x` выводится для нену-

левых степеней, а значение показателя степени (и знак "^" перед ним), — если эта степень больше единицы.

```
{ Вывод многочлена }
procedure WritePoly(P : tPoly);
var
   i : integer;
begin
   with P do
      for i := n downto 0 do begin
         if ( a[i]>0 ) and ( i<>n ) then
            Write(' + ')
         else if ( a[i]<0 ) and ( i=n ) then
            Write('-')
         else if a[i]<0 then
            Write(' - ');
         if   ( abs(a[i])>1 ) or
            ( i=0 ) and ( a[i] <> 0 ) or
            ( n=0 )
         then
            Write(abs(a[i]));
         if ( i>0 ) and ( a[i]<>0 ) then
            Write('x');
         if ( i>1 ) and ( a[i]<>0 ) then
            Write('^', i)
      end;
end;
```

Транслятор многочленов

Нам осталось реализовать транслятор (процедуру `GetPoly`), преобразующий введенную запись многочлена в массив коэффициентов. Его задача — считывать символы из входной строки, выполнять распознавание многочлена и вычислять его коэффициенты. Распознающие процедуры будут вложены внутрь `GetPoly`, а сама эта процедура лишь подготовит чтение входной строки, вызовет распознаватель и убедится, что за многочленом во входной строке ничего не содержится.

```
{ Ввод и трансляция многочлена }
procedure GetPoly(var P: tPoly);
const
   EOT = chr(0);     { Конец текста }
var
   Pos: integer;  { номер символа }
```

```
   Ch : char;      { очередной символ }
...
{ Здесь разместятся процедуры распознавателя }
...
begin
   Pos := 0;
   NextCh;
   Poly(P);
   if Ch <> EOT then
      Error('Ожидается конец текста');
end;
```

Разместим на синтаксических диаграммах значки семантических процедур и определим содержание этих процедур. Вначале на диаграмме многочлена (рис. 2.31).

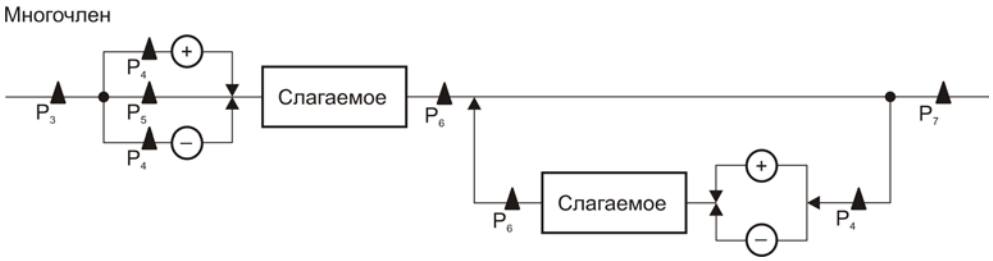


Рис. 2.31. Синтаксическая диаграмма многочлена с семантическими процедурами

Перед началом обработки массив коэффициентов многочлена заполняется нулями, а его степень принимается равной нулю. Эту работу выполнит семантическая процедура P_3, вызвав уже написанную нами процедуру `ClearPoly`.

```
P3: ClearPoly(P);
```

P_4 и P_5 отвечают за запоминание знака перед слагаемым. Знак сохраняется в локальной переменной `Op`.

```
P4: Op := Ch;
P5: Op := '+';
```

Каждое слагаемое многочлена имеет вид $a_k x^k$. Транслятор слагаемого вычислит значения коэффициента a и степени k. Получив эти значения, семантическая процедура P_6 в зависимости от

знака добавит или отнимет значение *a* из *k*-й ячейки массива коэффициентов:

```
P6: if Op = '+' then
       P.a[k] := P.a[k] + a
    else
       P.a[k] := P.a[k] - a;
```

Было бы неправильно просто записывать значение коэффициента в a[k], поскольку в записи многочлена могут быть несколько членов с одинаковой степенью *x* —синтаксис этого не запрещает. Складывая или вычитая каждый коэффициент с предыдущей суммой, транслятор как бы приводит подобные.

Определение степени *n* многочлена *P* выполняет семантическая процедура P_7.

```
P7: P.n := nmax;
    while (P.n > 0) and (P.a[P.n] = 0) do
       P.n := P.n-1;
```

Теперь, пользуясь диаграммой и вставляя в текст анализатора в соответствующих местах определенные нами семантические процедуры, можно записать распознаватель многочлена.

```
{ Многочлен }
procedure Poly(var P: tPoly);
var
   a     : integer;  { Модуль коэффициента }
   k     : integer;  { Степень слагаемого  }
   Op    : char;     { Знак операции       }
begin
   ClearPoly(P);                          { P3 }
   if Ch in ['+','-'] then begin
      Op := Ch;                           { P4 }
      NextCh;
      end
   else
      Op := '+';                          { P5 }
   Addend(a, k);
   if Op = '+' then                       { P6 }
      P.a[k] := P.a[k] + a                {    }
   else                                   {    }
      P.a[k] := P.a[k] - a;               {    }
   while Ch in ['+','-'] do begin
      Op := Ch;                           { P4 }
```

```
      NextCh;
      Addend(a, k);
      if Op = '+' then                    { P6 }
         P.a[k] := P.a[k] + a             {    }
      else                                {    }
         P.a[k] := P.a[k] - a;            {    }
   end;
   P.n := nmax;                           { P7 }
   while (P.n > 0) and (P.a[P.n] = 0) do  {    }
      P.n := P.n-1;                       {    }
end;
```

Задача транслятора слагаемого, как уже говорилось, состоит в определении коэффициента (точнее, модуля коэффициента) *a* и степени *k* слагаемого. В соответствии с этим предусмотрим и необходимые семантические процедуры (рис. 2.32).

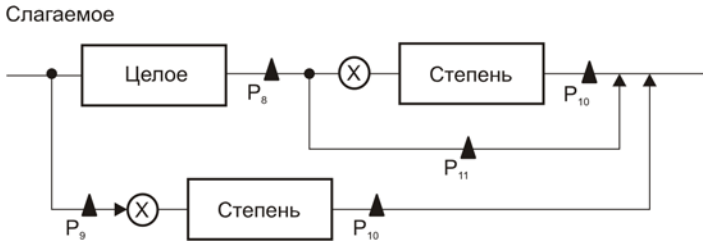


Рис. 2.32. Синтаксическая диаграмма слагаемого с семантическими процедурами

Процедура P_8 присваивает коэффициенту *a* значение, равное величине целого числа, которое вычисляется транслятором целого.

```
P8:   a := значение целого;
```

В программе это будет реализовано подстановкой переменной *a* в качестве параметра при вызове процедуры `Number`.

Если коэффициент отсутствует, он принимается равным единице.

```
P9:   a := 1;
```

Значение степени либо берется равным вычисленному распознавателем степени, либо нулевым, если в записи слагаемого отсутствует *x*.

```
P10:  k := значение степени;
```

```
P11:   k := 0;
```

Теперь записать распознаватель-транслятор слагаемого не составляет труда.

```
{ Слагаемое }
procedure Addend(var a, k : integer);
begin
   if Ch in ['0'..'9'] then begin
      Number(a);              { P8 }
      if Ch = 'x' then begin
         NextCh;
         Power(k);            { P10 }
         end
      else
         k := 0;              { P11 }
      end
   else if Ch = 'x' then begin
      a := 1;                 { P9 }
      NextCh;
      Power(k);               { P10 }
      end
   else
      Error('Ожидается число или "x"');
end;
```

Реализация распознавателя нетерминала "Степень" не вызывает затруднений и выполняется по соответствующей синтаксической диаграмме, на которой обозначены семантические процедуры P_{11} и P_{12} (рис. 2.33). Чтобы защитить программу от ошибки при обращении к массиву коэффициентов, в процедуре P_{11} предусмотрен контроль величины показателя степени. Вычисляемая степень обозначена p.

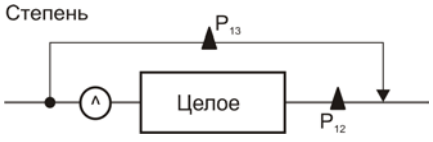


Рис. 2.33. Синтаксическая диаграмма степени с семантическими процедурами

```
P12:   p := значение целого;
       if p > nmax then
          Error('Слишком большая степень');
P13:   p := 1;
```

Обратите внимание, что в случае, когда семантическая процедура (P_{13}) расположена на ветви диаграммы, где не было терминальных и нетерминальных блоков, она сама играет роль блока, что несколько меняет структуру программы-распознавателя (в нашем случае появляется ветвь **else**).

```
{ Степень }
procedure Power(var p : integer);
begin
   if Ch = '^' then begin
      NextCh;
      Number(p);                                { P12 }
      if p > nmax then                          {     }
         Error('Слишком большая степень');      {     }
      end
   else
      p := 1;                                   { P13 }
end;
```

Обработка ошибок при трансляции

В рассмотренных примерах обработка ошибок выполнялась с помощью процедуры `Error`, которая после выдачи сообщения прекращала работу всей программы. Такой способ реакции на ошибку упрощает синтаксический анализатор, но далеко не идеален. Можно рассмотреть несколько вариантов его усовершенствования:

- Завершение работы распознавателя (транслятора) после обнаружения первой ошибки без прерывания работы вызвавшей его программы.
- Продолжение работы распознавателя после обнаружения первой ошибки с целью обнаружения других.

Второй вариант реакции на ошибки используется во многих трансляторах языков программирования. В литературе можно найти немало рекомендаций по способам восстановления распознавателя после обнаружения ошибки с целью продолжения анализа [Грис, 1975], [Хантер, 1984], [Вирт, 1985], [Wirth, 1996], [Ахо, 2001]. Однако, общего решения задачи не существует.

Многое зависит от конкретного языка. Качественное восстановление, обеспечивающее выдачу осмысленных сообщений, предполагает, в том числе, использование эмпирических приемов. Рассмотрение таких подходов выходит за рамки этой книги. Любознательным читателям могу рекомендовать посмотреть названные источники или проделать собственные опыты.

Обсудим и реализуем более простой первый вариант. Он совсем неплох. В условиях, когда скорость работы компьютеров неизмеримо возросла, время, необходимое для компиляции программы[10], невелико, и его потери, связанные с тем, что компилятор не сообщил программисту о нескольких ошибках сразу, исчезающе малы.

Итак, необходимо организовать реакцию на ошибку таким образом, чтобы после выдачи сообщения, анализатор не останавливал всю программу, а лишь завершил свою работу, давая возможность продолжить исполнение вызвавшей его программе.

Первый вариант усовершенствования состоит в том, чтобы после вызова каждой распознающей процедуры проверять успешность ее завершения и, если при анализе нетерминала обнаружена ошибка, пропускать оставшиеся части вызывающей процедуры. В этом случае процедура `Error` уже не прерывает программу с помощью `Halt`, а формирует признак ошибки, который может быть проверен в распознающих процедурах. Такой подход, однако, сильно загромоздит программы анализатора, сделает их неудобочитаемыми.

Используем другое решение. Проблема при обработке ошибок состоит лишь в том, чтобы после обнаружения ошибки завершить *все процедуры*, цепочка вызова которых привела к процедуре, обнаружившей ошибку. Это можно сделать, если при об-

[10] Точнее, одной единицы компиляции, например, модуля.

наружении ошибки процедура Error установит признак ошибки и прочитает текст до конца. Текущим символом станет "конец текста". Поскольку этот символ не может встретиться в анализируемом тексте, он будет отвергнут всеми частями анализатора, который завершит свою работу без прерывания программы в целом. В качестве признака ошибки разумно использовать номер ошибочного символа (обозначим ErrPos), ненулевое значение которого соответствует наличию ошибки и несет информацию о ее местоположении. Перед началом работы анализатора до первого вызова NextCh нужно выполнить ErrPos := 0. Исправленные процедуры Error и NextCh теперь выглядят так:

```
{ Ошибка }
procedure Error(Message : string);
   { Message - сообщение }
var
   e : integer;
begin
   if ErrPos = 0 then begin
      WriteLn('^': Pos);
      WriteLn('Синтаксическая ошибка: ', Message);
      e := Pos;
      while Ch <> EOT do NextCh;
      ErrPos := e;
   end;
end;
```

Процедура Error «самоблокируется», проверяя значение ErrPos и обходя выдачу сообщений при повторных вызовах. Пользоваться ею можно так же, как и раньше, как бы считая, что после вызова Error работа анализатора прерывается. Дополнительные проверки на наличие ошибки после обращения к распознавателям нетерминалов не нужны.

NextCh после обнаружения ошибки всегда возвращает «конец текста».

```
{ Читать следующий символ }
procedure NextCh;
begin
   if ErrPos <> 0 then
```

```
      Ch := EOT
   else
      …
end;
```

Надо, однако, беспокоиться о том, чтобы при обнаружении ошибки не оставались неопределенными или недопустимыми данные, связанные с семантической обработкой. Они могут использоваться в вычислениях, которые будут выполняться после выдачи сообщения об ошибке при возврате из распознающих процедур. При этом не должно возникнуть недопустимых ситуаций: деление на ноль, разыменование неопределенного или равного **nil** указателя, выход индекса за границы массива и т. п. Например, распознаватель степени, в задачу которого входит определение величины *p*, при использовании нашей технологии следует записать так:

```
{ Степень }
procedure Power(var p : integer);
begin
   if Ch = '^' then begin
      NextCh;
      Number(p);
      if p > nmax then begin
         Error('Слишком большая степень');
         p := 0;{Степень не должна остаться
                 слишком большой}
      end
      end
   else
      p := 1;
end;
```

Требуются еще одно уточнение. Поскольку по завершении работы распознавателя, обнаружившего ошибку, текущим символом станет EOT, его проверка уже не будет достаточным условием успешного окончания анализа. Использовавшееся выше завершение:

```
if Ch <> EOT then
   Error('Ожидается конец текста')
else
   WriteLn('Правильно');
```

должно быть заменено следующим:

```
if (ErrPos = 0) and (Ch = EOT) then
   WriteLn('Правильно')
else
   Error('Ожидается конец текста');
```

Существует мнение, что лучшим вариантом выхода из состояния ошибки в анализаторе, работающем по алгоритму рекурсивного спуска, является использование исключений, обработка которых предусмотрена в некоторых языках программирования. Вы можете поэкспериментировать и составить свое суждение на этот счет.

Табличный *LL*(1)-анализатор

Рассматривая автоматные языки, мы использовали детерминированный конечный автомат в роли эффективного универсального распознавателя. Один из вариантов его реализации — программная интерпретация таблицы переходов автомата. На похожих принципах может быть построен и распознаватель для *LL*(1)-грамматик, который будет представлять собой реализацию определенного подкласса упоминавшихся выше МП-автоматов.

Вначале модифицируем таблицу переходов конечного автомата. Ее обычный формат таков:

Таблица 2.3. Таблица переходов конечного автомата

Состояние	Символ				
	A	B	c	d	...
1					
2					
3					
4					
...					

Все состояния пронумерованы. В колонках символов для примера записаны буквы из начала латинского алфавита. Для реальных языков количество допустимых символов может быть довольно большим. Если же рассматривать практические аспекты реализации, то придется учесть, что на вход программы, моделирующей автомат, может поступать вообще любой символ.

Чтобы сократить размер таблицы, будем размещать различные символы в одном столбце. В каждом состоянии будет проверяться совпадение с единственным символом. Число состояний при этом может увеличиться.

Для примера возьмем конечный автомат, распознающий целые числа без знака. Таблица 2.4, имеющая традиционный вид, задает его переходы.

Таблица 2.4. Таблица переходов КА, распознающего целые без знака

Состояние	Символ	
	Цифра	Не цифра
1	2	*E*
2	2	*K*
E	*E*	*E*
K		

Здесь *E* означает состояние ошибки, а *K* — конечное состояние автомата.

Следующая таблица 2.5 имеет модифицированный вид. Символы записываются во втором столбце, состояние в которое переходит автомат при совпадении входного символа и символа в таблице — в третьем. В четвертом столбце отмечено, возникает ли ошибка, если входной символ не совпадает с символом, записанным в таблице для данного состояния. Если в графе «Ошиб-

ка» записано «Нет», то при несовпадении символов автомат переходит в следующее по порядку состояние.

Таблица 2.5. Модифицированная таблица переходов КА, распознающего целые

Состояние	Символ	Переход	Ошибка
10	Цифра	11	Да
11	Цифра	11	Нет
12	Любой	0	Нет

По причинам, которые мы вскоре выясним, состояния нового автомата пронумерованы не с 1. Переход в состояние 0, который происходит при получении автоматом, находящемся в состоянии 12, любого символа (например, символа «конец текста») означает завершение его работы с принятием (части) входной цепочки.

Автомат, распознающий целые, можно попробовать применить при построении автомата, распознающего, «Степень» из примера про многочлены (см. рис. 2.33), а также многочлен в целом. Начнем с автомата, распознающего степень. Построим таблицу переходов (табл. 2.6). Пусть состояния этого автомата начинаются с 20.

Таблица 2.6. Таблица переходов распознавателя степени

Состояние	Символ	Переход	Ошибка	Вызов	Читать
20	^	22	Нет	Нет	Да
21	Любой	0	Нет	Нет	Нет
22	Любой	10	Нет	Да	Нет
23	Любой	0	Нет	Нет	Нет

Потребовалось также добавить два новых столбца. Столбец «Читать» управляет чтением следующего символа: если в этом столбце стоит «Да», то при совпадении входного символа и символа, записанного во втором столбце для данного состояния, читается следующий символ входной цепочки. В предыдущей табл. 2.5 предполагалось, что при совпадении следующий символ считывается всегда.

Распознавание начинается, когда автомат находится в состоянии 20. Если текущим символом является «^», автомат переходит в состояние 22, считывая следующий символ. Если первый символ не равен «^» автомат переходит в состояние 21 и принимает часть входной цепочки, за которую он отвечает — запись степени в этом случае пуста.

Попав в состояние 22, автомат ожидает целое число. Распознаватель целого в нашем примере начинается с состояния 10 (нумерация не с единицы, чтоб подчеркнуть, что этот распознаватель не составляет законченный автомат, а только его часть). Поэтому в состоянии 22 запрограммирован переход в состояние 10. Но это не обычный переход, а переход с возвратом. По окончании распознавания целого должен произойти возврат из состояния 12 в состояние 23. Чтобы отметить особенность таких переходов, в таблицу введена графа «Вызов», в которой записывается «Да», если происходит переход с возвратом. Автомат, распознающий степень, как бы вызывает автомат, распознающий целое. Нетрудно догадаться, что суть происходящего подобна вызову одной подпрограммы из другой.

Значение 0 в графе «Переход» следует теперь воспринимать как возврат в состояние, следующее за тем, из которого произошел вызов. Важно понимать, что при разных вызовах это может быть разное состояние. Например, вызовы целого могут выполняться распознавателем степени, а могут — распознавателем слагаемого.

Табличный транслятор многочленов

Пользуясь выработанным форматом таблицы переходов, заполним ее для распознавателя многочленов (который будет, как свои части, включать уже рассмотренные распознаватели целого и степени). Имея в виду, что должен выполняться не только синтаксический анализ, но и трансляция многочлена, предусмотрим в таблице графу «Процедура», в которой будем записывать номер семантической процедуры (из числа предусмотренных раньше для примера про многочлены). Эта процедура будет вызываться при совпадении входного символа и символа в таблице. Если указан нулевой номер семантической процедуры, вызов не происходит (или процедура P_0 не выполняет никаких действий). Некоторые состояния добавляются в таблицу лишь для того, чтобы предусмотреть в нужные моменты выполнение необходимых семантических процедур (например, состояния 1, 6, 25).

Таблица 2.7. Таблица переходов *LL*(1)-распознавателя многочленов

Сост.	Символ	Переход	Ошибка	Вызов	Читать	Проц.	Примечание
Многочлен							
1	Л	2	–	–	–	3	Обнуление коэффициентов
2	+	5	–	–	+	4	
3	–	5	–	–	+	4	
4	Л	5	–	–	–	5	Нет знака спереди
5	Л	12	–	+	–	0	На 1-е слагаемое

Сост.	Символ	Переход	Ошибка	Вызов	Читать	Проц.	Примечание
6	Л	7	–	–	–	6	После 1-го слагаемого
7	+	10	–	–	+	4	Начало цикла
8	–	10	–	–	+	4	
9	⊥	0	+	–	–	7	Выход
10	Л	12	–	+	–	0	На слагаемое
11	Л	7	–	–	–	6	Конец цикла
Слагаемое							
12	Ц	15	–	–	–	0	
13	*x*	21	+	+	+	10	*a*=1
14	Л	0	–	–	–	9	После степени
15	Л	25	–	+	–	0	На целое
16	Л	17	–	–	–	8	После целого
17	*x*	19	–	–	+	0	
18	Л	0	–	–	–	11	*k*=0
19	Л	21	–	+	–	0	На степень
20	Л	0	–	–	–	9	Конец слагаемого
Степень							
21	^	23	–	–	+	0	

Сост.	Символ	Переход	Ошибка	Вызов	Читать	Проц.	Примечание
22	Л	0	–	–	–	13	p=1
23	Л	25	–	+	–	0	На целое
24	Л	0	–	–	–	12	
Целое							
25	Л	26	–	–	–	1	Инициализация
26	Ц	27	+	–	+	2	Первая цифра
27	Ц	27	–	–	+	2	Последующие цифры
28	Л	0	–	–	–	0	

Чтобы таблица была компактней, вместо «Да» и «Нет» в ней используются «+» и «–», «Л» обозначает любой символ, «Ц» — любую цифру, «⊥» — символ «конец текста».

Перед выполнением перехода с возвратом необходимо запоминать номер состояния, в которое автомат должен возвратиться. Недостаточно использовать для этого отдельную переменную, способную в каждый момент хранить номер только одного состояния. Дело в том, что вызовы могут быть вложенными: «Многочлен» вызывает «Слагаемое», «Слагаемое» вызывает «Целое» и «Степень», «Степень» — «Целое». Вызов, произошедший последним, закачивается первым. Это соответствует стековой дисциплине «последним пришел, первым ушел» (LIFO — Last In, First Out). Для запоминания номера состояния перед выполнением перехода с возвратом нужен стек.

Стек

Стек не раз потребуется нам в дальнейшем при рассмотрении различных методов трансляции. Определим его как абстрактный тип данных `tStack`.

Структуру данных назовем стеком, если:

1. Определены:
   - Тип данных, содержащихся в стеке (обозначим `tData`).
   - Процедура инициализации стека:
     ```
     Init(var S: tStack);
     ```
   - Процедура добавления элемента в стек:
     ```
     Push(var S: tStack; D: tData);
     ```
     (Push — проталкивать)
   - Процедура извлечения элемента из стека:
     ```
     Pop(var S: tStack; var D: tData);
     ```
     (Pop — выталкивать)
2. Процедуры `Push` и `Pop` подчиняются дисциплине «последним пришел — первым ушел».

На практике будут также полезны функции, позволяющие проверить, что стек не пуст и не заполнен:

- `NotEmpty(S: tStack): Boolean; { Стек не пуст }`
- `NotFull(S: tStack): Boolean;  { Стек не полон }`

Обратите внимание, что в этих определении ничего не говорится о том, как стек реализован и что конкретно он собой представляет. Это не мешает нам использовать вызовы названных процедур и функций. Принцип разделения спецификации и реализации — одна из фундаментальных основ программирования.

Реализация стека

Чтоб изложение было предметней, рассмотрим реализацию стека. Можно представить два основных варианта. В первом элементы хранятся в массиве, во втором — в списке. Запишем процедуры, предназначенные для работы со стеком, для случая, когда под стек используется массив. Вариант со списком предлагаю запрограммировать самостоятельно (хорошо, если при этом заголовки перечисленных выше процедур и функций не будут никак изменены).

Элементы помещаются в стек, начиная с первой ячейки массива. Число элементов, находящихся в данный момент в стеке (или, что то же самое, номер последнего занятого элемента массива) будем хранить в переменной `SP` (Stack Pointer — указатель стека). Массив элементов и `SP` будут полями записи типа `tStack`.

```
const
   nmax = … ; {Предельный объем стека}
type
   tStack = record
      a  : array [1..nmax] of tData;
      SP : integer;
   end;
```

После такого определения типа стека можно запрограммировать его инициализацию, которая сводится просто к обнулению указателя стека.

```
procedure Init(var S: tStack);
begin
   S.SP := 0;
end;
```

Не составляет труда и реализация процедур, выполняющих добавление и извлечение элементов.

```
procedure Push(var S: tStack; D: tData);
begin
   if S.SP < nmax then begin
      S.SP := S.SP + 1;
      S.a[S.SP] := D;
      end
```

```
   else
      Error('Переполнение стека');
end;

procedure Pop(var S: tStack; var D: tData);
begin
   if S.SP > 0 then begin
      D := S.a[S.SP];
      S.SP := S.SP - 1;
      end
   else
      Error('Стек пуст');
end;
```

Обе процедуры реагируют на ошибку с помощью вызова `Error`. Можно считать, что это та же процедура, которую мы используем в распознавателях, но, если стек применяется в иных обстоятельствах, это может быть другая процедура, решающая аналогичные задачи.

Осталось записать функции, позволяющие проверить состояние стека. Поскольку обычной работе соответствует ситуация, когда стек не пуст и не полон, в самих названиях двух этих функций содержится отрицание.

```
function NotEmpty(S: tStack): Boolean;
begin
   NotEmpty := S.SP > 0;
end;

function NotFull(S: tStack): Boolean;
begin
   NotFull := S.SP < nmax;
end;
```

Говорят, что элемент, помещенный на стек последним, находится на *вершине* стека, предпоследний — под вершиной. Это два *верхних* элемента стека. Такая терминология обусловлена естественными ассоциациями: стековой дисциплине подчиняются стопка подносов в столовой, монеты в пружинном кошельке для мелочи или патроны в магазине автоматического оружия. Во всех этих случаях элементы действительно добавляют сверху.

Поле SP в нашей реализации можно называть указателем *вершины* стека.

В дальнейшем, если в каком-либо алгоритме используется единственный стек, и тип его элементов ясен из контекста, мы будем записывать обращения к процедурам Push и Pop, опуская параметр-стек.

LL(1)-драйвер

Программу, которая выполняет распознавание, интерпретируя таблицу, называют драйвером. В нашей реализации драйвера будет использована таблица, тип которой определим как массив из записей.

```
tSynTable = array [1..n] of record
   Ch    : char;     { Символ    }
   Go    : integer;  { Переход   }
   Err   : Boolean;  { Ошибка    }
   Call  : Boolean;  { Вызов     }
   Read  : Boolean;  { Читать    }
   Proc  : integer;  { Процедура }
end;
```

Драйвер использует следующие переменные:

```
T     : tSynTable;   { Таблица          }
Ch    : char;        { Текущий символ   }
Err   : integer;     { Признак ошибки   }
Stack : tStack;      { Стек             }
i     : integer;     { Номер состояния  }
```

Приведенный в листинге 2.3 текст программы-драйвера не включает часть, которая заполняет таблицу перед началом работы. Программа содержит и другие условности, касающиеся обозначений произвольного символа и цифр в таблице.

Листинг 2.3. Драйвер табличного *LL*(1) анализатора

```
ResetText;
Init(Stack);
Push(Stack, 0);
Err := 0;
i := 1;
```

```
repeat
   if ( Ch=T[i].Ch ) or
      ( T[i].Ch = Любой ) or
      ( T[i].Ch = Цифра ) and (Ch in ['0'..'9'])
   then begin
      if T[i].Proc <> 0 then Proc(T[i].Proc);
      if T[i].Read then NextCh;
      if T[i].Go = 0 then
         Pop(Stack, i)
      else begin
         if T[i].Call then Push(Stack, i+1);
         i := T[i].Go;
      end;
      end
   else if T[i].Err then
      Err := i
   else
      i := i+1;
until (i=0) or (Err<>0);
```

Предполагается, что процедура `Proc` способна выполнять семантическую процедуру с заданным номером.

Драйвер завершает работу в двух случаях: когда в результате выполнения очередного возврата извлечен ноль из стека (он помещается в стек перед циклом), и при возникновении ошибки. Переменная `Err` получает значение, равное номеру состояния, в котором возникла ошибка.

Использование множеств символов

В ряде случаев одинаковые действия распознавателя выполняются для набора символов. Например, в задаче про многочлены знаки «+» и «–» всегда обрабатываются одинаково (строки 2, 3 и 7, 8 в таблице 2.7). Используя в таблице вместо отдельных символов множества, можно уменьшить число состояний (табл. 2.8).

```
tSynTable = array [1..n] of record
   ChSet : set of char; { Символы   }
   Go    : integer;     { Переход   }
   Err   : Boolean;     { Ошибка    }
   Call  : Boolean;     { Вызов     }
   Read  : Boolean;     { Читать    }
   Proc  : integer;     { Процедура }
end;
```

Таблица 2.8. Таблица переходов с множествами символов

Сост.	Символы	Переход	Ошибка	Вызов	Читать	Проц.	Примечание
Многочлен							
1	[#0..#255]	2	–	–	–	3	Обнуление коэффициентов
2	['+', '–']	4	–	–	+	4	
3	[#0..#255]	4	–	–	–	5	Нет знака спереди
4	[#0..#255]	10	–	+	–	0	На 1-е слагаемое
5	[#0..#255]	6	–	–	–	6	После 1-го слагаемого
6	['+', '–']	8	–	–	+	4	Начало цикла
7	[⊥]	0	+	–	–	7	Выход
8	[#0..#255]	10	–	+	–	0	На слагаемое
9	[#0..#255]	6	–	–	–	6	Конец цикла
Слагаемое							
10	['0'..'9']	13	–	–	–	0	
11	['x', 'X']	19	+	+	+	10	a=1
12	[#0..#255]	0	–	–	–	9	После степени

Сост.	Символы	Переход	Ошибка	Вызов	Читать	Проц.	Примечание
13	[#0..#255]	23	–	+	–	0	На целое
14	[#0..#255]	15	–	–	–	8	После целого
15	['x', 'X']	17	–	–	+	0	
16	[#0..#255]	0	–	–	–	11	k=0
17	[#0..#255]	19	–	+	–	0	На степень
18	[#0..#255]	0	–	–	–	9	После степени
Степень							
19	['^']	21	–	–	+	0	
20	[#0..#255]	0	–	–	–	13	p=1
21	[#0..#255]	23	–	+	–	0	
22	[#0..#255]	0	–	–	–	12	
Целое							
23	[#0..#255]	24	–	–	–	1	Инициали-зация
24	['0'..'9']	25	+	–	+	2	Первая цифра
25	['0'..'9']	25	–	–	+	2	Последую-щие цифры
26	[#0..#255]	0	–	–	–	0	

Множество [#0..#255] включает все символы, начиная с символа, имеющего порядковый номер 0, и заканчивая символом с порядковым номером 255.

Число состояний в нашем примере удалось уменьшить всего на два. Но получен еще ряд преимуществ: оказалось очень легко разрешить использование как строчной, так и прописной буквы *x* в записи многочлена (см. строку 15 в табл. 2.8), устранены условности, связанные с обозначением произвольного символа и множества цифр.

Использование множеств увеличивает расход памяти на представление таблицы. В случае если во второй графе таблицы помещается символ, он занимает один байт (при однобайтовой кодировке символов), если множество из символов — необходимо 32 байта для хранения каждого такого множества (1 бит на символ * 256 различных символов / 8 бит в одном байте).

Листинг 2.4. Драйвер табличного *LL*(1)-анализатора, использующий множества

```
ResetText;
Init(Stack);
Push(Stack, 0);
Err := 0;
i := 1;
repeat
   if Ch in T[i].ChSet then begin
      if T[i].Proc <> 0 then Proc(T[i].Proc);
      if T[i].Read then NextCh;
      if T[i].Go = 0 then
         Pop(Stack, i)
      else begin
         if T[i].Call then Push(Stack, i+1);
         i := T[i].Go;
      end;
      end
   else if T[i].Err then
      Err := i
   else
      i := i+1;
until (i=0) or (Err<>0);
```

Можно видеть, что программа-драйвер (листинг 2.4) совершенно не зависит от анализируемого языка. Это универсальный *LL*(1)-распознаватель (и транслятор). Синтаксис конкретного языка целиком определяется таблицей.

Обработка ошибок

Реакция на синтаксическую ошибку при использовании табличного анализатора может быть организована просто и естественно. Действительно, при обнаружении ошибки цикл анализа немедленно прекращается, а переменная `Err` при этом содержит номер состояния, в котором произошла ошибка. Значение `Err` однозначно определяет тип ошибки. Сообщения об ошибке могут быть помещены в отдельную графу таблицы, в те её строки, которые содержат «+» в столбце «Ошибка». Можно генерировать сообщения автоматически. Каждое такое сообщение может состоять из слова «Ожидается» с последующим перечислением множества символов `T[Err].ChSet`.

Кроме синтаксических необходимо обрабатывать ошибки, которые обнаруживаются семантическими процедурами. При использовании табличного распознавателя организовать реакцию на них также несложно. Одно из решений, позволяющее не менять программу-драйвер — присваивать переменной `Err` отрицательное значение при обнаружении ошибки семантической процедурой.

Рекурсивный спуск и табличный анализатор

Несмотря на совершенно разное устройство программ, принцип работы алгоритма рекурсивного спуска и табличного распознавателя похожи. Таблица состоит из частей, соответствующих нетерминалам грамматики, подобно тому, как программа, использующая рекурсивный спуск, состоит из распознающих процедур. Драйвер, работающий по таблице, реализует те же механизмы, которые действуют при вызовах процедур, в том числе

рекурсивных. Используемый драйвером явно стек, неявно присутствует при вызовах процедур.

Выбирая один из способов построения распознавателя, можно принимать во внимание следующие соображения.

- При табличном анализе лучше решается проблема выхода из состояния ошибки.
- Табличный анализатор допускает использование языка, не содержащего рекурсивных процедур.

Если речь идет о ручном программировании рекурсивного спуска и составлении таблицы разбора также вручную, то оказывается, что подготовка таблицы — это тоже своеобразное программирование на особом языке, предусматривающем условные переходы и переходы с возвратом[11].

- Программа-анализатор, написанная по методу рекурсивного спуска, наглядней таблицы. Программирование вручную рекурсивного спуска проще ручного составления таблицы.

Независимость программы-драйвера от входного языка должна, по идее, давать преимущества и создавать предпосылки для создания более гибких систем. В действительности, независимой от языка оказывается только часть, отвечающая за синтаксис. В то время как семантические процедуры, которые встраиваются в программу-транслятор, будут разными для различных языков.

- Независимость *LL*(1)-драйвера от входного языка создает преимущество при решении задачи синтаксического анализа. При наличии семантической обработки независимость теряется.

[11] Очень похоже на программирование на примитивных диалектах Бейсика с помощью IF, GOTO, GOSUB и нумерованных строк.

- Добавление семантической обработки к анализатору, написанному методом рекурсивного спуска, проще и естественней в силу лучшей структуры и наглядности такого анализатора.

Простое устройство синтаксической таблицы создает возможность ее автоматического формирования по описанию грамматики с помощью специальных программ — генераторов анализаторов. Однако с не меньшим успехом генератор анализаторов может порождать не таблицу, а программу на языке программирования, работающую по методу рекурсивного спуска. Проблемы, которые он при этом должен будет решать, будут в основном те же.

- Рекурсивный спуск и табличный *LL*(1)-анализ предоставляют сопоставимые возможности для автоматического построения анализатора.
- Анализатор, работающий по методу рекурсивного спуска, будет, скорее всего, быстрее табличного, интерпретирующего переходы по таблице, в то время как аналогичные переходы при рекурсивном спуске выполняются скомпилированной программой.

Трансляция выражений

Выражение — одно из основных понятий языков программирования. Подсистема, ответственная за трансляцию выражений, представляет собой важную и одну из самых сложных частей компилятора.

Польская запись

Обычная форма выражений предусматривает запись знака операции между операндами. Например:

$$a + b$$

$$a + b*c$$

$$(a + b)*c$$

Такую запись называют *инфиксной*.

В некоторых случаях используется префиксная (прямая польская) запись, когда обозначения операций записываются до операндов.

$$+ a\ b$$

$$+ a * b\ c$$

$$* + a\ b\ c$$

Постфиксная (обратная польская) запись требует, чтобы знаки операций записывались после соответствующих операндов. Приведенные выше инфиксные выражения в обратной польской записи будут выглядеть так:

$$a\ b +$$

$$a\ b\ c * +$$

$$a\ b + c *$$

Польской такая нотация названа в честь ее изобретателя — польского математика и логика Яна Лукасевича (Jan Ł ukasiewicz, 1878–1956). В этой книге обратная польская запись будет называться также *польской инверсной записью* (сокращенно ПОЛИЗ).

Польская запись не содержит скобок.

При записи в ПОЛИЗ операнды записываются в порядке следования в исходном выражении, операции — после своих операндов в порядке выполнения.

Таблица 2.9. Примеры записи выражений в обратной польской записи

Обычная (инфиксная) запись	Обратная польская запись
$(a + b)/(c + d)$	$a\ b + c\ d + /$

$a + b*c - a/(a + b)$	$a\ b\ c\ *\ +\ a\ a\ b\ +\ /\ -$
$\sqrt{1-\sin^2 x}$	1 x sin ^{2} – √

В последнем примере в табл. 2.9 «sin», «^{2}» и «√» следует рассматривать как знаки одноместных операций вычисления синуса, квадрата и квадратного корня соответственно.

Интерес к ПОЛИЗ обусловлен тем, что она удобна для вычисления выражений и как промежуточная форма представления выражений в трансляторе.

Принцип обратной польской записи может быть применен не только к выражениям, но и операторам языков программирования.

Алгоритм вычисления выражений в обратной польской записи

Выражения в обратной польской записи вычисляются с помощью стека. При чтении ПОЛИЗ слева направо операнды помещаются в стек, а операции применяются к верхним элементам стека. Результат выполнения операции возвращается в стек, заменяя собою операнды. По исчерпании выражения его значение находится на вершине стека.

В приведенном алгоритме (листинг 2.5) предполагается, что ПОЛИЗ состоит из отдельных *элементов*, и мы в состоянии отличать операнды от операций (с помощью множеств «Операнды», «Одноместные операции», «Двуместные операции»), а одноместные операции от двуместных. Предполагается также, что имеются только одноместные и двуместные операции[12].

[12] В общем случае, конечно, встречаются и многоместные операции. В этой роли могут выступать, например, функции нескольких переменных. Распространить алгоритм на этот случай не составляет труда.

Листинг 2.5. Алгоритм вычисления выражения в ПОЛИЗ

```
Читать(элемент); { Первый элемент ПОЛИЗ }
while элемент <> конец do begin
   if элемент in Операнды then
      Push(Значение(элемент))
   else if элемент in Одноместные операции then begin
      Pop(x);
      Push( Оп1(элемент, x) );
      end
   else if элемент in Двуместные операции then begin
      Pop(x2); Pop(x1);              {Обратите внимание }
      Push( Оп2(элемент, x1, x2) ); {на порядок x1 и x2}
   end;
   Читать(элемент);
end;
Pop(Значение выражения);
```

Операции со стеком в этом алгоритме выполняются с помощью процедур `Push` (в стек) и `Pop` (из стека), у которых параметр-стек для краткости не указан.

Функция `Значение` позволяет по записи элемента-операнда получить его значение. В зависимости от того, как кодируются операнды, является ли операнд константой или переменной, эта функция реализуется по-разному и здесь не детализируется.

Значением функций `Оп1` и `Оп2` является результат применения одноместной и двуместной операции, заданной значением первого параметра, к последующим параметрам. Например, если речь идет об операциях с вещественными, эти функции могут быть такими:

```
function Оп2(элемент: Элементы; x1, x2: real): real;
begin
   case элемент of
   сложение:  Оп2 := x1 + x2;
   вычитание: Оп2 := x1 - x2;
   умножение: Оп2 := x1*x2;
   деление:   Оп2 := x1/x2;
   end;
end;

function Оп1(элемент: Элементы; x: real): real;
begin
```

```
    case элемент of
    изменение знака: Оп1 := -x;
    синус:           Оп1 := sin(x);
    косинус:         Оп1 := cos(x);
    end;
end;
```

Схема трансляции выражений

Разные варианты использования обратной польской записи при трансляции выражений показаны на рисунке 2.34.

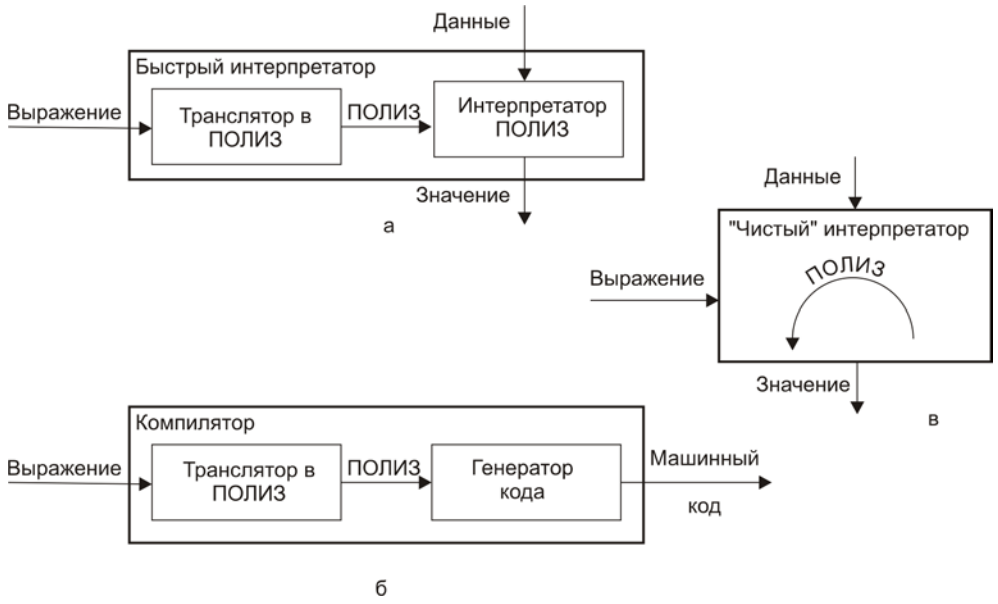


Рис. 2.34. Схемы трансляции выражений

Быстрый интерпретатор (рис. 2.34а) вначале преобразует выражение в ПОЛИЗ, а затем вычисляет его по алгоритму, приведенному выше. Компилятор (рис. 2.34б) использует ПОЛИЗ в роли промежуточного представления, по которому генератор формирует машинный код. В схеме «чистого интерпретатора» польская запись в явном виде не формируется, но совершаемые интерпретатором действия включают в себя те, что нужны для получения польской записи и те, что служат для вычисления с помощью стека.

Перевод выражений в обратную польскую запись

Преобразование инфиксного выражения в польскую запись может выполняться в ходе его синтаксического анализа. Чтобы определить действия, которые для этого нужны, достаточно обозначить на синтаксических диаграммах выражения семантические процедуры (рис. 2.35).

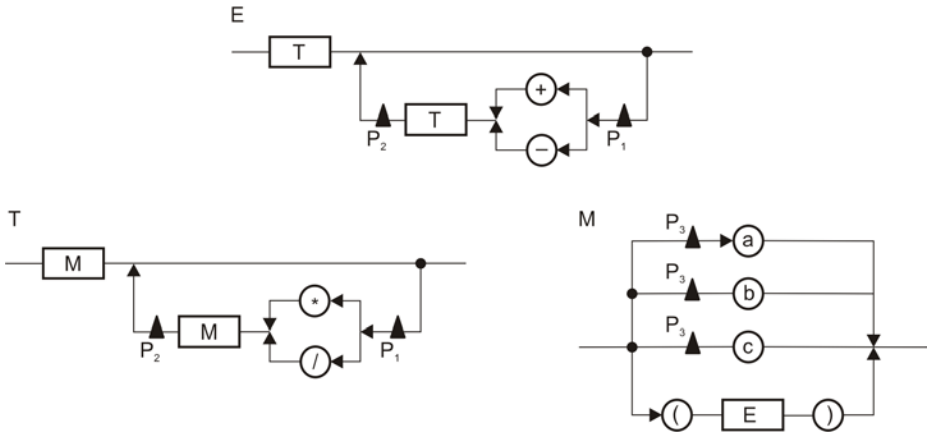


Рис. 2.35. Семантические процедуры для перевода выражения в обратную польскую запись

Определяя семантические процедуры, будем, как и всегда, считать, что переменная `Ch` содержит текущий символ входной цепочки. Вспомогательная переменная `Op` имеет символьный тип и способна хранить знак операции. Задача же состоит просто в том, чтобы вывести обратную польскую запись выражения.

```
P1: Запомнить(Ch);
P2: Вспомнить(Op); Write(Op);
P3: Write(Ch);
```

Реализация действий, обозначенных словами «Запомнить» и «Вспомнить», может быть различна, но в любом случае для запоминания знака операции необходимо использовать стек. Это обусловлено тем, что выражения (слагаемые, множители) могут быть вложенными и в определенные моменты работы алгоритма запомненными могут быть сразу несколько знаков. При этом запомненный первым должен быть «вспомнен» последним.

Если используется табличный распознаватель, то стек присутствует явно, и действие `Запомнить(Ch)` заменяется на `Push(Ch)`, а `Вспомнить(Op)` — на `Pop(Op)`.

Если применяется рекурсивный спуск, то достаточно сохранять знак операции в локальных переменных распознающих процедур «Выражение» и «Слагаемое». Не записывая программу целиком, приведем лишь процедуру для распознавания слагаемого.

```
procedure T;
var
   Op: char;
begin
   M; { Множитель }
   while Ch in ['+', '-'] do begin
      Op := Ch;
      NextCh;
      M;
      Write(Op);
   end;
end;
```

Алгоритм Э. Дейкстры перевода выражений в обратную польскую запись (метод стека с приоритетами)

Не следует думать, что синтаксически управляемая трансляция выражений в ПОЛИЗ, рассмотренная выше, — единственный способ такого перевода. Существуют и другие методы. Рассмотрим простой и эффективный алгоритм Э. Дейкстры, основанный на использовании стека операций. Он состоит в следующем:

1. Каждому знаку операции присваивается числовой приоритет. Операции, выполняемые в первую очередь, имеют больший приоритет. Приоритеты операций начинаются с 2.
2. Открывающим скобкам присваивается приоритет 0, закрывающим — 1.
3. Входная строка считывается слева направо. Операнды по мере их считывания помещаются в выходную строку.

4. Знаки перед записью в выходную строку помещаются в стек (рис. 2.36) в соответствии со следующей дисциплиной:

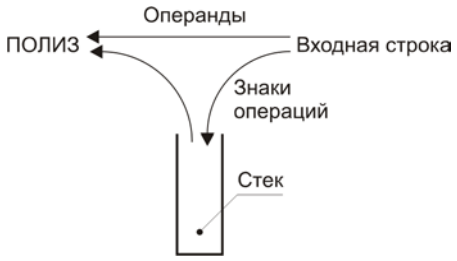


Рис. 2.36. Использование стека в алгоритме Э. Дейкстры

- Открывающая скобка (знак с приоритетом 0) помещается в стек.
- Если приоритет знака операции больше приоритета знака на вершине стека или стек пуст, новый знак добавляется в стек.
- Если приоритет знака меньше или равен приоритету знака на вершине стека, из стека в выходную строку «выталкиваются» все знаки с приоритетами большими или равными приоритету входного знака. После этого входной знак записывается в стек.
- Закрывающая скобка выталкивает в выходную строку все знаки до открывающей скобки. Открывающая скобка удаляется из стека. Закрывающая и открывающая скобка в выходную строку не записываются.

5. После просмотра всех символов входной строки из стека выталкиваются оставшиеся знаки.

Этот метод может быть распространен и на функции, переменные с индексами и др.

Интерпретация выражений

Рассмотрим, каким образом вычисляется выражение по его записи, без явного преобразования в ПОЛИЗ. Очевидно, что мож-

но действовать, комбинируя действия, связанные с переводом в ПОЛИЗ с действиями, которые выполняются при вычислении выражения по его польской записи. Вместо вывода операндов и операций в выходную строку надо помещать операнды в стек, а операции выполнять, применяя их к верхним элементам стека и возвращая на стек результат.

Способ решения задачи определен с помощью синтаксических диаграмм (рис. 2.37) и семантических процедур. В отличие от предыдущих примеров, добавлена возможность записывать знак перед первым слагаемым.

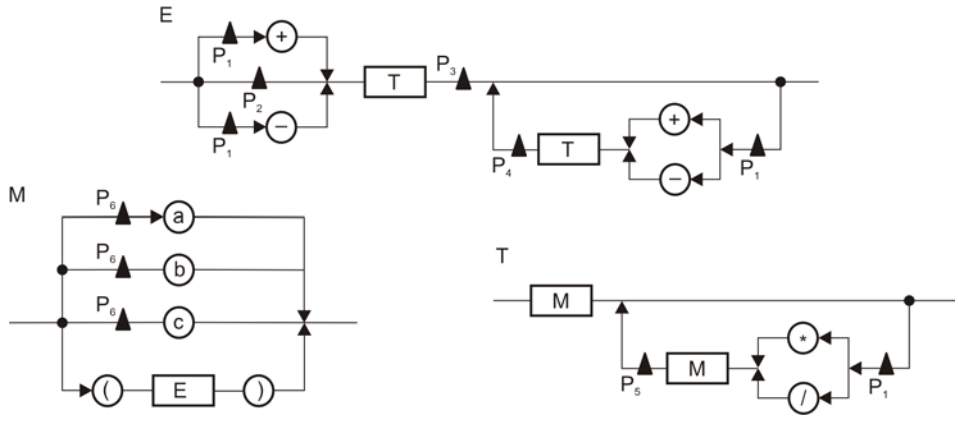


Рис. 2.37. Семантические процедуры для интерпретации выражений

```
P1: Запомнить(Ch);
P2: Запомнить('+');
P3: Вспомнить(Op);
    if Op = '-' then begin
       Pop(x); Push(-x);
    end;
P4: Вспомнить(Op);
    Pop(x2); Pop(x1);
    if Op = '+' then  Push(x1 + x2)
    else {Op = '-'}   Push(x1 – x2);
P5: Вспомнить(Op);
    Pop(x2); Pop(x1);
    if Op = '*' then  Push(x1*x2)
    else {Op = '/'}   Push(x1/x2);
P6: Push(Значение(Ch));
```

Здесь процедуры `Push` и `Pop` оперируют стеком операндов, в то время как `Запомнить` и `Вспомнить` явно или неявно обращаются к стеку операций.

Семантическое дерево выражения

Как уже говорилось, задача разбора состоит в построении дерева вывода (синтаксического дерева). Однако, ни один из рассмотренных нами алгоритмов не строит это дерево в явном виде. Структура входной цепочки, которую представляет дерево, лишь отражается в последовательности действий, совершаемых синтаксическим анализатором.

На практике важную роль может играть семантическое дерево. Его построение облегчает решение многих задач трансляции. Дерево может использоваться в качестве промежуточного представления программы.

Рассмотрим построение и использование семантического дерева на примере простых арифметических выражений, которые уже неоднократно использовались в этой главе (см. диаграммы на рис. 2.28–2.29). На рис. 2.38 показаны деревья трех выражений.

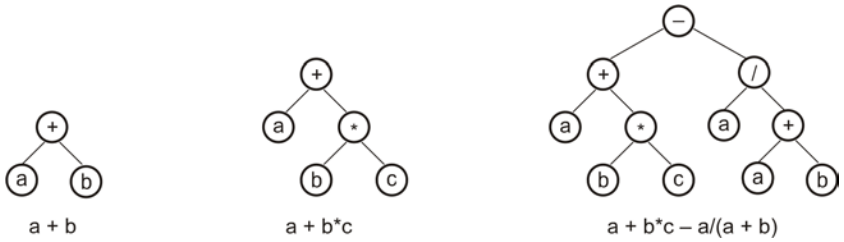


Рис. 2.38. Примеры семантических деревьев

Представление дерева

Для представления семантического дерева выражения будем использовать динамические переменные и указатели. Каждая вершина дерева содержит операнд или операцию. Поскольку в нашем примере используются однобуквенные операнды *a*, *b*, *c* и обозначения операций, состоящие из одного символа, достаточ-

но предусмотреть в каждой записи, соответствующей вершине дерева, поле Ch символьного типа.

```
type
   tPtr = ^tNode;     {Тип указателя на вершину}
   tNode = record     {Тип вершины дерева      }
      Ch     : char;     {Символ вершины                 }
      Counter: integer; {Счетчик временных переменных }
      Left   : tPtr;     {Указатель на левое поддерево }
      Right  : tPtr;     {Указатель на правое поддерево}
   end;
```

Поля Left и Right служат для установления связей в дереве. В нашем примере дерево будет двоичным, поскольку все операции двуместные. На поле Counter можно пока не обращать внимания, оно потребуется позже для решения одной из задач. На рис. 2.39 можно видеть дерево выражения *a* + *b***c*, состоящее из динамических переменных типа tNode.

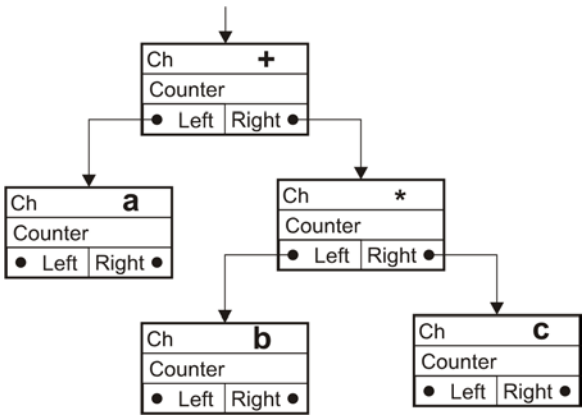


Рис. 2.39. Двоичное дерево выражения *a* + *b***c*

Построение дерева

Будем строить дерево с помощью программы, выполняющей синтаксический анализ выражения методом рекурсивного спуска. Потребуются следующие глобальные переменные:

```
var
   Ch       : char;     { Очередной символ              }
   Tree     : tPtr;     { Дерево                        }
   Counter  : integer;  { Счетчик временных переменных }
```

Задачей распознающих процедур Expression (выражение), Term (слагаемое) и Multiplier (множитель) кроме синтаксического анализа будет построение дерева того подвыражения, которое они распознали. Expression строит дерево выражения, Term — слагаемого, Multiplier — множителя. Построенное дерево будет выходным параметром каждой из этих процедур. Чтобы построить дерево Tree всего выражения обращаемся к Expression:

```
Expression(Tree);
```

Expression строит дерево выражения, обращаясь к распознавателю слагаемого для получения ссылок на деревья слагаемых. Связи в дереве выстраиваются так, чтобы это соответствовало выполнению операций «+» и «–» в цепочке слагаемых в порядке слева направо.

```
procedure Expression(var Tree : tPtr);
var
   t : tPtr;
begin
   Term(Tree);
   while Ch in ['+', '-'] do begin
      New(t);      {Новая вершина}
      t^.Ch := Ch; {содержит знак}
      t^.Left := Tree; {Слева - старое дерево          }
      NextCh;
      Term(t^.Right);  {Справа - новое слагаемое       }
      Tree := t;   {Указатель дерева - на новую вершину}
   end;
end;
```

Рисунок 2.40 иллюстрирует последовательность построения дерева для выражения $a - b - c$.

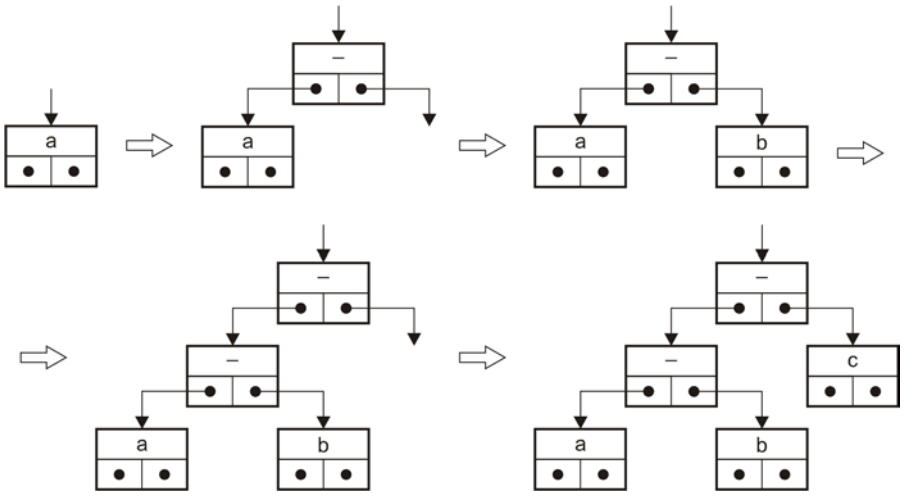


Рис. 2.40. Построение дерева выражения *a – b – c*

Анализатор слагаемого `Term`, действует подобно процедуре `Expression`.

```
procedure Term(var Tree : tPtr);
var
   t : tPtr;
begin
   Multiplier(Tree);
   while Ch in ['*', '/'] do begin
      New(t);
      t^.Ch := Ch;
      t^.Left := Tree;
      NextCh;
      Multiplier(t^.Right);
      Tree := t;
   end;
end;
```

Процедура-распознаватель множителя, встречая операнд, не заключенный в скобки, создает вершину, которая будет листом дерева.

```
procedure Multiplier(var Tree : tPtr);
begin
   if Ch in ['a', 'b', 'c'] then begin
      New(Tree);
      Tree^.Ch := Ch;
      Tree^.Left := nil;
      Tree^.Right := nil;
      NextCh
```

```
      end
   else if Ch = '(' then begin
      NextCh;
      Expression(Tree);
      if Ch = ')' then
         NextCh
      else
         Error('Ожидается ")"');
      end
   else
      Error('Ожидается a, b, c или "("');
end;
```

Дерево построено.

Получение постфиксной записи

Имея в распоряжении семантическое дерево, легко получить обратную польскую запись выражения. Для этого достаточно выполнить обход дерева (посещение всех его вершин) в порядке снизу вверх и слева направо, когда для каждой вершины вначале посещается ее левое поддерево, потом правое и в последнюю очередь сама вершина (рис. 2.41а). Посещая каждую вершину, надо просто выводить ее содержимое (значение поля `Ch`).

Рис. 2.41. Различный порядок обхода дерева

Рекурсивный обход дерева в указанном порядке выполняет процедура `Postfix`.

```
{ Обход дерева t снизу вверх }
procedure Postfix(t: tPtr);
begin
   if t <> nil then begin
      Postfix(t^.Left);  { Обход левого поддерева  }
      Postfix(t^.Right); { Обход правого поддерева }
      Write(t^.Ch);      { Посещение вершины t     }
   end;
end;
```

Тривиальным случаем, когда не происходит рекурсивных вызовов (и не выполняется вообще никаких действий) здесь является ситуация, когда дерево пусто (`t = nil`).

Чтобы выполнить обход фактически и получить обратную польскую запись, нужно вызвать `Postfix`, указав фактическим параметром имеющееся дерево:

```
Postfix(Tree);
```

Получение префиксной записи

Префиксная (прямая польская) запись получается обходом дерева сверху вниз и слева направо.

```
{ Обход дерева t сверху вниз }
procedure Prefix(t: tPtr);
begin
   if t <> nil then begin
      Write(t^.Ch);
      Prefix(t^.Left);
      Prefix(t^.Right);
   end;
end;
```

Обход дерева `Tree` выполняется вызовом `Prefix(Tree)`. В таблице 2.10 приведен результат работы процедур `Postfix` и `Prefix` для трех выражений, чьи деревья, можно видеть на рис. 2.38.

Таблица 2.10. Результат работы процедур Postfix и Prefix

Выражение	Postfix	Prefix
a + b	ab+	+ab
a + b*c	abc*+	+a*bc
a + b*c – a/(a + b)	abc*+aab+/–	–+a*bc/a+ab

Функциональная запись выражения

Согласитесь, что если обратная польская запись стала уже привычной, то префиксная воспринимается пока с трудом. Большую наглядность ей можно придать, если считать, что каждая опера-

ция — это функция с двумя аргументами. Обозначение функции записывается перед обозначением аргументов: *f*(*x*, *y*). Заменив знаки операций названиями функций: ADD — сложение, SUB — вычитание, MUL — умножение, DIV — деление, а также предусмотрев вывод запятых и скобок, можно получить запись выражения в функциональном стиле. При этом она, по сути, остается префиксной, лишь несколько меняя внешний вид.

При посещении вершин дерева надо различать, является ли вершина внутренней или листом. Для вершин-операций надо выводить название функции, скобки и запятую между аргументами. Попав в вершину-операнд, достаточно вывести символ, хранящийся в этой вершине, и даже не нужно посещать поддеревья, поскольку такая вершина их не имеет, являясь листом дерева.

Чтобы различать операции и операнды, используем множество

```
Operators = ['+', '-', '*', '/'].
```

Обход дерева по-прежнему надо выполнять сверху вниз и слева направо.

```
{ Функциональная запись дерева t }
procedure Functional(t: tPtr);
begin
   if t <> nil then begin
      if t^.Ch in Operators then begin
         WriteMnemo(t^.Ch); { Название функции }
         Write('(');
         Functional(t^.Left);
         Write(', ');
         Functional(t^.Right);
         Write(')');
         end
      else { Лист }
         Write(t^.Ch);
   end;
end;
```

Вывод названия функции, соответствующего знаку операции, выполняет такая процедура:

```
procedure WriteMnemo(Op: char);
begin
```

```
    case Op of
    '+': Write('ADD');
    '-': Write('SUB');
    '*': Write('MUL');
    '/': Write('DIV');
    end;
  end;
```

Чтобы получить функциональную запись выражения, для которого построено дерево `Tree`, надо написать: `Functional(Tree)`. Результат такого вызова для трех случаев показан в табл. 2.11.

Таблица 2.11. Функциональная запись выражений

Выражение	Функциональная запись
`a + b`	`ADD(a, b)`
`a + b*c`	`ADD(a, MUL(b, c))`
`a+b*c–a/(a+b)`	`SUB(ADD(a, MUL(b, c)), DIV(a, ADD(a, b)))`

Четверки

Четверки — одна из форм промежуточного представления программы в трансляторе. Представление четверками удобно для выполнения оптимизирующих преобразований программы, увеличивающих скорость ее работы и уменьшающих размер. Поэтому четверки используются в оптимизирующих компиляторах.

Четверку можно рассматривать как команду некоторого компьютера, имеющую следующий формат:

```
Операция, Операнд-1, Операнд-2, Результат
```

Например, выражение *a* + *b* может быть преобразовано в четверку

```
+, a, b, t
```

Здесь *t* обозначает временную переменную, в которую помещается результат сложения.

Некоторые реальные компьютеры действительно имели и имеют аналогичную (трехадресную) систему команд, когда за кодом

операции в команде следуют адреса двух операндов и адрес результата. Интересно заметить, что это либо старые машины 60-х годов (такие, как советская М-20), либо, наоборот, современные процессоры с сокращенным набором команд (RISC). Только если в старых трехадресных машинах в команде указывались адреса ячеек памяти, то в командах RISC-процессоров — номера регистров.

Не надо, однако, думать, что порождение четверок имеет смысл только при компиляции для трехадресных машин. Четверки — универсальный машинно-независимый способ промежуточного представления программы, по которому затем может быть получен машинный код для компьютеров различной архитектуры.

Одна операция в арифметическом выражении порождает одну четверку. Выражение в целом — последовательность четверок. В наших примерах в роли результата всегда будут фигурировать временные переменные, которые будем обозначать *t*1, *t*2, и т. д. (от temporary — временный). Номер четверки будет и номером временной переменной. Результат вычисления выражения сохраняется в последней временной переменной.

Получить четверки можно обходом семантического дерева снизу вверх. Последняя четверка соответствует последней выполняемой операции, которая находится в корне дерева. Посещать при обходе нужно только внутренние вершины дерева, соответствующие операциям.

Для назначения номеров временным переменным и, соответственно, четверкам мы уже предусмотрели переменную `Counter` и поле `Counter` записи о вершине дерева. При выводе очередной четверки значение `Counter` увеличивается на единицу (первоначально `Counter := 0`) и используется для нумерации временной переменной, являющейся результатом в этой четверке. Это же

значение записывается в вершину дерева, для которой выводилась четверка, становясь номером этой вершины.

Если левой (правой) дочерней вершиной той внутренней вершины, для которой выводится четверка, является элементарный операнд (*a*, *b* или *c*), его обозначение выводится в качестве первого (второго) операнда четверки. Если дочерняя вершина внутренняя, выводится «t» и номер дочерней вершины, записанный ранее в ее поле Counter.

```
{ Обход дерева t для получения четверок }
procedure Tetrads(t: tPtr);
begin
   if (t <> nil) and (t^.Ch in Operators) then begin
      Tetrads(t^.Left); {Четверки для левого  поддерева}
      Tetrads(t^.Right);{Четверки для правого поддерева}
      { Четверка для вершины t }
         Write{Mnemo}(t^.Ch); { Операция }
         Operand(t^.Left);    { Первый операнд }
         Operand(t^.Right);   { Второй операнд }
         { Результат }
            Counter := Counter + 1;
            WriteLn(', t', Counter);
      { Пронумеровать вершину }
         t^.Counter := Counter;
   end;
end;
```

Вывод операндов выполняется процедурой Operand:

```
procedure Operand(p : tPtr);
begin
   if p^.Ch in Operators then
      Write(', t', p^.Counter)
   else
      Write(',  ', p^.Ch);
end;
```

Получить четверки для выражения, по которому построено дерево Tree, можно вызовом Tetrads(Tree). Последовательность четверок для выражения, состоящего из единственного операнда, будет пуста.

Заменив Write(t^.Ch) на WriteMnemo(t^.Ch), можно получить четверки с мнемоническими обозначениями операций (см.

табл. 2.12). Такие четверки еще больше похожи на команды машинного языка.

Таблица 2.12. Четверки

Выражение	Четверки	Четверки с мнемоникой
`a + b`	`+, a, b, t1`	`ADD, a, b, t1`
`a + b*c`	`*, b, c, t1` `+, a, t1, t2`	`MUL, b, c, t1` `ADD, a, t1, t2`

`a + b*c – a/(a + b)`	`*, b, c, t1`
	`+, a, t1, t2`
	`+, a, b, t3`
	`/, a, t3, t4`
	`-, t2, t4, t5`
		`MUL, b, c, t1`
		`ADD, a, t1, t2`
		`ADD, a, b, t3`
		`DIV, a, t3, t4`
		`SUB, t2, t4, t5`

Вычисление (интерпретация) выражения по его семантическому дереву

Выполнив обход дерева снизу вверх, как при формировании обратной польской записи и четверок, можно вычислить значение выражения. Надо только уметь извлекать значения элементарных операндов. Вообще-то, *a*, *b* и *c*, исполняющие эту роль в наших примерах, могут быть переменными или константами, их значения могут считываться из ячейки памяти или, например, с какого-либо прибора, подключенного к компьютеру. В зависимости от этого и способ получения их значений будет различным. Но, чтобы не усложнять изложение, примем, что *a* просто равно 1, *b* равно 2, а *c*=3.

Процедуру `Calculate`, выполняющую обход дерева с целью вычисления значения выражения, снабдим выходным параметром `x`, обозначающим найденное значение. Будем считать, что это значение вещественное.

```
{ Обход дерева t для вычисления выражения }
procedure Calculate(t : tPtr; var x : real);
var
   x1, x2 : real;
begin
   if t<>nil then begin
      { Вычислить в левом поддереве }
         Calculate(t^.Left, x1);
      { Вычислить в правом поддереве }
         Calculate(t^.Right, x2);
      { Найти значение в вершине t }
         case t^.Ch of
         '+':  x := x1 + x2;
         '-':  x := x1 - x2;
         '*':  x := x1*x2;
         '/':  x := x1/x2;
         'a':  x := 1.0; { Так          }
         'b':  x := 2.0; { только       }
         'c':  x := 3.0; { для примера }
         end;
   end;
end;
```

Чтобы, допустим, напечатать значение выражения, для которого построено дерево `Tree`, можно записать:

```
var
   y : real;
…
Calculate(Tree, y);
Writeln('Значение равно ', y);
```

Поскольку результатом работы `Calculate` является единственное значение, имеет смысл оформить вычисление выражения по его семантическому дереву как функцию. Назовем ее `Value` — «значение».

```
{ Значение выражения, заданного деревом t }
function Value(t : tPtr): real;
begin
   if t<>nil then begin
      case t^.Ch of
      '+':  Value := Value(t^.Left) + Value(t^.Right);
      '-':  Value := Value(t^.Left) - Value(t^.Right);
      '*':  Value := Value(t^.Left) * Value(t^.Right);
      '/':  Value := Value(t^.Left) / Value(t^.Right);
      'a':  Value := 1.0;
      'b':  Value := 2.0;
```

```
        'c':  Value := 3.0;
        end;
    end;
end;
```

Согласитесь, получилось изящно. Теперь для печати значения достаточно написать:

```
Writeln('Значение равно ', Value(Tree));
```

Результат вычисления нескольких выражений с помощью `Calculate` и `Value` приведен в таблице 2.13.

Таблица 2.13. Значения выражений, вычисленные обходом дерева

Выражение	Значение
a	1.000
b	2.000
c	3.000
a + b	3.000
a + b*c	7.000
a + b*c – a/(a + b)	6.667

Хочется отметить важный нюанс. Программы, построенные по образцу `Calculate` с одной стороны и `Value` — с другой, могут оказаться неэквивалентными. Дело в том, что в некоторых языках программирования не фиксирован порядок вычисления операндов арифметической операции. В этом случае запись `Value(t^.Left) + Value(t^.Right)` совсем не обязательно означает, что вычисление значения в левом поддереве будет выполнено раньше вычисления в правом. Если операндами выражения являются функции с побочным эффектом, то в зависимости от порядка вычисления операндов значение выражения может оказаться различным. А вот программа подобная `Calculate` гарантирует, что вначале посещается левое поддерево и лишь потом — правое.

Получается, что `Value` будет всегда вычислять левый операнд раньше правого только в том случае, если это гарантирует реализация того языка, на котором сама функция `Value` (или подобная ей) написана.

Упражнения для самостоятельной работы

Ниже приводятся описания ряда конструкций (выражений, уравнений, элементов программ) и варианты их обработки. В качестве упражнений предлагается:

- Построить синтаксические диаграммы для этих конструкций.
- Запрограммировать синтаксический анализатор методом рекурсивного спуска. Анализатор должен либо сообщать о том, что конструкция записана верно, либо выдавать сообщение об ошибке с указанием места ее обнаружения.
- Построить синтаксическую таблицу и реализовать табличный *LL*(1)-анализатор.
- Дополнить анализаторы семантическими процедурами, выполняющими содержательную обработку.

Варианты заданий

1. Арифметическое выражение. Элементами выражения являются имена переменных, вещественные числа, функции (названия функций произвольные), знаки арифметических операций, круглые скобки.
2. Сумма — последовательность натуральных чисел и имен, разделенных знаками плюс и минус. Возможен и знак перед первым слагаемым.
3. Сумма вещественных чисел.
4. Квадратное уравнение с целыми коэффициентами.
5. Линейное алгебраическое уравнение с N неизвестными (X_k, k=1,2, ..., N) и постоянными целыми коэффициентами.

6. Сумма обыкновенных дробей.

7. Комплексное число (с целочисленными значениями действительной и мнимой частей).

8. Линейное однородное дифференциальное уравнение с постоянными целочисленными коэффициентами.

9. Произведение вида $(X - A_1)(X - A_2)(X - A_3) \ \ldots \ (X - A_n)$, где A_i, $(i=1..n)$ — целые числа.

10. Линейное уравнение с n неизвестными и постоянными целочисленными коэффициентами. Имена переменных произвольные.

11. Мультипликативная функция n переменных: $\Phi = A_0 \prod_{i=1}^{n} X_i$,где A_0 — целая константа; X_i, $(i=1..n)$ — имена.

12. Отношение *Операнд* ⊗ *Операнд*, где *Операнд* – целое или имя; ⊗ – знак отношения (>, <, =, <>, >=, <=).

13. Условие — отношения (см. выше), объединенные операциями «**и**» и «**или**». Операнды — однобуквенные имена.

14. Сумма произведений – последовательность слагаемых, представляющих собой произведение нескольких операндов. Операнды — имена.

15. Последовательность записанных через запятую элементов двумерных массивов. Индексы – натуральные числа.

16. Серия команд присваивания, разделенных «;». В каждой команде слева записано имя, справа — натуральное число. Знак присваивания «:=».

17. Сумма функций — последовательность функций с произвольными именами, разделенных знаками «+» и «–». Аргументы функций — имена или натуральные числа в скобках.

18. Бесскобочное арифметическое выражение с функциями. Все операнды — однобуквенные имена. Аргументы функций записываются в скобках.

19. Оператор `Write` (элементы списка — имена, строки; параметры форматов — целые числа).

20. Оператор `Read` с элементами списка ввода — именами простых переменных и элементами линейных массивов. Индексы массивов — целые числа.

21. Переменная языка Паскаль (включая элементы массива, компоненты записей и динамические переменные). Индексы — целые числа.

22. Простой тип языка Паскаль (имя, перечислимый тип, ограниченный тип). Константы ограниченного типа — целые числа или символы.

23. Раздел констант языка Паскаль. Предполагаются константы только целого и символьного типа.

24. Числовой ряд вида $1 + 1/n_1 + 1/n_2 + \ldots$, где n_1, n_2, ... — натуральные числа.

25. Числовой ряд вида $1 + 1/(n_1*m_1) + 1/(n_2*m_2) + \ldots$, где n_1, n_2, ..., m_1, m_2, ... — натуральные числа.

26. Числовой ряд вида $1 + 1/n_1\text{^}2 + 1/n_2\text{^}2 + \ldots$, где n_1, n_2, ... — натуральные числа.

27. Многочлен от x с рациональными коэффициентами.

28. Дробно-линейная функция от x: $(a_1*x + b_1)/(a_2*x + b_2)$, где a_1, b_1, a_2, b_2 — целые числа (при равенстве 0 могут не записываться).

29. Иррациональная сумма вида $x\text{^}(n_1/m_1) + x\text{^}(n_2/m_2) - \ldots + \ldots$, где n_1, n_2, ..., m_1, m_2, ... — натуральные числа.

30. Последовательность векторов: (x_1, y_1, z_1), (x_2, y_2, z_2), ..., где x, y, z — целочисленные координаты векторов.

31. Многочлен от x: $(x - x_1)^{k_1}*(x - x_2)^{k_2}*...$, где x_1, x_2, ... — целые числа; k_1, k_2, ... — неотрицательные целые.

32. Многочлен от x, записанный по схеме Горнера: $a_0 + x(a_1 + x(a_2 + x(...)))$, где a_0, a_1, ... — целые числа.

33. Дробно-рациональная функция: $(x - a_1)(x - a_2)$ .../((x - b_1)(x - b_2) ...), где a_1, a_2, ..., b_1, b_2, ... — целые числа.

34. Уравнение плоскости вида $Ax + By + Cz + D = 0$ с целыми коэффициентами.

35. Уравнение плоскости в отрезках: $x/a + y/b + z/c = 1$, где a, b, c — ненулевые целые числа.

36. Линейная функция двух переменных (x и y) с вещественными коэффициентами.

37. Множество-константа языка Паскаль с базовым типом `char`.

38. Множество-константа языка Паскаль с базовым типом `integer`.

Варианты обработки

1. Вычислить выражение.

2. Считая, что имена обозначают неизвестные, запросить у пользователя их значения и вычислить сумму. Значение одной и той же переменной, встречающейся в выражении больше одного раза должно запрашиваться только один раз.

3. Вычислить сумму.

4. Найти корни уравнения.

5. Программа обеспечивает ввод числовых значений неизвестных и проверяет, являются ли эти значения решениями уравнения. Вводятся только значения неизвестных, фактически

имеющихся в уравнении. Если некоторая неизвестная встречается дважды, программа должна запрашивать ее значение только один раз.

6. Вычислить сумму.
7. Вычисляется z^2, z^3, z^4... z^{10}, где z — введенное комплексное число.
8. Вычисляются корни характеристического уравнения.
9. Программа запрашивает значение x и вычисляет произведение.
10. Программа обеспечивает ввод числовых значений неизвестных и проверяет, являются ли эти значения решениями уравнения. Вводятся только значения неизвестных, фактически имеющихся в уравнении. Если некоторая неизвестная встречается дважды, программа должна запрашивать ее значение только один раз.
11. Программа запрашивает вещественные значения входящих в выражение переменных и вычисляет значение функции. Значение каждой переменной вводится лишь однажды.
12. Полагая, что имена — это идентификаторы целочисленных переменных, программа запрашивает их значения и проверяет истинность отношения.
13. Полагая, что имена суть идентификаторы целочисленных переменных, программа запрашивает их значения и проверяет истинность условия. Использовать короткую схему вычисления логических выражений, не требуя ввода значений переменных, если истинность условия может быть определена без них.
14. Программа запрашивает вещественные значения, входящих в выражение переменных и вычисляет значение суммы. Значение каждой переменной вводится лишь однажды.

15. Считая, что все массивы имеют размер 10*10, программа проверяет корректность задания индексов, и формирует текст программы на Паскале, которая заполняет все массивы, имена которых встретились в выражении, случайными целыми числами, элементам, встретившимся в выражении должны быть присвоены нулевые значения. Сгенерированная программа должна распечатывать содержимое всех этих массивов.
16. Сгенерировать программу на Паскале, состоящую из этих присваиваний и оператора `Write`, печатающего значения этих переменных.
17. Сгенерировать синтаксически правильную (не обязательно осмысленную) программу на Паскале, частью которой является оператор присваивания, содержащий в правой части исходное выражение.
18. Спрашивая у пользователя вещественное значение каждой переменной и значение каждой функции при заданном числовом значении аргумента программа-интерпретатор вычисляет выражение.
19. Выполнить оператор, запросив вещественные значения переменных.
20. Сгенерировать синтаксически правильную программу на Паскале, первым в которой записан данный оператор.
21. Сгенерировать синтаксически правильную программу на Паскале, содержащую описание этой переменной.
22. Сгенерировать программу на Паскале, которая содержит цикл, параметр которого меняется от минимального до максимального значения данного типа. Если тип задан именем, то предполагается, что это имя одного из стандартных дискретных типов. Программа должна проверить это.

23. Сгенерировать программу на Паскале, в которой данные константы определены как типизированные.

24. Вычислить точную сумму ряда. Результат представить в виде обыкновенной дроби.

25. Вычислить точную сумму ряда. Результат представить в виде обыкновенной дроби.

26. Вычислить точную сумму ряда. Результат представить в виде обыкновенной дроби.

27. Программа запрашивает рациональное значение *x*, а затем вычисляет значение многочлена, представив ответ в виде несократимой дроби.

28. Запросив целое значение *x*, вычислить точное значение функции, представив его в виде обыкновенной дроби.

29. Запросив вещественное значение x, вычислить сумму.

30. Вычислить длину ломаной, рассматривая данные векторы как радиус-векторы узлов ломаной в трехмерном пространстве.

31. Вычисляется многочлен при заданном *x*.

32. Вычисляется многочлен при заданном *x*.

33. Запросить рациональное значение *x*. Вычислить значение функции при заданном *x*, представив это значение в виде обыкновенной дроби.

34. Вычисляются коэффициенты плоскости, перпендикулярной заданной и проходящей через ось *Ox*.

35. Если плоскость не параллельна ни одной из координатных осей, вычислить объем тетраэдра, образованного заданной плоскостью и координатными плоскостями. Если параллельна, сообщить об этом.

36. Программа запрашивает значения x и y и вычисляет значение функции.

37. Программа запрашивает символ и отвечает на вопрос о принадлежности этого символа заданному множеству.

38. Программа запрашивает целое число и отвечает на вопрос о принадлежности этого числа заданному множеству.

ГЛАВА 3. ТРАНСЛЯЦИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Вооружившись теорией трансляции, перейдем к рассмотрению методов конструирования компиляторов. Будем рассматривать вопросы разработки транслятора на конкретном примере. Большая часть этой главы посвящена созданию компилятора (он же окажется интерпретатором) простого языка программирования.

Описание языков программирования

Нельзя говорить о разработке транслятора, если нет точного и строгого описания языка, для которого этот транслятор изготавливается. Конечно при создании новых языков ситуация не всегда так определённа. Документ, определяющий язык — его спецификация, — бывает, пишется одновременно с разработкой компилятора. В ходе реализации (создания транслятора) в язык могут вноситься изменения и уточнения. Но рано или поздно спецификация должна быть сформулирована. Для многих распространенных языков существуют международные стандарты. И, если речь идет об уже существующем языке, без его спецификации разработчику компилятора обойтись невозможно.

Важно подчеркнуть, что не любое описание языка годится на роль документа, руководствуясь которым можно программировать компилятор. О различных языках программирования написано множество книг. Это и учебники, и «наиболее полные руководства», и краткие справочники, и пособия «для чайников», а также краткие и вводные курсы. Но ни одна из таких книг не годится в качестве описания языка при создании компилятора. Эти книги, конечно, нужны для знакомства с языком, лучшего его понимания, как источники примеров. Но разработчику компилятора требуется Спецификация. Это может быть стандарт, если он существует, либо авторское описание языка. Спецификация — это максимально точное определение языка, его синтак-

сиса и семантики. Спецификация — весьма строгий и сухой документ, который не может служить учебником по программированию. Она обычно содержит не слишком много примеров, не дает рекомендаций по применению тех или иных конструкций, а лишь определяет их форму и смысл.

Метаязыки

Описание языка должно быть тоже записано на некотором языке, который выступает в этом случае в роли *метаязыка*. Для описания синтаксиса используются формальные нотации, эквивалентные порождающим грамматикам Н. Хомского. Синтаксис языков программирования обычно определяется с помощью контекстно-свободных грамматик, быть может, с несколько расширенными правилами.

Несмотря на то, что существуют подходы к формализации описания семантики (например, атрибутные грамматики), они не получили распространения в практике спецификации языков программирования. Семантику языковых конструкций определяют, пользуясь естественным языком — английским, русским.

Ниже мы рассмотрим варианты нотаций, использованных при описании синтаксиса известных языков программирования.

БНФ

Бэкуса-Наура Форма (БНФ) была впервые применена при описании Алгола-60. БНФ совпадает по сути с нотацией КС-грамматик, отличаясь лишь обозначениями. Предусмотрены следующие метасимволы:

< > — служат для выделения нетерминалов — понятий языка.

| — «или». Разделяет альтернативные правые части правил.

::= — «есть по определению». Заменяет стрелку, используемую при записи продукций КС-грамматик.

Терминальные символы записываются как есть, никаких специальных способов их выделения не предусмотрено.

Вот пример определений на БНФ, взятый из спецификации Алгола-60 — «Модифицированного сообщения»:

```
<простое арифметическое выражение> ::=
   <терм> | <знак операции типа сложения> <терм> |
   <простое арифметическое выражение>
      <знак операции типа сложения> <терм>
<знак операции типа сложения> ::= + | -
```

Как видно, для выражения повторений используется рекурсия, причем повсеместно — левая.

БНФ использована Н. Виртом при описании языка Паскаль. Хотя в нотацию были добавлены метаскобки { и }, обозначающие повторение, применены они лишь в отдельных случаях, в то время как, например, грамматика выражений леворекурсивна.

Синтаксические диаграммы

Нет особой необходимости знакомить читателя с синтаксическими диаграммами, поскольку они уже рассматривались в этой книге. Диаграммы стали популярны после выхода книги К. Йенсен и Н. Вирта «Паскаль». Они использованы в первой ее части — Руководстве — компактном учебнике языка. На рис. 3.1 показана одна из имеющихся там диаграмм.

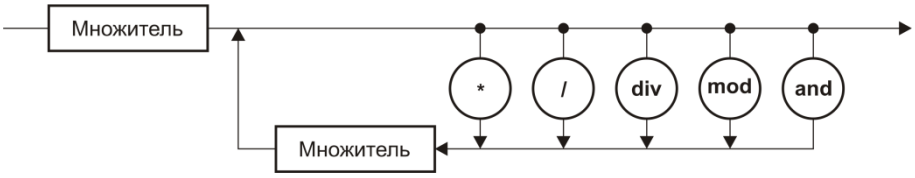


Рис. 3.1. Синтаксическая диаграмма слагаемого (язык Паскаль)

Расширенная форма Бэкуса-Наура (РБНФ)

Как уже говорилось, отсутствие в нотации формальных грамматик (и БНФ) средств явного задания повторений создает ряд трудностей. Во-первых, определения оказываются сложными

для понимания, недостаточно наглядными из-за обилия рекурсий. Во-вторых, возникают проблемы с тем, что грамматики, дающие подходящие семантические деревья, оказываются леворекурсивными.

При описании Модулы-2 и Оберона Н. Вирт использовал расширенную Бэкуса-Наура Форму (РБНФ)[13]. Главные модификации касаются введения скобок { и } для повторений, а [и] — для обозначения необязательного вхождения цепочек терминалов и нетерминалов в правые части правил. Соглашения относительно обозначений терминалов и нетерминалов также изменены, что не столь принципиально.

В дальнейшем мы будем пользоваться именно РБНФ. Вот как она определяется в спецификации Оберона-2[14]:

Варианты разделяются знаком |. Квадратные скобки [и] означают необязательность записанного внутри них выражения, а фигурные скобки { и } означают его повторение (возможно, 0 раз). Нетерминальные символы начинаются с заглавной буквы (например, `Оператор`). Терминальные символы или начинаются малой буквой (например, `идент`), или записываются целиком заглавными буквами (например, `BEGIN`), или заключаются в кавычки (например, `":="`).

К этому следует добавить, что в роли знака «есть по определению» в РБНФ используется «=», а каждое правило заканчивается точкой.

Вот так может быть определен синтаксис идентификатора (имени) с помощью РБНФ:

```
Имя = Буква { Буква | Цифра }.
```

[13] Эта нотация была предложена им в 1977 году [Вирт, 1977].

[14] Точнее, в русском переводе спецификации.

Являясь метаязыком, РБНФ должна быть пригодна для описания языков, имеющих практический интерес. В том числе с помощью РБНФ может быть определен и синтаксис самой РБНФ[15]:

```
Синтаксис = { Правило }.
Правило   = Имя "=" Выражение ".".
Выражение = Вариант { "|" Вариант }.
Вариант   = Элемент { Элемент }.
Элемент   = Имя | Цепочка | "(" Выражение ")" |
            "[" Выражение "]" | "{" Выражение "}".
Цепочка   = "'" { символ } "'" | '"' { символ } '"'.
```

В этих определениях не сделано различий между именами, обозначающими терминалы и нетерминалы, хотя сформулировать это на РБНФ было бы несложно. Различение имён вынесено за рамки синтаксиса и может быть специфицировано (и специфицируется, см. выше) отдельно. Подобным же образом часто поступают при определении языков программирования.

Описания синтаксиса языков семейства Си

В знаменитой книге Б. Кернигана и Д. Ритчи описание синтаксиса языка Си дано в нотации, которая эквивалентна БНФ, но использует другие соглашения об обозначениях терминалов и нетерминалов, альтернативных правых частей правил, необязательных конструкций.

Нетерминалы записываются курсивом, терминалы — прямым шрифтом. Альтернативные части правил выписываются в столбик по одному в строке или помечаются словами «one of» (один из). Необязательные части сопровождаются подстрочным индексом $_{opt}$ (от optional — необязательный; $_{необ.}$ — в некоторых русских переводах). Левая часть правила записывается отдельной строкой с отступом влево.

Вот пример определений конструкций языка Си:

---

[15] Подобное, конечно, справедливо и в отношении БНФ. РБНФ, как БНФ и как рассмотренный выше язык регулярных выражений — это КС-языки.

составной оператор

{ *список описаний*$_{opt}$ *список операторов*$_{opt}$ }

список операторов

оператор

оператор список операторов

Как видно, из-за отсутствия явного способа выражения повторений, определения изобилуют рекурсией.

Аналогичная нотация с минимальными изменениями использована для описания синтаксиса языков-потомков Си: Си++, Ява, Си#. Вот выдержка из стандарта ECMA-334 на язык Си#:

block:

{ *statement-list*$_{opt}$ }

statement-list:

statement

statement-list statement

Специального названия эта нотация, судя по всему, не имеет. И представляется, как минимум, странной. Она неудобна не только для чтения, но и для обработки на компьютере. Ее использование при описании новых языков трудно объяснить чем-либо, кроме дурно понятой необходимости следовать традициям.

Описания синтаксиса языка Ада

Контекстно-свободные грамматики языков Ада-83 и Ада-95 определены с помощью варианта БНФ, в который добавлены обозначения повторений и необязательных частей. Названия нетерминалов записываются обычным шрифтом с использованием знака подчеркивания, если название составное, а зарезервированные слова — жирным шрифтом. Поскольку ни квадратные, ни фигурные скобки в Аде не используются, как не используется

и знак «|» (все это метасимволы), никакого специального обозначения для терминалов не предусмотрено.

Определение синтаксиса оператора **if**, взятое из стандарта Ада-95, выглядит так:

```
if_statement ::=
   if condition then
      sequence_of_statements
   {elsif condition then
      sequence_of_statements}
   [else
      sequence_of_statements]
   end if;
```

Интересная и полезная особенность: синтаксические правила структурных конструкций представлены в виде, соответствующем их рекомендованному форматированию в программах (разделение на строки, отступы).

Язык программирования «О»

Все последующие вопросы, связанные с конструированием компиляторов, мы будем рассматривать на примере разработки транслятора для минимального подмножества языка Оберон-2 (Приложение 1). Такой конкретный и предметный подход наверняка сузит круг обсуждаемых тем, не позволит рассмотреть тонкие и сложные вопросы, связанные с оптимизацией кода, зато гарантирует полное понимание основ.

Оберон-2 — это расширенная версия языка Оберон, созданного выдающимся швейцарским специалистом Никлаусом Виртом, автором Паскаля и Модулы-2. Оберон очень прост и в то же время содержит в себе необходимые средства структурного, модульного и объектно-ориентированного программирования [Свердлов, 2007].

Тот простой язык, для которого мы напишем компилятор, будет называться «О». Это шуточное название, во-первых, подчеркивает некоторую несерьезность, учебный характер, «игрушеч-

ность» языка — уж очень он будет прост. Во-вторых, буква «О», похожа на ноль (сложность языка стремится к нулю) и в то же время это первая буква слова Оберон. Из других ассоциаций — «Операция «Ы».

Краткая характеристика языка «О»

- Язык «О» является точным подмножеством Оберона и Оберона-2, то есть любая программа на языке «О» является правильной программой на языках Оберон и Оберон-2.
- Программа на языке «О» состоит из единственного модуля. Выполнение программы начинается с первого оператора, записанного после слова **BEGIN**. Процедуры в языке «О» отсутствуют.
- Предусмотрены константы и переменные только целого (INTEGER) типа. Выражения логического типа (без логических операций) могут использоваться в операторах **IF** и **WHILE**. Массивов и записей нет.
- Выражения строятся по правилам языка Оберон. Допустимы все операции, применимые к целым и дающие результат целого типа: +, –, *, **DIV**, **MOD**. В логических выражениях используются операции отношения: =, #, <, >, <=, >=, которые применимы к целочисленным операндам.
- Набор операторов включает присваивание, вызов процедуры (стандартной), **IF** – **THEN** – **ELSIF** … **ELSE** – **END**, **WHILE** – **DO** – **END**.
- Предусмотрены стандартные процедуры и процедуры-функции ABS, DEC, HALT, INC, MAX, MIN, ODD. Их смысл такой же, как и в языке Оберон-2 (см. Приложение 1).
- Разрешается импорт стандартных (псевдо) модулей In и Out, предоставляющих процедуры ввода-вывода In.Open, In.Int, Out.Int, Out.Ln. Их свойства приведены в таблице.

Название процедуры	Параметры	Описание
`In.Open`	Нет	Открывает стандартный входной поток.
`In.Int(v)`	`v: INTEGER`	Вводит значение `v`
`Out.Int(x, n)`	`x, n: INTEGER`	Выводит значение `x`, используя `n` позиций
`Out.Ln`	Нет	Выводит перевод строки

- В записи программы разрешены комментарии, которые могут быть вложенными.
- Большие и малые буквы различаются.
- Кроме ключевых слов, используемых в языке «О», зарезервированными считаются также все остальные служебные слова языка Оберон-2. Их нельзя использовать в качестве идентификаторов в программах на «О». Это обеспечивает полную совместимость снизу вверх с Обероном-2.

Синтаксис «О»

Ниже приводится сводка синтаксиса языка «О» на РБНФ.

```
Модуль =
   MODULE Имя ";"
   [Импорт]
   ПослОбъявл
   [BEGIN
      ПослОператоров]
   END Имя ".".
Импорт =
   IMPORT Имя {"," Имя} ";".
ПослОбъявл =
   {CONST
      {ОбъявлКонст ";"}
   |VAR
      {ОбъявлПерем ";"} }.
ОбъявлКонст = Имя "=" КонстВыраж.
ОбъявлПерем = Имя {"," Имя} ":" Тип.
```

```
Тип = Имя.
ПослОператоров =
   Оператор {";"
   Оператор }.
Оператор = [
   Переменная ":=" Выраж
   |[Имя "."] Имя ["(" [Параметр {"," Параметр}] ")"]
   |IF Выраж THEN
      ПослОператоров
   {ELSIF Выраж THEN
      ПослОператоров}
   [ELSE
      ПослОператоров]
    END
   |WHILE Выраж DO
      ПослОператоров
    END
].
Параметр = Переменная | Выраж.
Переменная = Имя.
КонстВыраж = ["+" | "-"] (Число | Имя).
Выраж = ПростоеВыраж [Отношение ПростоеВыраж].
ПростоеВыраж = ["+"|"-"] Слагаемое {ОперСлож Слагаемое}.
Слагаемое = Множитель {ОперУмн Множитель}.
Множитель =
   Имя ["(" Выраж | Тип ")"]
   | Число
   | "(" Выраж ")".
Отношение = "=" | "#" | "<" | "<=" | ">" | ">=".
ОперСлож = "+" | "-".
ОперУмн = "*" | DIV | MOD.
Имя = буква {буква | цифра}.
Число = цифра {цифра}.
```

Можно заметить, что грамматика языка «О» не является *LL*(1)-грамматикой. Действительно, две первых альтернативы правила для операторов (первая соответствует присваиванию, вторая — вызову процедуры) нельзя различить по одному очередному символу — переменная представляет собой имя. Оба варианта в списке фактических параметров процедур и функций (правила для Оператора и Множителя) также могут начинаться именем. Можно было бы выполнить преобразование грамматики, но в этом нет особой нужды. Разделение этих случаев легко выполняется с помощью несложных контекстных проверок: если имя,

начинающее оператор, обозначает переменную, то впереди присваивание, иначе предполагаем вызов процедуры. Варианты анализа фактических параметров можно выбрать по виду соответствующих формальных.

Пример программы на «О»

Программа `Primes` печатает простые числа в диапазоне от 2 до *n* и подсчитывает их количество.

```
MODULE Primes;
(* Простые числа от 2 до n *)

IMPORT In, Out;

VAR
   n, c, i, d : INTEGER;

BEGIN
   In.Open;
   In.Int(n);
   c := 0; (* Счетчик простых *)
   i := 2;
   WHILE i <= n DO
      (* Делим на 2, 3, ... пока не разделится *)
         d := 2;
         WHILE i MOD d # 0 DO
            INC(d)
         END;
      IF d = i THEN (* i - простое *)
         INC(c);
         Out.Int(d, 8)
      END;
      INC(i);
   END;
   Out.Ln;
   Out.Int(c, 0);
END Primes.
```

Использованный здесь алгоритм весьма неэффективен[16], зато очень прост[17].

---

[16] Быть может, это самый неэффективный алгоритм поиска простых чисел.

[17] Возможно даже, что это самый простой алгоритм решения этой задачи.

Структура компилятора

Программы-компиляторы могут во многом отличаться друг от друга по внутреннему устройству, но есть несколько частей, которые присутствуют всегда или почти всегда. На рис. 3.2 показана типовая структура компилятора.

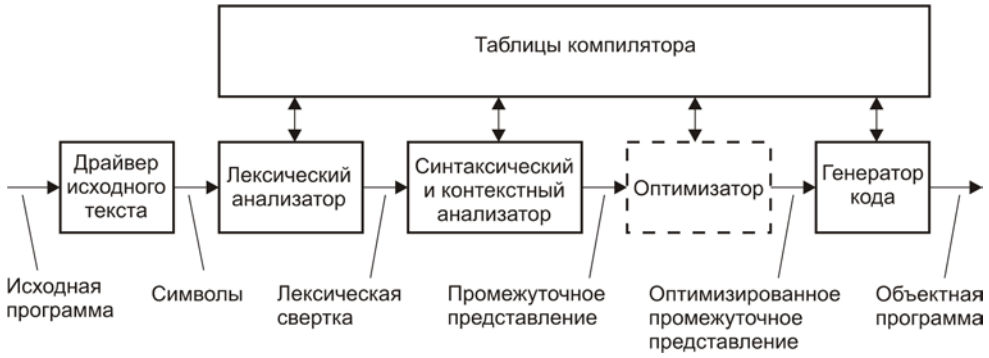


Рис. 3.2. Структура компилятора

Блок оптимизатора, показанный пунктиром, не является обязательным.

В дальнейшем мы подробно рассмотрим и запрограммируем модули компилятора, а сейчас — короткие пояснения.

Драйвер исходного текста. Непосредственно взаимодействует с текстом программы. Необходимость в этом модуле обусловлена тем, что источник текста бывает разным: программа может вводиться из файла или извлекаться из базы данных, считываться из окна текстового редактора, загружаться по сети и т. д. Драйвер может буферизовать ввод, обеспечивать вывод исходного текста по ходу трансляции, вести счет строк и символов, обеспечивать показ фрагмента программы, вызвавшего ошибку. Следующим модулям компилятора драйвер передает последовательность символов. При этом остальные части транслятора не зависят от источника и способа представления исходного текста. Основную функцию драйвера выполняет процедура `NextCh`.

Лексический анализатор (сканер). Разбивает программу на простые элементы-слова, называемые *лексемами*. Лексемами являются имена, служебные слова, числа, разделители, знаки операций. Удаляет из программы комментарии. Использование лексического анализатора упрощает последующие модули, которые уже не должны иметь дела с отдельными знаками, а могут оперировать более крупными неделимыми единицами.

Синтаксический и контекстный анализаторы. Суть работы синтаксического анализатора нам хорошо знакома. Синтаксического анализа недостаточно для того, чтоб убедиться в формальной правильности программы. Кроме синтаксиса, заданного КС-грамматикой, должны соблюдаться дополнительные правила: об обязательных описаниях объектов программы, соответствии типов и т.п. Выполнение этих правил проверяется с помощью таблиц, заполняемых в ходе трансляции, и составляет суть контекстного анализа. Синтаксический анализатор формирует промежуточное представление программы в форме обратной польской записи, семантического дерева, четверок.

Оптимизатор. В простых компиляторах может отсутствовать. Суть оптимизации состоит в эквивалентном преобразовании программы (промежуточного представления) с целью улучшения ее эффективности: увеличения быстродействия и уменьшения расхода памяти. Оптимизация, выполняемая на уровне промежуточного представления, не зависит от выходного языка компилятора, является машинно-независимой.

Генератор кода. Отвечает за формирование результирующей программы — машинных команд. В составе генератора кода также могут быть части, выполняющие машинно-зависимую оптимизацию. Программу, получаемую в результате компиляции, обычно называют объектной программой. Это может быть не полностью готовая к выполнению машинная программа, а от-

дельный модуль, который еще должен быть скомпонован с другими.

Таблицы компилятора. Компилятор содержит множество таблиц, с которыми взаимодействуют его модули: таблица зарезервированных слов, таблица имен, таблица типов и др. На поиск в таблицах тратится значительная часть времени работы транслятора.

Блоки компилятора, зависящие от входного языка, выполняющие анализ исходной программы и ее преобразование в промежуточное представление, образуют его анализирующую, входную, фронтальную часть (по-английски front-end). Блоки, ответственные за формирование выходной программы и не зависящие от входного языка образуют синтезирующую, выходную часть (back-end)[18].

Многопроходные и однопроходные трансляторы

Показанные на рис. 3.2 блоки транслятора (кроме драйвера исходного текста) можно рассматривать как последовательно выполняемые модули, осуществляющие отдельные фазы трансляции. Вначале лексический анализатор, прочитывая исходную программу от начала до конца, формирует последовательность лексем — лексическую свертку и сохраняет ее. Синтаксический анализатор вступает в работу по окончании работы сканера. Далее последовательно запускаются оптимизатор и генератор кода. Компилятор, работающий по такой схеме, называется *многопроходным*. Проходом считается прочтение транслятором программы от начала до конца в ее исходной или одной из промежуточ-

[18] К сожалению, устоявшиеся русские термины, эквивалентные полезным английским «front-end» и «back-end» не сформировались и, наверное, уже не сформируются. В устных беседах «компиляторостроители» обычно так и говорят: «фронт-энд», «бэк-энд». Ввести такие кальки с английского в письменную речь я не рискну.

ных форм: лексической свертки, промежуточных представлений. Не обязательно разбиение на проходы должно соответствовать разделению на блоки, показанные на рис. 3.2. Некоторые из блоков и фаз трансляции могут объединяться в один проход и наоборот, отдельные фазы могут выполняться в несколько проходов.

Последовательная работа основных блоков компилятора совсем не обязательна. Эти блоки можно рассматривать как логические части, действующие попеременно и выполняющие трансляцию за один просмотр программы. Ведущую роль при этом выполняет синтаксический анализатор. Обращаясь за лексемами к сканеру, он выполняет анализ и, распознав определенную часть программы, вызывает генератор кода для формирования порции машинных команд.

Несколько измененная схема, отражающая такой способ взаимодействия частей в однопроходном трансляторе, показана на рис. 3.3. В таблице 3.1 представлены преимущества и недостатки однопроходной и многопроходной организации.

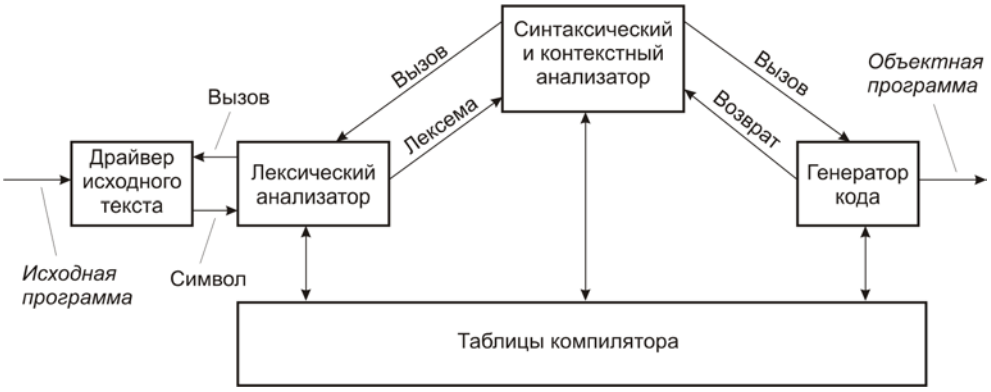


Рис. 3.3. Структура однопроходного транслятора

Таблица 3.1. Сравнение однопроходных и многопроходных трансляторов

	Однопроходный	**Многопроходный**
Преимущества	Простота. Быстрая работа. Отсутствие необходимости в промежуточных представлениях.	Нетребовательность к памяти. Лучшее разделение на модули. Лучшие возможности для оптимизации программы. Независимость анализатора от выходного языка. Независимость генератора от входного языка.
Недостатки	Затруднена оптимизация. Плохое разделение анализатора и генератора кода. Дополнительные требования к входному языку.	Сложность. Более медленная работа. Необходимость промежуточных представлений.

Первые компиляторы с языков высокого уровня были многопроходными. Это объяснялось тем, что ресурсы имевшихся компьютеров (объем оперативной памяти в первую очередь) просто не позволяли выполнить всю работу за один раз. Сам компилятор не помещался целиком в память, и его разбивали на

части, которые работали последовательно. Кроме того, однопроходная схема предполагает, что в оперативной памяти в ходе трансляции должна удерживаться значительная часть формируемого объектного кода (как минимум, код одной процедуры).

Количество частей и проходов могло быть довольно большим. Так, один из первых отечественных трансляторов с Алгола-60 ТА-2М состоял из 16 блоков и выполнял 16 проходов. В современных условиях уже нет технических ограничений, препятствующих однопроходной трансляции.

Не всякий язык программирования приспособлен для того, чтоб любая написанная на нем программа могла быть откомпилирована за один проход. Для однопроходной компиляции в частности требуется, чтобы употреблению объекта программы предшествовало его описание.

Компилятор языка «О»

Учебный компилятор языка «О» будет однопроходным. Простота устройства нам сейчас важнее возможностей оптимизации и скорости результирующей программы. Цель — написать работающий компилятор. И быть при этом в возможно большей степени уверенными, что работает он правильно[19]. Программировать будем на Паскале, который в этой книге играет роль основного языка. В приложениях вы можете также найти версии компилятора, написанные на Обероне, Си, Яве и Си#.

Приступим. Вот первый эскиз главной программы (листинг 3.1).

Листинг 3.1. Компилятор. Первый шаг детализации

```
program O;
```

[19] Среди всех мыслимых качеств программы главное, безусловно, — ее работоспособность. Как справедливо замечено, если не требуется, чтоб программа правильно работала, ее можно сделать сколь угодно быстрой и компактной. И, добавлю, генерирующей сколь угодно быстрый и компактный код.

```
{Учебный компилятор языка O}
   …
begin
   WriteLn('Компилятор языка O');
   Init;     {Инициализация}
   Compile;  {Компиляция   }
   Done;     {Завершение   }
end.
```

Может показаться, что этот шаг банален. Это не так. Разбив задачу на три последовательно выполняемые части, мы можем поочередно сконцентрироваться на каждой из них, забыв на время про остальные.

Подумаем об инициализации. Она может начинаться с подготовки текста исходной программы к чтению компилятором. Эту работу в предшествующих примерах выполняла процедура `ResetText`. Она же послужит нам и в этот раз. Процедура `ResetText` будет частью драйвера исходного текста.

По окончании компиляции, возможно, потребуется выполнить действия, завершающие работу с источником исходного текста: закрыть файл, отправить сообщение окну текстового редактора. За это будет отвечать процедура `CloseText`, также относящаяся к драйверу исходного текста.

Перед тем, как записать уточненную версию главной программы учтем, что `Compile` — основная процедура компилятора — это часть модуля синтаксического анализатора, который при однопроходной трансляции играет ведущую роль, координируя работу остальных частей. В нашей программе (листинг 3.2) синтаксический анализатор (parser) будет оформлен как отдельный модуль с названием `OPars`. Драйвер исходного текста назовем `OText`.

Листинг 3.2. Компилятор. Первый компилируемый вариант

```
program O;
{Учебный компилятор языка O}
uses
   OText, OPars;
```

```
procedure Init;
begin
   ResetText;
end;

procedure Done;
begin
   CloseText;
end;

begin
   WriteLn('Компилятор языка O');
   Init;     {Инициализация}
   Compile;  {Компиляция   }
   Done;     {Завершение   }
end.
```

В отличие от первого эскиза, это уже вариант, который может быть откомпилирован при условии, что имеются модули `OText` и `OPars`.

Для начала в роли процедуры `Compile` модуля `OPars` используем заглушку (листинг 3.3).

Листинг 3.3. Модуль синтаксического анализатора с заглушкой

```
unit OPars;
{Распознаватель}
interface
procedure Compile;

{=====================================================}

implementation
procedure Compile;
begin
   {Тут пусто. Это заглушка}
end;
end.
```

Вспомогательные модули компилятора

Я не буду программировать здесь полностью драйвер исходного текста, как и некоторые другие вспомогательные модули. Определим только их программные интерфейсы. С реализацией этих

модулей можно познакомиться в приложении, где приведен полный текст компилятора.

Драйвер исходного текста

Драйвер исходного текста (модуль `OText`, листинг 3.4) непосредственно взаимодействует с транслируемой программой. Другие части компилятора (в первую очередь сканер) обращаются к драйверу за очередным символом. Чтение символа выполняет процедура `NextCh`. Прочитанный символ она помещает в глобальную переменную `Ch`, экспортируемую драйвером. В программный интерфейс модуля `OText` входят также уже использовавшиеся процедуры `ResetText` и `CloseText`.

Листинг 3.4. Интерфейс драйвера исходного текста

```
unit OText;
{Драйвер исходного текста}

interface

const
   chSpace = ' ';     {Пробел      }
   chTab   = chr(9);  {Табуляция   }
   chEOL   = chr(10); {Конец строки}
   chEOT   = chr(0);  {Конец текста}

var
   Ch    : char;    {Очередной символ      }
   Line  : integer; {Номер строки         }
   Pos   : integer; {Номер символа в строке}

procedure ResetText;
procedure CloseText;
procedure NextCh;
{=====================================================}
```

Определены константы `chSpace`, `chTab` и `chEOL`[20], обозначающие пробел, табуляцию и конец строки, а также признак конца текста `chEOT`.

Драйвер исходного текста ведет подсчет строк и символов, помещая номер текущей строки в переменную `Line`, а номер текущего символа в строке — в переменную `Pos`. Эти значения могут использоваться при выдаче сообщений об ошибках.

После выполнения `ResetText` текущим (содержащимся в переменной `Ch`) должен стать первый символ компилируемой программы. Это может быть и незначащий символ — пробел, табуляция, разрыв строки. По ходу чтения исходный текст может печататься.

Приведенная в приложении базовая реализация драйвера выполняет ввод символов из текстового файла. Название файла процедура `ResetText` берет из командной строки.

Вот пример запуска нашего компилятора:

```
>O Primes.o
Компилятор языка O
>
```

Не полностью готового, а состоящего только из главной программы (листинг 3.2), модуля `OPars` с заглушкой вместо `Compile` (листинг 3.3) и драйвера исходного текста. А это пример неправильного вызова:

```
>O.exe
Компилятор языка O
Формат вызова:
   O <входной файл>
```

[20] То, что конец строки представлен значением chEOL, совсем не означает, что, строки реального текста обязательно разделяются одним специальным символом равным chr(10). Ситуация лишь представляется такой другим частям компилятора. Драйвер исходного текста, обнаружив границу строк ему одному известным способом, помещает в переменную Ch значение chEOL.

Обработка ошибок

За нее отвечает модуль `OError` (листинг 3.5). Он предоставляет другим частям транслятора процедуру `Error`, которая печатает переданное ей как параметр сообщение об ошибке, указывая место ошибки в исходном тексте. После этого процедура `Error` вызовом `Halt` прекращает работу компилятора. Такая простая реакция на ошибки нас вполне устраивает, поскольку создаваемый компилятор не является частью какой-либо интегрированной системы.

Листинг 3.5. Интерфейс модуля обработки ошибок

```
unit OError;
{Обработка ошибок}

interface

procedure Error(Mes: string);
procedure Expected(Mes: string);
procedure Warning(Mes: string);

{=====================================================}
```

Поскольку большинство сообщений компилятора об ошибках начинаются словом «Ожидается», предусмотрена процедура `Expected`, которая делает то же, что и `Error`, но сама печатает слово «Ожидается», избавляя нас от необходимости записывать его каждый раз. Потребуется также возможность печатать предупреждения. Для этого предусмотрена процедура `Warning`. В отличие от `Error` и `Expected`, она не останавливает работу программы.

Лексический анализатор (сканер)

Лексический анализатор решает следующие задачи:

- Выполняет «сборку» лексем из отдельных символов входного текста. Лексемы — это неделимые единицы программы — имена, числа, зарезервированные слова, разделители и знаки

операций, в том числе состоящие из нескольких символов, как, например, «:=».

- Удаляет комментарии и пробельные символы (пробел, табуляция, конец строки).

Рассмотрим для примера такой фрагмент программы на языке «О» (Оберон):

```
i := 2;
WHILE i <= n DO
```

В нем содержится 9 лексем: три лексемы «имя», лексемы «зарезервированное слово **WHILE**» и «слово **DO**», а также лексемы «присваивание», «меньше или равно», «целое число», «точка с запятой».

При использовании в компиляторе отдельного блока сканера грамматика языка программирования становится двухуровневой. С точки зрения сканера программа — это просто последовательность лексем. Правила записи лексем, равно как и синтаксис последовательности лексем могут быть заданы автоматной грамматикой. Терминалами этой грамматики являются отдельные знаки, в том числе пробельные символы. Это первый уровень определения языка — *лексическая грамматика*. Второй уровень — *синтаксическая грамматика*, которая используется синтаксическим анализатором. Терминалами синтаксической грамматики являются лексемы.

Использование сканера обусловлено такими причинами:

- Упрощение синтаксического анализатора. Синтаксический анализатор имеет дело не с отдельными символами, а с более крупными и содержательными единицами, избавляется от необходимости обрабатывать пробельные символы и комментарии.
- Повышается эффективность следующих за сканером фаз трансляции. Элементы программы на входе синтаксического

анализатора представляются не строками переменной длины, а данными фиксированного размера. Размер лексической свертки может быть меньше размера исходной программы.

- Естественным образом решается вопрос о пробельных символах. Пробелы, табуляции и концы строк запрещаются внутри лексем (кроме пробелов и табуляций в символьных строках[21]) и разрешаются между ними.
- Обеспечивается независимость следующих за сканером фаз трансляции от конкретного представления исходной программы и используемого набора символов. Облегчается изменение лексики языка. Замена или перевод на другой язык служебных слов не требуют изменения синтаксического анализатора и последующих блоков.

Виды и значения лексем

Взаимодействие сканера с последующими частями транслятора в многопроходном и однопроходном трансляторе строится несколько по-разному.

В многопроходном трансляторе лексический анализатор должен целиком сформировать кодированную последовательность лексем — лексическую свертку[22]. Для многих лексем достаточно было бы указать в свертке только их вид, то есть сообщить, что это за лексема. Виды лексем могут кодироваться целыми числами. Например, для лексем «присваивание» или «слово **`WHILE`**», достаточно указать, что это именно они, записав в свертку соответствующий код. Но для таких лексем как «число» и «имя»

---

[21] В некоторых языках, например в Си#, внутри символьных литералов разрешается помещать и разрыв строки, в некоторых языках не допускаются табуляции. В языке «О» символьные строки вообще не предусмотрены.

[22] Точнее, не в любом многопроходном, а лишь в том, где лексический анализ выполняется отдельным проходом, что в современных условиях вряд ли актуально.

указания их вида не достаточно. Синтаксическому анализатору безразлично, какое конкретно число записано в программе, но к моменту генерации кода эта информация потребуется. В связи с этим лексеме, наряду с ее видом, приписывается значение, а лексическая свертка представляется последовательностью пар вид-значение.

Значением лексемы «число» может быть величина этого числа или ссылка на запись в таблице констант. Значение лексемы «имя» — строка, содержащая символы конкретного имени, или ссылка на запись в таблице имен. Использование ссылок вместо самих значений позволяет обойтись в лексической свертке полями фиксированного размера.

В однопроходном трансляторе все проще. Поскольку формировать лексическую свертку целиком не требуется, сведения о текущей лексеме могут передаваться синтаксическому анализатору с помощью единственной переменной. Для лексем, которым приписывается значение, можно предусмотреть одно или несколько полей в записи о лексеме (или несколько отдельных переменных) — по одному для значений разных типов. Одно поле (переменная) хранит значения целых чисел, другое — вещественных и т. д. Вместо самих значений могут использоваться ссылки на записи в соответствующих таблицах, которые в этом случае должен формировать сканер.

Лексический анализатор языка «О»

В нашем компиляторе, который мы пишем на Паскале, лексический анализатор будет реализован в виде отдельного модуля `OScan`.

Виды лексем языка «О»

Лучшим вариантом кодирования видов лексем при программировании на Паскале будет использование перечислимого типа.

```
type
   tLex = (lexNone, …
```

Виды лексем будут константами типа `tLex`. Первая предусмотренная нами константа — первый вид лексем — `lexNone` — «никакая» лексема. Это особое значение будет соответствовать тем служебным словам Оберона-2, которые не используются в языке «О», например, **`RECORD`**, **`IS`**. Лексема вида `lexNone` не может присутствовать ни в одной правильной программе на языке «О». Продолжаем:

```
tLex =  (lexNone, lexName, lexNum,
         lexMODULE, lexIMPORT, lexBEGIN, lexEND,
         lexCONST, lexVAR, lexWHILE, lexDO,
         lexIF, lexTHEN, lexELSIF, lexELSE,
…
```

`lexName` и `lexNum` — это лексемы «имя» и «число». Сканер должен формировать для этих лексем еще и значения.

Служебным словам **`MODULE`**, **`IMPORT`**, **`BEGIN`** и т. д. соответствуют одноименные виды лексем. Значений для этих лексем не нужны. Если сканер выдает `lexBEGIN`, никакой другой информации от него не требуется.

Далее в определении типа `tLex` будут следовать лексемы, соответствующие знакам операций. Здесь возможны различные решения. Хотя знаки плюс и минус имеют разный смысл, с точки зрения синтаксического анализатора они совершенно равноправны и всегда обрабатываются им одинаково, как равноправны знаки операций типа умножения: «*», **`DIV`**, **`MOD`**, и отношения «=», «#», «<», «<=», «>», «>=». Поэтому можно отнести знаки операций типа сложения к одному виду, типа умножения — к другому, знаки отношений — к третьему. При этом соответствующие лексемы надо будет снабжать значениями, позволяющими на стадии генерации кода отличить плюс от минуса и умножение от деления.

Но можно каждый из перечисленных знаков относить к самостоятельному виду лексем.

При использовании пары вид-значение работа синтаксического анализатора немного упростится. Для проверки того, что текущая лексема это, например, операция типа сложения, придется сделать только одно сравнение на равенство. Если каждый знак закодирован по-особому, потребуется проверка на принадлежность текущей лексемы множеству значений.

Примем простое, «лобовое» решение — отнесем каждый знак к особому виду.

```
…
lexMult, lexDIV, lexMOD, lexPlus, lexMinus,
lexEQ, lexNE, lexLT, lexLE, lexGT, lexGE,
…
```

Обратите внимание на имена констант. Лексема, обозначающая «–», названа `lexMinus` — «минус», а не, скажем, `lexSub` — «вычитание», поскольку знак «–» может обозначать разные операции: одноместный минус — это перемена знака, двуместный — вычитание. Названия лексем, обозначающих отношения, выбраны с учетом традиции, восходящей к Фортрану: EQ (equal) — равно; NE (not equal) — не равно; LT (less than) — меньше чем; LE (less than or equal) — меньше или равно; GT (greater than) — больше чем; GE (greater than or equal) — больше или равно.

Наконец, замыкают определение типа `tLex` константы, обозначающие лексемы-разделители:

```
…
lexDot, lexComma, lexColon, lexSemi, lexAss,
lexLPar, lexRPar,
lexEOT);
```

Dot — точка; comma — запятая; colon — двоеточие; semi(colon) — точка с запятой; ass(ignment) — присваивание; l(eft) par(enthesis) — левая круглая скобка; r(ight) par(enthesis) —

правая круглая скобка. Константа `lexEOT`, означает лексему «конец текста» (её не следует путать с символом `chEOT`).

Программирование сканера

Правила записи лексем могут быть заданы автоматной грамматикой. Поэтому на роль сканера подошел бы конечный автомат. Мы могли бы изобразить диаграмму его переходов, построить таблицу. Другой вариант — программировать лексический анализатор по синтаксическим диаграммам. Но и в построении диаграмм нет реальной необходимости. Правила записи лексем языка «О» настолько просты, что сканер можно программировать «просто так». Построить диаграмму потребуется лишь для комментария. Кстати сказать, комментарии в языке «О» могут быть вложенными, поэтому их синтаксис задается контекстно-свободной, но не автоматной грамматикой.

В компиляторе, где лексический анализ не выделен в отдельный проход, собственно сканер — это процедура, выдающая по запросу синтаксического анализатора очередную лексему. Назовем эту процедуру `NextLex`. Задача `NextLex` состоит в том, чтобы, считывая символы, определить вид очередной лексемы и ее значение, (если для лексем данного вида значение требуется).

Перед первым вызовом процедуры `NextLex` текущим является первый символ входного текста. Он может быть и пробельным. Сканер должен пропустить пробелы, предшествующие лексеме, если они есть, прочитать символы лексемы и оставить текущим символ, следующий за лексемой. Такие же действия выполняются при каждом следующем вызове. Если запись лексемы содержит ошибку, сканер выдает соответствующее сообщение.

Начало процедуры `NextLex` будет таким:

```
procedure NextLex;
begin
   while Ch in [chSpace, chTab, chEOL] do NextCh;
   LexPos := Pos;
```

Напомню, что переменные `Ch` и `Pos`, процедура `NextCh` и константы `chSpace`, `chTab` и `chEOL` импортированы сканером из модуля `OText` (см. листинг 3.4).

Цикл **`while`** выполняет пропуск пробельных символов. По выходе из цикла текущим будет первый знак очередной лексемы (если какая-то лексема может начинаться с этого знака). Номер этого символа в строке хранится в переменной `Pos`. Полезно запомнить этот номер. Для этой цели служит переменная сканера `LexPos`. Она экспортируется сканером и может использоваться в модуле `OError`, для того, чтобы иметь возможность в диагностическом сообщении указать на начало лексемы, вызвавшей ошибку. Если для этих целей употребить `Pos`, то указать можно будет лишь на символ, следующий за лексемой, или на последний символ лексемы.

Далее, в зависимости от значения текущего символа (который предположительно является первым символом очередной лексемы), решаем, к какому виду эта лексема должна быть отнесена и при необходимости определяем её значение. Первый вариант выбора относится к случаю, когда символ — это буква, второй — когда символ — цифра.

```
case Ch of
'A'..'Z', 'a'..'z':
   Ident;
'0'..'9':
   Number;
…
```

С буквы может начинаться лексема «имя» (`lexName`) или одно из ключевых слов. Чтение символов лексемы, следующих за первой буквой, и определение того, является ли лексема именем или зарезервированным словом отложим, поручив процедуре `Ident`. Различать имена и ключевые слова, процедура `Ident` будет с помощью таблицы служебных слов.

С цифры в языке «О» (и многих других языках) может начинаться только число[23]. Чтение его последующих цифр и вычисление значения выполнит процедура `Number`.

Вид текущей лексемы (имя, служебное слово, число и т. д.) будем записывать в глобальную переменную `Lex`, которая экспортируется модулем `OScan`. Для значений числовых литералов предусмотрим глобальную переменную `Num`, а символы, составляющие имя, процедура `Ident` будет помещать в глобальную переменную `Name`.

Перед тем как продолжить распознавание лексем, зафиксируем интерфейс модуля `OScan` (листинг 3.6), который уже вполне определился.

Листинг 3.6. Интерфейс модуля сканера

```
unit OScan;
{ Сканер }

interface

const
   NameLen = 31; {Наибольшая длина имени}

type
   tName = string[NameLen];
   tLex  = (lexNone, lexName, lexNum,
            lexMODULE, lexIMPORT, lexBEGIN, lexEND,
            lexCONST, lexVAR, lexWHILE, lexDO,
            lexIF, lexTHEN, lexELSIF, lexELSE,
            lexMult, lexDIV, lexMOD, lexPlus, lexMinus,
            lexEQ, lexNE, lexLT, lexLE, lexGT, lexGE,
            lexDot, lexComma, lexColon, lexSemi, lexAss,
            lexLpar, lexRpar,
```

[23] Здесь полезно будет ввести в обиход понятие «литерал». В данном случае — это число, записанное с помощью цифр. Это числовой литерал. В то же время в программе могут быть числовые константы, которым даны символические имена, и которые литералами не являются. Литерал — это константа, записанная как есть, буквально. Наряду с числовыми, могут быть строковые, символьные и другие литералы.

```
            lexEOT);

  var
     Lex   : tLex;    {Текущая лексема            }
     Name  : tName;   {Строковое значение имени   }
     Num   : integer; {Значение числовых литералов}
     LexPos: integer; {Позиция начала лексемы     }

  procedure InitScan;
  procedure NextLex;

  {=====================================================}
```

Несколько пояснений. Определив тип `tName` для имён и установив максимальную длину имени, мы ввели ограничение, которого не было в спецификации языка. Это особенность нашей реализации языка «О». Такие ограничения представляются вполне допустимыми. Другим решением могла быть попытка не вводить предельной длины имени. Это, несомненно, привело бы к усложнению способа хранения имен, но вряд ли принесло бы реальную пользу, поскольку принятое ограничение в 31 символ достаточно для подавляющего большинства программ. Наш компилятор будет предупреждать о превышении этого лимита.

Процедура `InitScan`, упомянутая в интерфейсе модуля, будет отвечать за приведение сканера в исходное состояние. В числе ее задач — прочитать первую лексему программы и заполнить таблицу служебных слов.

Продолжим распознавание лексем. При выборе порядка следования вариантов в **`case`** можно учитывать частоту использования лексем разных видов в программах. Если лексемы, встречающиеся чаще, распознавать в первую очередь, сканер, возможно, будет работать несколько быстрей[24]. Одна из самых употреб-

[24] Оптимизирующие компиляторы создают код для оператора **`case`** (и подобных ему в других языках), основанный на использовании таблицы переходов. В этом случае скорость выполнения **`case`** не зависит от упорядочения вариантов.

ляемых лексем — точка с запятой, часто используются точка и запятая:

```
';':
   begin
      NextCh;
      Lex := lexSemi;
   end;
'.':
   begin
      NextCh;
      Lex := lexDot;
   end;
',':
   begin
      NextCh;
      Lex := lexComma;
   end;
```

Распознавание этих лексем элементарно. Переменная `Lex` просто получает соответствующее значение. Вызов `NextCh` делает текущим следующий за лексемой символ.

Немного сложнее обстоит дело с обработкой двоеточия. С этого знака могут начинаться лексемы двух видов: собственно двоеточие (лексема `lexColon`) и знак присваивания (`lexAss`). Чтобы различить эти два случая достаточно прочитать еще один символ. Если это будет знак «=», значит лексема — присваивание, иначе — двоеточие. В первом случае перед выходом из `NextLex` надо прочитать еще один символ.

```
':':
   begin
      NextCh;
      if Ch = '=' then begin
         NextCh;
         Lex := lexAss;
         end
      else
         Lex := lexColon;
   end;
```

Следующие лексемы — знаки операций отношения. Их шесть видов. Но начинаться эти лексемы могут только с четырех различных символов:

```
'=':
   begin
      NextCh;
      Lex := lexEQ;
   end;
'#':
   begin
      NextCh;
      Lex := lexNE;
   end;
'<':
   begin
      NextCh;
      if Ch='=' then begin
         NextCh;
         Lex := lexLE;
         end
      else
         Lex := lexLT;
   end;
'>':
   begin
      NextCh;
      if Ch='=' then begin
         NextCh;
         Lex := lexGE;
         end
      else
         Lex := lexGT;
   end;
```

Открывающая круглая скобка может представлять саму себя (лексема `lexLPar`), но с этого же знака начинается комментарий. Различим эти два случая, прочитав еще один символ:

```
'(':
   begin
      NextCh;
      if Ch = '*' then begin
         Comment;
         NextLex; {Рекурсия}
         end
      else
```

```
            Lex := lexLpar;
        end;
```

Обработку комментария (которая состоит в пропуске всех его символов) выполнит процедура `Comment`, которую запишем позже. А пока отметим, что в момент начала её работы текущим является символ «*». После пропуска комментария процедура `NextLex` вызывается рекурсивно — сам комментарий лексемой не является, а `NextLex` не может завершить работу, не прочитав лексемы.

Распознавание остальных видов лексем не составляет труда:

```
    ')':
       begin
          NextCh;
          Lex := lexRpar;
       end;
    '+':
       begin
          NextCh;
          Lex := lexPlus;
       end;
    '-':
       begin
          NextCh;
          Lex := lexMinus;
       end;
    '*':
       begin
          NextCh;
          Lex := lexMult;
       end;
    chEOT:
       Lex := lexEOT;
    else
       Error('Недопустимый символ');
    end {case};
  end {NextLex};
```

Если символ, ставший текущим после пропуска пробелов, не совпадает ни с одним из перечисленных вариантов, сканер сообщает о лексической ошибке.

Распознавание имен

Выполняется процедурой Ident (листинг 3.7). В ее задачу входит считывание символов имени и их запись в переменную сканера Name. Если считанное имя не совпадает ни с одним из зарезервированных слов, процедура Ident должна занести значение lexName в переменную сканера Lex, в противном случае Lex получает значение, соответствующее зарезервированному слову. В начале работы процедуры Ident переменная Ch уже содержит первую букву идентификатора.

Листинг 3.7. Сканирование идентификаторов

```
procedure Ident;
var
   i : integer;
begin
   i := 0;
   repeat
      if i < NameLen then begin
         i := i + 1;
         Name[i] := Ch;
         end
      else
         Error('Слишком длинное имя');
      NextCh;
   until not (Ch in ['A'..'Z', 'a'..'z', '0'..'9']);
   Name[0] := chr(i); {Длина строки Name теперь равна i}
   Lex := TestKW;     {Проверка на ключевое слово}
end;
```

Символы заносятся в строку Name прямым обращением к ее элементам: Name[i] := Ch. По завершении цикла в нулевой элемент строки записывается ее длина — используется особенность представления строк во многих реализациях языка Паскаль.

При превышении допустимой длины имени сканер сообщает об ошибке и прекращает работу компилятора. Другим возможным решением было бы продолжение чтения символов имени, без их записи в переменную Name. В этом случае компилятор будет раз-

решать использовать имена произвольной длины, но различаться они будут по первым `NameLen` символам[25]. Первый, более строгий вариант представляется предпочтительным, поскольку исключает какие-либо недоразумения.

Таблица ключевых слов

Определение вида лексемы по содержимому переменной `Name` выполняет функция `TestKW` (testing keywords — проверка ключевых слов), значением которой является вид лексемы. Функция `TestKW` использует таблицу, ключ поиска в которой — строка, содержащая идентификатор. Для каждого служебного слова в таблице записан вид соответствующей лексемы (значение типа `tLex`). Если для какого-то имени поиск в таблице оканчивается неудачей (имя не найдено, поскольку не совпадает ни с одним из ключевых слов), функция `TestKW` возвращает значение `lexName`.

Таблица ключевых слов может быть организована по-разному. При ее конструировании можно стремиться сделать поиск по возможности быстрым. Обращение к таблице служебных слов происходит всякий раз, когда в компилируемой программе встречается идентификатор, поэтому от времени поиска заметно зависит скорость работы всего компилятора. Подходящими вариантами для таблицы будут двоичный поиск в упорядоченном по ключам массиве, перемешанная (хэш) таблица, таблица, организованная в виде двоичного дерева. Можно учесть, что содержание таблицы неизменно — в ней содержатся 34 зарезервированных слова языка «О» (Оберона-2). Это позволяет заранее разместить их так, чтобы максимально ускорить поиск. Разумно принять во внимание и частоту использования различных служебных слов в реальных программах.

---

[25] Такой подход используется в диалектах языка Паскале компании Borland, где значащими являются первые 63 и 255 символов имени соответственно.

В начале работы компилятора таблица зарезервированных слов заполняется данными. Занесение отдельного слова в таблицу будем выполнять с помощью процедуры EnterKW. При ее вызове указываются ключевое слово и вид лексемы. Вот так может выглядеть занесение в таблицу первых трех слов:

```
EnterKW('ARRAY', lexNone);
EnterKW('BY',    lexNone);
EnterKW('BEGIN', lexBEGIN);
```

Устройство процедуры EnterKW и порядок занесения слов зависят от выбранного способа организации таблицы.

ЗАДАНИЕ

> Сконструируйте и реализуйте таблицу служебных слов так, чтобы скорость работы сканера была по возможности наибольшей. При этом желательно не жертвовать наглядностью программы и не превышать разумного расхода памяти.

Здесь же мы используем простейший вариант организации таблицы служебных слов — массив, в котором выполняется последовательный, линейный поиск. Это самый медленный способ. Но его простота позволяет быстрее продвинуться в разработке компилятора, сохраняя уверенность в правильной работе сканера. В дальнейшем линейный поиск можно заменить более эффективным алгоритмом.

Предусмотрим в секции реализации модуля OScan следующие описания:

```
const
   KWNum = 34;   {Размер таблицы}
type
   tKeyWord = string[9]; {Длина слова PROCEDURE}
var
   nkw     : integer; {Число занесенных в таблицу слов}
   KWTable : array [1..KWNum] of
      record
         Word  : tKeyWord;
         Lex   : tLex;
      end;
```

Заполнение таблицы выполняется при инициализации сканера процедурой `InitScan` (листинг 3.8).

Листинг 3.8. Инициализация сканера

```
procedure InitScan;
begin
   nkw := 0; {Вначале таблица пуста}

   EnterKW('ARRAY',     lexNone);
   EnterKW('BY',        lexNone);
   EnterKW('BEGIN',     lexBEGIN);
   EnterKW('CASE',      lexNone);
   EnterKW('CONST',     lexCONST);
   EnterKW('DIV',       lexDIV);
   EnterKW('DO',        lexDO);
   EnterKW('ELSE',      lexELSE);
   EnterKW('ELSIF',     lexELSIF);
   EnterKW('END',       lexEND);
   EnterKW('EXIT',      lexNone);
   EnterKW('FOR',       lexNone);
   EnterKW('IF',        lexIF);
   EnterKW('IMPORT',    lexIMPORT);
   EnterKW('IN',        lexNone);
   EnterKW('IS',        lexNone);
   EnterKW('LOOP',      lexNone);
   EnterKW('MOD',       lexMOD);
   EnterKW('MODULE',    lexMODULE);
   EnterKW('NIL',       lexNone);
   EnterKW('OF',        lexNone);
   EnterKW('OR',        lexNone);
   EnterKW('POINTER',   lexNone);
   EnterKW('PROCEDURE', lexNone);
   EnterKW('RECORD',    lexNone);
   EnterKW('REPEAT',    lexNone);
   EnterKW('RETURN',    lexNone);
   EnterKW('THEN',      lexTHEN);
   EnterKW('TO',        lexNone);
   EnterKW('TYPE',      lexNone);
   EnterKW('UNTIL',     lexNone);
   EnterKW('VAR',       lexVAR);
   EnterKW('WHILE',     lexWHILE);
   EnterKW('WITH',      lexNone);

   NextLex; {Чтение первой лексемы}
   {возможно только после заполнения таблицы}
end;
```

Вызов `InitScan` следует предусмотреть при инициализации компилятора, за которую отвечает процедура `Init` в основной программе:

```
procedure Init;
begin
   ResetText;
   InitScan;
end;
```

Занесение в таблицу данных об одном служебном слове выполняет процедура `EnterKW`:

```
procedure EnterKW(Name: tKeyWord; Lex: tLex);
begin
   nkw := nkw + 1;
   KWTable[nkw].Word := Name;
   KWTable[nkw].Lex := Lex;
end;
```

Функция `TestKW` (листинг 3.9) выполняет линейный поиск значения `Name` в таблице `KWTable`:

Листинг 3.9. Линейный поиск в таблице ключевых слов

```
function TestKW: tLex;
var
   i : integer;
begin
   i := nkw;
   while (i>0) and (Name<>KWTable[i].Word) do
      i := i-1;
   if i>0 then
      TestKW := KWTable[i].Lex
   else
      TestKW := lexName;
end;
```

Сканирование числовых литералов

В языке «О» предусмотрены только целые числа. Их обработка выполняется достаточно просто (листинг 3.10). Вычисленное значение числового литерала записывается в переменную сканера `Num`.

Листинг 3.10. Сканирование чисел

```
procedure Number;
var
   d : integer;
begin
   Lex := lexNum;
   Num := 0;
   repeat
      d := ord(Ch) - ord('0');
      if (Maxint - d) div 10 >= Num then
         Num := 10*Num + d
      else
         Error('Слишком большое число');
      NextCh;
   until not (Ch in ['0'..'9']);
end;
```

При вызове процедуры `Number` переменная `Ch` уже содержит первую цифру числа.

Пропуск комментариев

Выполняется процедурой `Comment`. Ее задача — прочитать все символы, включая закрывающую комментарий пару «*)». В языке «О», как и в Обероне, комментарии могут быть вложенными, поэтому должно быть правильно учтено соответствие открывающих «(*» и закрывающих «*)» пар символов.

При обработке комментария необходимо учитывать, что текст программы может закончиться до того, как встретятся символы «*)». Поэтому нужна явная проверка символа «конец текста». Синтаксическая диаграмма для комментария представлена на рис. 3.4.

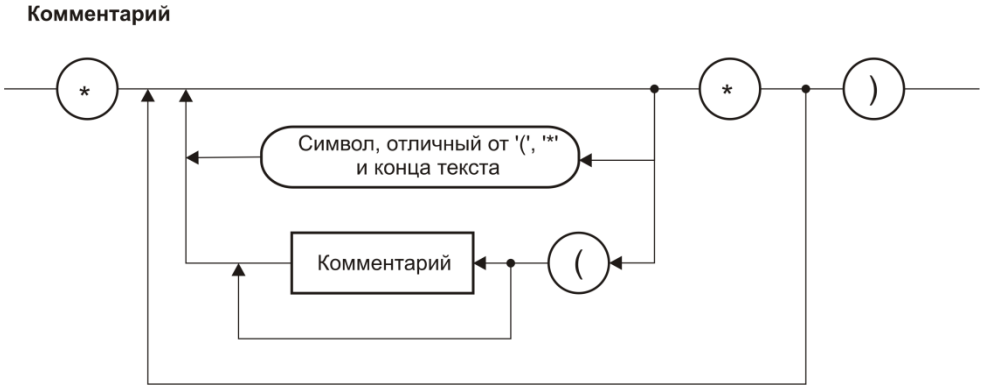


Рис. 3.4. Синтаксическая диаграмма комментария

Вызов процедуры `Comment` (листинг 3.11) происходит, когда текущим символом является «*», поэтому формально удобно считать, что комментарий — это конструкция, начинающаяся символом «*». Такому соглашению подчиняется и рекурсивное обращение к комментарию, которое можно видеть на диаграмме. В соответствии со структурой диаграммы распознающая процедура содержит два вложенных цикла и рекурсивно обращается к себе.

Листинг 3.11. Рекурсивный пропуск комментариев

```
procedure Comment;
begin
   NextCh;
   repeat
      while (Ch <> '*') and (Ch <> chEOT) do
         if Ch = '(' then begin
            NextCh;
            if Ch = '*' then Comment;
            end
         else
            NextCh;
      if Ch = '*' then NextCh;
   until Ch in [')', chEOT];
   if Ch = ')' then
      NextCh
   else begin
      LexPos := Pos;
      Error('Не закончен комментарий');
   end;
end;
```

Чтобы при выдаче диагностического сообщения «Не закончен комментарий» было правильно указано место ошибки (место, где закончился текст), переменной `LexPos` присваивается значение `Pos`, равное номеру текущего символа в строке. В обычном случае `LexPos` обозначает номер символа, начинающего лексему, но комментарий — не лексема, может занимать несколько строк, поэтому сохранять значение `LexPos`, соответствующее месту, где комментарий начался, было бы неправильно.

Пропуск комментариев может быть выполнен и по-другому. Программа, приведенная в листинге 3.12, использует счетчик уровня вложенности комментариев `Level`. Счетчик увеличивается на единицу при входе в комментарий и уменьшается при выходе из него. Нулевое значение `Level` соответствует положению вне комментария.

Листинг 3.12. Нерекурсивный пропуск комментариев

```
procedure Comment;
var
   Level : integer;
begin
   Level := 1;
   NextCh;
   repeat
      if Ch = '*' then begin
         NextCh;
         if Ch = ')' then begin
            Level := Level - 1;
            NextCh
         end;
         end
      else if Ch = '(' then begin
         NextCh;
         if Ch = '*' then begin
            Level := Level + 1;
            NextCh
         end;
         end
      else {if Ch <> chEOT then}
         NextCh;
   until (Level = 0) or (Ch = chEOT);
   if Level <> 0 then begin
```

```
        LexPos := Pos;
        Error('Не закончен комментарий');
    end;
end;
```

Работа этой процедуры подобна поведению автомата с магазинной памятью. Роль стека здесь исполняет счетчик `Level`. В стек как бы помещаются открывающие скобки комментария, а каждая закрывающая скобка удаляет из стека соответствующую открывающую. Но, поскольку в стек всегда заносятся одни и те же элементы, запоминать их самих нет нужды, достаточно подсчитывать их количество.

Тестирование сканера

Лексический анализатор не выполняет в нашем компиляторе самостоятельного просмотра исходной программы. Сканер — лишь процедура, которую вызывает синтаксический анализатор, если ему требуется очередная лексема. Чтобы не откладывать тестирование сканера до момента, когда будет готов синтаксический анализатор, и для того, чтоб быстрее увидеть, как синтаксический анализатор взаимодействует со сканером, заменим пустую заглушку, использовавшуюся в роли процедуры `Compile` модуля `OPars` (см. листинг 3.3) содержательной заглушкой (листинг 3.13). Она будет читать лексемы до исчерпания входного текста, то есть до тех пор, пока очередной лексемой не станет `lexEOT`. Можно даже поручить процедуре `Compile` подсчет числа прочитанных лексем.

Листинг 3.13. Промежуточная версия синтаксического анализатора с подсчетом лексем

```
unit OPars;
{Распознаватель}
interface

procedure Compile;

{=====================================================}
```

```
implementation

uses OScan;

procedure Compile;
var
   n : integer;
begin
   n := 0;
   while Lex <> lexEOT do begin
      n := n + 1;
      NextLex;
   end;
   Writeln('Число лексем ', n );
end;

end.
```

Подсчет числа лексем интересен сам по себе. Количество лексем может служить мерой объема исходного текста программы. Ведь на размер программного текста в строках или байтах влияют многие вещи, которые не имеют отношения к содержанию программы, а зависят от индивидуального стиля программиста и особенностей языка программирования. Размер в байтах меняется в зависимости от длины имен, выбираемых программистом, длины служебных слов языка, количества пробелов. Число строк тоже зависит от индивидуальных привычек. Например, в программах, написанных Н. Виртом, можно видеть длинные строки, содержащие по несколько операторов, в то время как во многих других источниках, в том числе в этой книге, культивируется стиль, предполагающий запись не более одного оператора в строке.

Измерение числа лексем позволяет объективней оценить объем программистской работы и размеры программ.

Получившаяся у нас промежуточная версия компилятора способна обработать любую правильную программу на языке «О» и может сообщить о лексической ошибке и нарушении ограничений реализации при записи идентификаторов и чисел. В то же

время наш анализатор будет без возражений обрабатывать текст, представляющий собой синтаксически неправильную последовательность правильно записанных лексем. Например, два приведенных в таблице 3.2 текста никаких ошибок не вызовут.

Таблица 3.2. Последовательности лексем

Правильная программа	Правильные лексемы
`MODULE Module;` `END Module.`	`.Module END` `;Module MODULE`

Синтаксический анализатор

Имея в распоряжении сканер, будем программировать синтаксический анализатор методом рекурсивного спуска, считая лексемы терминальными символами.

Структура распознающих процедур определяется синтаксисом соответствующих конструкций. Мы не будем пользоваться для задания синтаксиса диаграммами, поскольку формулы, записанные на РБНФ, и так достаточно наглядны. В тексте распознавателя перед каждой распознающей процедурой в качестве комментария будем записывать РБНФ-выражение, определяющее синтаксис соответствующей конструкции.

Все процедуры синтаксического анализатора располагаются в секции реализации модуля `OPars`. Интерфейс модуля (см. листинги 3.3 и 3.13) при этом не изменяется.

Начальным нетерминалом грамматики языка «О» является «Модуль». Соответствующая процедура распознавателя будет называться `Module`, а ее вызовом из процедуры `Compile` начнется синтаксический анализ:

```
procedure Compile;
begin
   Module;
   Writeln('Компиляция завершена');
end;
```

Находившуюся на этом месте заглушку, считавшую число лексем, пришлось удалить.

Процедура `Module` записывается в соответствии с синтаксисом, заданным формулой для нетерминала «Модуль».

```
(* MODULE Имя ";" [Импорт] ПослОбъявл
   [BEGIN ПослОператоров] END Имя "." *)
procedure Module;
begin
   if Lex = lexMODULE then {Слово MODULE   }
      NextLex
   else
      Expected('MODULE');
   if Lex = lexName then   {Имя модуля     }
      NextLex
   else
      Expected('имя модуля');
   if Lex = lexSemi then   {Точка с запятой}
…
```

Можно было бы и дальше писать в таком стиле, но уже в самом начале мы столкнулись с тремя идущими подряд проверками соответствия текущей и ожидаемой лексемы. Проверки эти однотипны, и есть смысл поручить их выполнение специальной процедуре. Тем более, что и в дальнейшем подобные ситуации будут встречаться. Назовем такую процедуру `Check` (проверка, контроль). Её параметрами будут вид ожидаемой анализатором лексемы `L` и строка `M` (от message — «сообщение»), которая будет передана процедуре `Expected` и вставлена ею после слова «Ожидается» в сообщение об ошибке.

```
procedure Check(L: tLex; M: string);
begin
   if Lex <> L then
      Expected(M)
   else
      NextLex;
end;
```

Теперь, анализатор модуля запишется проще (листинг 3.14).

Листинг 3.14. Синтаксический анализатор модуля

```
(* MODULE Имя ";" [Импорт] ПослОбъявл
   [BEGIN ПослОператоров] END Имя "." *)
procedure Module;
begin
   Check(lexMODULE, 'MODULE');
   Check(lexName, 'имя модуля');
   Check(lexSemi, '";"');
   if Lex = lexIMPORT then
      Import;
   DeclSeq;    {Последовательность объявлений}
   if Lex = lexBEGIN then begin
      NextLex;
      StatSeq; {Последовательность операторов}
   end;
   Check(lexEND, 'END');
   Check(lexName, 'имя модуля');
   Check(lexDot, '".."');
end;
```

Эта процедура выполняет синтаксический анализ в чистом виде, не пытаясь делать какие-либо дополнительные контекстные проверки. Так, не контролируется соответствие имени, записанного после слова **END** в конце модуля и названия модуля, идущего вслед за **MODULE**. Когда компилятор будет дополнен средствами контекстного анализа, такие проверки, конечно, будут выполняться. Приведенную же версию процедуры `Module` следует рассматривать как предварительную, в дальнейшем она будет дорабатываться.

Продолжим программирование анализатора, реализуя распознающие процедуры, обращение к которым уже предусмотрено процедурой `Module`. Первая в очереди — процедура-анализатор списка импорта (листинг 3.15). При ее рассмотрении также надо иметь в виду, что это чистый синтаксический анализ. Никаких попыток выполнить импорт фактически и даже проверить, действительно ли встречающиеся имена — это имена модулей, пока не предпринимается.

Листинг 3.15. Синтаксический анализатор списка импорта

```
(* IMPORT Имя {"," Имя} ";" *)
procedure Import;
begin
   Check(lexIMPORT, 'IMPORT');
   Check(lexName, 'имя модуля');
   while Lex = lexComma do begin
      NextLex;
      Check(lexName, 'имя модуля');
   end;
   Check(lexSemi, '";"');
end;
```

Далее (листинг 3.16) следует анализатор последовательности объявлений.

Листинг 3.16. Распознающая процедура для последовательности объявлений

```
(* {CONST {ОбъявлКонст ";"}
    |VAR {ОбъявлПерем ";"} } *)
procedure DeclSeq;
begin
   while Lex in [lexCONST, lexVAR] do begin
      if Lex = lexCONST then begin
         NextLex;
         while Lex = lexName do begin
            ConstDecl; {Объявление константы}
            Check(lexSemi, '";"');
         end;
         end
      else begin
         NextLex; { VAR }
         while Lex = lexName do begin
            VarDecl;   {Объявление переменных}
            Check(lexSemi, '";"');
         end;
      end;
   end;
end;
```

Это окончательный текст процедуры `DeclSeq` (от declarations sequence — последовательность объявлений). Никаких контекстных проверок и действий по генерации машинного кода на этом уровне анализа происходить не будет, поскольку любые такие действия относятся к объявлениям конкретных констант и

переменных, а их обработка будет сконцентрирована в процедурах `ConstDecl` и `VarDecl`.

Анализатор последовательности операторов (листинг 3.17) также в дальнейшем не изменится.

Листинг 3.17. Анализатор последовательности операторов

```
(* Оператор {";" Оператор} *)
procedure StatSeq;
begin
   Statement;    {Оператор}
   while Lex = lexSemi do begin
      NextLex;
      Statement; {Оператор}
   end;
end;
```

С нетерминалом «последовательность операторов» в грамматике языка «О» связано самовложение. Последовательность операторов состоит из операторов, а некоторые операторы содержат в себе последовательности операторов. В программе-анализаторе это порождает косвенную рекурсию. Чтобы заголовки участвующих в рекурсии процедур были известны до места вызова этих процедур, предусмотрим опережающее описание процедуры `StatSeq`, которое можно разместить в начале секции реализации модуля `OPars`:

```
procedure StatSeq; forward;
```

Цепочку распознающих процедур можно было бы разворачивать и дальше. Реализуя алгоритм рекурсивного спуска, это можно сделать легко и быстро. В ходе реальной разработки так и следует поступить, завершив синтаксический анализатор полностью. Но здесь и сейчас я не буду выписывать все распознающие процедуры, поскольку большинство из них подвергнутся модернизации при создании контекстного анализатора и генератора кода. Тогда их и запишем. Принцип же, надеюсь, ясен. Приведу для примера лишь процедуру, анализирующую слагаемое (листинг 3.18), — она является частью анализатора выражений.

Листинг 3.18. Синтаксический анализатор слагаемого

```
(* Множитель {ОперУмн Множитель} *)
procedure Term; {Слагаемое}
begin
   Factor;    {Множитель}
   while Lex in [lexMult, lexDIV, lexMOD] do begin
      NextLex;
      Factor; {Множитель}
   end;
end;
```

Контекстный анализ

В ходе контекстного анализа должно быть проверено соблюдение тех правил языка, которые не выражаются с помощью контекстно-свободных грамматик[26]. Примерами являются проверка правильности употребления имен и контроль соответствия типов.

Таблица имен

Корректность использования имени в конкретном месте программы может быть проверена, если известно, какой объект программы этим именем обозначен: константа, тип, переменная, процедура, модуль. Для имени константы, переменной, процедуры (функции) необходимо также знать, к какому типу они относятся. Могут быть важны и другие характеристики, связанные с именем.

---

[26] Точнее сказать, не выражены с помощью КС-грамматики авторами спецификации языка. В некоторых случаях, требования, которые могут быть в принципе заданы с помощью синтаксических правил, разумнее формулировать словесно, чтобы не усложнять формальную грамматику. Примером может служить проблема «висячего else» в языках Паскаль, Си, Ява, Си#, связанная с потенциально неоднозначной трактовкой конструкции, содержащей последовательность `then if` (в Си-подобных языках: if(…) if …). В спецификации Паскаля и Си# просто замечено, что `else` всегда относится к ближайшему предшествующему `if`, в то время как в спецификации языка Ява это выражено с помощью правил КС-грамматики, что заметно ее усложнило.

Атрибуты каждого имени, используемого в программе, хранятся в специальной таблице транслятора — таблице имен. Заполняется таблица имен при трансляции объявлений и списка импорта. Предопределенные имена (имена стандартных типов, процедур) могут быть занесены в таблицу заранее. При трансляции операторов обращение к таблице имен позволяет определить атрибуты каждого встретившегося имени, и служит для выявления необъявленных имен.

Блочная структура и области видимости

Структура таблицы имен должна отражать блочную структуру программы. Одно и то же имя может обозначать несколько объектов в программе, если они определены в разных ее блоках. Поэтому сведения об именах также надо хранить в разных блоках таблицы имен. И хотя в языке «О» понятие «блок» не используется, мы учтем блочную структуру программы при организации таблицы имен. Во-первых, это будет полезно при последующем расширении языка, во-вторых, как скоро выяснится, пригодится уже при существующем его состоянии, в-третьих, позволит представить организацию таблицы имен в трансляторах «настоящих» языков.

В языке Оберон блоки образуются модулями и процедурами. Рассмотрим программу на Обероне (листинг 3.19) — модуль, содержащий вложенные процедуры.

Листинг 3.19. Блочная структура и области видимости

```
MODULE M;
VAR
   v1, v2, v3 : INTEGER;
PROCEDURE P1(…);
VAR
   v1, v2 : INTEGER;
   PROCEDURE P11(…);
   VAR
      v1 : INTEGER;
   BEGIN
```

```
        {Здесь видны локальная переменная v1,
         переменная v2 процедуры P1
         а также глобальная переменная v3.
         Видны также имена P11, P1 и M}
    …
    END P11;
BEGIN
    {Здесь видны P11, P1, M,
     локальные v1 и v2 процедуры P1,
     а также глобальная переменная v3}
    …
END P1;
PROCEDURE P2(…);
VAR
    v2 : INTEGER;
BEGIN
    {Здесь видны P2, P1, M,
     локальная переменная v2 процедуры P2,
     а также глобальные v1 и v3}
    …
END P2;
BEGIN
    {Здесь видны P2, P1, M и глобальные v1, v2, v3}
    …
END M.
```

Область видимости[27] имени простирается от точки его объявления до конца блока (модуля, процедуры), в начале которого это объявление находится. Из нее исключаются области видимости одноименных объектов, объявленных во вложенных блоках.

Правильно считать «точкой объявления» имени то место в программе, где это имя записано, а не конец объявления, в котором это имя содержится. Например, началом области видимости переменной `v1`, содержащейся в объявлении

```
VAR
    v1{точка 1}, v2, v3: INTEGER;{точка 2}
```

следует считать точку 1, а не точку 2.

[27] Используется также название «область действия». Соответствующий английский термин — scope.

Блоки программы можно рассматривать как *пространства имен* — пространство имен модуля, пространство имен процедуры. Области видимости отдельных идентификаторов вложены в соответствующие пространства имен.

Блоки таблицы имен создаются компилятором при обработке начала каждого блока программы (модуля, процедуры). После того, как однопроходный компилятор закончил трансляцию процедурного блока, соответствующий блок таблицы имен может быть уничтожен, ведь области видимости локальных имен этой процедуры закончились, и эти имена не могут быть видны в последующих частях программы.

Последовательность создания и уничтожения блоков таблицы имен при трансляции программы, приведенной в листинге 3.19, получается такой.

1. Создание блока для пространства имен модуля (блок M).
2. Создание блока для процедуры P1 (блок P1) при входе в процедуру P1, то есть при трансляции заголовка процедуры P1.
3. Создание блока для процедуры P11 (блок P11).
4. Уничтожение блока P11 по окончании трансляции процедуры P11.
5. Уничтожение блока P1.
6. Создание блока для процедуры P2 (блок P2).
7. Уничтожение блока P2.
8. Уничтожение блока M по окончании трансляции модуля.

Можно видеть, что создание и уничтожение блоков таблицы имен подчиняется стековой дисциплине: блок, созданный последним, уничтожается первым. Соответственно, и в таблице имен блоки должны образовывать стек.

Блок стандартных идентификаторов

Кроме имен, которые программист определяет сам, в программе могут использоваться предопределенные идентификаторы: имена стандартных типов, процедур, функций. В Обероне (и языке «О») такими именами являются, например, `INTEGER`, `ABS`, `MAX`. Их область видимости распространяется на весь текст модуля. Но такие имена не являются зарезервированными словами (в отличие, например, от **`BEGIN`**), и программист может придать им в программе другой смысл. Если записать

```
VAR MAX: INTEGER;
```

то в остальной части блока, в котором находится это объявление, `MAX` будет обозначать переменную целого типа, а не стандартную функцию `MAX`.

Таким образом, по отношению к стандартным идентификаторам могут быть применены обычные правила, относящиеся к пространствам имен, если считать, что эти идентификаторы определены в блоке, охватывающем модуль. Блок стандартных идентификаторов открывается в таблице имен перед началом трансляции модуля[28]. Стандартные идентификаторы заносятся в таблицу, после чего может быть открыт блок для пространства имен модуля.

В связи с необходимостью правильной обработки стандартных идентификаторов блочная структура таблицы имен оказывается актуальной и для компилятора языка «О».

Таблица имен компилятора «О»

Выберем простой, хоть и не очень эффективный (из-за медленного поиска имени) способ организации таблицы имен — линейный список. Если добавление и удаление элементов выпол-

[28] В языке «О» модуль и программа — это одно и то же.

нять с одного конца списка, он ведет себя подобно стеку, что и требуется для таблицы имен. Чтобы разграничить блоки таблицы, используем элементы, которые в одном из своих полей будут содержать специальный признак (рис. 3.5).

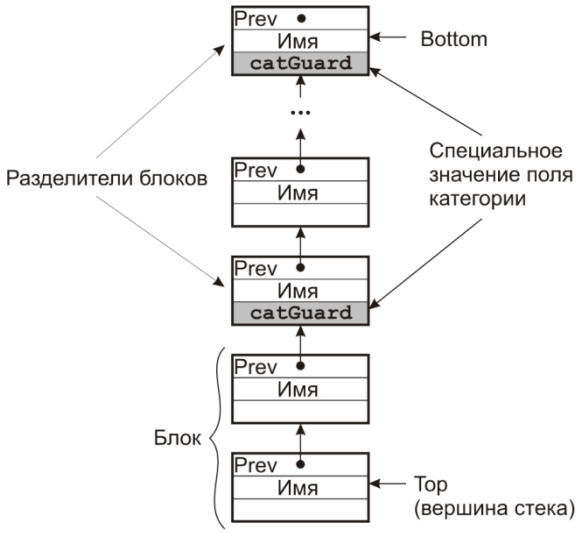


Рис. 3.5. Устройство таблицы имен компилятора языка «О»

Информация в таблице имен

Для каждого имени, встретившегося в программе на языке «О», в таблице имен будет храниться следующая информация:

- Само имя в виде строки символов. Будет служить ключом при поиске.
- Категория имени. Что имя обозначает: константу, переменную, тип, стандартную процедуру, модуль.
- Тип. Должен быть указан для имен переменных, констант, процедур-функций.
- Значение. Для имени константы это будет ее числовое значение. Для переменных и процедур это поле также пригодится.

Программный модуль OTable

Теперь можно определить программный интерфейс модуля `OTable` (листинг 3.20), который в нашем компиляторе будет отвечать за работу с таблицей имен.

Листинг 3.20. Интерфейс модуля для работы с таблицей имен

```
unit OTable;
{ Таблица имен }

interface

uses OScan;

type
   {Категории имён}
      tCat = (catConst, catVar, catType,
              catStProc, catModule, catGuard);

   {Типы}
      tType = (typNone, typInt, typBool);

   tObj = ^tObjRec;      {Тип указателя на запись таблицы}
   tObjRec = record      {Тип записи таблицы имен}
      Name  : tName;     {Ключ поиска             }
      Cat   : tCat;      {Категория имени         }
      Typ   : tType;     {Тип                     }
      Val   : integer;   {Значение                }
      Prev  : tObj;      {Указатель на пред. имя }
   end;

{Инициализация таблицы}
   procedure InitNameTable;
{Добавление элемента}
   procedure Enter
      (N: tName; C: tCat; T: tType; V: integer);
{Занесение нового имени}
   procedure NewName
      (Name: tName; Cat: tCat; var Obj: tObj);
{Поиск имени}
   procedure Find(Name: tName; var Obj: tObj);
{Открытие области видимости (блока)}
   procedure OpenScope;
{Закрытие области видимости (блока)}
   procedure CloseScope;

{=======================================================}
```

Среди значений перечислимого типа `tCat`, обозначающего категории объектов в таблице имен, предусмотрена константа `catGuard`, которая будет обозначать специальный элемент таблицы, разграничивающий ее блоки (guard по-английски — страж, ограждение; frontier guard — пограничник). Значение `typNone` — «никакой», использованное в определении `tType`, потребуется для указания об отсутствии типа у объекта программы. Например, стандартные процедуры (не функции) не имеют типа, чем и отличаются от процедур-функций.

Инициализация таблицы, организованной как линейный список, может быть сведена к установке в значение **`nil`** указателя на список. Переменную, обозначающую указатель на начало (вершину) списка, как и указатель на его конец (дно) определим как глобальные в секции реализации модуля `OTable`:

```
var
   Top    : tObj;  {Указатель на вершину списка    }
   Bottom : tObj;  {Указатель на конец (дно) списка}

{Инициализация таблицы имен}
procedure InitNameTable;
begin
   Top := nil;
end;
```

Добавление элемента к списку будет выполнять процедура `Enter` (листинг 3.21). Ее параметрами являются значения полей записи таблицы имен.

Листинг 3.21. Добавление элемента в таблицу имен

```
procedure Enter(N: tName; C: tCat; T: tType; V: integer);
var
   P : tObj;
begin
   New(P);
   P^.Name := N;
   P^.Cat := C;
   P^.Typ := T;
   P^.Val := V;
   P^.Prev := Top;
```

```
    Top := P;
end;
```

Процедуры `OpenScope` и `CloseScope` (листинг 3.22) отвечают за открытие и закрытие (уничтожение) блоков таблицы имен, соответствующих пространствам имен в программе.

Листинг 3.22. Открытие и закрытие областей видимости

```
procedure OpenScope;
begin
   Enter('', catGuard, typNone, 0);
   if Top^.Prev = nil then
      Bottom := Top;
end;

procedure CloseScope;
var
   P : tObj;
begin
   while Top^.Cat <> catGuard do begin
      P := Top;
      Top := Top^.Prev;
      Dispose(P);
   end;
   P := Top;
   Top := Top^.Prev;
   Dispose(P);
end;
```

Процедура `OpenScope` помещает в список элемент, разделяющий блоки таблицы. Этот элемент содержит значение `catGuard` в поле категории имени. Если открывается первый блок таблицы имен, указатель `Bottom` устанавливается на добавленный пограничный элемент. Указатель `Bottom` будет использоваться при поиске имен.

Процедура `CloseScope` удаляет блок таблицы имен, находящийся на вершине стека (открытый последним). Для этого уничтожаются все записи от вершины списка до ближайшего пограничного элемента включительно.

Заполнение таблицы имен

Происходит при трансляции объявлений, которые могут располагаться в начале каждого блока программы. Когда транслятор встречает объявление имени, он должен занести его и сопутствующую информацию в тот блок таблицы имен, который соответствует текущему блоку программы. Блок, в который помещаются данные, является последним по времени открытым блоком таблицы. Перед занесением необходимо проверить, нет ли уже в этом последнем блоке такого же имени. По правилам Оберона, а вслед за ним языка «О», как и по правилам многих других языков, один идентификатор не может быть объявлен в одном блоке дважды.

Добавление в таблицу нового имени `Name` будет выполнять процедура `NewName` (листинг 3.23). Входной параметр `Cat` определяет категорию имени. Указатель `Obj` на созданную в таблице запись возвращается как результат работы `NewName`. Другие атрибуты помещенного в таблицу идентификатора (тип обозначаемого объекта, значение), могут быть сформированы вызывающей программой уже после вызова `NewName`. Значение `Val` при инициализации устанавливается равным 0, что будет использовано в дальнейшем.

Перед записью имени в таблицу проверяется, нет ли **в пределах текущего блока** такого же. Если обнаружено совпадение, сообщается об ошибке.

Листинг 3.23. Запись нового имени в таблицу имен

```
procedure NewName(Name: tName; Cat: tCat; var Obj: tObj);
begin
   Obj := Top;
   while (Obj^.Cat<>catGuard) and (Obj^.Name<>Name) do
      Obj := Obj^.Prev;
   if Obj^.Cat = catGuard then begin
      New(Obj);
      Obj^.Name := Name;
      Obj^.Cat := Cat;
```

```
      Obj^.Val := 0;
      Obj^.Prev := Top;
      Top := Obj;
      end
   else
      Error('Повторное объявление имени');
end;
```

Поиск имен

Занесение нового имени в таблицу происходит при обработке его *определяющего вхождения*, то есть, когда имя объявляется. При трансляции *использующего вхождения* выполняется поиск имени в таблице. По правилам Оберона любому использующему вхождению должно предшествовать определяющее. Исключение составляют стандартные идентификаторы, которые являются предопределенными и, как мы уже решили, считаются описанными во внешнем по отношению к модулю блоке.

Процедура `Find` (листинг 3.24) ищет в таблице имя `Name`. Поиск начинается с вершины списка и **не останавливается на границах блоков**. Если во вложенных блоках программы объявлены одинаковые имена, найдено будет то, которое определено в ближайшем внутреннем блоке. И только, если имя не объявлено в текущем блоке и ни в одном из охватывающих его, сообщается об ошибке.

Листинг 3.24. Поиск имени

```
procedure Find(Name: tName; var Obj: tObj);
begin
   Bottom^.Name := Name;
   Obj := Top;
   while Obj^.Name <> Name do
      Obj := Obj^.Prev;
   if Obj = Bottom then
      Error('Необъявленное имя');
end;
```

Результатом поиска является ссылка `Obj`, указывающая на запись о найденном имени. С помощью этой ссылки можно получить всю информацию об объекте, обозначенном этим именем.

При поиске используется «барьер» — перед началом поиска в поле Name последнего в списке элемента (на который указывает Bottom), помещается искомый ключ.

Напомню, что процедура Error, которая может быть вызвана при неудачной попытке занесения имени в таблицу и при неудачном поиске, останавливает работу всего компилятора.

Контекстный анализ модуля

Контекстный анализатор не является самостоятельным блоком компилятора. Действия, связанные с проверкой имен и соответствия типов, встраиваются в синтаксический анализатор. Написанные раньше части распознавателя должны быть модернизированы с целью добавления в них действий по контекстному анализу. Первой изменим основную процедуру анализатора — Compile (листинг 3.25). В ее задачу теперь будет входить инициализация таблицы имен, открытие в таблице блока стандартных идентификаторов, занесение в этот блок стандартных имен, открытие блока, соответствующего области видимости идентификаторов модуля, вызов, как и прежде, распознавателя Module и закрытие обоих открытых блоков таблицы имен.

Листинг 3.25. Основная процедура распознавателя

```
procedure Compile;
begin
   InitNameTable;
   OpenScope; {Блок стандартных имен}
   Enter( 'ABS', catStProc, typInt, spABS );
   Enter( 'MAX', catStProc, typInt, spMAX );
   Enter( 'MIN', catStProc, typInt, spMIN );
   Enter( 'DEC', catStProc, typNone, spDEC );
   Enter( 'ODD', catStProc, typBool, spODD );
   Enter( 'HALT', catStProc, typNone, spHALT );
   Enter( 'INC', catStProc, typNone, spINC );
   Enter( 'INTEGER', catType, typInt, 0 );
   OpenScope;  {Блок модуля}
   Module;
   CloseScope; {Блок модуля}
   CloseScope; {Блок стандартных имен}
```

```
    WriteLn;
    WriteLn('Компиляция завершена');
end;
```

Занесение в таблицу стандартных идентификаторов выполняется с помощью процедуры `Enter`. Для имен стандартных процедур в качестве значений заносятся их условные номера, которые в дальнейшем будут использованы при обработке вызовов процедур. Определены эти номера могут быть в секции реализации модуля `OPars`:

```
const
    spABS    = 1;
    spMAX    = 2;
    spMIN    = 3;
    spDEC    = 4;
    spODD    = 5;
    spHALT   = 6;
    spINC    = 7;
    spInOpen = 8;
    spInInt  = 9;
    spOutInt = 10;
    spOutLn  = 11;
```

Для процедур-функций указывается их тип. В поле типа для `INC` и `DEC`, которые функциями не являются, заносится `typNone`. Значение для идентификатора `INTEGER` не требуется, а в соответствующее поле просто записывается ноль.

Обозначения процедур ввода-вывода при инициализации таблицы имен в нее не заносятся. Хотя можно видеть, что константы, задающие их условные номера, (`spInOpen` – `spOutLn`) уже определены. Идентификаторы из стандартных модулей `In` и `Out` будут занесены в таблицу имен при импорте соответствующего модуля.

Первая возможность продемонстрировать работу с таблицей имен предоставляется при модернизации распознающей процедуры `Module` (листинг 3.26). Если раньше синтаксический анализатор (см. листинг 3.13) лишь убеждался в наличии какого-либо имени после слова **`MODULE`** в начале и после слова **`END`** в конце, то

в ходе контекстного анализа нужно проверить, одинаковы ли эти имена. Для этого идентификатор, записанный после слова **MODULE**, заносится в таблицу, а имя, встретившееся после **END**, сравнивается с именем из таблицы. Для сохранения ссылки на запись об имени модуля используется локальная переменная `ModRef` (от Module Reference — ссылка на модуль).

Листинг 3.26. Распознаватель модуля

```
(* MODULE Имя ";" [Импорт] ПослОбъявл
   [BEGIN ПослОператоров] END Имя "." *)
procedure Module;
var
   ModRef: tObj; {Ссылка на имя модуля в таблице}
begin
   Check(lexMODULE, 'MODULE');
   if Lex <> lexName then
      Expected('имя модуля')
   else {Имя модуля - в таблицу имен}
      NewName(Name, catModule, ModRef);
   NextLex;
   Check(lexSemi, '";"');
   if Lex = lexIMPORT then
      Import;
   DeclSeq;
   if Lex = lexBEGIN then begin
      NextLex;
      StatSeq;
   end;
   Check(lexEND, 'END');

   {Сравнение имени модуля и имени после END}
      if Lex <> lexName then
         Expected('имя модуля')
      else if Name <> ModRef^.Name then
         Expected('имя модуля "' + ModRef^.Name + '"')
      else
         NextLex;
   Check(lexDot, '"."');
end;
```

Конечно, сравнить имя, записанное после **MODULE**, с именем после соответствующего **END** можно и не обращаясь к таблице имен. Достаточно в начале запомнить (в локальной переменной) строковое значение переменной сканера `Name`, а затем сравнить

его с именем, обнаруженным после **END**. Однако два этих решения не эквивалентны. Занесение имени модуля в таблицу означает, что в последовательности объявлений модуля уже нельзя будет определить другой объект с тем же именем. Если же имя модуля в таблицу не заносится, то такое становится возможным. Другой нюанс: если имя модуля занесено в таблицу, оно может перекрыть видимость одного из стандартных идентификаторов. Например, назвав модуль `INTEGER`, мы сделаем невозможным использование стандартного типа `INTEGER` в последовательности объявлений этого модуля.

Которое из двух решений правильное — вопрос совсем не однозначный. Имя модуля должно быть «видно» в той среде, где модуль используется. Это может быть среда Оберон-системы или операционная система. В этом смысле имя модуля не должно быть локальным внутри самого модуля[29], и решение не заносить его в таблицу имен вполне правомерно. С другой стороны, по крайней мере, в языке «О», где программа состоит из единственного модуля, нет причин разрешать применение имени модуля в каком-либо ином смысле.

Трансляция списка импорта

Синтаксический анализатор списка импорта приведен выше в листинге 3.14. Однако этот анализатор лишь убеждается, что в списке имеется одно или больше имен, не проверяя, действительно ли это имена существующих модулей, нет ли в списке повторов и попытки импортировать в модуле его самого. Действия по трансляции импорта отдельного модуля поручим специальной процедуре `ImportModule`. Распознаватель списка импорта, вызывающий `ImportModule`, приведен в листинге 3.27.

[29] Немного забегая вперед, можно заметить, что именно так обстоит дело с именами процедур: они относятся к блоку, в котором располагается сама процедура, а не к блоку процедуры.

Листинг 3.27. Анализатор списка импорта

```
(* IMPORT Имя { "," Имя } ";" *)
procedure Import;
begin
   Check(lexIMPORT, 'IMPORT');
   ImportModule;    {Импорт модуля}
   while Lex = lexComma do begin
      NextLex;
      ImportModule; {Импорт модуля}
   end;
   Check(lexSemi, '";"');
end;
```

Процедура `ImportModule` (листинг 3.28), убедившись, что текущая лексема — имя, заносит его в таблицу для того, чтобы это имя было видно в оставшейся части компилируемой программы. При добавлении в таблицу будет проверено, что имя не упомянуто дважды. В частности, при попытке импортировать модулем самого себя также будет сообщено о повторном объявлении, поскольку имя компилируемого модуля уже занесено в тот же блок таблицы.

Листинг 3.28. Импорт модуля

```
procedure ImportModule;
var
   ImpRef: tObj;
begin
   if Lex = lexName then begin
      NewName(Name, catModule, ImpRef);
      if Name = 'In' then begin
         Enter('In.Open', catStProc, typNone, spInOpen);
         Enter('In.Int', catStProc, typNone, spInInt);
         end
      else if Name = 'Out' then begin
         Enter('Out.Int', catStProc, typNone, spOutInt);
         Enter('Out.Ln', catStProc, typNone, spOutLn);
         end
      else
         Error('Неизвестный модуль');
      NextLex;
      end
   else
      Expected('имя импортируемого модуля');
end;
```

После того, как имя импортируемого модуля занесено в таблицу, выполняется собственно импорт.

При трансляции с «настоящего» языка Оберон это означало бы поиск файла модуля или файла спецификации его интерфейса в среде Оберон-системы. В упрощенном учебном компиляторе языка «О» предусмотрены лишь два стандартных модуля `In` и `Out`. Они, по существу, встроены в язык. Никаких внешних файлов, в которых хранится код или спецификация этих модулей не предусматривается. Компилятор «знает» про существование модулей `In` и `Out`. При их импорте в таблицу добавляются заранее известные компилятору имена экспортированных этими модулями процедур. Обозначения процедур `In.Open`, `In.Int`, `Out.Int` и `Out.Ln` заносятся в текущий блок таблицы имен вместе с именем модуля и точкой. Это позволяет в программе на языке «О» обратиться к таким процедурам только по их уточненным (квалифицированным) именам, включающим имя модуля, и только при условии, что соответствующий модуль программой импортирован.

Трансляция описаний

Распознаватель последовательности объявлений можно видеть выше в листинге 3.16. Никаких изменений, связанных с контекстным анализом, вносить в процедуру `DeclSeq` из листинга 3.16 не требуется, поскольку она отвечает лишь за контроль синтаксиса в последовательности объявлений констант и переменных, не имея дела с конкретными переменными и константами. Вся работа по контекстному анализу выполняется процедурой `ConstDecl`, обрабатывающей определение отдельной константы и процедурой `VarDecl`, ответственной за одно описание переменных.

Трансляция определений констант

Начнем с трансляции объявлений констант (листинг 3.29). Как и всегда, встретив определяющее вхождение имени, транслятор заносит его в таблицу, сопровождая сведениями о том, к какой категории это имя относится (при вызове `ConstDecl` имя уже является текущей лексемой). В нашем случае речь идет об имени константы (категория `catConst`), но при вызове `NewName` укажем категорию `catGuard`. Делается это для того, чтобы предотвратить тавтологию — определение константы через саму себя. Очевидно, что конструкции вроде

```
CONST c = c;
```

должны быть запрещены. Если же в момент записи в таблицу имен сразу назначить определяемому имени категорию константы, его тут же можно будет использовать в этой роли. Значение `catConst` занесем в поле категории лишь после того, как константа будет определена полностью.

Листинг 3.29. Трансляция объявления константы

```
(* Имя "=" КонстВыраж *)
procedure ConstDecl;
var
   ConstRef: tObj; {Ссылка на имя в таблице}
begin
   NewName(Name, catGuard, ConstRef);
   NextLex;
   Check(lexEQ, '"="');
   ConstExpr(ConstRef^.Val);
   ConstRef^.Typ := typInt; {Констант других типов нет}
   ConstRef^.Cat := catConst;
end;
```

Для анализа и вычисления константного выражения, которое записывается в определении константы после знака «=», будет служить процедура `ConstExpr`. Ее выходным параметром является числовое значение выражения. В качестве фактического параметра при вызове `ConstExpr` подставим поле значения (поле `Val`) той записи таблицы имен, где хранятся сведения об опреде-

ляемой константе. Эта запись доступна через указатель `ConstRef`.

Напомню, что в языке «О» определены константные выражения лишь специального вида. В константном выражении можно использовать число или имя константы (со знаком или без него). Приведенная в листинге 3.30 процедура `ConstExpr` распознает константное выражение и вычисляет его, присваивая найденное значение своему выходному параметру `V`.

Листинг 3.30. Анализ и вычисление константного выражения

```
(* ["+" | "-"] (Число | Имя) *)
procedure ConstExpr(var V: integer);
var
   X  : tObj;
   Op : tLex;
begin
   Op := lexPlus;
   if Lex in [lexPlus, lexMinus] then begin
      Op := Lex;
      NextLex;
   end;
   if Lex = lexNum then begin
      V := Num;
      NextLex;
      end
   else if Lex = lexName then begin
      Find(Name, X);
      if X^.Cat = catGuard then
         Error('Нельзя определять константу через себя')
      else if X^.Cat <> catConst then
         Expected('имя константы')
      else
         V := X^.Val;
      NextLex;
      end
   else
      Expected( 'константное выражение' );
   if Op = lexMinus then
      V := -V;
end;
```

Трансляция описаний переменных

Задача процедуры `VarDecl` (листинг 3.31), отвечающей за трансляцию одного описания переменных (список переменных, за которым следует тип) довольно проста. Имена переменных должны быть занесены в таблицу со значением атрибута «категория» равным `catVar` и снабжены указанием об их типе. Дело облегчается тем, что в языке «О» существуют лишь переменные типа `INTEGER`, поэтому значение поля `Typ` в записи об имени можно заполнить сразу, даже до распознавания самого типа.

Листинг 3.31. Трансляция описания переменных

```
(* Имя {"," Имя} ":" Тип *)
procedure VarDecl;
var
   NameRef : tObj;
begin
   if Lex <> lexName then
      Expected('имя')
   else begin
      NewName(Name, catVar, NameRef);
      NameRef^.Typ := typInt;
      NextLex;
   end;
   while Lex = lexComma do begin
      NextLex;
      if Lex <> lexName then
         Expected('имя')
      else begin
         NewName(Name, catVar, NameRef);
         NameRef^.Typ := typInt;
         NextLex;
      end;
   end;
   Check(lexColon, '":"');
   ParseType;
end;
```

Распознающая процедура для типа названа `ParseType` (распознать тип), а не `Type`, поскольку **`type`** в языке Паскаль зарезервировано.

Можно обратить внимание, что в записях об именах переменных осталось незаполненным поле значения (поле `Val`). Пока у нас нет для этого необходимой информации. Она появится лишь при рассмотрении генерации машинного кода.

Трансляция объявлений констант и переменных связана лишь с заполнением таблицы имен и не порождает никакого машинного кода.

Контекстный анализ выражений

Выражения — важнейший элемент любого языка программирования. В языке «О» предусмотрены арифметические (типа `INTEGER`) и логические выражения. Последние могут использоваться только в операторах **`IF`** и **`WHILE`**. Синтаксис выражений языка «О» определяется следующими РБНФ-формулами:

```
Выраж = ПростоеВыраж [Отношение ПростоеВыраж].
ПростоеВыраж = ["+"|"-"] Слагаемое {ОперСлож Слагаемое}.
Слагаемое = Множитель {ОперУмн Множитель}.
Множитель = Имя ["(" Выраж | Тип ")"]
   | Число | "(" Выраж ")".
```

В задачу контекстного анализатора при трансляции выражений входит проверка соответствия операций и типов операндов. Поскольку синтаксис выражений иерархичен (выражение → простое выражение → слагаемое → множитель), распознающие процедуры нижнего уровня должны сообщать процедурам верхнего уровня тип соответствующего подвыражения. Каждый распознаватель в иерархии выражений должен определять (скажем даже «вычислять») тип соответствующего подвыражения.

Итак, распознаватели, участвующие в анализе выражений, проверяют соответствие типов операндов и операций и *вычисляют типы* подвыражений. Ни о какой генерации кода и вычислении значений выражений речь пока не идет — «вычисляются» лишь типы выражений.

Каждый распознаватель снабдим выходным параметром (**var** T: tType), обозначающим тип подвыражения. Вначале программируем (листинг 3.32) процедуру Expression — выражение.

Листинг 3.32. Распознаватель выражений

```
(* ПростоеВыраж [Отношение ПростоеВыраж] *)
procedure Expression(var T : tType);
begin
   SimpleExpr(T); {Получить тип первого подвыражения}
   if Lex in [lexEQ, lexNE, lexGT, lexGE, lexLT, lexLE]
   then begin
      if T <> typInt then
         Error('Несоответствие операции типу операнда');
      NextLex;
      SimpleExpr(T); {Правый операнд отношения}
      if T <> typInt then
         Expected('выражение целого типа');
      T := typBool;
   end; {иначе тип равен типу первого простого выражения}
end;
```

Обратите внимание, что контроль типов происходит только в связи с распознаванием операции.

Далее по иерархии следует распознаватель простых выражений (листинг 3.33).

Листинг 3.33. Распознаватель простого выражения

```
(* ["+"|"-"] Слагаемое {ОперСлож Слагаемое} *)
procedure SimpleExpr(var T : tType);
begin
   if Lex in [lexPlus, lexMinus] then begin
      NextLex;
      Term(T);
      if T <> typInt then
         Expected('выражение целого типа');
      end
   else
      Term(T);
   if Lex in [lexPlus, lexMinus] then begin
      if T <> typInt then
         Error('Несоответствие операции типу операнда');
      repeat
         NextLex;
         Term(T);
```

```
            if T <> typInt then
               Expected('выражение целого типа');
         until not( Lex in [lexPlus, lexMinus] );
      end;
   end;
```

Как можно видеть, необходимость выполнения контекстных проверок несколько изменила реализацию распознавателя. Так, вместо цикла

```
   while Lex in [lexPlus, lexMinus] do begin
      NextLex;
      Term
   end;
```

который в синтаксическом анализаторе выполнял бы обработку второго и последующих слагаемых, использована конструкция из **if** и **repeat**, позволяющая вовремя проверить тип первого слагаемого.

За исключением случая, когда простое выражение состоит из одного слагаемого (терма), его тип будет целым. Если слагаемое одно, тип простого выражения совпадает с типом этого слагаемого.

Аналогично строится распознаватель слагаемого (листинг 3.34).

Листинг 3.34. Распознаватель слагаемого

```
   (* Множитель {ОперУмн Множитель} *)
   procedure Term(var T: tType);
   begin
      Factor(T);
      if Lex in [lexMult, lexDIV, lexMOD] then begin
         if T <> typInt then
            Error('Несоответствие операции типу операнда');
         repeat
            NextLex;
            Factor(T);
            if T <> typInt then
               Expected('выражение целого типа');
         until not( Lex in [lexMult, lexDIV, lexMOD] );
      end;
   end;
```

Множитель представляет собой первичное выражение, элементарный операнд. В этой роли могут выступать переменная, именованная константа, вызов процедуры-функции, число, наконец, выражение в скобках. Распознаватель множителя (листинг 3.35) серией последовательных проверок отделяет эти варианты. Поскольку первые три вида множителя (переменная, константа, функция) начинаются с имени, для их разделения используется обращение к таблице имен. В зависимости от категории имени обрабатывается тот или иной вариант.

Листинг 3.35. Распознаватель множителя

```
(* Имя ["(" Выраж | Тип ")"] | Число | "(" Выраж ")" *)
procedure Factor(var T: tType);
var
   X : tObj;
begin
   if Lex = lexName then begin
      Find(Name, X);
      if X^.Cat = catVar then begin
         {Переменная}
            T := X^.Typ;
            NextLex;
         end
      else if X^.Cat = catConst then begin
         {Константа}
            T := X^.Typ;
            NextLex;
         end
      else if (X^.Cat=catStProc) and (X^.Typ<>typNone)
      then begin
         {Процедура-функция}
            NextLex;
            Check(lexLPar, '"("');
            StFunc(X^.Val, T);
            Check(lexRPar, '")"');
         end
      else
         Expected(
         'переменная, константа или процедура-функция'
         );
      end
   else if Lex = lexNum then begin
      {Число}
         T := typInt;
```

```
            NextLex
         end
      else if Lex = lexLPar then begin
         {Выражение в скобках}
            NextLex;
            Expression(T);
            Check(lexRPar, '")"');
         end
      else
         Expected('имя, число или "("');
   end;
```

Обработку списка фактических параметров стандартной функции выполнит процедура `StFunc`. Она же вычислит тип множителя, представляющего собой вызов процедуры-функции. Заголовок этой `StFunc` будет таким:

```
procedure StFunc(F: integer; var T: tType);
```

Здесь `F` — номер функции (`stABS`, `stMAX`, ...); `T` — выходной параметр — тип функции. Запрограммирована эта процедура будет позже.

Контекстный анализ операторов

В языке «О» имеется четыре вида операторов: присваивание, вызов процедуры, оператор **`if`** и оператор **`while`**. Как следует из синтаксической формулы (листинг 3.36), оператор также может быть пустым.

Листинг 3.36. Синтаксис оператора языка «О»

```
Оператор = [
   Переменная ":=" Выраж
   |[Имя "."] Имя ["(" [Параметр {"," Параметр}] ")"]
   |IF Выраж THEN
      ПослОператоров
   {ELSIF Выраж THEN
      ПослОператоров}
   [ELSE
      ПослОператоров]
    END
   |WHILE Выраж DO
      ПослОператоров
    END
].
```

Синтаксис всех видов операторов определяется единым РБНФ-правилом. Если буквально следовать технологии рекурсивного спуска, нужно записать одну распознающую процедуру, транслирующую все операторы. Такая процедура будет достаточно громоздкой. Предусмотрим свои распознающие процедуры для каждого из видов операторов, в то время как на распознаватель `Statement` (оператор) будет лишь возложена обязанность, определить, с каким из четырех случаев он имеет дело и вызвать соответствующую распознающую процедуру.

Первые два варианта (присваивание и вызов процедуры) не могут быть различены из анализа одной текущей лексемы: и тот и другой оператор начинается с имени. Распознавание может быть выполнено с привлечением контекстной информации. Если имя принадлежит переменной, то далее следует ожидать присваивание, иначе можно предположить обозначение процедуры — стандартной или из импортированного стандартного модуля.

Итак, если первая лексема оператора — имя, выполняем поиск в таблице имен с целью определения его категории:

```
if Lex = lexName then begin
   Find(Name, X);
```

Здесь `X` — переменная типа `tObj`, а `Name` — глобальная переменная, экспортированная сканером.

Далее необходимо учесть, что обозначение процедуры может быть составным — состоять из имени модуля и имени процедуры, разделенных точкой. Поэтому первое встретившееся имя может оказаться именем модуля. В общем случае составное обозначение может быть и у переменной, константы, типа, если они импортированы из другого модуля. Однако стандартные модули `In` и `Out` языка «О» экспортируют только процедуры.

Если найденное в таблице имя принадлежит модулю, то проверяется наличие точки, имени импортируемого объекта и выпол-

няется новый поиск. При этом в качестве ключа процедуре `Find` передается (после проверки длины) составное обозначение, состоящее из имени модуля, точки и имени искомого объекта, например, '`In.Int`'. Именно таким образом обозначения процедур из стандартных модулей были занесены в таблицу имен при ее инициализации.

```
if X^.Cat = catModule then begin
   NextLex;
   Check(lexDot, '"."');
   if    (Lex = lexName) and
      (Length(Name) + Length(X^.Name) < NameLen)
   then
      Find(X^.Name + '.' + Name, X)
   else
      Expected('имя из модуля ' + X^.Name);
end;
```

Результат поиска составного имени (ссылка на найденный в таблице имен объект) вновь помещается в переменную `X`. Если же имя модуля не встретилось, то последующей части программы будет передана прежняя ссылка `X`, а эта последующая часть даже «не узнает», было обозначение составным или нет.

Дальнейший анализ не составляет труда. Окончательный текст распознавателя операторов приведен в листинге 3.37.

Листинг 3.37. Распознавание операторов

```
procedure Statement;
var
   X : tObj;
begin
   if Lex = lexName then begin
      Find(Name, X);
      if X^.Cat = catModule then begin
         NextLex;
         Check(lexDot, '"."');
         if (Lex = lexName) and
            (Length(X^.Name) + Length(Name) < NameLen)
         then
            Find(X^.Name + '.' + Name, X)
         else
            Expected('имя из модуля ' + X^.Name);
```

```
        end;
        if X^.Cat = catVar then
           AssStatement          {Присваивание}
        else if (X^.Cat = catStProc) and (X^.Typ = typNone)
        then
           CallStatement(X^.Val) {Вызов процедуры}
        else
           Expected(
              'обозначение переменной или процедуры'
           );
        end
     else if Lex = lexIF then
        IfStatement
     else if Lex = lexWHILE then
        WhileStatement
  end;
```

Полагаю, принцип контекстных проверок, выполняемых с помощью таблицы имен, понятен. Рассмотренные выше трансляция описаний, анализ выражений и распознавание видов операторов дают достаточно примеров. Мы не будем сейчас подробно обсуждать контекстный анализ конкретных операторов языка «О». В дальнейшем эти части программы все равно пришлось бы переписывать еще раз для внедрения действий по генерации кода. Чтобы не загромождать изложение, рассмотрим контекстный анализ отдельных операторов позже, вместе с генерацией кода для них.

Генерация кода

Интересное дело: существенная часть компилятора уже написана, а еще не было сказано ни слова о процессоре, в код которого должна транслироваться программа на языке «О». И это должно радовать, поскольку свидетельствует об универсальности использованных алгоритмов и хорошем проектировании программы, отдельные части которой максимально независимы.

Виртуальная машина

Мы не будем здесь рассматривать генерацию кода для какого-либо реального процессора или семейства процессоров. Во-

первых, потому, что различных процессоров существует множество, и нам пришлось бы выбирать какой-то из них. Во-вторых, генерация кода для реального процессора была бы сопряжена с необходимостью учета множества технических деталей, не так уж важных в принципиальном плане. Немало места пришлось бы уделить рассмотрению самой системы команд процессора.

Поступим по-другому. Сконструируем собственный процессор, обладающий простой и удобной системой команд. Используя его, мы сможем рассмотреть основные принципы генерации кода, не отвлекаясь на частности. Речь, конечно, не идет об изготовлении прибора в металле и кремнии. Вместо этого используем программу, имитирующую работу процессора.

В выбранном подходе можно увидеть еще ряд достоинств. Программа, оттранслированная в код не существующего реально, а моделируемого программно (виртуального) процессора сможет быть выполнена в любой системе, где будет способен работать имитатор-интерпретатор. А поскольку написан он будет на Паскале[30], это означает, что программы на языке «О» можно будет выполнить везде, где есть подходящий компилятор языка Паскаль. Наконец, выбранный нами подход похож на технологии, использованные при реализации языков Ява и Си#, а еще раньше — Паскаля. Это позволяет познакомиться с принципами, лежащими в основе этих технологий.

Несуществующий абстрактный компьютер, работа которого реализуется на реальной машине с помощью программных средств, называют *виртуальной машиной*. Примером является Java Virtual Machine (JVM) — виртуальная Ява-машина, представляющая собой модель стекового процессора, в код которого транслируются программы на языке Ява. Дадим название и на-

[30] В приложениях вы также можете найти текст компилятора-интерпретатора, записанный на других языках программирования.

шему компьютеру. Называться он будет ОВМ — виртуальная О-машина.

Исторически использование программного моделирования гипотетического компьютера связано с одной из первых реализаций языка Паскаль в начале 1970-х годов. Виртуальный код был назван тогда П-кодом. Позднее такая же техника использовалась при реализации Visual Basic. В системе Microsoft .NET код виртуальной машины носит название «промежуточный язык» (intermediate language, сокращенно — IL).

Архитектура виртуальной машины

Как и любой компьютер, ОВМ (рис. 3.6) будет содержать процессор и память. Процессор способен выполнять определенный набор команд, а также содержит ряд регистров — специальных ячеек памяти, используемых командами. В памяти хранятся программа и данные.

Память

Поскольку в языке «О» используются лишь данные целого типа, память ОВМ будет состоять из некоторого количества слов-ячеек, каждая из которых может хранить одно целое число. Каждое слово имеет уникальный адрес, который выражается целым числом. В программе, моделирующей работу виртуальной машины, это будет выглядеть следующим образом:

```
const
   MemSize = 8*1024;

var
   M: array [0..MemSize-1] of integer;
```

Константа `MemSize` определяет размер памяти в словах. Для примера ее значение взято равным 8К слов. Такой памяти вполне достаточно для размещения небольших демонстрационных программ, написанных на языке «О», а также их данных. Поскольку

массивы в языке не предусмотрены, объем данных не может быть слишком велик.

Массив `m` (от memory — память) — это память виртуальной машины. Номера (адреса) его элементов начинаются с нуля.

Разрядность слова виртуальной машины зависит от разрядности типа `integer` в том компиляторе Паскаля, с помощью которого будет транслироваться интерпретатор виртуальной машины. Собственно, реальных вариантов два: при 16-разрядном представлении (Turbo Pascal, Free Pascal) ОВМ будет 16-разрядной; если используется 32-разрядный тип `integer` (Delphi и др.), ОВМ тоже будет 32-разрядной.

Разрядность слова определяет не только диапазон целых чисел, которые можно обрабатывать, но и размер адресного пространства машины. Поскольку адреса ячеек памяти сами будут храниться в таких же ячейках памяти, максимально возможное значение адреса 16-разрядной машины составит 32 767; 32-разрядной — 2 147 483 647. Таким образом, максимальный объем памяти 16-разрядной ОВМ мог бы составить 32К слов (64 Кбайт); 32-разрядной — 2 гигаслова (8Гбайт). Реально, ни то, ни другое недостижимо: 16-разрядная реализация Паскаля не позволит создать массив объемом 64К, реальный 32-разрядный компьютер, для которого создают код 32-разрядные компиляторы, не может иметь больше 4Гбайт памяти.

Процессор

Одной из форм представления программы при трансляции является обратная польская запись (ПОЛИЗ). Преобразование программы в ПОЛИЗ (генерация кода), и ее последующее исполнение (интерпретация) выполняются простым и естественным способом с использованием стека. В связи с этим, в роли виртуального процессора используем стековую машину с безадресной

(нуль-адресной[31]) системой команд. Программа для такой машины представляет собой последовательность операндов и операций, соответствующих обратной польской записи программы. Встретившиеся в программе операнды заносятся в стек, операции выполняются над верхними элементами стека, результат операции заменяет собой операнды на вершине стека.

Каждый операнд (константа, адрес) и код каждой операции в программе для ОВМ будет занимать одно слово памяти. Чтобы различать операнды и операции, предусмотрим кодирование операций отрицательными целыми числами. Тогда для операндов остаются зарезервированы неотрицательные целые значения. Получается, что отрицательный операнд не может быть непосредственно указан в программе. Это, однако, не должно создать серьезных проблем, поскольку отрицательные числа встречаются в реальных программах гораздо реже неотрицательных. К тому же, числовые литералы в языке «О» рассматриваются как числа без знака. Поэтому, например, в выражении `x-1` операндами являются `x` и 1, а минус — это знак двуместной операции. Чтобы было возможно выполнение действий, подобных `x := -1`, в системе команд ОВМ будет предусмотрена операция «перемена знака» — унарный минус.

Запрет отрицательных констант в машинном коде ОВМ не означает отказ от отрицательных чисел вообще, то есть запрет на получение в стеке отрицательных величин в ходе вычислений и хранение отрицательных значений в ячейках данных, отведенных для переменных.

---

[31] В том смысле, что сама команда не содержит адресов своих операндов. Операнды всегда в стеке. В противоположность этому возможны одноадресные, двухадресные, трехадресные и даже четырехадресные команды и системы команд, а также системы команд с переменной адресностью.

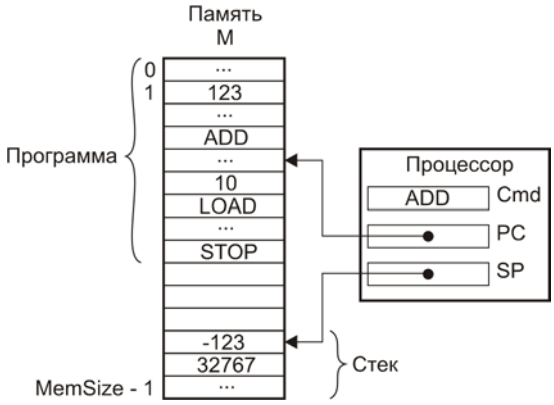


Рис. 3.6. Архитектура ОВМ

Программный счетчик

Команды программы располагаются в памяти ОВМ (массиве `M`). Процессор выполняет команды одну за другой. При отсутствии переходов команды выполняются в порядке их расположения в памяти. Каждая команда — это либо операнд, либо операция. Условимся, что первой выполняется команда, записанная в ячейке с адресом 0.

Адрес очередной команды, подлежащей исполнению, записан в регистре процессора, обозначаемом `PC` (от program counter — программный счетчик). Начальное значение `PC` в соответствии с только что принятым соглашением равно 0.

Стек и указатель стека

Стек, используемый ОВМ при вычислениях, будет располагаться в старших адресах памяти. На текущую вершину стека указывает регистр процессора `SP` (stack pointer — указатель стека). Значение `SP` в каждый момент времени равно адресу ячейки, являющейся вершиной стека. Значение `SP` уменьшается при добавлении элементов в стек, и увеличивается при их извлечении — стек растет в сторону меньших адресов. Перед выполнением программы стек пуст, а значение `SP` равно `MemSize`.

Система команд

Как уже говорилось, программа для ОВМ представляет собой записанную в память (массив `M`) последовательность операндов и операций. Каждый операнд и каждая операция занимают одно слово. При выполнении программы операнды заносятся в стек, операции выполняются над данными с вершины стека. Операнды можно рассматривать как операции (с кодами от 0 до `maxint`). Действие такой операции состоит в том, что в стек заносится ее код. Команды перед выполнением считываются в регистр команд процессора, обозначаемый `Cmd`.

Команды ОВМ перечислены в таблице 3.3. Для пояснения их действия в графе «Стек» приводятся состояния стека до и после выполнения команды. Вершина стека считается расположенной справа. Например, запись `x, y` → `x+y` означает, что до выполнения команды (`ADD`) на стеке[32] располагались числа `x` и `y`, причем, `y` — на вершине стека, а `x` — под вершиной, а после выполнения — их сумма. Естественно, что кроме `x` и `y` в глубине стека перед выполнением команды `ADD` могли размещаться и другие данные, но поскольку они не участвуют в ее выполнении, то и не упоминаются.

Такой способ обозначений заимствован из описаний языка Форт, да и вообще, система команд ОВМ устроена по тому же принципу, что и Форт.

[32] Выражение «на стеке» — программистский жаргон. По правилам русского языка следовало, пожалуй, сказать «в стеке». Но было бы не очевидно, что речь идет о верхних элементах. «На стеке» — сокращенный вариант оборота «на вершине стека».

Таблица 3.3. Система команд ОВМ

Код	Обозначение	Название	Стек	Действие
c>=0	Нет	Константа	→ c	
-1	STOP	Останов	Не меняется	
Арифметические операции				
-2	ADD	Сложение	x, y → x+y	
-3	SUB	Вычитание	x, y → x-y	
-4	MUL	Умножение	x, y → x*y	
-5	DIV	Деление	x, y → x **DIV** y	
-6	MOD	Остаток	x, y → x **MOD** y	
-7	NEG	Изменение знака	x → -x	
Операции с памятью				
-8	LOAD	Загрузка	A → M[A]	
-9	SAVE	Сохранение	A, x →	M[A] := x
Операции со стеком				
-10	DUP	Дублирование	x → x, x	
-11	DROP	Сброс	x →	
-12	SWAP	Обмен	x, y → y, x	
-13	OVER	Наверх	x, y → x, y, x	

Код	Обозначение	Название	Стек	Действие
Команды перехода				
-14	GOTO	Безусловный переход	A →	PC := A
-15	IFEQ	Переход, если равно	x, y, A →	**if** x=y **then** PC := A
-16	IFNE	Переход, если не равно	x, y, A →	**if** x<>y **then** PC := A
-17	IFLE	Переход, если меньше или равно	x, y, A →	**if** x<=y **then** PC := A
-18	IFLT	Переход, если меньше	x, y, A →	**if** x<y **then** PC := A
-19	IFGE	Переход, если больше или равно	x, y, A →	**if** x>=y **then** PC := A
-20	IFGT	Переход, если больше	x, y, A →	**if** x>y **then** PC := A
Операции ввода и вывода				
-21	IN	Ввод	→ введенное число	SP := SP-1; Write('?'); Readln(M[SP])
-22	OUT	Вывод	x, w →	Write(x: w)
-23	OUTLN	Перевод строки	Не меняется	WriteLn

Краткие пояснения по поводу некоторых команд. Как уже говорилось, все арифметические команды берут свои операнды с вершины стека. Двуместные операции (ADD, SUB, MUL, DIV, MOD) заменяют два операнда, взятые с вершины стека, результатом операции. При этом число элементов в стеке уменьшается на единицу. Операция NEG меняет знак элемента, находящегося на вершине стека.

Команда LOAD загружает на вершину стека значение, хранящееся в памяти по указанному адресу A. Её действие сводится к замене на стеке адреса A значением M[A]. Такое действие называется «разыменованием». Команда SAVE сохраняет взятое со стека значение в ячейке памяти с указанным адресом. Перед выполнением SAVE на вершине стека должно быть сохраняемое значение, под вершиной — адрес.

Команды, оперирующие элементами на вершине стека, имеют тот же смысл и обозначения, что и в языке Форт. DUP дублирует элемент, находящийся на вершине стека, DROP уничтожает верхний элемент, SWAP обменивает два верхних элемента стека, OVER дублирует на вершине стека элемент, находившийся под вершиной.

Команда GOTO выполняет переход по адресу, находящемуся на вершине стека. Реализация перехода сводится к присваиванию значения с вершины стека программному счетчику PC. После этого следующей выполняемой командой будет команда, находящаяся в памяти по адресу, занесенному в PC.

Команды условных переходов требуют трех операндов на стеке. Сравниваемые значения x и y находятся под вершиной стека, адрес перехода A — на вершине. Происходит переход по адресу A, если выполняется заданное отношение для x и y, в противном случае выполняется следующая команда.

Команды ввода-вывода соответствуют имеющимся в языке «O» возможностям. Команда `IN` печатает знак «?», запрашивает вводимое число и заносит его на вершину стека. `OUT` выводит целое значение `x`, находящееся под вершиной стека, используя `w` позиций. `OUTLN` выполняет перевод строки.

Вполне можно представить себе аппаратный компьютер, имеющий такую систему команд. Разве что ввод и вывод организованы в ОВМ нетрадиционно. Конкретные операции ввода и вывода, как правило, не входят в систему команд реальных процессоров. Связь с внешними устройствами осуществляется через специальные порты ввода-вывода, либо для обмена с внешними устройствами резервируется определенная часть адресного пространства. В последнем случае обмен сводится к записи и считыванию данных по определенным адресам памяти. И уж совсем нетипично, что в набор команд ОВМ входят такие действия как печать целого и перевод строки. В реальных системах речь могла бы идти о выводе отдельного символа, а вывод числа и перевод строки были бы реализованы программно.

Еще одна особенность, отличающая систему команд ОВМ от команд реальных процессоров, состоит в том, что для хранения 24 различных кодов операций используется целое машинное слово длиной 16 или даже 32 бита, в то время как было бы достаточно всего 5 бит.

Программирование в коде виртуальной машины

Чтобы освоить систему команд ОВМ и понять принципы программирования для этой машины, рассмотрим примеры. Возьмем задачу нахождения наибольшего общего делителя (НОД) двух натуральных чисел и запрограммируем ее вначале на языке «O», используя для решения алгоритм Евклида: пока числа не сравняются, уменьшать большее из них на величину меньшего (листинг 3.38).

Листинг 3.38. Нахождение НОД по алгоритму Евклида

```
(* Наибольший общий делитель *)
MODULE Euqlid;
IMPORT In, Out;
VAR
   X, Y : INTEGER;
BEGIN
   In.Open;
   In.Int(X);
   In.Int(Y);
   WHILE X # Y DO
      IF X > Y THEN
         X := X - Y
      ELSE
         Y := Y - X
      END;
   END;
   Out.Int(X, 0);
   Out.Ln;
END Euqlid.
```

Программирует компилятор

Представим, какой машинный код должен создать компилятор для такой программы на языке «О». Запишем этот код, имея в виду следующее. Компилятор назначает каждой переменной свою ячейку памяти. Для получения значения переменной генерируются команды, загружающие это значение из памяти. Наш компилятор не выполняет никакой оптимизации, программируя «в лоб».

Вообще-то, если речь идет о машинном коде, следовало бы записать программу как последовательность чисел. Но читать ее в таком виде трудно. Используем мнемонические коды команд вместо числовых. Применим и некоторые другие обозначения. В языках машинного уровня (ассемблерах) часто используют точку с запятой для обозначения комментариев. Часть строки, следующая за точкой с запятой — комментарий. Строки программы на «О», породившие соответствующий машинный код будем записывать в форме комментария. В той же строке, что и команда, будем отражать состояние стека после выполнения этой коман-

ды (вершина стека справа). Адреса команд отделяются круглой скобкой.

Распределим память под переменные X и Y. Поскольку код программы наверняка получится короче 100 команд, будем хранить значение X в ячейке 100, а значение Y — в ячейке 101. Константы 100 и 101 в листинге 3.39 означают адрес X и адрес Y соответственно.

Листинг 3.39. Машинный код программы Euqlid

```
; Наибольший общий делитель
; MODULE Euqlid;
; IMPORT In, Out;
; VAR
; X, Y : INTEGER;
; BEGIN
; In.Int(X);

   0) 100     ; 100
   1) IN      ; 100, X
   2) SAVE

; In.Int(Y);

   3) 101     ; 101
   4) IN      ; 101, Y
   5) SAVE

; WHILE X # Y DO

   6)  100    ; 100
   7)  LOAD   ; X
   8)  101    ; X, 101
   9)  LOAD   ; X, Y
   10) 36     ; X, Y, 36
   11) IFEQ

; IF X > Y THEN

   12) 100    ; 100
   13) LOAD   ; X
   14) 101    ; 101
   15) LOAD   ; Y
   16) 27     ; X, Y, 27
   17) IFLE
```

```
; X := X – Y

   18) 100   ; 100
   19) 100   ; 100, 100
   20) LOAD  ; 100, X
   21) 101   ; 100, X, 101
   22) LOAD  ; 100, X, Y
   23) SUB   ; 100, X-Y
   24) SAVE

; ELSE

   25) 34
   26) GOTO

; Y := Y – X

   27) 101   ; 101
   28) 101   ; 101, 101
   29) LOAD  ; 101, Y
   30) 100   ; 101, Y, 100
   31) LOAD  ; 101, Y, X
   32) SUB   ; 101, Y-X
   33) SAVE

; END;
; END;

   34) 6
   35) GOTO

; Out.Int(X, 0);

   36) 100   ; 100
   37) LOAD  ; X
   38) 0     ; X, 0
   39) OUT

; Out.Ln;

   40) OUTLN

; END Euqlid.

   41) STOP
```

Программа заняла 42 машинных слова. Теперь, зная размер кода, можно было бы изменить адреса X и Y, предусмотрев размещение этих переменных сразу за кодом программы. Переменной X можно назначить адрес 42, переменной Y — 43. Компилятор, по-видимому, должен будет действовать аналогично, ведь он не может выдвигать гипотезы о будущем размере кода и назначать адреса еще до начала компиляции «с запасом», как поступили мы. Переписывать программу, заменяя константы 100 и 101 на 42 и 43, не будем. Но будем иметь в виду, что при дальнейшей разработке компилятора задачу назначения адресов придется решать.

Программируем вручную

Программа, представленная в листинге 3.39, далеко не оптимальна. Оперируя всего двумя величинами X и Y, она постоянно занята загрузкой и сохранением их значений. Но зачем сохранять значения в памяти, если они тут же потребуются в следующем цикле? Можно в ходе выполнения удерживать X и Y на стеке. Действуя по такому принципу, напишем новый вариант программы (листинг 3.40).

Листинг 3.40. Нахождение НОД(X, Y). Программирование вручную

```
0)  IN    ; X
1)  IN    ; X, Y
2)  OVER  ; X, Y, X
3)  OVER  ; X, Y, X, Y
4)  15    ; X, Y, X, Y, 15
5)  IFEQ  ; X, Y    На выход, если X=Y
6)  OVER  ; X, Y, X
7)  OVER  ; X, Y, X, Y
8)  11    ; X, Y, X, Y, 11
9)  IFLT  ; X, Y    В обход SWAP, если X>Y
10) SWAP  ; Y, X    На вершине большее
11) OVER  ; Min(X, Y), Max(X, Y), Min(X, Y)
12) SUB   ; Новое X, Новое Y
13) 2     ; X, Y, 2
14) GOTO  ; X, Y    На начало цикла
15) DROP  ; X       Одно значение было лишним
16) 0     ; X, 0
```

```
17) OUT
18) OUTLN
19) STOP
```

Достаточно трудно представить, что компилятор может породить такой код. Приведенная программа основана на неочевидных манипуляциях с вершиной стека. Такая манера характерна для ручного программирования на языке Форт. Сравнение листингов 3.39 и 3.40 наглядно демонстрирует причины меньшей эффективности программ, полученных трансляцией с языка высокого уровня[33] в сравнении с написанными вручную. Код, полученный вручную, оказался вдвое короче, и работать будет быстрее, поскольку в цикле написанной вручную программы выполняется 12 или 13 команд, а в откомпилированной программе — 21 или 23.

Для нахождения наибольшего общего делителя (НОД) двух натуральных чисел может быть написана еще более компактная программа[34] (листинг 3.41). Вместо вычитаний она использует вычисление остатка от деления.

Листинг 3.41. Нахождение НОД с вычислением остатка

```
0)  IN    ; X
1)  IN    ; X, Y
2)  SWAP  ; Y, X
3)  OVER  ; Y, X, Y
4)  MOD   ; Y, X mod Y
5)  DUP   ; Y, X mod Y, X mod Y
6)  0     ; Y, X mod Y, X mod Y, 0
7)  2
8)  IFNE
9)  DROP
10) 0
11) OUT
12) OUTLN
13) STOP
```

[33] С помощью простого неоптимизирующего компилятора

[34] Автор Ф. Меньшиков.

Реализация виртуальной машины

Спроектировав архитектуру виртуальной машины и даже поупражнявшись в программировании ОВМ, займемся её воплощением. Роль виртуальной машины будет исполнять ее программная модель — интерпретатор. Ресурсы, предоставляемые виртуальной машиной, будут сосредоточены в программном модуле `OVM`. В его интерфейсной секции (листинг 3.42) определяются константа `MemSize`, задающая размер памяти, константы, обозначающие коды операций, и массив `M` — память виртуальной машины.

Листинг 3.42. Интерфейс модуля виртуальной машины

```
unit OVM;
{Виртуальная машина}

interface

const
   MemSize = 8*1024;

   cmStop   = -1;

   cmAdd    = -2;
   cmSub    = -3;
   cmMult   = -4;
   cmDiv    = -5;
   cmMod    = -6;
   cmNeg    = -7;

   cmLoad   = -8;
   cmSave   = -9;

   cmDup    = -10;
   cmDrop   = -11;
   cmSwap   = -12;
   cmOver   = -13;

   cmGOTO   = -14;
   cmIfEQ   = -15;
   cmIfNE   = -16;
   cmIfLE   = -17;
   cmIfLT   = -18;
   cmIfGE   = -19;
```

```
    cmIfGT   = -20;

    cmIn     = -21;
    cmOut    = -22;
    cmOutLn  = -23;

var
    M: array [0..MemSize-1] of integer;

procedure Run;
```

Процедура `Run` выполняет программу, записанную в память виртуальной машины, начиная с команды, находящейся в `M[0]`. Эта процедура реализует работу процессора ОВМ.

Использование модуля `OVM` предполагается строить по следующей схеме. Генератор кода записывает команды программы прямо в память виртуальной машины (массив `M`). Массив памяти и коды операций определены в интерфейсе модуля OVM, чтобы быть доступными другим модулям компилятора. После того как код сгенерирован и записан в память ОВМ, программа может быть выполнена вызовом процедуры `Run`. Поместим этот вызов в главную программу нашего компилятора (листинг 3.43).

Листинг 3.43. Главная программа транслятора языка «О»

```
program O;
{Компилятор языка О}

uses
    OText, OScan, OPars, OVM, OGen;

procedure Init;
begin
    ResetText;
    InitScan;
    InitGen;
end;

procedure Done;
begin
    CloseText;
end;
```

```
begin
   WriteLn('Компилятор языка O');
   Init;    {Инициализация}
   Compile; {Компиляция}
   Run;     {Выполнение}
   Done;    {Завершение}
end.
```

После такого решения транслятор приобрел черты интерпретатора, поскольку отвечает теперь и за выполнение программы. Если иметь в виду учебный характер проекта, такое сочетание компиляции и интерпретации безусловно полезно — позволяет понять реализацию как одного, так и другого. В реальных системах интерпретаторы обычно строятся по схожей схеме — интерпретируется не исходная программа, а ее промежуточное представление. Это позволяет достичь большей эффективности.

Приведенный в листинге 3.43 текст главной программы транслятора языка «О» является окончательным. Обратите внимание, что в предложении **uses** упомянут модуль генератора кода `OGen`, а в процедуре `Init` вызывается инициализация генератора кода (`InitGen`).

Реализация процедуры `Run` достаточно проста (листинг 3.44). Она строится по спецификации ОВМ. Регистры виртуальной машины `PC` и `SP` и `Cmd` превращаются в локальные переменные процедуры `Run`. Кроме этого используется локальная переменная `Buf`, необходимая для реализации команды `SWAP`. В начале работы регистры процессора приводятся в исходное состояние: `SP = MemSize` (стек пуст); `PC = 0` (выполнение начинается с команды, расположенной по адресу 0).

Обратите внимание, что сразу после считывания очередной команды в регистр команд `Cmd` и еще до начала выполнения команды программный счетчик `PC` увеличивается на единицу. Это означает, что при выполнении данной команды `PC` указывает на следующую по порядку команду.

Напомню, что стек ОВМ растет в сторону меньших адресов, поэтому при добавлении элементов в стек регистр SP уменьшается, при удалении — увеличивается. M[SP] означает текущую вершину стека, M[SP+1] — элемент, находящийся под вершиной, M[SP+2] — второй от вершины элемент в глубине стека. При реализации двуместных арифметических операций, уменьшающих число элементов в стеке на единицу, вначале изменяется указатель стека, после чего M[SP] — это «новая» вершина стека, а M[SP–1] — «старая».

Листинг 3.44. Процессор виртуальной машины

```
procedure Run;
var
   PC    : integer;
   SP    : integer;
   Cmd   : integer;
   Buf   : integer;
begin
   PC := 0;
   SP := MemSize;
   Cmd := M[PC];
   while Cmd <> cmStop do begin
      PC := PC + 1;
      if Cmd >= 0 then begin
         SP := SP - 1;
         M[SP] := Cmd;
         end
      else
         case Cmd of
         cmAdd:
            begin
               SP := SP + 1;
               M[SP] := M[SP] + M[SP-1];
            end;
         cmSub:
            begin
               SP := SP + 1;
               M[SP] := M[SP] - M[SP-1];
            end;
         cmMult:
            begin
               SP := SP + 1;
               M[SP] := M[SP]*M[SP-1];
            end;
```

```
cmDiv:
   begin
      SP := SP + 1;
      M[SP] := M[SP] div M[SP-1];
   end;
cmMod:
   begin
      SP := SP + 1;
      M[SP] := M[SP] mod M[SP-1];
   end;
cmNeg:
   M[SP] := -M[SP];
cmLoad:
   M[SP] := M[M[SP]];
cmSave:
   begin
      M[M[SP+1]] := M[SP];
      SP := SP + 2;
   end;
cmDup:
   begin
      SP := SP - 1;
      M[SP] := M[SP+1];
   end;
cmDrop:
   SP := SP + 1;
cmSwap:
   begin
      Buf := M[SP];
      M[SP] := M[SP+1];
      M[SP+1] := Buf;
   end;
cmOver:
   begin
      SP := SP - 1;
      M[SP] := M[SP+2];
   end;
cmGOTO:
   begin
      PC := M[SP];
      SP := SP + 1;
   end;
cmIfEQ:
   begin
      if M[SP+2] = M[SP+1] then
         PC := M[SP];
      SP := SP + 3;
   end;
cmIfNE:
```

```
            begin
               if M[SP+2] <> M[SP+1] then
                  PC := M[SP];
               SP := SP + 3;
            end;
         cmIfLE:
            begin
               if M[SP+2] <= M[SP+1] then
                  PC := M[SP];
               SP := SP + 3;
            end;
         cmIfLT:
            begin
               if M[SP+2] < M[SP+1] then
                  PC := M[SP];
               SP := SP + 3;
            end;
         cmIfGE:
            begin
               if M[SP+2] >= M[SP+1] then
                  PC := M[SP];
               SP := SP + 3;
            end;
         cmIfGT:
            begin
               if M[SP+2] > M[SP+1] then
                  PC := M[SP];
               SP := SP + 3;
            end;
         cmIn:
            begin
               SP := SP - 1;
               Write('?');
               Readln( M[SP] );
            end;
         cmOut:
            begin
               Write(M[SP+1]:M[SP]);
               SP := SP + 2;
            end;
         cmOutLn:
            WriteLn;
         else begin
            WriteLn('Недопустимый код операции');
            M[PC] := cmStop;
         end;
         end;
      Cmd := M[PC];
   end;
```

```
    WriteLn;
    if SP < MemSize then
       WriteLn('Код возврата ', M[SP]);
    Write('Нажмите ВВОД');
    ReadLn;
end;
```

Если после выполнения команды `STOP` стек не будет пустым, то находящееся на его вершине число воспринимается как код возврата, переданный программой в «окружающую среду» и свидетельствующий о характере завершения программы. Значение 0 обычно интерпретируется как нормальное завершение, а отличные от нуля коды возврата соответствуют разным вариантам аварийного завершения.

Генератор кода

Было бы неправильно разрешать анализатору (модуль `OPars`) оперировать непосредственно с памятью виртуальной машины, записывать туда команды формируемой программы. Надо, насколько возможно, разделить анализирующую и генерирующую части компилятора. Это улучшит его структуру и придаст гибкость. Порождение машинных команд с помощью небольшого набора процедур, сосредоточенных в отдельном модуле, позволит лучше контролировать процесс генерации и при необходимости вносить в него изменения.

За непосредственную генерацию кода будет отвечать модуль `OGen`. Только процедурам этого модуля будет разрешено обращаться к памяти виртуальной машины для записи туда кодов машинных команд. Но решения о том, какие именно команды должны быть порождены, будут приниматься по ходу анализа входной программы, и вызовы процедур генерации будут размещены в недрах распознавателя.

В листинге 3.45 можно видеть первую версию модуля генератора кода. Основной его процедурой является `Gen`. Она записывает в очередную свободную ячейку памяти (массив `M`) команду, код

которой передан при вызове (параметр Cmd). Адрес свободной ячейки хранится в глобальной и видимой из других модулей переменной PC. Это программный счетчик периода генерации. После записи в память очередной команды его значение увеличивается на единицу и снова начинает указывать на свободную ячейку. Перед началом генерации кода и всего процесса компиляции память виртуальной машины пуста. Начальное значение PC, равное нулю, устанавливается процедурой InitGen, которая вызывается из главной программы компилятора.

Листинг 3.45. Предварительная версия генератора кода

```
unit OGen;
{Генератор кода}

interface

var
   PC : integer;

procedure InitGen;
procedure Gen(Cmd: integer);

{=====================================================}

implementation

uses
   OVM;

procedure InitGen;
begin
   PC := 0;
end;

procedure Gen(Cmd: integer);
begin
   M[PC] := Cmd;
   PC := PC+1;
end;

end.
```

Не следует путать переменную PC, определенную в модуле OGen и программный счетчик виртуальной машины. Последний — это

локальная переменная виртуальной машины и недоступна извне. Программный счетчик периода генерации будет использоваться в модуле распознавателя (но только для того, чтоб узнать его текущее значение).

Распределение памяти

Каждой переменной программы на языке «О» должна быть назначена ячейка памяти, где будет храниться значение этой переменной. Можно представить себе несколько вариантов размещения переменных в памяти ОВМ. Эти варианты показаны на рисунке 3.7.

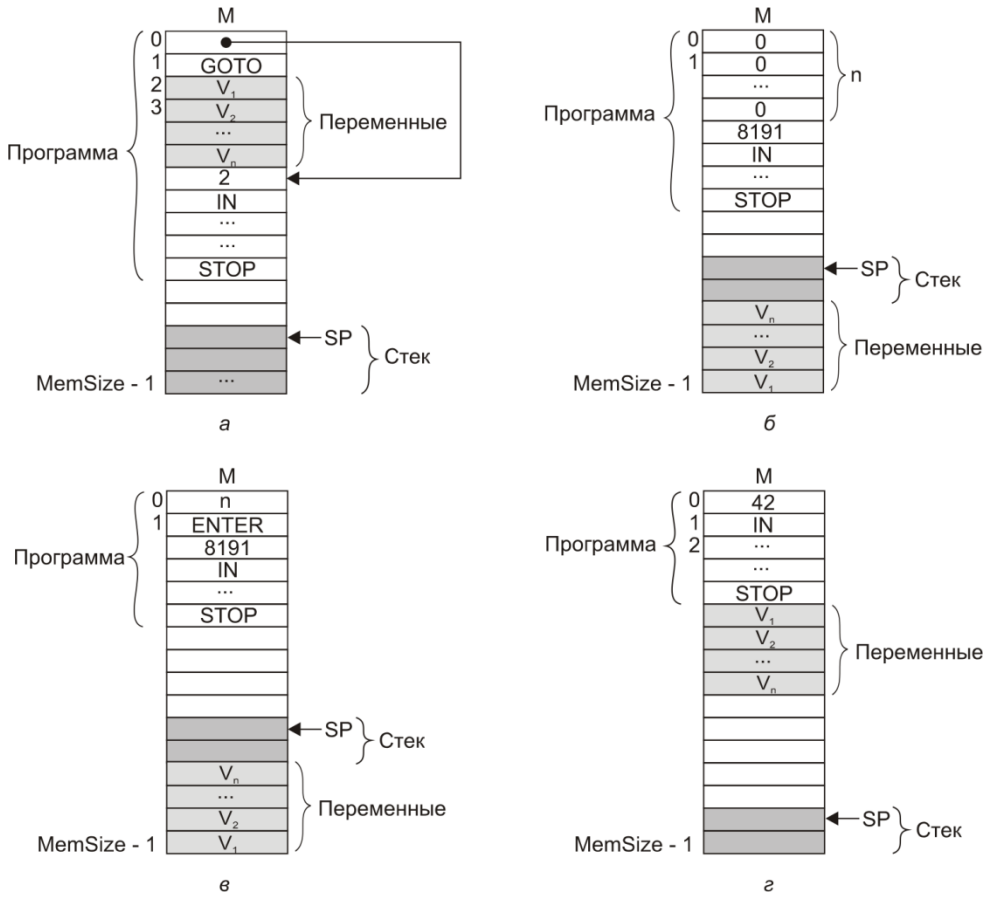


Рис. 3.7. Варианты распределения памяти под переменные

Можно разместить переменные перед кодом программы в начале памяти (рис. 3.7*а*). Поскольку ОВМ при запуске начинает выполнение с команды по адресу 0, компилятор должен поместить в ячейки с адресами 0 и 1 команды, выполняющие переход на основную часть программы в обход участка памяти, отведенного под переменные. Адреса переменным компилятор может назначить в ходе трансляции описаний и хранить эти адреса в поле `Val` записей таблицы имен, относящихся к переменным. Первая переменная получает адрес 2, следующая — 3 и т.д.

Другой возможный вариант размещения переменных — в старших адресах памяти, на дне стека (рис. 3.7*б*). Первая переменная программы размещается в ячейке с адресом `MemSize-1`, вторая по адресу `MemSize-2` и т.д. Назначение адресов может происходить уже в ходе трансляции описаний переменных. Первыми командами машинная программа должна выполнить резервирование места под переменные. Указатель стека должен быть продвинут вверх на столько ячеек, сколько переменных имеется в программе (значение `SP` должно быть уменьшено на число переменных программы). Резервирование заданного количества ячеек в стеке может быть выполнено серией команд, записывающих в стек константу (например, 0). Это можно сделать в цикле или индивидуально для каждой переменной. Запись определенной константы в отведенную переменной ячейку при резервировании памяти означало бы гарантированную инициализацию переменных. В тех языках, где такая инициализация предусмотрена, подобный механизм распределения был бы весьма подходящим. В языке «О» обязательная инициализация не предусмотрена.

На рис. 3.7*в* показан вариант с таким же размещением переменных. Но резервирование места в стеке выполняется специальной командой `ENTER`, которой пока нет в системе команд ОВМ. Эта команда будет рассмотрена позже.

Одним из недостатков размещения переменных на дне стека является то, что код программы оказывается зависим от объема памяти виртуальной машины. Программа, откомпилированная для машины с памятью, например, 16К слов не сможет выполняться на машине с памятью в 8К слов, даже если этого объема памяти было бы достаточно. Дело в том, что в программе предназначенной для машины с памятью 16К слов, переменные размещены в ячейках с адресами 16 383, 16 384, 16 381, ..., которых просто нет у машины с 8К слов.

Наиболее естественным представляется размещение переменных сразу за кодом программы после команды `STOP` (рис. 3.7*г*). Именно такой вариант и будет использован в нашем компиляторе. Вопрос только в том, что объем кода неизвестен как во время трансляции описаний, так и при генерации команд, в которых должны использоваться адреса переменных. Проблема, однако, решается. При этом не требуется формировать никаких дополнительных машинных команд для обхода области размещения переменных или резервирования ячеек. После того как мы приобретём опыт генерации кода, вернемся к этому вопросу и реализуем размещение переменных несложным, хоть и не тривиальным способом. При этом окажется, что задержка с назначением адресов переменным до завершения генерации кода не препятствует этой генерации.

Для полноты картины упомянем вариант, когда переменные размещаются после любой имеющейся в программе команды безусловного перехода. Это не требует генерации никаких дополнительных команд, но представляется ненужной экзотикой и не имеет каких-либо преимуществ. Вообще говоря, память под переменные даже не обязательно распределять единым массивом.

Под константы также можно было бы отводить отдельные ячейки памяти и при необходимости доступа к их значениям загру-

жать константы на стек из этих ячеек. Но в нашем случае в этом нет нужды, поскольку константы могут быть просто встроены в код.

Генерация кода для выражений

Общий принцип генерации кода для выражений состоит в том, что каждая распознающая процедура, участвующая в трансляции — транслятор выражения, простого выражения, слагаемого, множителя — должна сформировать такой машинный код, который вычисляет значение соответствующего подвыражения, оставляя вычисленное значение на вершине стека. Или по-другому: каждый распознаватель формирует обратную польскую запись подвыражения, за которое этот распознаватель отвечает — ведь машинный код ОВМ — не что иное, как обратная польская запись программы.

Генерация кода для множителей

Множитель — это элементарное выражение, атомарный операнд: константа, переменная, вызов функции, выражение в скобках.

Код для константы, встретившейся в выражении, должен поместить значение этой константы на стек. Для неотрицательных констант достаточно сгенерировать команду, совпадающую со значением константы, для отрицательных предусматривается еще перемена знака.

Код, порождаемый при распознавании переменной в составе выражения, должен загружать значение переменной на стек. Для этого генерируется адрес переменной и команда `LOAD`.

Для вызова каждой стандартной функции код генерируется индивидуально. Формирование кода для выражения в скобках распознаватель множителя поручает распознавателю выражений.

В листинге 3.46 приведена процедура `Factor` (ее предыдущий вариант — в листинге 3.35) — анализатор множителя, в текст которой добавлены действия по генерации кода, которые выделены.

Листинг 3.46. Генерация кода для множителя

```
procedure Factor(var T : tType);
var
   X : tObj;
begin
   if Lex = lexName then begin
      Find(Name, X);
      if X^.Cat = catVar then begin
         GenAddr(X);    {Адрес переменной}
         Gen(cmLoad);
         T := X^.Typ;
         NextLex;
         end
      else if X^.Cat = catConst then begin
         GenConst(X^.Val);
         T := X^.Typ;
         NextLex;
         end
      else if (X^.Cat=catStProc) and (X^.Typ<>typNone)
      then begin
         NextLex;
         Check(lexLPar, '"("');
         StFunc(X^.Val, T);
         Check(lexRPar, '")"');
         end
      else
         Expected(
         'переменная, константа или процедура-функции'
         );
      end
   else if Lex = lexNum then begin
      T := typInt;
      GenConst(Num);
      NextLex
      end
   else if Lex = lexLPar then begin
      NextLex;
      Expression(T);
      Check(lexRPar, '")"');
      end
   else
```

```
        Expected('имя, число или "("');
  end;
```

Явным образом здесь генерируется лишь команда LOAD для загрузки на стек значения переменной (код команды задан константой cmLoad, определенной в модуле виртуальной машины, см. листинг 3.42). Для генерации кода для константы вызывается процедура GenConst, которая порождает одну или две команды в зависимости от знака константы. Отрицательное число не может непосредственно быть записано в код ОВМ, поскольку в этом случае его не отличить от кода операции. Напомню, что все коды операций ОВМ — отрицательные. Процедуру GenConst (листинг 3.47) размещаем в модуле OGen.

Листинг 3.47. Генерация кода для константы (модуль OGen)

```
  procedure GenConst(C: integer);
  begin
     Gen(abs(C));
     if C < 0 then
        Gen(cmNeg);
  end;
```

Должно быть понятно, что процедура GenAddr(X), не может записать в память ОВМ адрес переменной, который будет этой переменной окончательно назначен. Код программы еще не сформирован полностью и невозможно узнать адрес ячейки, где могла бы разместиться та переменная, на запись об имени которой в таблице имен ссылается указатель X. В действительности GenAddr записывает в то место машинной программы, где должен быть адрес, некоторую служебную информацию, которая в дальнейшем позволит поместить сюда правильный адрес. До рассмотрения алгоритма назначения адресов и реализации GenAddr мы можем считать, что действие этой процедуры соответствует ее названию.

Трансляция вызовов стандартных функций

В языке «О» предусмотрены четыре стандартных функции: `ABS`, `MAX`, `MIN` и `ODD`. Это то подмножество стандартных функций Оберона, которое сохраняет смысл в языке, где есть переменные и константы только целого типа, а также простейшие логические выражения.

Обычно основанием для включения функции или процедуры непосредственно в язык является то, что она не может быть запрограммирована с помощью других средств этого языка, поскольку имеет нестандартный синтаксис фактических параметров, необычные правила, относящиеся к типам параметров и типу значения. Если иметь в виду Оберон, это, безусловно, относится к `MAX` и `MIN`, поскольку аргументом этих функций является тип, в то время как обычные процедуры и функции Оберона не могут иметь аргументов-типов. Функция `ABS` в Обероне также своеобразна: тип ее значения зависит от типа аргумента. Абсолютная величина вещественного числа будет иметь вещественный тип, целого — целый. Что касается `ODD`, то она могла быть запрограммирована обычными средствами и присутствует в Обероне, вероятно, по традиции, обеспечивая эффективное преобразование целого типа в логический.

Такой статус стандартных функций обуславливает и способ из реализации в компиляторе. Во-первых, для вызовов стандартных функций зачастую генерируются не команды вызова подпрограмм, а машинный код, вычисляющий значение функции прямо в месте вызова. Во-вторых, обработка списка фактических параметров стандартных функций и процедур выполняется индивидуально.

В нашем компиляторе обработку списка фактических параметров выполняет процедура `StFunc`.

```
procedure StFunc(F: integer; var T: tType);
```

Входным параметром StFunc является номер стандартной функции (stABS, stMAX, ...); выходным — тип ее результата. В момент вызова StFunc (см. листинги 3.35, 3.46) текущей лексемой является первая лексема списка фактических параметров.

Рассмотрим вначале, как должны транслироваться отдельные функции и их списки аргументов.

Функция ABS

ABS имеет единственный аргумент, который может быть выражением. Транслируем это выражение, вычисляя его тип и генерируя для него код:

```
Expression(T);
```

В языке «О» аргумент ABS может иметь только целый тип. Проверяем это:

```
if T <> typInt then
   Expected('выражение целого типа');
```

Транслятор выражений Expression генерирует команды, в результате исполнения которых значение фактического параметра ABS будет помещено на вершину стека. Теперь наша задача состоит в том, чтобы сформировать машинный код, который изменит знак числа, находящегося на вершине стека, если число отрицательно, и оставит число без изменения в противном случае. Напомню, что команды ОВМ, выполняющие проверки (команды условных переходов), удаляют свои операнды со стека. Поэтому перед проверкой необходимо продублировать с помощью DUP проверяемое значение. Код должен получаться такой:

```
A-5) DUP  ; X, X
A-4) 0    ; X, X, 0
A-3) A    ; X, X, 0, A
A-2) IFGE ; X
A-1) NEG  ; -X
  A) …    ; теперь на стеке ABS(X)
```

Здесь X — значение аргумента функции; A — адрес команды, следующей за формируемым фрагментом, он же — адрес условного перехода при X>=0. В комментариях после точки с запятой, как обычно, показано состояние стека после выполнения каждой команды. Все приведенные команды, кроме той, что представляет собой адрес перехода A, могут быть легко сгенерированы вызовом процедуры Gen с соответствующим аргументом. Адрес перехода вычисляется добавлением к текущему значению счетчика команд (переменная PC модуля OGen) смещения ячейки, на которую выполняется переход, относительно ячейки, в которую записывается адрес этого перехода (ячейка с адресом A-3). Это смещение равно 3. Генерацию такого кода выполнит процедура GenAbs (листинг 3.48), которую поместим в модуль OGen.

Листинг 3.48. Генерация кода для ABS

```
procedure GenAbs;
begin
   Gen(cmDup);
   Gen(0);
   Gen(PC+3);
   Gen(cmIfGE);
   Gen(cmNeg);
end;
```

Вызовом GenAbs и заканчивается трансляция списка фактических параметров стандартной процедуры-функции ABS.

Вызов транслятора выражений Expression с последующей проверкой типа выражения будет несколько раз использоваться и в дальнейшем. Предусмотрим для этих целей специальные процедуры (листинг 3.49). Одна из них, IntExpression, будет вызываться, если надо транслировать выражение и проверить, что оно имеет целый тип, другая, BoolExpression, вызывается, если ожидается выражение логического типа.

Листинг 3.49. Трансляция и проверка типа выражений

```
procedure IntExpression;
var T : tType;
```

```
begin
   Expression(T);
   if T <> typInt then
      Expected('выражение целого типа');
end;

procedure BoolExpression;
var
   T : tType;
begin
   Expression(T);
   if T <> typBool then
      Expected('логическое выражение');
end;
```

Обе эти процедуры располагаются в секции реализации модуля `OPars`.

Процедура `StFunc`, включающая окончательный вариант фрагмента, отвечающего за трансляцию `ABS`, показана в листинге 3.50. Схема трансляции трех других функций та же самая: вызывается процедура, распознающая аргумент, генерируется код самой функции, фиксируется тип функции.

Листинг 3.50. Трансляция стандартных функций

```
procedure StFunc(F: integer; var T: tType);
begin
   case F of
   spABS:
      begin
         IntExpression;
         GenAbs;
         T := typInt;
      end;
   spMAX:
      begin
         ParseType;
         Gen(MaxInt);
         T := typInt;
      end;
   spMIN:
      begin
         ParseType;
         GenMin;
         T := typInt;
      end;
```

```
      spODD:
         begin
            IntExpression;
            GenOdd;
            T := typBool;
         end;
      end;
   end;
```

Функции MAX и MIN

Функция `MAX`, которая в языке «О» может быть использована только в форме `MAX(INTEGER)`, возвращает число, обозначаемое в Паскале `MaxInt`. Машинный код, порождаемый `MAX`, будет состоять из одной команды-константы, равной 32 767, если реализация 16-разрядная, и равной 2 147 483 647, если 32-разрядная. Кроме генерации команды фрагмент, отвечающий за `MAX`, проверяет аргумент вызовом `ParseType` и задает тип: `T := typInt`.

Несколько сложнее дело обстоит с трансляцией `MIN`. Виртуальная О-машина не предусматривает непосредственную загрузку отрицательной константы на стек. Формировать значение `MIN(INTEGER)` на стеке приходится в соответствии с формулой `-MAX(INTEGER) - 1`. Эту работу выполняет процедура `GenMin` (листинг 3.51) модуля `OGen`.

Листинг 3.51. Генерация кода для функции MIN

```
   procedure GenMin;
   begin
      Gen(MaxInt);
      Gen(cmNeg);
      Gen(1);
      Gen(cmSub);
   end;
```

Функция ODD

Функция `ODD`, в отличие от `ABS`, `MAX` и `MIN`, имеет логический тип. Это, однако, не означает, что сгенерированный для нее код будет вычислять логическое значение, оставляя его на стеке. Логические выражения вообще и функция `ODD` в частности использу-

ются в языке «О» только в операторах **IF** и **WHILE**. И в том и в другом случае при истинности логического выражения должны выполняться операторы, машинный код которых следует сразу за условием (рис. 3.8). Если условие ложно, должен выполняться переход на участок машинного кода, расположенный дальше (переход вперед).

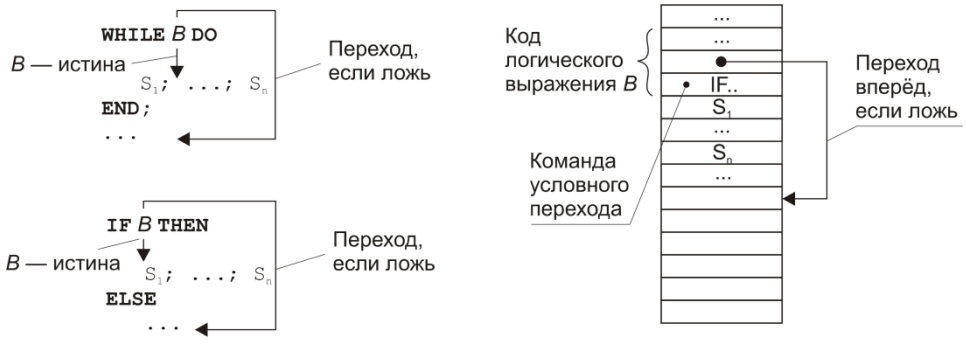


Рис. 3.8. Схема трансляции логических выражений

Таким образом, код логического выражения должен заканчиваться командой условного перехода, выполняемого, если выражение ложно. Это весьма универсальный подход, применимый в большинстве случаев и при трансляции логических выражений общего вида, включающих операции «И», «ИЛИ», «НЕ». Можно также заметить, что и при трансляции оператора **REPEAT** – **UNTIL**, имеющегося в Обероне и других языках, логическое выражение также должно порождать «переход, если ложь». Что касается языка «О», то сформулированный принцип будет использован во всех случаях трансляции логических выражений.

Итак, код, который генерируется для `ODD`, должен вычислить остаток от деления находящегося на стеке числа на 2 и, если этот остаток равен 0 (число четно, значение `ODD` «ложь»), выполнить переход вперед. Однако, адрес этого перехода в момент генерации кода для `ODD` неизвестен. Его можно определить лишь после того, как будут сгенерированы команды для участка программы, выполняемого в случае истинности логического выражения. Ес-

ли речь о цикле **WHILE**, то это следующее после слова **DO** тело цикла, а если логическое выражение используется в операторе **IF**, то это последовательность операторов, следующая за **THEN.**

Решение проблемы состоит в том, что вместо реального адреса перехода генерируется фиктивное значение (например, 0). Это позволяет продолжить генерацию, зарезервировав в машинном коде место, куда позднее будет помещен адрес перехода вперед. Местоположение ячейки, в которую занесен фиктивный адрес, запоминается, а после того, как становится известно место, куда должен быть выполнен переход, в эту ячейку записывается нужное значение.

Процедура, генерирующая код для ODD в соответствии с рассмотренной схемой, показана в листинге 3.52.

Листинг 3.52. Генерация кода для функции ODD

```
procedure GenOdd;
begin
   Gen(2);
   Gen(cmMod);
   Gen(0);
   Gen(0); {Фиктивный адрес перехода вперед}
   Gen(cmIfEQ);
end;
```

Запоминание местоположения ячейки, в которую записан фиктивный адрес перехода вперед, выполняется не внутри процедуры GenOdd, или процедур транслирующих другие варианты логических выражений, а в тех частях транслятора, которые отвечают за обработку операторов **IF** и **WHILE**. При трансляции этих операторов детальные сведения о коде, сформированном для логического выражения, следующего за **IF** и **WHILE** (а также **ELSIF**), уже будут недоступны. Однако остается известным, что машинный код логического выражения обязательно завершается командой условного перехода, а во второй от конца этого кода ячейке записан фиктивный адрес перехода вперед.

Невозможность определить адрес перехода вперед в момент формирования команды перехода — одна из причин того, что ранние компиляторы должны были выполнять несколько проходов. Не имея возможности удерживать в памяти сколько-нибудь значительные участки формируемого кода, многопроходный транслятор вначале запоминает ячейки, в которых должны быть записаны адреса переходов вперед, определяет эти адреса, и уже на другом проходе помещает их в соответствующие места формируемого кода.

Наличие памяти, позволяющей хранить весь формируемый машинный код (как в нашем случае) или хотя бы код одной процедуры, дает возможность выполнить формирование адресов переходов вперед однопроходному транслятору.

Генерация кода для слагаемых

Слагаемое (Term) — это один или несколько множителей, соединенных знаками операций типа умножения (*, **DIV**, **MOD**).

```
Слагаемое = Множитель {ОперУмн Множитель}.
```

Задача транслятора при преобразовании слагаемого в машинный код ОВМ состоит в том, чтобы породить команды, вычисляющие значение слагаемого и оставляющие это значение на вершине стека. При этом выражение вида

$$F_1 \otimes F_2,$$

где F_1, F_2 — множители, а $\otimes$ — операция типа умножения, транслируется в обратную польскую запись:

Машинный код для F_1
Машинный код для F_2
$\otimes$

Генерация кода для множителей выполняется вызовом процедуры `Factor`. Транслятору слагаемого (процедуре Term) остается лишь формировать команды, выполняющие операции умноже-

ния деления и получения остатка. Для этого знак операции запоминается в момент, когда соответствующая лексема является текущей, а после трансляции очередного множителя формируется машинная команда, соответствующая запомненному знаку.

Связанные с генерацией кода вставки, которые нужно сделать в процедуру `Term` из листинга 3.34, выполнявшую синтаксический и контекстный анализ слагаемого, в листинге 3.53 выделены.

Листинг 3.53. Транслятор слагаемого

```
procedure Term(var T: tType);
var
   Op : tLex;
begin
   Factor(T);
   if Lex in [lexMult, lexDIV, lexMOD] then begin
      if T <> typInt then
         Error('Несоответствие операции типу операнда');
      repeat
         Op := Lex;
         NextLex;
         Factor(T);
         if T <> typInt then
            Expected('выражение целого типа');
         case Op of
         lexMult: Gen(cmMult);
         lexDIV:  Gen(cmDIV);
         lexMOD:  Gen(cmMOD);
         end;
      until not( Lex in [lexMult, lexDIV, lexMOD] );
   end;
end;
```

Генерация кода для простых выражений

Генерация кода для простых выражений выполняется аналогично трансляции множителей. Разница в том, что вместо умножения и деления речь идет о формировании машинных команд для операций сложения и вычитания, а также в том, что если перед первым слагаемым есть минус, то генерируется команда перемены знака.

Добавим в процедуру SimpleExpr, выполняющую синтаксический и контекстный анализ простого выражения (см. листинг 3.33) фрагменты, отвечающие за генерацию кода. В листинге 3.54 они выделены.

Листинг 3.54. Трансляция простого выражения

```
(* ["+"|"-"] Слагаемое {ОперСлож Слагаемое} *)
procedure SimpleExpr(var T : tType);
var
   Op : tLex;
begin
   if Lex in [lexPlus, lexMinus] then begin
      Op := Lex;
      NextLex;
      Term(T);
      if T <> typInt then
         Expected('выражение целого типа');
      if Op = lexMinus then
         Gen(cmNeg);
      end
   else
      Term(T);
   if Lex in [lexPlus, lexMinus] then begin
      if T <> typInt then
         Error('Несоответствие операции типу операнда');
      repeat
         Op := Lex;
         NextLex;
         Term(T);
         if T <> typInt then
            Expected('выражение целого типа');
         case Op of
         lexPlus:  Gen(cmAdd);
         lexMinus: Gen(cmSub);
         end;
      until not( Lex in [lexPlus, lexMinus] );
   end;
end;
```

Генерация кода для выражений общего вида

Выражение языка «О» (и языка Оберон) — это одно простое выражение или два простых выражения, соединенных знаком операции отношения. В первом случае (одно простое выражение)

никаких специальных действий по генерации кода не требуется, код формируется вызовом процедуры `SimpleExpr`.

Если же имеется знак отношения, то его следует запомнить в тот момент, когда этот знак прочитан, а после формирования кода для правого операнда отношения (вызовом `SimpleExpr`) нужно сгенерировать условный переход вперед, выполняемый, если отношение ложно. Таким образом, выдерживается общее правило формирования кода для логических выражений: машинный код логического выражения заканчивается условным переходом вперед, выполняемым, если выражение ложно. Второй от конца этого кода командой является адрес перехода, который вначале заменяется фиктивным значением, а окончательно формируется распознавателем конструкции, в которой используется логическое выражение.

Вставки в анализатор выражений, отвечающие за генерацию кода, в листинге 3.55 отмечены.

Листинг 3.55. Генерация кода для выражений

```
(* ПростоеВыраж [Отношение ПростоеВыраж] *)
procedure Expression(var T : tType);
var
   Op : tLex;
begin
   SimpleExpr(T);
   if Lex in [lexEQ, lexNE, lexGT, lexGE, lexLT, lexLE]
   then begin
      Op := Lex;
      if T <> typInt then
         Error('Несоответствие операции типу операнда');
      NextLex;
      SimpleExpr(T); {Правый операнд отношения}
      if T <> typInt then
         Expected('выражение целого типа');
      GenComp(Op);   {Генерация условного перехода}
      T := typBool;
   end; {иначе тип равен типу первого простого выражения}
end;
```

Формирование условного перехода, замыкающего машинный код отношения (сравнения) выполняет процедура GenComp (от generate comparison — сформировать сравнение). Эту процедуру поместим в модуль OGen. В качестве параметра процедуре GenComp передается знак отношения, записанный в выражении. GenComp (листинг 3.56) формирует переход, используя машинную команду, соответствующую противоположному отношению. Так, например, если исходное сравнение имело вид A<=B, то будет сгенерирован «переход, если больше» — команда IFGT, поскольку переход должен выполняться, если условие ложно.

Листинг 3.56. Генерация кода для сравнений

```
procedure GenComp(Op: tLex);
begin
   Gen(0); {Фиктивный адрес перехода вперед}
   case Op of
   lexEQ : Gen(cmIfNE);
   lexNE : Gen(cmIfEQ);
   lexLE : Gen(cmIfGT);
   lexLT : Gen(cmIfGE);
   lexGE : Gen(cmIfLT);
   lexGT : Gen(cmIfLE);
   end;
end;
```

Генерация кода для операторов

Распознаватели операторов присваивания и вызова процедуры, операторов **IF** и **WHILE** вызываются из уже написанной при рассмотрении контекстного анализа процедуры Statement (см. листинг 3.37). Теперь запрограммируем эти распознаватели. В их задачу входит синтаксический и контекстный анализ соответствующих операторов и генерация кода для них.

Трансляция оператора присваивания

Транслятор оператора присваивания строится в точном соответствии с синтаксисом этой конструкции:

```
Переменная ":=" Выраж.
```

Для анализа и генерации кода для переменной и выражения вызываются (листинг 3.57) распознаватели `Variable` и `IntExpression`. При трансляции переменной должен быть сформирован код, оставляющий на вершине стека адрес этой переменной. Транслятор выражения формирует команды, вычисляющие выражение и оставляющие его значение на стеке. Напомню, что `IntExpression` (см. листинг 3.49) не только выполняет синтаксический анализ и генерацию кода для выражения, но и проверяет, имеет ли выражение целый тип.

Процедуре `AssStatement` остается сформировать команду `SAVE`, которая запишет значение, находящееся на вершине стека по адресу, находящемуся под вершиной.

Листинг 3.57. Трансляция оператора присваивания

```
(* Переменная ":=" Выраж *)
procedure AssStatement;
begin
   Variable;
   if Lex = lexAss then begin
      NextLex;
      IntExpression;
      Gen(cmSave);
      end
   else
      Expected('":="')
end;
```

Таким образом, код, формируемый для присваивания вида `v := E`, где `v` — переменная, а `E` — выражение, имеет следующую структуру:

```
Av
Код для E
SAVE
```

Здесь `Av` обозначает адрес переменной `v`. Транслятор переменной — процедура `Variable` – показана в листинге 3.58.

Листинг 3.58. Трансляция переменной

```
(* Переменная = Имя. *)
procedure Variable;
var
   X : tObj;
begin
   if Lex <> lexName then
      Expected('имя')
   else begin
      Find(Name, X);
      if X^.Cat <> catVar then
         Expected('имя переменной');
      GenAddr(X);
      NextLex;
   end;
end;
```

В языке «О» в роли переменной может использоваться только имя. Выполняется поиск имени в таблице имен, проверка того, что оно принадлежит переменной и формирование команды, представляющей адрес этой переменной. В качестве параметра процедуре `GenAddr`, ответственной за генерацию адреса, передается ссылка на переменную в таблице имен (`X`).

Трансляция вызовов стандартных процедур

Анализатор операторов `Statement` (см. листинг 3.37), встретив имя процедуры, определяет ее порядковый номер (`P`) по таблице имен и вызывает процедуру `CallStatement` (листинг 3.59), передав ей этот номер в качестве параметра. Текущей лексемой в момент вызова является имя процедуры. `CallStatement` начинает с того, что пропускает эту лексему. Предусмотренная при этом проверка с помощью `Check`, по сути, является фиктивной, поскольку вызывающая программа уже определила, что имя действительно принадлежит процедуре. Вызов `Check` вместо `NextLex` использован здесь лишь для наглядности.

Листинг 3.59. Транслятор операторов вызова процедуры

```
(* Имя [ "(" Параметр {"," Параметр} ")" ] *)
procedure CallStatement(P : integer);
begin
```

```
    Check(lexName, 'имя процедуры');
    if Lex = lexLPar then begin
       NextLex;
       StProc(P);
       Check(lexRPar, '")"');
       end
    else if P in [spOutLn, spInOpen] then
       StProc(P)
    else
       Expected('"("');
end;
```

Вызов процедуры без параметров, а их в языке «О» две: `Out.Ln` и `In.Open`, может содержать пустую пару скобок или не содержать скобок вообще. Если же отсутствуют скобки при вызове процедур, которые обязаны содержать параметры, сообщается об ошибке. Замечу, что значение `P` здесь всегда корректно и соответствует одной из предусмотренных в языке «О» стандартных процедур.

Трансляция списка фактических параметров и генерация кода для конкретных процедур выполняется подпрограммой `StProc`. Как и при обработке вызовов стандартных функций (см. листинг 3.50), список фактических параметров каждой процедуры транслируется индивидуально (листинг 3.60).

Листинг 3.60. Трансляция фактических параметров стандартных процедур

```
procedure StProc(P: integer);
var
   c : integer;
begin
   case P of
   spDEC:
      begin
         Variable;
         Gen(cmDup);
         Gen(cmLoad);
         if Lex = lexComma then begin
            NextLex;
            IntExpression;
            end
         else
            Gen(1);
```

```
            Gen(cmSub);
            Gen(cmSave);
         end;
      spINC:
         begin
            Variable;
            Gen(cmDup);
            Gen(cmLoad);
            if Lex = lexComma then begin
               NextLex;
               IntExpression;
               end
            else
               Gen(1);
            Gen(cmAdd);
            Gen(cmSave);
         end;
      spInOpen: { Пусто };
      spInInt:
         begin
            Variable;
            Gen(cmIn);
            Gen(cmSave);
         end;
      spOutInt:
         begin
            IntExpression;
            Check(lexComma, '','');
            IntExpression;
            Gen(cmOut);
         end;
      spOutLn:
         Gen(cmOutLn);
      spHalt:
         begin
            ConstExpr(c);
            GenConst(c);
            Gen(cmStop);
         end;
      end;
  end {case};
```

Трансляция оператора `DEC(v)`, где `v` — переменная, порождает такой код:

```
Av
DUP
LOAD
1
```

```
SUB
SAVE
```

Если вызов `DEC` имеет вид `DEC(v, n)`, где `n` — выражение, то генерируются команды:

```
Av
DUP
LOAD
Код для n
SUB
SAVE
```

Аналогично транслируется вызов процедуры `INC`. Процедура `In.Open` включена в язык «О» исключительно для совместимости с существующими реализациями Оберона и никакого кода не порождает. Трансляция `In.Int`, `Out.Int` и `Out.Ln` представляется очевидной. Аргументом процедуры `HALT` может быть константное выражение, значение которого остается на стеке после остановки программы и выводится в качестве кода возврата по окончании работы виртуальной машины (см. листинг 3.44).

Трансляция оператора `WHILE`

Синтаксический и контекстный анализ конструкции

```
WHILE ЛогическоеВыражение DO
   ПоследовательностьОператоров
END
```

очень прост. Необходимые проверки выполняются несколькими вызовами процедур:

```
Check(lexWHILE, 'WHILE');
BoolExpression;
Check(lexDO, 'DO');
StatSeq;
Check(lexEND, 'END');
```

Генерация сводится к формированию двух команд перехода: условного, выполняемого за пределы цикла, если условие цикла ложно, и безусловного перехода назад на начало цикла, выполняемого по завершении последовательности операторов. Структура кода должна быть такой:

```
WhilePC: Код для логического выражения
         завершается условным переходом:
         EndPC   ; адрес перехода вперед
         IF..    ; команда условного перехода
CondPC:
         Код для
         последовательности
         операторов

         WhilePC ; адрес начала цикла
         GOTO

EndPC:   …
```

В этой схеме использованы метки, отделенные от обозначений команд двоеточием — обычное решение для языков ассемблера. Метки соответствуют адресам тех команд, на которые ссылаются. Значением метки `WhilePC` является адрес первой команды, вычисляющей логическое выражение; `CondPC` — это адрес участка кода, соответствующего последовательности операторов; `EndPC` — адрес участка программы, следующего за циклом. Во время компиляции `WhilePC`, `CondPC` и `EndPC` — это значения программного счетчика времени компиляции (переменная `PC` модуля `OGen`) перед началом трансляции цикла, после трансляции логического выражения и после трансляции `END` соответственно.

Для формирования команд выполняется следующее:

1. Перед трансляцией цикла (логического выражения) запоминается в локальной переменной `WhilePC` текущее значение программного счетчика.
2. Значение `PC` по окончании трансляции логического выражения запоминается в локальной переменной `CondPC`.
3. После трансляции последовательности операторов генерируется адрес перехода назад на начало цикла, равный значению `WhilePC` и команда `GOTO`. Можно считать, что эти команды порождаются как результат трансляции слова `END`, завершающего цикл.

4. По адресу, равному `CondPC-2` записывается текущее значение `PC` (в предыдущих рассуждениях было обозначено `EndPC`). Тем самым завершается формирование перехода вперед.

Перечисленные действия выполняет процедура `WhileStatement`, приведенная в листинге 3.61. За генерацию перехода вперед с фиктивным адресом отвечает процедура `BoolExpression`. Запись текущего значения `PC` по адресу `CondPC-2`, завершающая генерацию этого перехода, выполняется вызовом `Fixup(CondPC)` (fixup — адресная привязка — общепринятый термин, обозначающий именно то, что и поручено этой процедуре, — фиксацию значений адресов, которые не могли быть определены раньше).

Листинг 3.61. Трансляция оператора цикла

```
procedure WhileStatement;
var
   WhilePC  : integer;
   CondPC   : integer;
begin
   WhilePC := PC;
   Check(lexWHILE, 'WHILE');
   BoolExpression;
   CondPC := PC;
   Check(lexDO, 'DO');
   StatSeq;
   Check(lexEND, 'END');
   Gen(WhilePC);
   Gen(cmGOTO);
   Fixup(CondPC);
end;
```

Поскольку операторы цикла могут быть вложенными, принципиально важно, что `WhilePC` и `CondPC` локальны в `WhileStatement`.

Трансляция оператора `IF`

Подход к трансляции оператора **`IF`** сходен с методами, использованными при компиляции цикла **`WHILE`**. Но **`IF`** может порождать только переходы вперед, к тому же, количество таких переходов из-за присутствия частей **`ELSIF`** заранее неизвестно.

Правила записи конструкции задаются формулой:

```
IF Выраж THEN
   ПослОператоров
{ELSIF Выраж THEN
   ПослОператоров}
[ELSE
   ПослОператоров]
END
```

а синтаксический и контекстный анализ выполняются следующим образом:

```
Check(lexIF, 'IF');
BoolExpression;  {Логическое выражение}
Check(lexTHEN, 'THEN');
StatSeq;
while Lex = lexELSIF do begin
   NextLex;
   BoolExpression; {Логическое выражение         }
   Check(lexTHEN, 'THEN');
   StatSeq;        {Последовательность операторов}
end;
if Lex = lexELSE then begin
   NextLex;
   StatSeq;        {Последовательность операторов}
end;
Check(lexEND, 'END');
```

Генерация кода должна обеспечить формирование по одному условному переходу на каждое условие (логическое выражение). Каждый такой переход должен быть направлен в обход той последовательности операторов, которая выполняется при истинности данного условия (рис. 3.9). Сами команды перехода формируются при трансляции логических выражений. Задача транслятора **IF** — лишь зафиксировать (с помощью `Fixup`) адреса этих переходов.

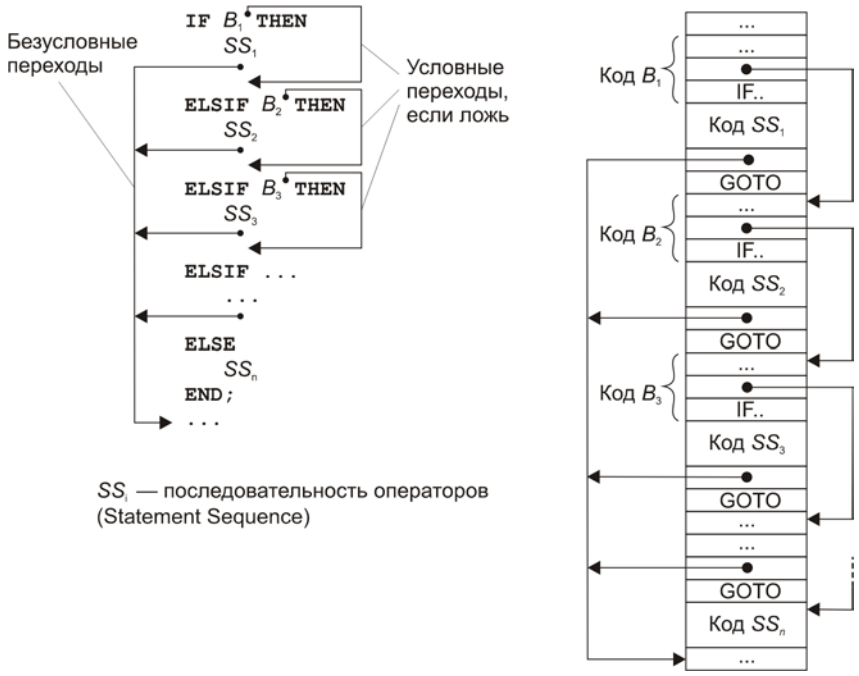


Рис. 3.9. Схема трансляции оператора **IF**

Безусловные переходы должны быть сгенерированы после кода каждой последовательности операторов кроме последней. Адрес всех переходов один и тот же, все они должны выполняться на команду, следующую за оператором **IF**. Оказывается, что адрес **заранее неизвестного количества** безусловных переходов не может быть определен до окончания трансляции всего оператора, а при трансляции завершающего **END** нужно занести текущее значение счетчика команд PC в несколько ячеек. Адреса этих ячеек должны запоминаться по ходу трансляции. Для этих целей можно было бы использовать массив или список, но мы поступим по-другому.

Последовательность адресов ячеек, в которые нужно занести адрес безусловного перехода вперед, будем хранить в самих этих ячейках! Каждая следующая (с большим адресом) ячейка будет хранить адрес предыдущей. В самую первую ячейку (не имеющую предшествующей) запишем 0. Адрес последней (с наибольшим адресом) ячейки будем хранить в локальной перемен-

ной (LastGOTO). На самом деле в каждой из упомянутых ячеек и в переменной LastGOTO будет храниться не сам адрес, а его значение, увеличенное на 2. Это позволяет унифицировать подходы, запоминая места как условных (с помощью CondPC), так и безусловных переходов (с помощью LastGOTO) **после** формирования самих команд перехода.

Листинг 3.62. Трансляция условного оператора

```
procedure IfStatement;
var
   CondPC   : integer;
   LastGOTO : integer;
begin
   Check(lexIF, 'IF');
   LastGOTO := 0;     {Предыдущего перехода нет        }
   BoolExpression;
   CondPC := PC;      {Запомн. положение усл. перехода }
   Check(lexTHEN, 'THEN');
   StatSeq;
   while Lex = lexELSIF do begin
      Gen(LastGOTO);  {Фиктивный адрес, указывающий    }
      Gen(cmGOTO);    {на место предыдущего перехода.  }
      LastGOTO := PC; {Запомнить место GOTO            }
      NextLex;
      Fixup(CondPC);  {Зафикс. адрес условного перехода}
      BoolExpression;
      CondPC := PC;   {Запомн. положение усл. перехода }
      Check(lexTHEN, 'THEN');
      StatSeq;
   end;
   if Lex = lexELSE then begin
      Gen(LastGOTO);  {Фиктивный адрес, указывающий    }
      Gen(cmGOTO);    {на место предыдущего перехода   }
      LastGOTO := PC; {Запомнить место последнего GOTO }
      NextLex;
      Fixup(CondPC);  {Зафикс. адрес условного перехода}
      StatSeq;
      end
   else
      Fixup(CondPC);   {Если ELSE отсутствует          }
   Check(lexEND, 'END');
   Fixup(LastGOTO);    {Направить сюда все GOTO      }
end;
```

Процедура Fixup (листинг 3.63) может обеспечить как фиксацию адреса одиночного перехода, так и заполнение цепочки ячеек, когда это требуется при компиляции **IF**. Фиксация одиночного перехода оказывается частным случаем формирования цепочки адресов при условии, что при генерации кода для одиночного перехода вперед в качестве фиктивного адреса был записан 0. Это требование соблюдается (см. листинги 3.52, 3.56). Можно заметить, что значение 0 не может быть ссылкой на предыдущий переход, поскольку это означало бы, что адрес такого перехода располагается в ячейке с адресом –2, которой не существует.

Листинг 3.63. Адресная привязка

```
procedure Fixup(A: integer);
var
   temp: integer;
begin
   while A > 0 do begin
      temp := M[A-2];
      M[A-2] := PC;
      A := temp;
   end;
end;
```

В качестве входного параметра процедуре Fixup передается адрес A, указывающий на место последней (или просто одной) команды перехода (сам адрес перехода располагается в ячейке A-2). Fixup записывает на место фиктивных адресов текущее значение программного счетчика времени компиляции (переменная PC модуля OGen). Процедуру Fixup имеет смысл разместить в модуле OGen.

Завершение генерации

Завершающим шагом в формировании машинного кода будет генерация команд, останавливающих программу. При распознавании точки в конце транслируемого модуля запишем в последовательность машинных команд константу 0, которая будет служить кодом нормального завершения программы, и команду

STOP. Необходимые действия по генерации добавляются в распознаватель модуля (листинг 3.65).

Назначение адресов переменным

Машинный код для программы сформирован. Известен его размер. Он равен значению программного счетчика времени компиляции (PC), которое счетчик примет после генерации последней команды программы — команды STOP.

Теперь самое время вспомнить о необходимости назначения адресов переменным, поскольку только по завершении генерации команд, то есть после формирования команды STOP, стал известен адрес той ячейки памяти, начиная с которой можно разместить переменные. Этот адрес равен значению PC.

Всем переменным программы назначим последовательные адреса, начиная с текущего значения PC. Для каждой переменной языка «О» требуется ровно одна ячейка памяти. Перечень всех переменных можно получить, просматривая таблицу имен.

Назначив переменной адрес, необходимо занести этот адрес во все места машинной программы, где должна быть ссылка на эту переменную. Список адресов ячеек памяти, в которые должен быть записан адрес данной переменной, можно хранить в самих этих ячейках, подобно тому, как при генерации кода для **IF** запоминалось местоположение безусловных переходов (рис. 3.10).

В роли указателя на цепочку адресов для данной переменной используем поле Val записи об этой переменной в таблице имен. В каждую ячейку, в которую надо записать адрес переменной, будем в ходе генерации команд помещать адрес предыдущей такой же ячейки. Эту работу выполнит процедура GenAddr (листинг 3.64), вызовы которой уже неоднократно использовались, но реализация до настоящего момента оставалась тайной.

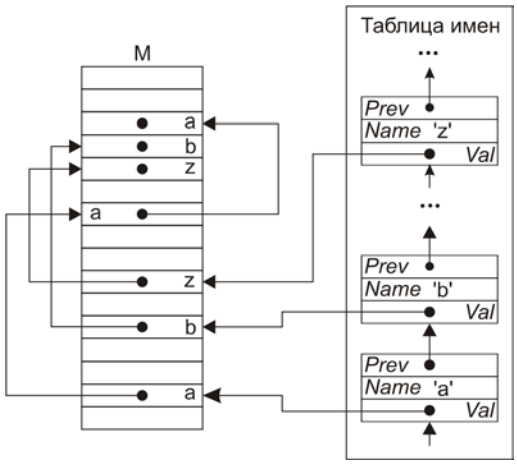


Рис. 3.10. Схема адресной привязки переменных

Листинг 3.64. Формирование цепочки ячеек под видом генерации адреса

```
procedure GenAddr(X: tObj);
begin
   Gen(X^.Val); {В текущую ячейку адрес предыдущей + 2}
   X^.Val := PC+1; {Адрес+2 = PC+1}
end;
```

Чтобы унифицировать подходы, будем хранить в образующих цепочку ячейках не сами адреса, а их значения, увеличенные на 2. Это позволит для прохода по цепочкам с целью окончательной фиксации адресов использовать ту же программу адресной привязки `Fixup` (см. листинг 3.63).

В качестве параметра процедура `GenAddr` получает ссылку `X` на запись о переменной в таблице имен. Значение `X^.Val`, содержащее увеличенный на 2 адрес предыдущей ячейки, где должна быть ссылка на ту же переменную, генерируется в качестве очередной команды. Увеличенный на 2 адрес ячейки, куда эта команда попала, становится новым значением поля `Val` записи о переменной.

Чтобы предлагаемая схема работала, необходимо в качестве начального значения поля `Val` в записи таблицы имен для каждой переменной взять 0. Это обеспечивается при добавлении нового имени процедурой `NewName` (см. листинг 3.23).

Для размещения переменных в памяти нужно по завершении генерации кода просмотреть таблицу имен, каждой имеющейся там переменной назначить адрес на единицу больший адреса предыдущей и записать этот адрес во все ячейки цепочки, относящейся к данной переменной. Эти действия выполнит процедура `AllocateVariables` (allocate variables — разместить переменные) модуля `OGen`. Ее вызов можно видеть в листинге 3.65, а реализацию в листинге 3.66.

Листинг 3.65. Завершение генерации кода

```
procedure Module;
var
   ModRef: tObj; {Ссылка на имя модуля в таблице}
begin
   Check(lexMODULE, 'MODULE');
   …
   Check(lexDot, '".".');
   Gen(0);             {Код возврата}
   Gen(cmStop);        {Команда останова}
   AllocateVariables; {Размещение переменных}
end;
```

Первой переменной назначается адрес, равный значению `PC` после генерации кода. Далее `PC` продолжает увеличиваться и, как всегда, используется в роли программного счетчика периода компиляции, обозначая теперь адрес, назначаемый очередной переменной.

Листинг 3.66. Размещение переменных в памяти (назначение адресов)

```
procedure AllocateVariables;
var
   VRef: tObj; {Ссылка на переменную в таблице имен}
begin
   FirstVar(VRef);          {Найти первую переменную}
   while VRef <> nil do begin
      if VRef^.Val = 0 then
         Warning(
         'Переменная ' + VRef^.Name + ' не используется'
         )
      else begin
         Fixup(VRef^.Val); {Адресная привязка переменной}
         PC := PC + 1;
```

```
      end;
      NextVar(VRef);        {Найти следующую переменную}
   end;
end;
```

Изменение значения поля `Val` в записях о переменных происходит только при вызове `GenAddr`, то есть в момент трансляции фактического использования переменной в программе. Это позволяет обнаруживать неиспользованные переменные, не распределять для них память и выдавать соответствующие предупреждения, если значение `Val` осталось равным 0.

Просмотр таблицы имен с целью поиска в ней записей о переменных выполняется процедурами `FirstVar` и `NextVar` (листинг 3.67) модуля `OTable`. `FirstVar` служит для поиска первой переменной, `NextVar` — следующих. Результат поиска — ссылка `VRef` на запись о переменной.

Листинг 3.67. Просмотр таблицы имен с целью поиска переменных

```
var
   CurrObj: tObj;

procedure FirstVar(var VRef : tObj);
begin
   CurrObj := Top;
   NextVar(VRef);
end;

procedure NextVar(var VRef : tObj);
begin
   while (CurrObj<>Bottom) and (CurrObj^.Cat<>catVar) do
      CurrObj := CurrObj^.Prev;
   if CurrObj = Bottom then
      VRef := nil
   else begin
      VRef := CurrObj;
      CurrObj := CurrObj^.Prev;
   end
end;
```

Если переменная не найдена, выходной параметр `VRef` будет равен **`nil`**. Процедуры `FirstVar` и `NextVar` используют глобальную

(в секции реализации модуля `OTable`) переменную `CurrObj` — указатель на очередной объект таблицы имен.

Трансляция процедур

В языке «О» нет процедур. Есть только возможность вызывать стандартные процедуры и процедуры-функции. Мы не будем «официально» вводить процедуры в язык и не будем переписывать компилятор для языка «О с процедурами». Рассмотрим, не вдаваясь излишне в детали, лишь принципиальные моменты, возникающие при компиляции описаний и вызовов процедур и процедур-функций. Этого должно оказаться достаточно, чтобы при желании самостоятельно реализовать соответствующие механизмы в трансляторе.

Будем считать, что язык «О» дополнен процедурами в соответствии с правилами Оберона при следующих упрощениях:

- Параметры процедур могут быть только целого типа.
- Процедуры не могут быть вложенными.

Будем рассматривать различные варианты трансляции процедур, продвигаясь от простых частных случаев к общему.

Расширенный набор команд виртуальной машины

Для поддержки процедур введем в систему команд ОВМ несколько дополнительных инструкций, обходиться без которых было бы, если не невозможно, то затруднительно. Новые команды, во многом подобные инструкциям реальных процессоров, представлены в таблице 3.4.

Таблица 3.4. Дополнительные команды ОВМ

Код	Обозн.	Название	Стек	Действие
Вызов процедуры и возврат из нее				
-24	`CALL`	Вызов	`A → PC` `(PA → RA)`	`M[SP] ↔ PC`

-25	RET	Возврат	P0, P1, …, Pn-1, RA, n →	PC:= RA; SP:= SP+n+ 2
Выделение и освобождение памяти в стеке				
-26	ENTER	Выделение	n → x1, x2, … xn	SP := SP–n+1
-27	LEAVE	Освобождение	x1, x2, … xn, n →	SP := SP+n+1
Операции с регистром базы				
-28	GETBP	Получить BP	→ BP	M[SP] := BP
-29	SETBP	Установить BP	A →	BP := A
Загрузка и сохранение локальных переменных				
-30	LLOAD	Загрузка лок.	A → M[BP–A]	M[SP]:=M[BP-A]
-31	LSAVE	Сохранение лок.	A, V →	M[BP-A] := V
Получение указателя стека				
-32	SP	Получить SP	→ SP	M[SP-1]:=SP; SP:=SP–1

Кроме дополнительных инструкций в архитектуру ОВМ добавлен регистр базы BP (base pointer — указатель базы). Он предназначается для относительной адресации локальных переменных. Назначение конкретных команд будет разъясняться по мере их использования.

Процедуры без параметров и локальных переменных

Рассмотрим описание процедуры Pr, не имеющей формальных параметров и локальных переменных:

```
PROCEDURE Pr;
BEGIN
   …
END Pr;
```

Такая процедура может оперировать только глобальными переменными. Её операторы транслируются как обычно, поэтому сейчас для нас несущественны и заменены многоточием.

Трансляция заголовка процедуры

При обработке заголовка (`PROCEDURE Pr;`) компилятор должен поместить идентификатор процедуры (`Pr`) в таблицу имен, в пространство имен модуля. В поле значения имени `Val` следует записать текущее значение счетчика команд `PC` — это будет адрес процедуры, то есть номер ячейки, начиная с которой в памяти будут располагаться относящиеся к процедуре машинные команды. Обозначим этот адрес `PA` — адрес процедуры (procedure address).

Трансляция вызова процедуры

Оператор вызова процедуры в рассматриваемом случае состоит из имени этой процедуры:

```
Pr;
```

Код, порождаемый для этого оператора, должен обеспечить переход по адресу процедуры `PA` с последующим возвратом на оператор, следующий за вызовом процедуры. Именно для такой работы и предназначена команда `CALL` — вызов подпрограммы, выполняющая «переход с возвратом».

Команда `CALL` присваивает программному счетчику значение, взятое с вершины стека, обеспечивая тем самым переход на команду по этому адресу. При этом на вершину стека записывается значение, которое имел перед этим программный счетчик. Напомню, что при исполнении данной машинной команды программный счетчик уже указывает на следующую. В результате выполнения `CALL` на вершине стека окажется адрес команды, следующей за ней. Это и будет адрес возврата. Будем обозначать его `RA` (return address — адрес возврата). По завершении работы

процедуры этот адрес, сохраненный на стеке, позволит выполнить возврат к командам вызывающей программы.

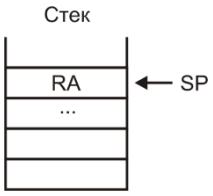
Можно заметить, что действие команды CALL сводится к обмену значений PC и значения с вершины стека, что и отражено в таблице 3.4.

Таким образом, при трансляции вызова процедуры без параметров и локальных переменных компилятор должен поместить в генерируемую последовательность команд адрес процедуры PA и команду CALL:

Исходный текст	Машинный код
Pr;	PA ; Адрес процедуры CALL ; Переход с возвратом

Вход в процедуру и возврат из процедуры

После вызова процедуры, то есть перехода по адресу PA на вершине стека находится адрес возврата. При таком состоянии стека процедура начинает работать.



В рассматриваемом простейшем случае нет необходимости формировать какие-либо специальные команды, которые будут выполняться при входе в процедуру.

Операторы, составляющие процедуру, транслируются как обычно.

Каждый оператор оставляет стек в том же состоянии, что и получил. Поэтому после выполнения последнего оператора процедуры стек будет в том же состоянии, что и сразу после ее вызова. На вершине стека будет располагаться RA. Чтобы обеспечить

возврат по адресу, находящемуся на вершине стека, достаточно сгенерировать в конце кода процедуры (при распознавании **END**) команду GOTO.

Исходный текст	Машинный код
PROCEDURE Pr;	PA) …
BEGIN	…
…	…
END Pr;	GOTO

Процедуры с параметрами-значениями без локальных переменных

Рассмотрим теперь процедуру с *n* параметрами-значениями следующего вида:

```
PROCEDURE Pr(P0, P1, …, Pn-1: INTEGER);
BEGIN
   …
END Pr;
```

Первый вопрос, который нужно решить — это способ размещения параметров процедуры в памяти.

Распределение памяти для формальных параметров и локальных переменных

Можно предложить несколько способов размещения в памяти параметров и локальных переменных процедур.

Статическое распределение памяти

В этом случае каждому параметру каждой процедуры и каждой локальной переменной[35] может быть отведена собственная постоянная ячейка памяти. При этом может использоваться тот же

[35] С точки зрения распределения памяти и организации доступа к ней нет существенной разницы между формальными параметрами и локальными переменными, особенно если речь идет о параметрах-значениях.

способ назначения адресов, который применялся для глобальных переменных. Ниже приведен эскиз модуля, содержащего две процедуры, и распределение памяти под переменные и параметры.

Исходный текст	Машинный код
MODULE M;	
VAR X : INTEGER;	
PROCEDURE Pr1(P1: INTEGER);	
BEGIN … **END** Pr1;	…
PROCEDURE Pr2(P2: INTEGER);	
BEGIN … **END** Pr2;	…
BEGIN	
…	…
END M.	STOP
	X
	P1
	P2

При статическом распределении параметры процедур и локальные переменные занимают место в памяти даже тогда, когда соответствующая процедура не исполняется.

Такой способ распределения памяти использовался в ранних версиях Фортрана. К его преимуществам можно отнести эффективность доступа к данным и простоту. Недостатки — увеличенный расход памяти и невозможность рекурсии.

Распределение памяти в стеке

Это наиболее распространенный способ размещения локальных переменных и параметров процедур. Он позволяет реализовать рекурсию, не требует выделения памяти под данные тех процедур, которые не исполняются в данный момент. Этот вариант и будет использован.

Трансляция вызова процедуры с параметрами-значениями

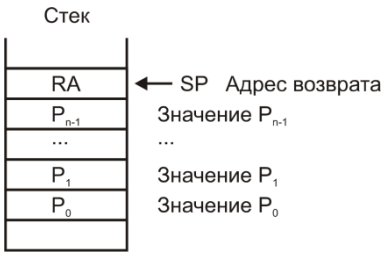
При трансляции вызова процедуры с *n* параметрами-значениями

```
Pr(P0, P1, …, Pn-1);,
```

где P0, P1, …, Pn-1 — выражения, определяющие фактические параметры, компилятор сформирует машинный код для вычисления каждого параметра. Далее генерируется адрес процедуры и команда перехода с возвратом:

```
<Код, вычисляющий P0>
<Код, вычисляющий P1>
…
<Код, вычисляющий Pn-1>
PA   ;Адрес процедуры
CALL ;Вызов
```

Этот код обеспечит при своем выполнении вычисление выражений фактических параметров, их значения останутся в стеке. После этого команда CALL поместит на вершину стека адрес возврата RA:



В таком состоянии стек будет передан вызванной процедуре.

Доступ к параметрам процедуры

Для загрузки и сохранения параметров будут применяться команды локальной загрузки LLOAD и сохранения LSAVE. Они используют относительную адресацию с помощью регистра BP (см. таблицу 3.4). Параметр, который эти команды берут с вершины стека, означает величину смещения вверх (в сторону меньших адресов) относительно значения, записанного в базовый регистр BP. Если в регистр базы записать адрес P0, то загрузка параметра Pi может быть выполнена с помощью следующей пары команд:

```
i     ;Относительный адрес
LLOAD ;Команда загрузки по относительному адресу
```

Такое соглашение мы и будем использовать. Регистр базы будет указывать на первую, считая от дна стека, ячейку памяти, относящуюся к процедуре. Участок стека, хранящий данные процедуры, обычно называют кадром (stack frame).

Трансляция описания процедуры

После распознавания слова **PROCEDURE,** распознавания имени процедуры (например, `Pr`) и занесения этого имени в пространство глобальных имен транслятор должен открыть с помощью `OpenScope` новый блок в таблице имен — блок локальных данных процедуры.

При трансляции формальных параметров их идентификаторы заносятся в таблицу имен, а в поле значения `Val` записывается порядковый номер параметра (от 0 до *n*-1), который будет служить его относительным адресом.

Вход в процедуру

В случае, когда у процедуры имеются параметры и, значит, будет использована адресация относительно регистра базы, компилятор должен сгенерировать код, который будет выполняться в начале работы процедуры (при входе в процедуру) и обеспечит установку регистра `BP` на дно кадра стека[36]. Код, который выполняет подготовительные действия перед выполнением операторов, составляющих процедуру, часто называют *прологом*. Можно считать, что пролог порождается компилятором при распознавании слова **BEGIN**.

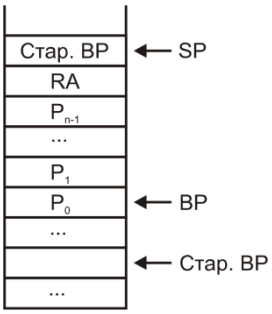
[36] «Установить регистр на…» означает: «занести в регистр значение, равное адресу нужной ячейки».

Поскольку одна процедура может вызываться из другой[37], то при возврате из вызванной процедуры регистр базы должен снова указывать на кадр стека вызывающей. Следовательно, перед тем как BP будет установлен в новое значение при входе в процедуру, его старое значение, соответствующее среде вызывающей процедуры, должно быть запомнено с тем, чтобы перед выходом его можно было восстановить. Запоминать старое значение BP, естественно, следует на стеке.

Таким образом, в прологе должны быть выполнены (и запрограммированы компилятором) следующие действия:

1. Запомнить на стеке старое (текущее) значение регистра базы BP.
2. Записать в регистр базы адрес параметра P0.

В результате стек должен оказаться в таком состоянии:



Запоминание старого значения BP выполняется просто одной командой GETBP. А вот чтобы записать в регистр базы адрес первого параметра процедуры, его придется вычислить, прибавив к значению указателя стека SP (он в этот момент указывает на расположенное на вершине стека старое значение BP) величину n+1 — n параметров плюс ячейка, занятая адресом возврата RA. В результате пролог получается таким:

[37] Речь идет о вложенности *вызовов* процедур (одна процедура может вызывать другую), но не о вложенности самих процедур, которой в нашем языке нет.

```
GETBP ;Сохранить старое значение BP
SP    ;Значение SP на стек
n+1   ;Это известная компилятору константа
ADD
SETBP ;Установить новое значение BP
```

Выход из процедуры

Перед возвратом из процедуры стек будет в том же состоянии, как и сразу после выполнения пролога (см. схему выше). Задача эпилога (фрагмента кода, выполняемого непосредственно перед возвратом из процедуры) состоит в следующем:

1. Восстановить старое значение BP (командой SETBP).
2. Перейти по адресу RA (который после выполнения п. 1 находится на вершине стека), удалив из стека *n* параметров. Для выполнения именно таких действий предназначена команда RET (см. табл. 3.4). В качестве параметра RET получает на вершине стека количество снимаемых со стека значений (не считая RA).

Эпилог, который должен быть сгенерирован компилятором:

```
SETBP  ;Восстановить BP
n      ;Известная компилятору константа
RET    ;Возврат из процедуры
```

При выходе из процедуры компилятор также должен закрыть с помощью CloseScope область видимости локальных данных процедуры.

Процедуры с параметрами-значениями и локальными переменными

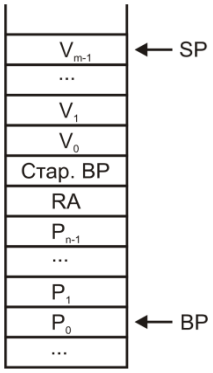
Пусть *n* — число формальных параметров-значений, *m* — количество локальных переменных, а процедура имеет такой вид:

```
PROCEDURE Pr(P0, P1, …, Pn-1: INTEGER);
VAR
   V0, V1, …, Vm-1 : INTEGER;
BEGIN
   …
END Pr;
```

Трансляция заголовка процедуры в этом случае происходит так же, как и в предыдущем.

Трансляция описаний локальных переменных

Задача компилятора при трансляции локальных переменных — разместить их в памяти, т.е. назначить адреса. Значения фактических параметров и адрес возврата записываются в стек при вызове процедуры. Локальные переменные можно разместить в стеке поверх адреса возврата и сохраненного старого значения `BP`, как показано на схеме.



По мере распознавания идентификаторов локальных переменных компилятор назначает им относительные адреса, начиная со значения n+2. То есть, переменная `Vi` (i = 0..n–1) получает адрес i+n+2. Назначенные адреса записываются в поле значения `Val` таблицы имен.

Доступ к локальным переменным для чтения и записи полностью аналогичен доступу к параметрам-значениям и выполняется по относительным адресам с помощью команд `LLOAD` и `LSAVE`.

Вход в процедуру

Резервирование памяти для локальных переменных предусмотрим в прологе. Вначале выполняются те же действия, что в предыдущем случае: сохранение старого и вычисление нового значения `BP`. Затем с помощью команды `ENTER` указатель стека под-

нимается на *m* ячеек вверх и тем самым резервируется место для локальных переменных:

```
GETBP ;Сохранить старое значение BP
SP
n+1
ADD
SETBP ;Установить новое значение BP
m     ;Зарезервировать место для
ENTER ;m переменных
```

Выход из процедуры

Последовательность действий, совершаемых при выходе из процедуры, определяется состоянием стека на этот момент. Это состояние показано на схеме выше. Компилятор должен запрограммировать следующее:

1. Освободить память, занятую локальными переменными. Реализуется командой `LEAVE`, которая выполняет действия обратные `ENTER`.
2. Восстановить `BP`.
3. Возвратиться в вызывающую программу с удалением *n* параметров из стека с помощью команды `RET`.

Код эпилога получается таким:

```
m     ;Известная компилятору константа
LEAVE ;Удалить переменные
SETBP ;Восстановить BP
n     ;Удаление n параметров и
RET   ;возврат
```

Простейшая оптимизация кода

В ряде частных случаев генерации кода для пролога и эпилога универсальные последовательности команд могут быть заменены более короткими. Некоторые такие оптимизации представлены в таблице 3.5.

Таблица 3.5. Оптимизация кода пролога и эпилога

Неоптимизированный код	Оптимизированный код
1 ENTER	0
1 LEAVE	DROP
0 RET	GOTO

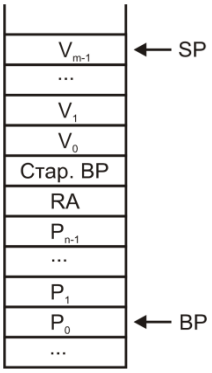
Процедуры-функции с параметрами-значениями и локальными переменными

Процедуры-функции вызываются при вычислении выражений. Следуя алгоритму вычисления выражений, представленных в обратной польской записи, функция должна заменять на стеке значения своих аргументов вычисленным значением функции.

Пусть процедура-функция `Pr` имеет *n* аргументов, использует *m* локальных переменных и ее описание имеет такой вид:

```
PROCEDURE Pr(P0, P1, …, Pn-1: INTEGER): INTEGER;
VAR
   V0, V1, …, Vm-1 : INTEGER;
BEGIN
   …
END Pr;
```

Сразу после входа и выполнения пролога стек, переданный функции, имеет такой вид:



Возвращая управление вызывающей программе, функция должна поместить вычисленное значение на дно стекового кадра, то есть в ячейку, занятую параметром `P0` (ячейку, адрес которой хранится в `BP`). После возврата из функции эта ячейка должна стать вершиной стека вызывающей программы, то есть не должна освобождаться командой `RET`. Это достигается уменьшением на единицу параметра команды `RET`: вместо пары команд

```
n
RET
```

должна быть сгенерирована последовательность

```
n-1
RET
```

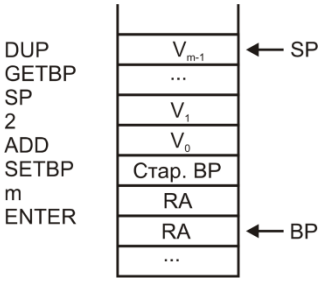
Отдельно нужно рассмотреть случай, когда функция не имеет параметров (*n*=0). Если не принять специальных мер, то в этой ситуации ячейки, куда следует записать вычисленное значение функции, просто не будет: параметр `P0` отсутствует.

Вход в процедуру-функцию

При наличии параметров вход в процедуру-функцию не отличается от обычного.

При *n*=0 можно было бы в вызывающей программе предусмотреть запись в стек фиктивного фактического параметра, например числа 0. Более изящное решение состоит в том, что в пролог добавляется команда `DUP`, которая дублирует значение адреса возврата `RA`. Верхняя копия `RA` исполняет свою основную роль,

нижняя — резервирует место для значения функции. Таким образом, при n=0 (и $m \neq 0$) пролог и стек после его выполнения будут такими:



Выход из процедуры-функции

Ниже в левой колонке приведен код, порождаемый компилятором для эпилога функции, имеющей больше одного аргумента ($n > 1$), в правой колонке — для случая, когда $n = 0$ или $n = 1$.

```
M
LEAVE
SETBP ;Восстановить BP
n-1   ;На 1 меньше
обычного
RET
```

```
m
LEAVE
SETBP ;Восстановить BP
GOTO  ;возврат
```

В обоих вариантах нижняя ячейка стекового кадра передается вызывающей программе. Можно считать, что приведенный эпилог порождается компилятором при распознавании слова `END`, завершающего процедуру-функцию. В обычной ситуации выполнение функции не должно доходить до `END`, поскольку в этом случае значение функции не будет определено[38]. Вычисление функции должно завершаться оператором `RETURN`.

[38] Можно заметить, что при нашей реализации значением функции в этом случае будет либо адрес возврата (если параметров нет), либо значение первого параметра (если он есть). Надеюсь, что никому из программирующих на языке «О с процедурами» не придет в голову использовать эту особенность реализации.

Трансляция оператора RETURN

В собственно процедурах (не функциях) оператор **RETURN** транслируется так же, как **END**, завершающий процедуру. То есть компилятор либо порождает эпилог, либо может генерировать безусловный переход на эпилог, размещенный в конце кода процедуры, либо может не порождать никакого кода, если **RETURN** записан непосредственно перед **END.**

При использовании внутри функции оператор возврата записывается в такой форме:

RETURN <Выражение>

При его трансляции нужно обеспечить вычисление выражения, запись его значения в ячейку результата, на которую указывает BP, и обычный возврат из функции. Код получается таким:

```
GETBP ;Адрес ячейки результата
<Код для Выражения>
SAVE  ;Сохранить результат
m     ;Если есть
LEAVE ;локальные переменные
SETBP ;Восстановить BP
n-1   ;Если параметров больше одного
RET   ;Возврат
```

Особенность трансляции параметров-переменных

При трансляции вызовов процедур, имеющих параметры-значения и параметры-переменные, компилятор может использовать контекстную информацию для выбора одного из вариантов распознавания фактических параметров. Если очередной формальный параметр — переменная, то соответствующий фактический должен также быть переменной — вызывается распознаватель переменной. Если следует параметр-значение, вызывается распознаватель выражений.

Подстановка фактического параметра вместо параметра-переменной

Примем соглашение. Для формального параметра-переменной в стек при вызове процедуры всегда записывается **абсолютный адрес** фактического параметра.

Пусть заголовок вызываемой процедуры имеет вид:

```
PROCEDURE Pr(VAR V: INTEGER);
```

а вызов этой процедуры таков:

```
Pr(X);
```

Рассмотрим несколько случаев генерации кода для вызова `Pr` в зависимости от того, что такое переменная `X`.

1. `X` — глобальная переменная. Тогда задача решается непосредственно. Для переменной (как в компиляторе языка «О» без процедур) генерируется ее (абсолютный) адрес:

```
X    ;Абсолютный адрес X
Pr   ;Адрес процедуры Pr
CALL ;Вызов
```

Здесь и далее адрес (абсолютный или относительный) объекта программы (переменной, процедуры) будем обозначать при записи фрагментов машинного кода просто именем этого объекта. В реальный код адреса глобальных переменных компилятор записывает, используя алгоритм, рассмотренный выше в разделе «Назначение адресов переменным». Адреса процедур, их параметров и локальных переменных извлекаются из таблицы имен, где они хранятся в поле значения `Val` записи о соответствующем объекте.

2. `X` — локальная переменная или формальный параметр-значение. В этом случае абсолютный адрес ячейки, где хранится значение `X`, вычисляется как разность содержимого регистра базы `BP` и относительного адреса `X`.

```
GETBP ;Значение регистра базы
X     ;Относительный адрес X
```

```
SUB   ;Теперь на стеке абсолютный адрес X
Pr    ;Адрес процедуры Pr
CALL  ;Вызов
```

3. X — параметр-переменная. Значит, в ячейке с известным относительным адресом, соответствующей формальному параметру X, хранится абсолютный адрес фактического параметра. Используя команду локальной загрузки, получаем на стеке этот абсолютный адрес:

```
X     ;Относительный адрес
LLOAD ;Теперь на стеке абсолютный адрес
Pr    ;Адрес процедуры Pr
CALL  ;Вызов
```

Доступ к параметрам-переменным

Извлечение (загрузка на вершину стека) значения параметра-переменной сводится к получению абсолютного адреса фактического параметра и последующему применению команды абсолютной загрузки LOAD:

```
V     ;Относительный адрес
LLOAD ;Теперь на стеке абсолютный адрес
LOAD  ;На стеке значение
```

Если значение параметру-переменной нужно присвоить, т.е. выполнить действие

```
V := E,
```

где E — выражение, то вначале должен быть получен абсолютный адрес фактического параметра, затем вычисляется выражение E (его значение остается на стеке), после этого можно применить команду SAVE, использующую абсолютную адресацию:

```
X           ;Относительный адрес
LLOAD       ;Теперь на стеке абсолютный адрес
<Код для E>
SAVE        ;Сохранить
```

Пример программы на языке «О с процедурами»

Для иллюстрации возможностей языка, трансляцию которого мы подробно обсудили, приведу пример законченной программы на

«О с процедурами». Программа (листинг 3.68) решает известную головоломку «Ханойские башни»: требуется перенести *n* дисков со стержня 1 на стержень 2, используя стержень 3 в качестве вспомогательного (рис. 3.11).

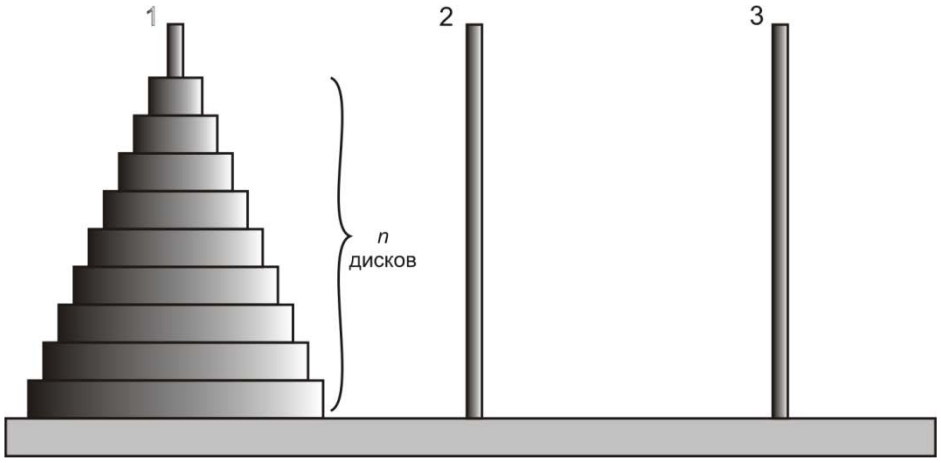


Рис. 3.11. Головоломка «Ханойские башни»

Можно переносить по одному диску. Больший диск никогда не должен находиться поверх меньшего.

Листинг 3.68. «Ханойские башни» на «О с процедурами»

```
MODULE Towers;
IMPORT In, Out;

VAR
   n : INTEGER;

PROCEDURE WriteLn(X, Y: INTEGER);
BEGIN
   Out.Int(X, 1); Out.Int(Y, 2); Out.Ln;
END WriteLn;

PROCEDURE Hanoy(n, X, Y, Z: INTEGER);
BEGIN
   IF n > 0 THEN
      Hanoy(n-1, X, Z, Y);
      WriteLn(X, Y);
      Hanoy(n-1, Z, Y, X);
   END;
END Hanoy;
```

```
BEGIN
   In.Open;
   In.Int(n);
   Hanoy(n, 1, 2, 3);
END Towers.
```

Программа печатает решение задачи в виде последовательности пар чисел. Первое число пары обозначает стержень, с которого надо взять диск, второе — стержень, на который перенести. Вот пример выполнения программы для *n*=3:

```
?3
1 2
1 3
2 3
1 2
3 1
3 2
1 2
```

В листинге 3.69 можно видеть код виртуальной машины, который будет сформирован компилятором для программы «Ханойские башни». Участки кода записаны за теми фрагментами исходной программы, которые этот код порождают. Строки исходного кода оформлены как комментарии.

Листинг 3.69. Результат компиляции программы «Ханойские башни»

```
;MODULE Towers;
;IMPORT In, Out;

;VAR
;   n : INTEGER;

;PROCEDURE WriteLn(X, Y: INTEGER);
0)  62
1)  GOTO  ; В обход процедур

;BEGIN
2)  GETBP ; Сохранить стар. BP
3)  SP
4)  3     ; Потому что параметров 2
5)  ADD
6)  SETBP ; Установить BP = SP+3

;   Out.Int(X, 1); Out.Int(Y, 2); Out.Ln;
```

```
7)  0     ; Относительный адрес X
8)  LLOAD ; X
9)  1     ; X, 1
10) OUT
11) 1     ; Относительный адрес Y
12) LLOAD ; Y
13) 2     ; Y, 2
14) OUT
15) OUTLN

;END WriteLn;
16) SETBP ; Восстановить BP
17) 2     ; Удалить 2 параметра
18) RET   ; Возврат из WriteLn

;PROCEDURE Hanoy(n, X, Y, Z: INTEGER);
;BEGIN
19) GETBP ; Сохранить стар. BP
20) SP
21) 5
22) ADD
23) SETBP ; Установить BP = SP+5 (4 параметра)

;   IF n > 0 THEN
24) 0     ; - Относительный адрес n
25) LLOAD ; n
26) 0     ; n, 0
27) 59
28) IFLE

;      Hanoy(n-1, X, Z, Y);
29) 0     ; - Относительный адрес n
30) LLOAD ; n
31) 1     ; n, 1
32) SUB   ; n-1
33) 1     ; - Относительный адрес X
34) LLOAD ; n-1, X
35) 3     ; - Относительный адрес Z
36) LLOAD ; n-1, X, Z
37) 2     ; - Относительный адрес Y
38) LLOAD ; n-1, X, Z, Y
39) 19    ; - Адрес процедуры Hanoy
40) CALL

;      WriteLn(X, Y);
41) 1     ; - Относительный адрес X
42) LLOAD ; X
43) 2     ; - Относительный адрес Y
```

```
44) LLOAD ; X, Y
45) 2     ; - Адрес процедуры WriteLn
46) CALL

;      Hanoy(n-1, Z, Y, X);
47) 0     ; Относительный адрес n
48) LLOAD ; n
49) 1     ; n, 1
50) SUB   ; n-1
51) 3     ; - Относительный адрес Z
52) LLOAD ; n-1, Z
53) 2     ; - Относительный адрес Y
54) LLOAD ; n-1, Z, Y
55) 1     ; Относительный адрес X
56) LLOAD ; n-1, Z, Y, X
57) 19    ; - Адрес процедуры Hanoy
58) CALL

;   END;
;END Hanoy;
59) SETBP ; Восстановить BP
60) 4     ; Удалить 4 параметра
61) RET   ; Возврат

;BEGIN
;   In.Open;
;   In.Int(n);
62) 74    ; Адрес глобальной n
63) IN    ; 74, n
64) SAVE

;   Hanoy(n, 1, 2, 3);
65) 74    ; Адрес n
66) LOAD  ; n
67) 1
68) 2
69) 3
70) 19    ; Адрес процедуры Hanoy
71) CALL

;END Towers.
72) 0
73) STOP
```

Конструкция простого ассемблера

Обсуждая генерацию кода компилятором языка «О», мы постоянно использовали мнемоническую запись команд машинного

кода. Однако такая запись рассматривалась лишь как условная, предназначенная для внешнего представления машинных команд. Правила этой записи не были строго определены и время от времени менялись, поскольку не предполагалось, что программы, записанные таким мнемоническим кодом, могут быть непосредственно введены в компьютер и выполнены. Между тем, для этого нет принципиальных препятствий. Нужно лишь формализовать правила записи и написать программу или программы, которые бы выполняли преобразование мнемонического кода в числовой, обеспечивали его загрузку в память ОВМ и запуск. Получится *ассемблер*.

Слово «ассемблер» используют в двух различных смыслах. Во-первых, ассемблер — это программа-транслятор, преобразующая мнемокод в машинные команды. Во-вторых, ассемблером называют сам язык мнемонических команд. Правильнее было бы во втором случае говорить о «языке ассемблера», но сложившуюся практику не отменишь.

Язык ассемблера виртуальной машины

Сформулируем правила языка ассемблера ОВМ. Многие из них будут не новы, поскольку уже использовались при мнемонической записи фрагментов машинного кода.

- Команды виртуальной машины имеют мнемонические обозначения, приведенные в таблицах 3.3 и 3.4. Названия команд всегда записываются заглавными буквами. Заглавные и строчные буквы считаются различными.
- В роли команд могут использоваться целые константы без знака.
- В программу могут быть включены комментарии. Все символы от точки с запятой, включая ее саму, до конца строки являются комментарием и не влияют на смысл программы.

- Строки программы могут быть помечены, в роли меток используются идентификаторы (имена), за которыми следует двоеточие.
- Имена меток могут использоваться в качестве команд для ссылки на ячейки памяти, соответствующие помеченным строкам. Метка-команда означает адрес ячейки и не сопровождается двоеточием.
- Строка программы может содержать не больше одной команды.
- При записи программы могут использоваться пробелы и пустые строки. Пробелы и разделители строк не должны встречаться внутри обозначений команд и меток.

Пример программы на ассемблере

В листинге 3.70 приведена записанная на ассемблере программа нахождения наибольшего общего делителя двух натуральных чисел, эквивалентная программе из листинга 3.40.

Листинг 3.70. Нахождение НОД(X, Y). Программа на ассемблере ОВМ.

```
;НОД по алгоритму Евклида

      IN  ; X
      IN  ; X, Y

Loop: OVER   ; X, Y, X
      OVER   ; X, Y, X, Y
      Quit
      IFEQ   ; X, Y    На выход (Quit), если X=Y
      OVER   ; X, Y, X
      OVER   ; X, Y, X, Y
      NoSwap
      IFLT   ; X, Y    В обход SWAP, если X>Y
      SWAP   ; Y, X    На вершине большее
NoSwap:
      OVER   ; Min(X, Y), Max(X, Y), Min(X, Y)
      SUB    ; Новое X, Новое Y
      Loop
      GOTO   ; X, Y    На начало цикла

Quit: DROP   ; X       Одно значение было лишним
```

```
0       ; X, 0
OUT
OUTLN
STOP
```

Основное отличие этого текста от приведенного в листинге 3.40 — использование меток и отказ от записи числовых адресов. Это значительно облегчает разработку программы, избавляя от необходимости вести счет номеров ячеек. Возможность применять символические метки — ключевое усовершенствование языка ассемблера. При этом важно заметить, что метки в программе из листинга 3.70 означают в точности то же, что и соответствующие адреса в программе из листинга 3.40. Можно даже говорить, что значением метки `Loop` является адрес 2, метки `Quit` — 15, а `NoSwap` — 11.

Формальный синтаксис языка ассемблера

Запишем правила языка ассемблера на РБНФ:

```
Программа = Строка {перевод_строки Строка}.
Строка = [Метка][Число|Имя|Код].
```

Это синтаксическая грамматика ассемблера. Простейшие элементы программы: метки, числа, имена и коды команд, как и в случае языка высокого уровня, будем называть лексемами. Их вид определяется лексической грамматикой:

```
Метка = буква {буква | цифра} ":".
Число = цифра {цифра}.
Имя = буква {буква | цифра}.
Код = Имя.
```

Синтаксически понятия «Метка» и «Имя» разделены. Но содержательно они тесно связаны. Метка определяет адрес некоторой точки в программе, имя представляет собой ссылку на этот адрес. В дальнейшем метку с двоеточием будем называть также определяющим вхождением имени, одноименную ссылку без двоеточия — использующим вхождением.

Важно понимать, что метки могут употребляться не только для условных и безусловных переходов, но и для обозначения ячеек памяти, когда эти ячейки используются как хранилища данных, то есть в роли переменных.

Рассмотрим простейшую программу, которая складывает два введенных числа (листинг 3.71). Введенные значения вначале записываются в память, а потом извлекаются оттуда и суммируются. Вообще-то, действовать так нет особенной нужды, можно было бы, поместив числа на стек при вводе, тут же их сложить. Но в этом примере нам важно, чтобы программа обращалась к памяти.

Листинг 3.71. Резервирование памяти в программе на ассемблере

```
; Сложение двух чисел
   X    ; Адрес X
   IN   ; Ввести
   SAVE ; Сохранить по адресу X
   Y    ; Адрес Y
   IN   ; Ввести
   SAVE ; Сохранить по адресу Y
   X
   LOAD ; Загрузить X
   Y
   LOAD ; Загрузить Y
   ADD  ; Сложить
   0    ; Это формат вывода
   OUT  ; Вывести
   OUTLN
   STOP
X: 0    ; Резервирование места для X
Y: 0    ; Резервирование места для Y
```

Самое важное здесь — метки `X` и `Y`. Они помечают ячейки, в которые записан 0 и которые расположены сразу за ячейкой, где хранится команда `STOP`. Заносить именно ноль туда, где будут храниться слагаемые вообще-то совсем не обязательно. Начальные значения никак этой программной не используются и записывать их в ячейки `X` и `Y` потребовались лишь для того, чтобы за-

резервировать место. Другого способа занять две ячейки сразу за кодом и пометить их наш ассемблер не предоставляет. Вообще-то, в ячейку Y можно было ничего первоначально не записывать. Концовка программы могла быть такой:

```
    …
    STOP
X:  0     ; Резервирование места для X
Y:        ; Резервирование места для Y
```

И в этом случае значением метки `Y` будет адрес той же ячейки.

Программируя на ассемблере, совсем не обязательно размещать переменные сразу за командой `STOP`. С таким же успехом можно спланировать расположение переменных за любой командой безусловного перехода `GOTO` или возврата `RET`.

Наконец, возможно[39] использование нестандартных приемов при размещении переменных. Так, в примере про сложение можно назначить для хранения X[40] первую ячейку из занимаемых кодом программы, а для Y — вторую (листинг 3.72).

Листинг 3.72. Трюки при резервировании памяти в программе на ассемблере

```
;!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
;!! Не пробуйте повторить! Опасно !!
;!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

X: X     ; Адрес X
Y: IN    ; Сначала здесь команда, потом Y
   SAVE  ; Сохранить по адресу X
   Y     ; Адрес Y
   IN    ; Ввести
   SAVE  ; Сохранить по адресу Y
   X
   LOAD  ; Загрузить X
   Y
```

[39] «Возможно» — не значит «рекомендуется».

[40] При таком использовании в программе про X и Y удобней говорить как про переменные, хотя такого понятия в языке ассемблера «официально» не существует.

```
LOAD ; Загрузить Y
ADD  ; Сложить
0    ; Это формат вывода
OUT  ; Вывести
OUTLN
STOP
```

Дело в том, что команда (адрес X), находящаяся в ячейке, предназначенной для хранения X, будет при работе программы выполнена до того, как в эту ячейку будет записано введенное значение. То же относится и к команде `IN`, на место которой будет записано второе введенное значение.

Получилась программа, которая в ходе исполнения модифицирует себя. Нет никаких причин использовать подобные приемы. Они лишь увеличивают риск ошибок. Экономия нескольких ячеек памяти не стоит того. Отрицательным свойством самомодифицирующейся программы является и ее нереентерабельность[41]. Такая программа может быть выполнена лишь один раз после загрузки в память. Попытка ее повторного выполнения без перезагрузки кода приведет к ошибке. Этот пример приведен здесь лишь для иллюстрации средств языка ассемблера.

Программа, печатающая себя

В качестве еще одного примера программирования на ассемблере рассмотрим программу, печатающую саму себя (листинг 3.73). Речь, правда, не идет об известном сюжете — интроспективной программе, печатающей собственный *исходный* текст. В нашем примере печатается *машинный код*, то есть числовые коды команд, составляющих программу.

Листинг 3.73. Программа, печатающая свой машинный код

```
;Метка Код     Стек
Begin: Begin ; Begin
Loop:  DUP   ; Begin, Begin
```

[41] Реентерабельность — свойство программы, позволяющее повторный вход в нее.

```
        LOAD  ; Begin, M[Begin]
        0
        OUT
        OUTLN ; Begin
        1
        ADD   ; Begin+1
        DUP   ; Begin+1, Begin+1
        End   ; Begin+1, Begin+1, End
        Loop
        IFLE
        DROP
End:    STOP
```

Комментарии, показывающие состояние стека, относятся к первому проходу цикла.

Это еще один пример нетривиального использования меток. Метка `Loop` используется для перехода, `Begin` и `End` отмечают начало и конец того участка кода, который будет напечатан[42]. Адреса, соответствующие `Begin` и `End`, участвуют в арифметических операциях и сравнениях. В таких случаях говорят об использовании адресной арифметики. В языках, предназначенных для надежного программирования, она не разрешена. Ассемблер относится к другому классу.

Реализация ассемблера

Приступим к разработке программы-ассемблера, т.е. транслятора с языка ассемблера в код ОВМ. Как и в случае с компилятором «О», ассемблер будет помещать формируемый код прямо в память виртуальной машины. После этого полученная программа сразу запускается на исполнение (в случае успешной компиляции, разумеется).

Мы будем использовать классическую двухпроходную схему ассемблирования. Хотя ресурсы современного компьютера по-

[42] Можно заметить, что такая программа может печатать не обязательно свой код, а содержимое любого участка памяти от ячейки, помеченной `Begin`, до ячейки, имеющей метку `End`.

зволяют обойтись и одним проходом при трансляции с ассемблера, использование двух проходов для языка, в котором определение имени не обязано предшествовать использованию, оказывается естественней и проще.

Главная программа и вспомогательные модули

Модульная структура ассемблера будет во многом повторять конструкцию компилятора «О». Предусматриваются драйвер исходного текста (`AsmText`), модуль для работы с таблицей имен (`AsmTable`), лексический анализатор (`AsmScan`). Это вспомогательные модули. За собственно ассемблирование будет отвечать `AsmUnit`, объединяющий функции распознавателя и генератора кода. Модуль виртуальной машины используется, разумеется, без всяких изменений. Отвечающий за реакцию на ошибки, модуль `OError` также применен неизменным. Главная программа (листинг 3.74) организует всю работу.

Листинг 3.74. Главная программа ассемблера

```
program OASM;
{Ассемблер виртуальной машины OBM}
uses
   AsmText, AsmScan, AsmTable, AsmUnit, OVM;
procedure Init;
begin
   AsmTable.InitNameTable;
   AsmText.OpenText;
   AsmScan.InitScan;
end;

procedure Done;
begin
   AsmText.CloseText;
end;

begin
   WriteLn('Ассемблер виртуальной О-машины');
   Init;
   AsmUnit.Assemble; {Компилировать}
   OVM.Run;
   Done;
end.
```

Чтобы структура обсуждаемой программы была понятней, я счел необходимым записать вызовы процедур вспомогательных модулей с указанием имени модуля.

Хотя функции драйвера исходного текста, сканера и модуля работы с таблицей остаются в основном теми же, что и в компиляторе языка «О», специфика ассемблера требует внесения изменений в интерфейс этих модулей.

Поскольку транслятор будет двухпроходным, нужна возможность читать исходный текст программы дважды. Для этого в драйвере исходного текста (листинг 3.75) наряду с процедурой `ResetText` предусматривается процедура `OpenText`. `OpenText` должна быть вызвана один раз, чтобы открыть текст (обычно это файл). `ResetText` вызывается, всякий раз, когда необходимо читать исходный текст с начала, то есть перед каждым проходом ассемблера.

Листинг 3.75. Интерфейс драйвера исходного текста

```
unit AsmText;
{Драйвер исходного текста}

interface

const
   chSpace  = ' ';
   chTab    = chr(9);
   chEOL    = chr(13);
   chEOT    = chr(0);

var
   Ch    : char;
   Pos   : integer;
   Line  : integer;

procedure OpenText;
procedure ResetText;
procedure CloseText;
procedure NextCh;

{=====================================================}
```

Сканер для ассемблера (листинг 3.76) оказывается проще, чем для языка высокого уровня, поскольку разновидностей лексем в ассемблере намного меньше.

Листинг 3.76. Интерфейс модуля сканера

```
unit ASMScan;
{Сканер для ассемблера}

interface

const
   NameLen = 31;

type
   tName = string[NameLen];
   tLex = (
      lexLabel,  {Метка}
      lexOpCode, {Код операции}
      lexNum,    {Число}
      lexName,   {Имя}
      lexEOL,    {Конец строки}
      lexEOT     {Конец текста}
   );

var
   Lex      : tLex;    {Текущая лексема}
   Num      : integer; {Значение числа}
   OpCode   : integer; {Значение кода операции}
   Name     : tName;   {Строка имени}

   LexPos   : integer;

procedure InitScan;
procedure NextLex;

{=====================================================}
```

Под лексемой `lexLabel` подразумевается определяющее вхождение метки, сопровождаемое двоеточием, в то время как `lexName` обозначает использующее вхождение — имя метки, употребленное на правах команды.

Таблица имен в нашем простом ассемблере не имеет блочной структуры. Именами снабжаются лишь метки. Для каждой метки в таблицу имен записывается соответствующий ей адрес.

Листинг 3.77. Интерфейс модуля таблицы имен

```
unit AsmTable;
{Таблица имен}

interface

uses
   AsmScan;

type
   tObj = ^tObjRec;
   tObjRec = record
      Name : tName;
      Addr : integer;
      Prev : tObj;
   end;

procedure InitNameTable;
procedure NewName(Addr: integer); {Добавить имя в
таблицу}
procedure Find(var Addr: integer);{Поиск имени}

{=====================================================}
```

Процедура `NewName` (листинг 3.77) добавляет в таблицу имя, содержащееся в глобальной переменной сканера `Name`, и заносит в запись об этом имени адрес `Addr`, переданный в качестве параметра. Процедура `Find` ищет имя `Name` в таблице и при удачном поиске возвращает в выходном параметре `Addr` соответствующий этому имени адрес.

Ассемблирование

Ассемблирование — это трансляция с языка ассемблера, за которую в нашей программе будет отвечать модуль `AsmUnit`.

Листинг 3.78. Интерфейс основного модуля ассемблера

```
unit AsmUnit; {Модуль ассемблера}

interface

procedure Assemble;

{=====================================================}
```

Первый и второй проходы ассемблера

Замена мнемонических обозначений операций их машинными кодами тривиальна. Обработка числовых констант также не составляет труда. Основная же работа ассемблера связана с трансляцией имен.

Каждое использующее вхождение имени должно быть заменено адресом. Адрес задается местоположением одноименной метки (определяющим вхождением). Поскольку определяющее вхождение может располагаться после использующего, адрес может быть неизвестен к моменту обработки использующего вхождения при первом проходе. Проблему решает использование двух проходов по исходной программе.

На первом проходе обрабатываются только определяющие вхождения и заполняется таблица имен. Транслятор ведет счет машинных команд с помощью программного счетчика периода компиляции, который, как и прежде, обозначим `PC`. Перед первым проходом `PC=0`. Затем счетчик увеличивается на единицу каждый раз, когда в программе встречаются код операции, константа или имя — каждый из этих элементов занимает в машинной программе одну ячейку. Адресом, назначенным данному имени, будет значение `PC` в момент распознавания определяющего вхождения этого имени. Само определяющее вхождение имени (метка) не меняет значения `PC`.

На втором проходе генерируется машинный код, в который записываются адреса, определенные при первом проходе. Адреса извлекаются из таблицы имен. Определяющие вхождения имен на втором проходе игнорируются.

Программирование распознавателя

Как и в случае трансляции с языка высокого уровня, в ассемблере ведущую роль будет играть синтаксический анализатор. Специфика же состоит в том, что предусматриваются два прохода, в

каждом из которых нужно решать задачу распознавания. В обоих проходах распознаватель программы действует в соответствии с синтаксисом:

```
Программа = Строка {перевод_строки Строка}.
```

Отличие между проходами состоит в том, что при разборе отдельной строки выполняется различная семантическая обработка. В то же время распознавание следования строк в соответствии с приведенной выше формулой в первом и втором проходах должны выполняться одинаково.

Чтобы не переписывать распознаватель программы дважды, используем немного необычное, но очень подходящее решение: распознающая процедура нетерминала «Программа» будет получать в качестве параметра процедуру-распознаватель строки. Чтобы оформить параметр процедурного типа, определим в секции реализации модуля `AsmUnit` соответствующий тип (листинг 3.79). В этом же месте программного кода поместим описание программного счетчика периода компиляции `PC`.

Листинг 3.79. Глобальные описания секции реализации модуля AsmUnit

```
type
   tLineProc = procedure; {Тип распознавателя строки}
var
   PC : integer;
```

Теперь запишем процедуру `Assemble` (листинг 3.80), которая запустит сначала первый, а потом второй проход и напечатает сообщения по завершении компиляции.

Листинг 3.80. Запуск первого и второго проходов ассемблера

```
procedure Assemble;
begin
   Pass(LineFirst);  {Первый проход}
   Pass(LineSecond); {Второй проход}
   WriteLn;
   WriteLn('Компиляция завершена');
   WriteLn('Размер кода ', PC);
   WriteLn;
end;
```

Напомню, что при обнаружении ошибок (например, при первом проходе) вызывается процедура Error, которая прерывает выполнение программы.

Оба прохода выполняются вызовом процедуры Pass (pass — проход; прогон; просмотр), которой для выполнения первого прохода передается процедура-обработчик строки LineFirst, для второго — LineSecond.

Структура Pass (листинг 3.81) определяется синтаксисом программы. Кроме того, перед каждым проходом вызовами ResetText и NextLex подготавливается чтение исходного текста программы с начала. Отсчет PC при каждом проходе начинается с нуля.

Листинг 3.81. Проход ассемблера

```
(* Программа = Строка { перевод_строки Строка } *)
procedure Pass(Line : tLineProc);
begin
   ResetText;
   NextLex;
   PC := 0;
   Line;    {Распознавание строки}
   while Lex = lexEOL do begin
      NextLex;
      Line; {Распознавание строки}
   end;
   if Lex <> lexEOT then
      Error('Так нельзя');
end;
```

Обработчик строки на первом проходе (процедура LineFirst, листинг 3.82) должен при распознавании метки (определяющего вхождения имени) заносить имя в таблицу, назначая в качестве адреса текущее значение PC. При распознавании числа, кода операции или использующего вхождения имени достаточно увеличивать PC.

Листинг 3.82. Распознаватель-обработчик строки на первом проходе

```
(* Строка = [метка][число|имя|код] *)
procedure LineFirst;
begin
   if Lex = lexLabel then begin
      NewName(PC);
      NextLex;
   end;
   if Lex in [lexName, lexNum, LexOpCode] then begin
      PC := PC + 1;
      NextLex;
   end;
end;
```

Распознаватель строки, предназначенный для второго прохода ассемблера (процедура `LineSecond`, листинг 3.83), работает в соответствии с тем же синтаксисом, но выполняет другие семантические действия. Определяющие вхождения имен (метки с двоеточием) пропускаются, а для кодов операций, чисел и имен генерируется машинный код — по одной команде на каждую из названных лексем.

Листинг 3.83. Распознаватель-обработчик строки при втором проходе

```
(* Строка = [метка][число|имя|код] *)
procedure LineSecond;
var
   Addr : integer;
begin
   if Lex = lexLabel then
      NextLex;
   case Lex of
   lexName:
      begin
         Find(Addr);
         Gen(Addr);
         NextLex;
      end;
   lexNum:
      begin
         Gen(Num);
         NextLex;
      end;
```

```
    lexOpCode:
       begin
          Gen(OpCode);
          NextLex;
       end;
    end;
end;
```

Имя порождает генерацию адреса, который берется из заполненной на первом проходе таблицы имен. При распознавании константы в качестве команды в машинный код записывается она сама. Мнемоническое обозначение операции порождает генерацию соответствующей машинной команды, код которой предоставляет сканер в глобальной переменной `OpCode`.

Для записи команд в машинный код, который помещается в память виртуальной машины, используется процедура `Gen` (листинг 3.84).

Листинг 3.84. Генерация кода

```
procedure Gen(Cmd: integer);
begin
   if PC < MemSize then begin
      M[PC] := Cmd;
      PC := PC+1;
      end
   else
      Error('Недостаточно памяти');
end;
```

Являясь частью модуля `AsmUnit`, она отвечает и за продвижение программного счетчика периода компиляции `PC` на втором проходе ассемблера.

Автоматизация построения и мобильность трансляторов

Не обязательно все работы по программированию компилятора, построению, анализу и преобразованию грамматик должны выполняться вручную.

Автоматический анализ и преобразование грамматик

Для ряда задач теории формальных языков и грамматик, существуют алгоритмы, решающие эти задачи. Будучи реализованы в виде специальных программ, они могут быть использованы и используются для анализа и преобразования грамматик при разработке языков программирования и специализированных языков информационных систем. Иногда алгоритма не существует, но автоматизированная попытка поиска решения, тем не менее, может быть предпринята. В качестве примера рассмотрим два сюжета такого рода.

Проверка наличия *LL*(1) **свойства у контекстно-свободной грамматики**. Для описания синтаксиса языков программирования в подавляющем большинстве случаев используются КС-грамматики. Если такая грамматика является еще и *LL*(1) грамматикой, то это дает возможность построить эффективный детерминированный распознаватель языка, работающий, например, методом рекурсивного спуска. Существует алгоритм, позволяющий для произвольной КС-грамматики определить, является ли она *LL*(1)-грамматикой.

Преобразование произвольной КС-грамматики в эквивалентную *LL*(1)**-грамматику**. Если КС-грамматика не обладает *LL*(1)-свойством, можно предпринять попытку ее преобразования в эквивалентную *LL*(1)-грамматику. Задача такого преобразования является, однако, алгоритмически неразрешимой. Это означает, что не существует и не может существовать алгоритма, решающего ее. Тем не менее, попытки преобразования могут быть предприняты. Есть программы, которые пытаются выполнять такое преобразование, и основываются на использовании эвристических приемов. Результатом работы такой программы может быть успешное преобразование грамматики либо неудача. В первом случае результатами преобразования можно восполь-

зоваться. Неудача же не обязательно означает, что преобразование невозможно, быть может, его просто не удалось найти.

Достоинство автоматических анализаторов и преобразователей грамматик состоит в том, что в случае достижения с их помощью результата его надежность оказывается весьма высокой, если не абсолютной. Например, при успешном автоматическом преобразовании КС-грамматики к виду *LL*(1) гарантируется отсутствие искажения языка, то есть эквивалентность грамматик. Проблема может возникнуть, только если сама программа-преобразователь содержит ошибку. При преобразовании вручную шансы ошибиться значительно больше.

Автоматическое построение компилятора и его частей

Другое направление автоматизации при создании транслятора — автоматическая генерация его модулей. Для этого могут быть разработаны специальные программы.

Конструкторы сканеров

Лексический анализатор — один из самых простых блоков транслятора. Правила записи лексем обычно могут быть заданы с помощью автоматной грамматики. Задача разбора для автоматной грамматики эффективно решается конечным автоматом. Нетрудно представить себе программу, которая по заданной автоматной грамматике порождает либо программу-распознаватель (рис. 3.12а), либо таблицу переходов конечного автомата (рис. 3.12б).

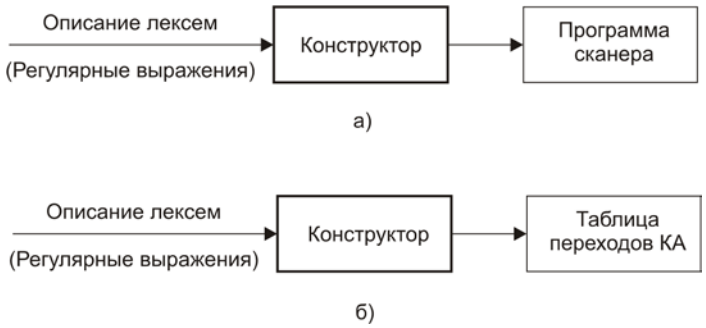


Рис. 3.12. Конструкторы сканеров

Описания лексем языка, которые поступают на вход конструктора сканеров, обычно записываются с помощью регулярных выражений.

Самым известным конструктором лексических анализаторов является программа Lex. Существуют ее реализации, использующие различные языки программирования, в том числе Си и Паскаль. Как результат своей работы Lex порождает программу на одном из этих языков. При необходимости в порождаемую конструктором программу могут быть встроены части, заранее написанные вручную. Правила записи лексем определяются с помощью несколько расширенной нотации регулярных выражений. Lex позволяет исключать из программы комментарии, однако не предоставляет готовой возможности обработки вложенных комментариев, какие имеются, например, в Обероне.

Генераторы синтаксических анализаторов

При соблюдении некоторых ограничений на КС-грамматику (например, наличие $LL(k)$ свойства) оказывается возможным автоматически построить эффективный распознаватель для такой грамматики. Программа генератор синтаксических анализаторов, получая на входе описание грамматики, может порождать распознаватель, работающий по таблице, и саму таблицу (рис. 3.13а), либо генерировать программу распознавателя, действующего, например, по методу рекурсивного спуска (рис. 3.13б).

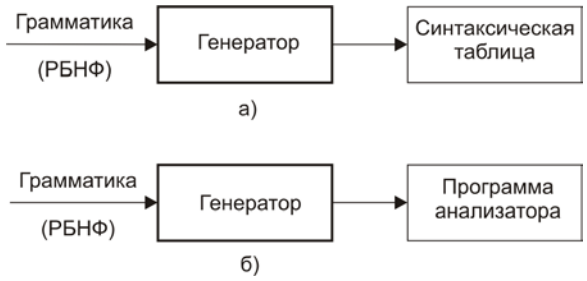


Рис. 3.13. Генераторы синтаксических анализаторов

Табличные распознаватели, применяемые в генераторах анализаторов, могут работать как на основе нисходящих, так и восходящих алгоритмов разбора. Для описания грамматики используются различные варианты БНФ, как правило, расширенные.

Компиляторы компиляторов

Если возможность автоматического порождения лексических анализаторов и синтаксических анализаторов для ограниченного, но вполне содержательного класса КС-грамматик не вызывает сомнений и является обыденной практикой, то полностью автоматическое построения всего компилятора, включая контекстный анализатор и генератор кода — скорее идеал, чем реальная возможность. Между тем, системы, в той или иной мере приближающиеся к этому идеалу, существуют и используются. Их называют компиляторами компиляторов (compiler-compiler). В литературе на русском языке используется термин «система построения трансляторов», сокращенно — СПТ. Общая схема автоматизированной генерации компилятора представлена на рис. 3.14.

В качестве формализма, на основе которого строится описание семантики, обычно используются атрибутные грамматики, введенные в обиход Д. Кнутом. Не надо, однако, представлять дело так, что семантика языка описывается некими формулами, подобными РБНФ. Дело сводится к тому, что разработчик должен написать на используемом в СПТ языке программирования (Си,

Паскаль) семантические процедуры, которые обеспечат выполнение контекстного анализа и генерации кода, а СПТ предоставит возможность интегрировать эти процедуры в синтаксический анализатор.

Рис. 3.14. Компилятор компиляторов

Самым известным компилятором компиляторов является система YACC. Ее название — аббревиатура, происходящая от ироничного Yet Another Compiler Compiler — еще один компилятор компиляторов. К началу 1970-х годов, когда появился YACC, тема автоматического порождения компиляторов была не только популярной, но даже несколько избитой.

Первоначально ориентированный на операционную систему Unix, YACC использовал Си в качестве базового языка. В дальнейшем были разработаны различные варианты YACC-совместимых компиляторов компиляторов, предназначенных для разных операционных систем, использующие отличные от Си базовые языки. Получила известность СПТ Bison[43] — некоммерческая, свободно распространяемая версия YACC.

[43] Очевидно, такое название происходит от созвучия YACC и yak — як, ведь бизоны — родственники яков.

Компиляторы компиляторов, совместимые с YACC, ориентированные на различные операционные системы и языки программирования, можно найти в Интернете.

YACC интегрируется с конструктором сканеров Lex. На вход СПТ подается описание языка, содержащее фрагменты кода на базовом языке (например, Си), описания лексем и правила трансляции, представляющие собой спецификацию синтаксиса на расширенной БНФ, дополненной возможностью связать терминалы и нетерминалы правил с семантическими процедурами. Результатом работы является текст программы-транслятора (ее основного модуля) на базовом языке.

YACC основан на использовании восходящего *LALR*(1)-распознавателя, работающего по таблице, и ориентированного на *LR*(1)-грамматики — подмножество КС-грамматик, позволяющее определить более широкий класс языков, чем *LL*(1)-грамматики.

Компиляторы компиляторов и программирование вручную

Практика создания компиляторов показывает, что автоматизированная разработка не имеет радикального преимущества в сравнении с программированием вручную. Объясняется это в первую очередь тем, что семантическая составляющая работы компилятора почти не автоматизируется. Что касается лексического и синтаксического анализаторов, то для разумно спроектированного языка трудоемкость реализации этих компонент невелика и составляет очень малую часть всех затрат на разработку.

Программирование компилятора вручную предоставляет бо́льшую гибкость. Получаемая программа имеет ясную структуру, читается намного легче, чем сгенерированная СПТ.

В свою очередь СПТ предоставляют возможность использовать более мощный класс *LR*(1)-грамматик и *LALR*(1)-распознава-

телей. Такие распознаватели предполагают использование таблиц, построение которых вручную сложно и чревато ошибками.

Использование СПТ может провоцировать усложнение грамматики языка. При ручном программировании неоднозначности грамматики и проблемы детерминированного распознавания часто могут быть решены с помощью несложных контекстных проверок, в то время как анализатор, порождаемый автоматически, требует разрешения таких вопросов уже на уровне синтаксических определений.

Можно предположить также, что написанный вручную код окажется эффективней порожденного компилятором компиляторов.

Некоторые сравнения можно сделать из рассмотрения двух вариантов компилятора языка «О». Один — работающий по методу рекурсивного спуска, другой получен[44] с помощью свободно распространяемой СПТ Turbo Pascal Lex and Yacc Version 4.1. Функционально компиляторы одинаковы. Вариант, полученный с помощью YACC и Lex, использует ряд модулей первого компилятора, в том числе, разумеется, виртуальную машину.

Таблица 3.6. Сравнение компиляторов языка «О»

Характеристика	Рекурсивный спуск	Lex + YACC
Размер исходного кода, написанного вручную		
строк	1368	1064
байт	23 957	25 164
лексем	5149	4837
В том числе сканер		
строк	262	126

[44] Работа выполнена В. Ёжкиным.

Характеристика	Рекурсивный спуск	Lex + YACC
байт *лексем*	5131 955	2809 534
Размер кода полученного компилятора на Паскале *строк* *байт* *лексем*	 1368 23 957 5149	 4650 85 861 20 225
Размер исполняемого файла компилятора (байт)	16 656	34 336
Время компиляции тестовой программы[45] (с)	0,37	0,77

Как видно, объем кода, который пришлось написать вручную, отличается в двух вариантах незначительно. Если в качестве единицы измерения использовать число лексем, которое меньше зависит от индивидуального стиля программирования, то при использовании СПТ уменьшение объема ручной работы составило всего 6%, причем вся экономия достигнута на генерации лексического анализатора. Lex действительно позволяет очень быстро получить простой сканер. Так, для выполнения измерений, результаты которых представлены в таблице, с помощью Lex в кратчайший срок был изготовлен сканер, позволяющий подсчитывать число лексем в программах на Паскале и входных файлах Lex и YACC.

[45] 1606 строк на языке «О», компьютер на базе процессора Pentium с тактовой частотой 100МГц.

Эффективность компилятора, полученного с помощью СПТ, оказалась заметно ниже, чем запрограммированного вручную — он имеет больший размер исполняемого файла и работает медленней.

Известны примеры крупных разработок, выполненных в 1990-е годы и основанных как на использовании СПТ, так и на программировании вручную. Так, при создании компилятора Си++ в лаборатории Открытых Информационных Технологий ВМиК МГУ (руководитель В. А. Сухомлин, ведущие разработчики Е. А. Зуев, А. Н. Кротов) был использован компилятор компиляторов Bison. Разработка велась на Си и Си++. В то же время компилятор нового языка Си#, созданный компанией Microsoft, также написан на Си++, но с использованием рекурсивного спуска.

Использование языков высокого уровня

Первые трансляторы, появившиеся в 1950-е годы, программировались на машинном языке или на языке ассемблера. В дальнейшем роль языков низкого уровня, как инструментов создания компиляторов, постепенно снижалась. Уже в 1960-е годы некоторые трансляторы программировались на таких языках как Фортран и Алгол-60.

В настоящее время не осталось причин, препятствующих использованию языков высокого уровня при программировании компиляторов и интерпретаторов. Во-первых, существует большой выбор трансляторов для разнообразных языков и программно-аппаратных платформ. Эти языки и эти трансляторы могут служить инструментами для создания новых языков и новых трансляторов.

Во-вторых, мощность компьютеров, на которых исполняются компиляторы[46], настолько возросла, что превышает обычную потребность транслятора в ресурсах, и, следовательно, нет необходимости добиваться максимально возможной эффективности компилятора за счет перехода при его разработке на язык низкого уровня. К тому же, задача трансляции (если не иметь в виду продвинутые методы оптимизации кода) имеет не слишком высокую временную сложность, и ускорение трансляции вряд ли должно быть приоритетным требованием.

В-третьих, достижения в методах оптимизации кода таковы, что хороший оптимизирующий компилятор может создавать программу, почти не уступающую по эффективности написанной вручную. Если есть возможность использовать такой компилятор при разработке, то о программировании на ассемблере следует забыть.

Наконец, трудно себе представить, что компиляторы таких сложных языков, как, например, Си++, Ява, Си# могут быть написаны вручную на ассемблере с разумными затратами и приемлемой надежностью.

Поскольку язык ассемблера для каждой платформы сугубо специфичен, компилятор, написанный на ассемблере, будет обладать низкой мобильностью. Для переноса на другую платформу его придется перепрограммировать полностью или почти полностью. Транслятор же, существующий в виде программы на языке высокого уровня, может быть перекомпилирован для использования на разных системах.

[46] Отнюдь не все современные компьютеры обладают высокой мощностью. Встроенный компьютер, управляющий работой, например, микроволновой печи не имеет ни быстрого процессора, ни памяти большого объема. Но ведь никто и не пробует запускать компилятор на компьютере печки!

Т-диаграммы трансляторов

С помощью Т-диаграмм рассмотрим возможности трансформации трансляторов, написанных на языках высокого уровня. Транслятор связан с тремя языками: входным (исходным), выходным (целевым, объектным) и инструментальным — языком, на котором написан или существует он сам. Будем, как и раньше, изображать транслятор в виде Т-диаграммы. Слева на такой диаграмме записывается исходный язык, справа — целевой, внизу — инструментальный (рис. 3.15).

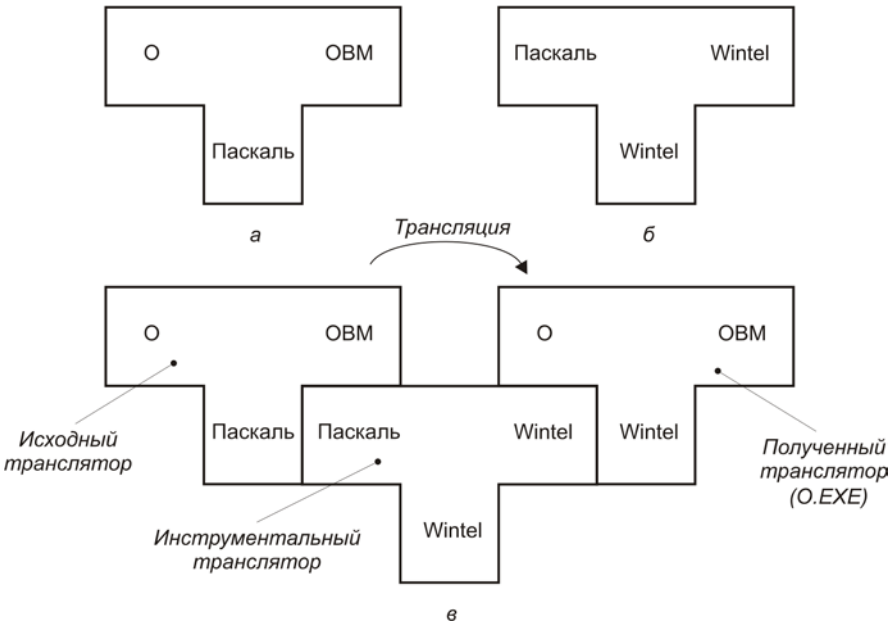


Рис. 3.15. Т-диаграммы:
а — компилятор «О»; *б* — компилятор Паскаля; *в* — трансляция компилятора «О»

На рис. 3.15*а* можно видеть Т-диаграмму компилятора, который транслирует с языка «О» в код ОВМ и написан на Паскале. Его мы разрабатывали в этой главе. Рисунок 3.14*б* соответствует компилятору языка Паскаль, работающему на платформе «Wintel» (так называют совокупность операционной системы Windows и компьютера на базе процессора Intel). Инструментальный и целевой языки в этом случае — машинный код процессора In-

tel, к которому добавлены соглашения программного интерфейса Windows. Если компилятор доступен в виде исполнимого кода, то этот код и можно считать инструментальным языком. Диаграмме с рис. 3.15*б* соответствуют Turbo Pascal, Free Pascal, Delphi и другие трансляторы Паскаля для Windows. Их и можно использовать для трансляции компилятора языка «О». Эту трансляцию проиллюстрируем с помощью Т-диаграмм, как показано на рис. 3.15*в*. Компилятор *Паскаль-Wintel-Wintel*[47] транслирует компилятор *О-Паскаль-ОВМ*, преобразуя программу на Паскале в код Wintel, в результате чего получается компилятор *О-Wintel-ОВМ* — программа `O.EXE`.

При создании программ для встроенных микрокомпьютеров (микроконтроллеров), управляющих работой различных технических устройств и систем, используют кросскомпиляторы. Кросскомпилятор, работая на одном компьютере (например, персональном), генерирует машинный код для другого (например, встроенного). Действительно, как уже говорилось, запустить программу-компилятор на микроконтроллере микроволновой печи затруднительно. Другой пример: микроконтроллеры мобильных телефонов обладают памятью и быстродействием, сопоставимыми с возможностями персональных компьютеров недавнего прошлого, но и в этом случае вести разработку программ для телефона на самом телефоне никому не приходит в голову. Пример диаграммы кросскомпилятора можно видеть на рисунке 3.16а. Язык Си часто используется для программирования микроконтроллеров. Кросскомпилятором можно считать и транслятор *О-Wintel-ОВМ* (рис 3.15в).

[47] Таким способом, перечисляя фигурирующие на Т-диаграмме языки слева направо, условимся обозначать транслятор в тексте.

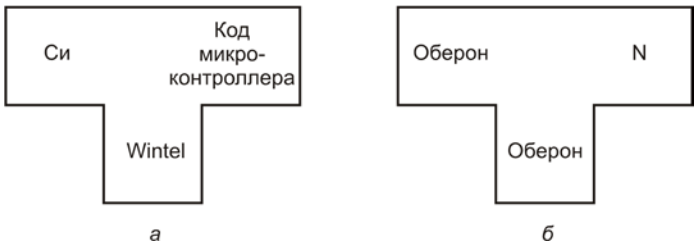


Рис. 3.16. Кросскомпилятор (*а*) и самокомпилятор (*б*)

Самокомпилятор. Раскрутка

Компилятор может быть написан на своем собственном входном языке, в таком случае его называют самокомпилятором. Как это может быть реализовано и зачем это нужно?

Реализация старого языка для новой машины

Пусть, к примеру, имеется компилятор *Оберон-Оберон-N*, написанный на Обероне и транслирующий с Оберона в код машины[48] *N* (рис. 3.16б). Представим, что *N* — это новый тип компьютера, для которого еще нет ни одного компилятора никакого языка высокого уровня[49]. И мы решили, что первым языком на этой машине будет Оберон.

Оберон — не новый язык. Существует много компиляторов для него. Используем компилятор *Оберон-M-M*, имеющийся на машине *M* (рис. 3.17а). Собственно, с его помощью на машине *M* и велась разработка самокомпилятора. Транслируем самокомпилятор (рис. 3.17б). Получится кросскомпилятор *Оберон-M-N*, способный работать на машине *M* и транслирующий в код ма-

[48] Будем для упрощения говорить о «машине» (вычислительной) вместо того, чтобы каждый раз использовать понятие программно-аппаратной платформы. В 1960-е, 1970-е и даже 1980-е годы компилятор действительно мог исполняться на «голом железе», не будучи погруженным в среду операционной системы. Сейчас такого уже нет.

[49] Конечно, новые типы машин появляются не так уж часто, но ведь появляются. К тому же, *N* может быть не реальным компьютером, а виртуальным.

шины *N*. С его помощью исходный компилятор *Оберон-Оберон-N* транслируется еще раз (рис. 3.17в). В результате получается *Оберон-N-N*, способный работать на машине *N* и производить код для нее же. Получение нового компилятора двукратной трансляцией самокомпилятора может быть проиллюстрировано Т-диаграммами и немного по-другому (рис. 3.17г).

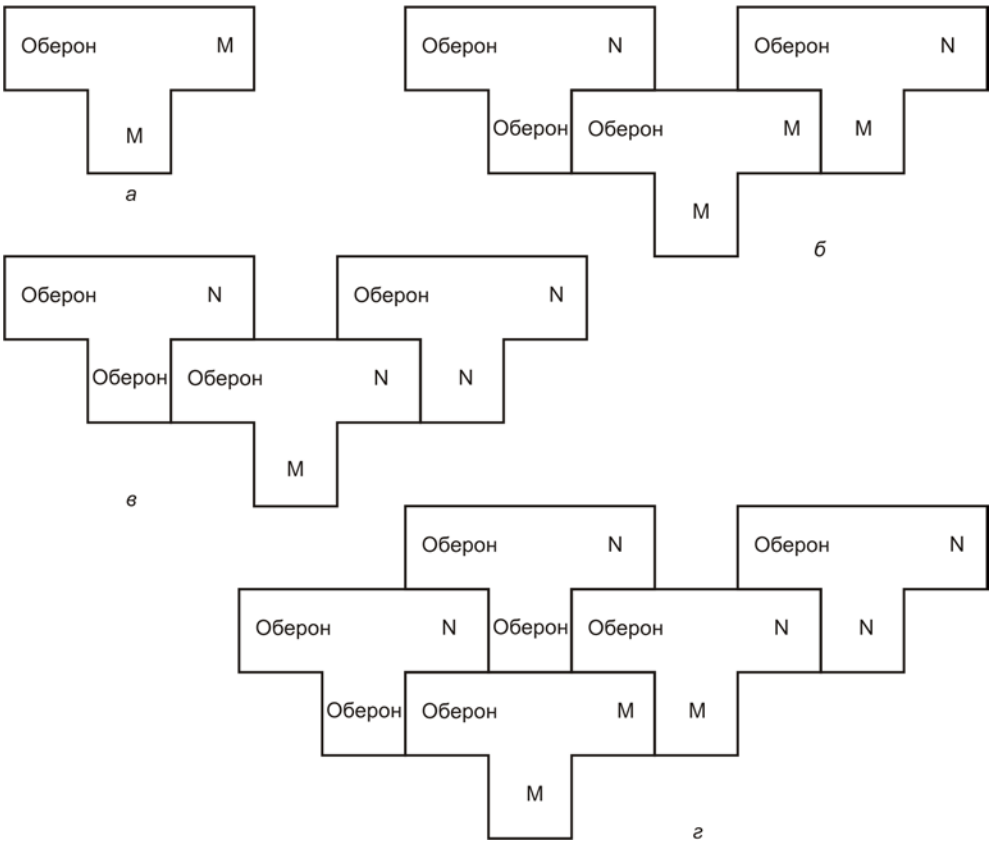


Рис. 3.17. Трансляция самокомпилятора: *а* — инструментальный компилятор; *б* — получение кросскомпилятора; *в* — самокомпиляция; *г* — полная схема раскрутки

Реализация языка Оберон для новой машины выполнена. При этом не пришлось программировать в коде машины *N*.

Использование самокомпилятора облегчает перенос существующего языка на новую машину.

Если инструментальный компилятор *Оберон-М-М*, который был использован при разработке, доступен в исходном коде и тоже является самокомпилятором (*Оберон-Оберон-М*), то создание нового транслятора Оберона могло сводиться всего лишь к модификации существующего — перепрограммированию его генератора кода.

Разработка компилятора нового языка раскруткой

Теперь рассмотрим другую ситуацию — создание компилятора нового языка, для которого никаких компиляторов еще не существует. Назовем новый язык *L*. Поставим задачу его реализации для машины *M*. Написать сразу самокомпилятор *L-L-M* возможно, скорее, теоретически. Да и как его использовать? Ведь компиляторов *L*, способных работать на машине *M*, нет, и компилировать этот компилятор нечем. В принципе, возможна его трансляция в ассемблерный код вручную, но мы стремимся избежать программирования на языке низкого уровня.

Поступим по-другому. Вначале запрограммируем компилятор *L* на каком-либо из существующих и реализованных для машины *M* языков, например, на Си (рис. 3.18*а*). После этого созданный компилятор можно переписать с языка Си на язык *L* (рис. 3.18*б*), используя в качестве инструмента при таком переносе первоначальный компилятор (рис 3.17*в*).

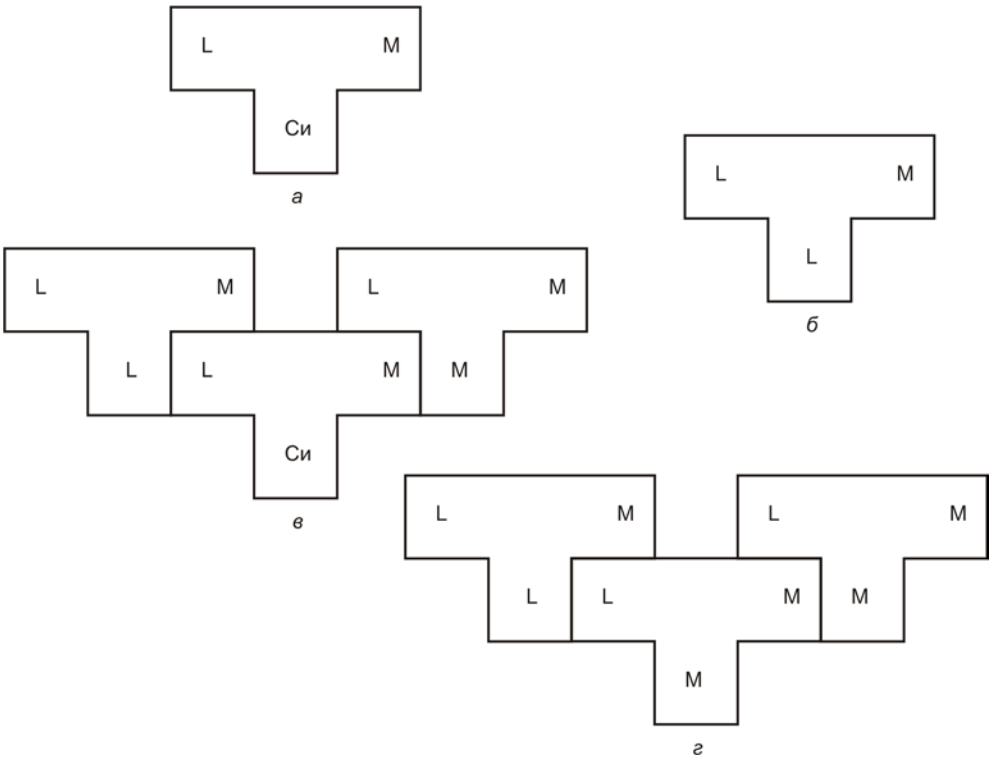


Рис. 3.18. Реализация нового языка: *а* — первоначальный компилятор; *б* — самокомпилятор; *в* — трансляция самокомпилятора; *г* — самокомпиляция

Самокомпилятор, существующий в виде программы в коде той машины, для которой он компилирует, при трансляции своего исходного текста порождает себя (рис. 3.18*г*).

Пользоваться языком *L* на машине *M* можно и, не имея самокомпилятора, достаточно применять *L*-*M*-*M*, полученный компиляцией *L*-*Си*-*M*. Такой транслятор даже может быть перепрограммированием генератора кода перенесен на другую машину, где есть компилятор Си. Напомню, что самокомпилятор переносится на новую машину с такими же затратами, даже если там нет другого компилятора.

Другой способ получить самокомпилятор нового языка *L* для машины *M* состоит в следующем. Вначале на каком-либо из существующих языков (например, Си) программируется компилятор для такого подмножества *L*, которое с одной стороны доста-

точно просто для того, чтобы не требовались большие затраты на его реализацию, с другой — достаточно богато, чтобы на этом языке можно было программировать компилятор. Обозначим такое подмножество L_0. Диаграмма начального компилятора L_0 показана на рис. 3.19*а*.

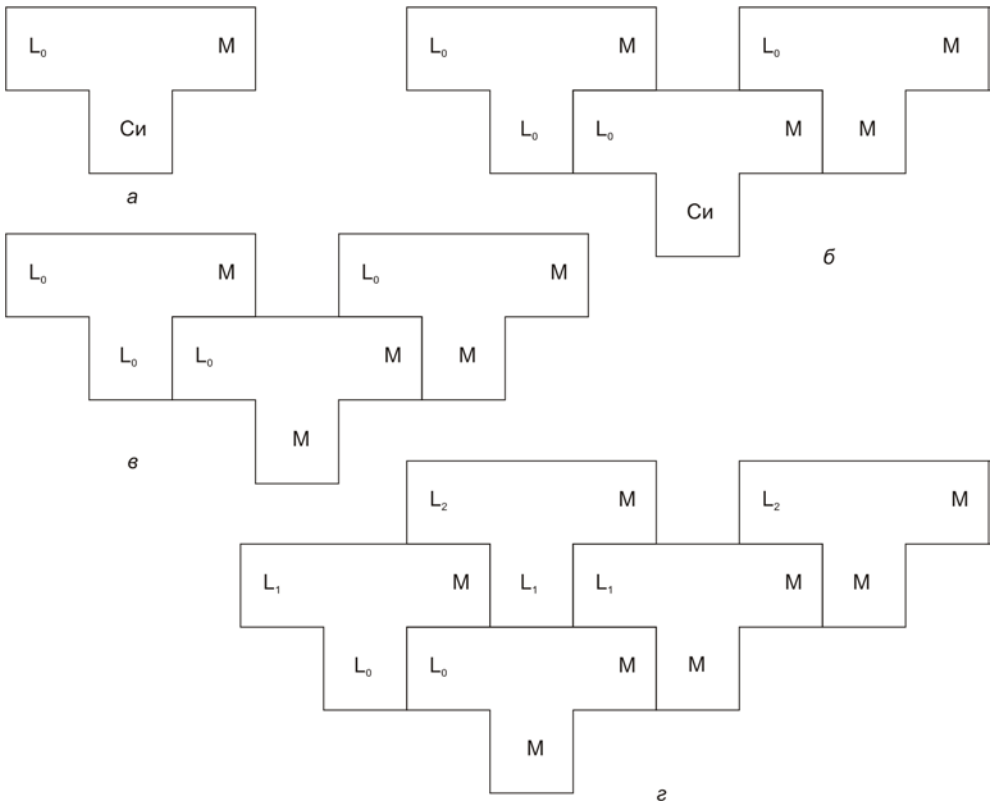


Рис. 3.19. Раскрутка

Далее на языке L_0 программируется компилятор L_0, который компилируется с помощью L_0-*Си*-*М* (рис. 3.19*б*), после чего способен компилировать себя (рис. 3.19*в*). После этого начинается постепенное расширение L_0 до тех пор, пока он не превратится в L. Сначала реализуется какое-либо из ранее не реализованных средств L. С помощью предыдущей версии компилируется новая, после чего вновь реализованное средство можно использовать в программе-компиляторе для его дальнейшего расширения (рис. 3.19*г*). Такой процесс называется *раскруткой*.

Собственно, частным случаем раскрутки является и процесс, рассмотренный в первой части этого раздела, когда на Си пишется компилятор сразу полного языка *L*. Это раскрутка, выполняемая за один шаг, когда L_0=*L*.

Часто раскруткой называют вообще любой процесс создания самокомпилятора.

Улучшение качества кода при раскрутке

Самокомпилятор обладает полезным свойством — **способностью к самосовершенствованию.** Если он модернизируется с целью улучшения качества создаваемого кода, то через самокомпиляцию повышается и качество кода его самого.

Самокомпилятор является тестом для самого себя. Способность без ошибок компилировать такую нетривиальную программу, какой является транслятор, добавляет уверенности в надежности компилятора. Надо, однако, понимать, что самокомпилятор совсем не обязательно обеспечивает собственное полное тестовое покрытие[50].

Не могу удержаться еще от одного замечания. По моему убеждению, **только компилятор, написанный на своем собственном входном языке, может быть эстетически безупречен**. Если, конечно, речь идет о языке, который годится для написания компиляторов.

[50] Тестовое покрытие характеризует, в какой степени различные части программы подвергаются проверке при тестировании. Считается, что минимальным требованием является исполнение каждого оператора тестируемой программы в ходе испытаний хотя бы один раз. В этом случае можно говорить о полном тестовом покрытии.

Примеры раскрутки

Известно достаточно много проектов, связанных с реализацией языков программирования, в ходе которых использовалась раскрутка.

Оптимизирующий компилятор Фортрана, известный как Fortran H, был написан на Фортране в конце 1960-х для операционной системы IBM OS/360. Раскрутка была выполнена в три стадии. Компилятор обеспечивал очень высокое качество объектного кода, но за счет использования четырех проходов и разнообразных оптимизаций сам работал медленно. Этот компилятор широко использовался в нашей стране в 1970-80-е годы на ЕС ЭВМ в операционной системе ОС ЕС, которая была аналогом OS/360.

Первый компилятор Паскаля был написан в 1970 году на Паскале (Н. Вирт, У. Амман (U. Ammann), Е. Мармье (E. Marmier), Р. Шилд (R. Schild)). Затем он был вручную транслирован Р. Шилдом в код компьютера CDC6000. После чего прошел еще несколько стадий раскрутки.

С помощью раскрутки Н. Виртом и Ю. Гуткнехтом разрабатывались Оберон-система и компилятор Оберона. Первый вариант компилятора был получен из существующего на компьютере Lilith компилятора для Модулы-2. Это была смесь Модулы и Оберона, позволявшая обрабатывать подмножество Оберона (Оберон0). Затем компилятор Оберон0 был переведен на его собственный язык. Это позволило перенести его на компьютер Ceres, для которого транслятор и предназначался. Далее последовала длинная серия шагов раскрутки. Каждый шаг состоял из двух фаз: вначале в язык вносились изменения, затем эти изменения начинали использоваться в самом компиляторе.

Унификация промежуточного представления

Затраты на разработку трансляторов могут быть уменьшены, а мобильность повышена, если использовать унифицированное промежуточное представление программы при трансляции.

В роли такого представления может выступать язык программирования, язык некоторой (виртуальной) машины, различные формы внутреннего представления программы в компиляторе.

Промежуточные языки

Представим, что необходимо реализовать n языков для m машин. Если для каждой пары язык — машина создается компилятор, то всего потребуется $n \times m$ компиляторов. Можно использовать промежуточный язык. Обозначим его ПЯ. Для всех языков создается компилятор (конвертор), транслирующий в ПЯ. А для каждой машины — компилятор, транслирующий с ПЯ в код этой машины. Всего нужно разработать $n+m$ трансляторов. Даже, если речь идет о двух машинах (m=2), схема становится выгодна уже при n=3. Промежуточный язык должен быть достаточно выразительным, чтобы транслировать на ПЯ было легко, и достаточно простым, чтобы оставались разумными затраты на создание трансляторов с ПЯ в машинный код.

Идея использования промежуточного языка возникла еще в конце 1950-х годов, когда обсуждался проект UNCOL (Universal Computer Oriented Language) — универсального компьютерного языка. Однако реального воплощения UNCOL тогда не получил.

Примером универсального промежуточного языка, специально созданного для этой роли, является АЛМО, разработанный в 1966 году в Институте прикладной математики АН СССР. Основанная на этом языке универсальная система программирования была реализована на нескольких типах отечественных ЭВМ.

На роль промежуточного языка хорошо подходит Си. Представляя собой по выражению Н. Вирта, «синтаксически усовершенствованный ассемблер», он прост и эффективен. Компиляторы Си имеются практически на любой платформе. Язык хорошо стандартизован. И, если недостатком Си, как языка для программирования вручную, является низкая надежность, то при его использовании в роли промежуточного языка проблема исчезает. Конверторы, порождающие код на Си, неоднократно использовались при реализации разных языков, например, Си++.

Примером использования промежуточного языка при построении систем программирования является применение компанией Microsoft языка IL (от Intermediate Language — промежуточный язык) в многоязыковой системе .NET. IL представляет собой машинный код виртуальной стековой машины. Компиляторы языков Си#, Си++, Visual Basic, JScript, входящие в состав .NET, транслируют в IL. Затем IL-код преобразуется в машинный код платформы, на которой работает система. Такое преобразование может происходить непосредственно в момент загрузки IL-модуля. Основная форма существования IL-кода — двоичные файлы. Однако, существует и IL-ассемблер, который может быть использован для программирования на IL.

В принципе, в систему .NET могут быть добавлены и компиляторы других языков.

Заслуживает внимания технология, предложенная учеником Н. Вирта Михаэлем Францем (Michael Franz) в 1994 году. Вначале программа транслируется в промежуточное представление, которое содержит компактную кодированную запись семантического дерева программы (SDE-представление). Кодирование основано на использовании так называемого семантического словаря (Semantic-Dictionary Encoding) и схоже с методами, используемыми при сжатии данных без потерь. SDE-представление полностью сохраняет информацию исходной программы, вклю-

чая блочную структуру и сведения о типах выражений, облегчая тем самым последующую оптимизацию и генерацию кода.

Промежуточное представление программы преобразуется в код целевой платформы в момент загрузки, «на лету». Благодаря структуре SDE-представления кодогенерация может выполняться весьма эффективно с получением высококачественного кода. SDE-представление в два раза более компактно, чем машинный код традиционных процессоров и более компактно, чем байт-код Явы.

П-код и виртуальные машины

Другой способ использования промежуточного представления — трансляция с языка высокого уровня в код гипотетической (виртуальной) машины и последующая интерпретация полученного кода на целевой платформе с помощью программного симулятора. Такой подход нам хорошо знаком, именно он использован в этой книге.

Одним из первых примеров его применения стала разработка в 1973 году в Федеральном техническом университете (ETH) Цюриха группой Н. Вирта П-кода (в оригинале — P-code), П-машины и П-компилятора языка Паскаль. П-код представляет собой наилучшим образом приспособленный для трансляции с Паскаля машинный код гипотетической П-машины — компьютера со стековой архитектурой, реализованного с помощью интерпретатора, написанного на Паскале. Появление П-системы способствовало широкому распространению языка Паскаль, поскольку позволяло легко переносить его на различные платформы перепрограммированием П-машины.

Вариант П-кода долгое время использовался в системе программирования Visual Basic компании Microsoft.

В первой половине 1990-х идея П-кода была применена при реализации языка Ява компанией Sun Microsystems. Транслятор

языка Ява преобразует программу в файлы байт-кода, который представляет собой машинный язык стекового компьютера. Байт-код исполняется интерпретатором, получившим название виртуальной Ява-машины (Java Virtual Machine, JVM).

Существуют реализации виртуальной Ява-машины для различных платформ. Все они могут исполнять одни и те же файлы байт-кода. Из-за того, что интерпретация существенно снижает быстродействие, разработаны различные схемы трансляции байт-кода в родной код той платформы, на которой исполняется программа. Часто такая трансляция происходит «на лету» при загрузке файлов байт-кода в память непосредственно перед исполнением. Выполняющие такую трансляцию подсистемы называют JIT-компиляторами (Just in time compilers, от Just in Time — вовремя).

Идея использования промежуточного языка была реализована при создании отечественного многопроцессорного вычислительного комплекса «Эльбрус»[51], объектный код которого был фактически постфиксной записью. Во время выполнения программы на «Эльбрусе» происходила аппаратная «just-in-time» компиляция объектного кода в обычный трехадресный регистровый код.

За несколько лет до появления Ява-технологии использование виртуальной машины для унификации программного обеспечения было предложено в работах Ю.В. Матиясевича, А.Н. Терехова, Б.А. Федотова [Матиясевич, 1984], [Матиясевич, 1990].

[51] Идейным предшественником “Эльбруса” была американская вычислительная система Burroughs 6700/7700.

Виртуальная машина, использованная в этой книге, безусловно, также происходит от П-кода[52]. Только в ней, в отличие от П-кода и JVM, реализована «чистая» стековая архитектура, когда операнды любых команд всегда находятся в стеке. Операнды команд П-кода и байт-кода Явы могут содержаться в самих командах.

Модульные компиляторы

Использование промежуточного представления в двух рассмотренных случаях (промежуточный язык и П-код) предполагает его формирование в явном виде с записью в файл. Кроме того, в процессе участвуют две программы. В первом варианте — это компилятор с исходного языка на промежуточный и компилятор с промежуточного языка в машинный код, во втором — компилятор в промежуточный код и интерпретатор этого кода.

Использование и унификация промежуточного представления программы полезны и внутри компилятора или семейства компиляторов. При этом промежуточное представление может существовать лишь в виде некоторой внутренней структуры данных, а работа по трансляции выполняется единой программой. Пользователь компилятора может даже не знать о существовании промежуточного представления.

В компиляторе, построенном по модульному принципу, часть, зависящая от входного языка, отделяется от генерирующей части, которая зависит от целевой машины. Интерфейсом между этими частями является промежуточное представление. В компиляторе может присутствовать языково-независимый и машинно-независимый оптимизатор, который работает на уровне про-

[52] Впервые эта машина под названием М-11 была применена в одном из учебных курсов в 1995 году, когда сведения о JVM были еще недоступны.

межуточного представления, порождая его оптимизированный вариант (рис 3.20).

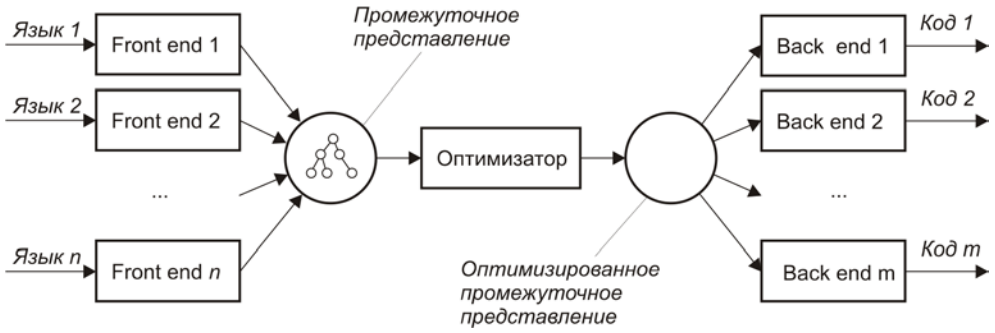


Рис. 3.20. Модульный компилятор

Фронтальная, анализирующая часть транслятора (front end) отвечает за лексический, синтаксический и контекстный анализ исходной программы и порождает промежуточное представление. При изменении входного языка фронтальная часть может быть заменена без того, чтобы это затронуло другие части транслятора.

Соответственно, синтезирующая часть (back end) не должна зависеть от входного языка и может подвергаться замене при изменении целевой машины.

Часто в роли промежуточного представления используется семантическое дерево программы. Хотя языки высокого уровня синтаксически различаются, семантическая основа широкого класса языков может быть систематизирована, а их промежуточное семантическое представление унифицировано.

Комбинирование различных анализирующих и синтезирующих частей позволяет получить семейство компиляторов для разных языков и целевых платформ примерно с теми же затратами, что и при использовании промежуточных языков. Однако модульный компилятор удобней в использовании, поскольку представляет собой единую программу.

В однопроходном же компиляторе, в частности том, который рассмотрен в этой книге, отвечающие за анализ и синтез части тесно переплетены. Эффективная и простая однопроходная схема не обладает той гибкостью в отношении входного языка и целевой машины, которая свойственна модульному компилятору.

По схеме с промежуточным представлением построены компиляторы XDS новосибирской компании Excelsior. На общей основе разработаны оптимизирующие компиляторы для Модулы-2, Оберона, Явы, байт-кода Явы (!) и двух специализированных языков, генераторы кода для процессоров Intel x86, Motorola 68K, SPARC, PowerPC, VAX, а также конверторы в Си и Си++. Модульную архитектуру имеют компиляторы Фортрана, Си, Си++ и Явы компании IBM.

ПРИЛОЖЕНИЕ 1. ЯЗЫК ПРОГРАММИРОВАНИЯ ОБЕРОН-2

Х. Мёссенбёк, Н. Вирт
Институт компьютерных систем, ETH Цюрих
Перевод с английского[53] С. Свердлова

От переводчика

Язык программирования Оберон создан автором Паскаля и Модулы-2 Никлаусом Виртом в 1987 году в ходе разработки одноименной операционной системы для однопользовательской рабочей станции Ceres. Язык и операционная система названы именем одного из спутников планеты Уран — Оберона, открытого английским астрономом Уильямом Гершелем ровно за двести лет до описываемых событий.

«Сделай так просто, как возможно, но не проще того» — это высказывание А.Эйнштейна Вирт выбрал эпиграфом к описанию языка. Удивительно простой и даже аскетичный Оберон является, вероятно, минимальным универсальным языком высокого уровня. Он проще Паскаля и Модулы-2 и в то же время обогащает их рядом новых возможностей. Важно то, что автором языка руководили не сиюминутные коммерческие и конъюнктурные соображения, а глубокий анализ реальных программистских потребностей и стремление удовлетворить их простым, понятным, эффективным и безопасным образом, не вводя по возможности новых понятий. Являясь объектно-ориентированным языком, Оберон даже не содержит слова object.

Оберон представляется идеальным языком для изучения программирования. Сочетание простоты, строгости и неизбыточности предоставляет начинающему программисту великолепную

[53] Публикуется с любезного разрешения проф. Х. Мёссенбёка и проф. Н. Вирта.

возможность, не заблудившись в дебрях, выработать хороший стиль, освоив при этом и структурное и объектно-ориентированное и модульно-компонентное программирование.

Сотрудничество Н. Вирта с Ханспетером Мёссенбёком привело к добавлению в язык ряда новых средств. Новая версия получила название Оберон-2. Описание именно этого языка дается в настоящем переводе. Оберон-2 представляет собой почти правильное расширение Оберона. В Оберон-2 добавлены:

- связанные с типом процедуры;
- экспорт только для чтения;
- открытые массивы в качестве базового типа для указателей;
- оператор WITH с вариантами;
- оператор FOR.

Отдельного внимания заслуживает само описание, с которым вам предстоит познакомиться. Вирт и его соавтор достигли совершенства не только в искусстве разработки, но, несомненно, и в деле описания языков программирования. Поражают изумительная точность и краткость этого документа. Почти каждая его фраза превращается при написании компилятора в конкретные строки программного кода.

Возникшие при переводе описания Оберона-2 на русский язык терминологические вопросы решались исходя из следующих соображений: предпочтительным является буквальный перевод; недопустимо добавление терминов, отсутствующих в оригинале; должны быть учтены отечественные традиции в терминологии алголоподобных языков; предпочтительно использование терминов, привычных широкому кругу программистов, взамен узкоспециальных. Ниже приведен список терминов, перевод которых не представляется очевидным.

(direct) base type	(непосредственный) базовый тип
array compatible	совместимый массив
array type	тип массив
assignment compatible	совместимый по присваиванию
basic type	основной тип
browser	смотритель
case statement	оператор CASE
character	символ, знак
declaration	объявление
designator	обозначение
direct extension	непосредственное расширение
equal types	равные типы
exit statement	оператор выхода
expression compatible	совместимое выражение
for statement	оператор FOR
function procedure	процедура-функция
if statement	оператор IF

loop statement	оператор LOOP
matching	совпадение
operator	операция
pointer type	тип указатель
predeclared	стандартный
private field	скрытое поле
proper procedure	собственно процедура
public field	доступное поле
qualified	уточненный
real	вещественный
record type	тип запись
repeat statement	оператор REPEAT
return statement	оператор возврата
same type	одинаковый тип
scale factor	порядок
scope	область действия
statement	оператор
string	строка
symbol	слово

type extension	расширение типа
type guard	охрана типа
type inclusion	поглощение типа
type tag	тег
type test	проверка типа
type-bound procedures	связанные с типом процедуры
while statement	оператор WHILE
with statement	оператор WITH

1. Введение

Оберон-2 — язык программирования общего назначения, продолжающий традицию языков Паскаль и Модула-2. Его основные черты — блочная структура, модульность, раздельная компиляция, статическая типизация со строгим контролем соответствия типов (в том числе межмодульным), а также расширение типов и связанные с типами процедуры.

Расширение типов делает Оберон-2 объектно-ориентированным языком. Объект — это переменная абстрактного типа, содержащая данные (состояние объекта) и процедуры, которые оперируют этими данными. Абстрактные типы данных определены как расширяемые записи. Оберон-2 перекрывает большинство терминов объектно-ориентированных языков привычным словарем языков императивных, обходясь минимумом понятий в рамках тех же концепций.

Этот документ не является учебником программирования. Он преднамеренно краток. Его назначение — служить справочни-

ком для программистов, разработчиков компиляторов и авторов руководств. Если о чем-то не сказано, то обычно сознательно: или потому, что это следует из других правил языка, или потому, что потребовалось бы определять то, что фиксировать для общего случая представляется неразумным.

В приложении A[54] определены некоторые термины, которые используются при описании правил соответствия типов Оберона-2. В тексте эти термины выделены курсивом, чтобы подчеркнуть их специальное значение (например, *одинаковый* тип).

2. Синтаксис

Для описания синтаксиса Оберона-2 используются Расширенные Бэкуса-Наура Формы (РБНФ). Варианты разделяются знаком |. Квадратные скобки [и] означают необязательность записанного внутри них выражения, а фигурные скобки { и } означают его повторение (возможно 0 раз). Нетерминальные символы начинаются с заглавной буквы (например, Оператор). Терминальные символы или начинаются малой буквой (например, идент), или записываются целиком заглавными буквами (например, BEGIN), или заключаются в кавычки (например, ":=").

3. Словарь и представление

Для представления терминальных символов предусматривается использование набора знаков ASCII. Слова языка — это идентификаторы, числа, строки, операции и разделители. Должны соблюдаться следующие лексические правила. Пробелы и концы строк не должны встречаться внутри слов (исключая комментарии и пробелы в символьных строках). Пробелы и концы строк игнорируются, если они не существенны для отделения двух по-

[54] Здесь и далее в спецификации Оберона-2 имеются в виду приложения к спецификации (они обозначены буквами латинского алфавита), а не к книге (эти приложения пронумерованы). *Примеч. перев.*

следовательных слов. Заглавные и строчные буквы считаются различными.

1. *Идентификаторы* — последовательности букв и цифр. Первый символ должен быть буквой.

    идент = буква {буква | цифра}.

Примеры: `x  Scan  Oberon2  GetSymbol  firstLetter`

2. *Числа* — целые или вещественные (без знака) константы. Типом целочисленной константы считается минимальный тип, которому принадлежит ее значение (см. 6.1). Если константа заканчивается буквой `H`, она является шестнадцатеричной, иначе — десятичной.

Вещественное число всегда содержит десятичную точку. Оно может также содержать десятичный порядок. Буква E (или D) означает "умножить на десять в степени". Вещественное число относится к типу REAL кроме случая, когда у него есть порядок, содержащий букву D. В этом случае оно относится к типу LONGREAL.

```
число = целое | вещественное.

целое = цифра {цифра} | цифра {шестнЦифра} "H".

вещественное = цифра {цифра} "." {цифра} [Порядок].

Порядок = ("E" | "D") ["+" | "-"] цифра {цифра}.

шестнЦифра = цифра | "A" | "B" | "C" | "D" | "E" | "F".

цифра = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

Примеры:

`1991`	INTEGER	1991
`0DH`	SHORTINT	13
`12.3`	REAL	12.3

`4.567E8`	REAL	456 700 000
`0.57712566D-6`	LONGREAL	0.00000057712566

3. *Символьные* константы обозначаются порядковым номером символа в шестнадцатеричной записи, оканчивающейся буквой X.

   символ = цифра {шестнЦифра} "X".

4. *Строки* — последовательности символов, заключенные в одиночные (') или двойные (") кавычки. Открывающая кавычка должна быть такой же, что и закрывающая и не должна встречаться внутри строки. Число символов в строке называется ее *длиной*. Строка длины 1 может использоваться везде, где допустима символьная константа и наоборот.

   строка = '"' {символ} '"' | "'" {символ} "'".

Примеры: `"Oberon-2"` `"Don't worry!"` `"x"`

5. *Операции* и *разделители* — это специальные символы, пары символов или зарезервированные слова, перечисленные ниже. Зарезервированные слова состоят исключительно из заглавных букв и не могут использоваться как идентификаторы.

```
+     :=    ARRAY    IMPORT   RETURN
-     ^     BEGIN    IN       THEN
*     =     BY       IS       TO
/     #     CASE     LOOP     TYPE
~     <     CONST    MOD      UNTIL
&     >     DIV      MODULE   VAR
.     <=    DO       NIL      WHILE
,     >=    ELSE     OF       WITH
;     ..    ELSIF    OR
|     :     END      POINTER
(     )     EXIT     PROCEDURE
[     ]     FOR      RECORD
{     }     IF       REPEAT
```

6. *Комментарии* могут быть вставлены между любыми двумя словами программы. Это произвольные последовательности символов, начинающиеся скобкой `(*` и оканчивающиеся `*)`.

Комментарии могут быть вложенными. Они не влияют на смысл программы.

4. Объявления и области действия

Каждый идентификатор, встречающийся в программе, должен быть объявлен, если это не стандартный идентификатор. Объявления задают некоторые постоянные свойства объекта, например, является ли он константой, типом, переменной или процедурой. После объявления идентификатор используется для ссылки на соответствующий объект.

Область действия объекта *x* распространяется текстуально от точки его объявления до конца блока (модуля, процедуры или записи), в котором находится объявление. Для этого блока объект является *локальным*. Это разделяет области действия одинаково именованных объектов, которые объявлены во вложенных блоках. Правила для областей действия таковы:

1. Идентификатор не может обозначать больше чем один объект внутри данной области действия (то есть один и тот же идентификатор не может быть объявлен в блоке дважды);
2. Ссылаться на объект можно только изнутри его области действия;
3. Тип *T* вида POINTER TO *T1* (см. 6.4) может быть объявлен в точке, где *T1* еще неизвестен. Объявление *T1* должно следовать в том же блоке, в котором *T* является локальным;
4. Идентификаторы, обозначающие поля записи (см. 6.3) или процедуры, связанные с типом, (см. 10.2) могут употребляться только в обозначениях записи.

Идентификатор, объявленный в блоке модуля может сопровождаться при своем объявлении экспортной меткой ("*" или "–"), чтобы указать, что он экспортируется. Идентификатор *x*, экспортируемый модулем *M*, может использоваться в других модулях,

если они импортируют *M* (см. гл. 11). Тогда идентификатор обозначается в этих модулях *M.x* и называется *уточненным идентификатором*. Переменные и поля записей, помеченные знаком "–" в их объявлении, предназначены *только для чтения* в модулях-импортерах.

```
УточнИдент      = [идент "."] идент.
ИдентОпр  = идент ["*" | "–"].
```

Следующие идентификаторы являются стандартными; их значение определено в указанных разделах:

ABS	(10.3)	LEN	(10.3)
ASH	(10.3)	LONG	(10.3)
BOOLEAN	(6.1)	LONGINT	(6.1)
CAP	(10.3)	LONGREAL	(6.1)
CHAR	(6.1)	MAX	(10.3)
CHR	(10.3)	MIN	(10.3)
COPY	(10.3)	NEW	(10.3)
DEC	(10.3)	ODD	(10.3)
ENTIER	(10.3)	ORD	(10.3)
EXCL	(10.3)	REAL	(6.1)
FALSE	(6.1)	SET	(6.1)
HALT	(10.3)	SHORT	(10.3)
INC	(10.3)	SHORTINT	(6.1)
INCL	(10.3)	SIZE	(10.3)
INTEGER	(6.1)	TRUE	(6.1)

5. Объявления констант

Объявление константы связывает ее идентификатор с ее значением.

ОбъявлениеКонстанты = ИдентОпр "="
КонстантноеВыражение.

КонстантноеВыражение = Выражение.

Константное выражение — это выражение, которое может быть вычислено по его тексту без фактического выполнения программы. Его операнды — константы (Гл. 8) или стандартные функции (Гл. 10.3), которые могут быть вычислены во время компиляции. Примеры объявлений констант:

```
N = 100
limit = 2*N – 1
fullSet = {MIN(SET)..MAX(SET)}
```

6. Объявления типов

Тип данных определяет набор значений, которые могут принимать переменные этого типа, и набор применимых операций. Объявление типа связывает идентификатор с типом. В случае структурированных типов (массивы и записи) объявление также определяет структуру переменных этого типа. Структурированный тип не может содержать сам себя.

ОбъявлениеТипа = ИдентОпр "=" Тип.

Тип = УточнИдент | ТипМассив | ТипЗапись |
ТипУказатель | ПроцедурныйТип.

Примеры:

```
Table = ARRAY N OF REAL
Tree = POINTER TO Node
Node =  RECORD
 key : INTEGER;
 left, right: Tree
END
CenterTree = POINTER TO CenterNode
```

```
CenterNode = RECORD (Node)
 width: INTEGER;
 subnode: Tree
END
Function = PROCEDURE(x: INTEGER): INTEGER
```

6.1 Основные типы

Основные типы обозначаются стандартными идентификаторами. Соответствующие операции определены в 8.2, а стандартные функции в 10.3. Предусмотрены следующие основные типы:

1. BOOLEAN	логические значения TRUE и FALSE.
2. CHAR	символы расширенного набора . ASCII (0X .. 0FFX).
3. SHORTINT	целые в интервале от MIN(SHORTINT) до MAX(SHORTINT).
4. INTEGER	целые в интервале от MIN(INTEGER) до MAX(INTEGER).
5. LONGINT	целые в интервале от MIN(LONGINT) до MAX(LONGINT).
6. REAL	вещественные числа в интервале от MIN(REAL) до MAX(REAL).
7. LONGREAL	вещественные числа от MIN(LONGREAL) до MAX(LONGREAL).
8. SET	множество из целых от 0 до MAX(SET).

Типы от 3 до 5 — *целые типы*, типы 6 и 7 — *вещественные типы*, а вместе они называются *числовыми типами*. Эти типы образуют иерархию; больший тип поглощает меньший тип:

LONGREAL >= REAL >= LONGINT >= INTEGER >= SHORTINT

6.2 Тип массив

Массив — структура, состоящая из определенного количества элементов одного типа, называемого *типом элементов*. Число

элементов массива называется его *длиной*. Элементы массива обозначаются индексами, которые являются целыми числами от 0 до длины массива минус 1.

ТипМассив = ARRAY [Длина {"," Длина}] OF Тип.

Длина = КонстантноеВыражение.

Тип вида

```
ARRAY L0, L1, ..., Ln OF T
```

понимается как сокращение

```
ARRAY L0 OF
   ARRAY L1 OF
   …
      ARRAY Ln OF T
```

Массивы, объявленные без указания длины, называются *открытыми массивами*. Они могут использоваться только в качестве базового типа указателя (см. 6.4), типа элементов открытых массивов и типа формального параметра (см. 10.1). Примеры:

```
ARRAY 10, N OF INTEGER
ARRAY OF CHAR
```

6.3 Тип запись

Тип запись — структура, состоящая из фиксированного числа элементов, которые могут быть различных типов и называются *полями*. Объявление типа запись определяет имя и тип каждого поля. Область действия идентификаторов полей простирается от точки их объявления до конца объявления типа запись, но они также видимы внутри обозначений, ссылающихся на элементы переменных-записей (см. 8.1). Если тип запись экспортируется, то идентификаторы полей, которые должны быть видимы вне модуля, в котором объявлены, должны быть помечены. Они называются *доступными полями*; непомеченные элементы называются *скрытыми полями*.

```
ТипЗапись =
        RECORD ["(" БазовыйТип ")"]
        СписокПолей {";" СписокПолей} END.

БазовыйТип = УточнИдент.

СписокПолей = [СписокИдент ":" Тип].
```

Тип запись может быть объявлен как расширение другого типа запись. В примере

```
T0 = RECORD x: INTEGER END
T1 = RECORD (T0) y: REAL END
```

T1 — (непосредственное) расширение *T0*, а *T0* — (непосредственный) базовый тип *T1* (см. Прил. А). Расширенный тип *T1* состоит из полей своего базового типа и полей, которые объявлены в *T1*. Все идентификаторы, объявленные в расширенной записи, должны быть отличны от идентификаторов, объявленных в записи(записях) ее базового типа.

Примеры объявлений типа запись:

```
RECORD
   day, month, year: INTEGER
END
RECORD
   name, firstname: ARRAY 32 OF CHAR;
   age: INTEGER;
   salary: REAL
END
```

6.4 Тип указатель

Переменные-указатели типа *P* принимают в качестве значений указатели на переменные некоторого типа *T*. *T* называется базовым типом указателя типа *P* и должен быть типом массив или запись. Типы указатель заимствуют отношение расширения своих базовых типов: если тип *T1* — расширение *T* и *P1* — это тип POINTER TO *T1*, то *P1* — также является расширением *P*.

```
ТипУказатель = POINTER TO Тип.
```

Если *p* — переменная типа *P* = POINTER TO *T*, вызов стандартной процедуры *NEW(p)* (см. 10.3) размещает переменную типа *T* в свободной памяти. Если *T* — тип запись или тип массив с фиксированной длиной, размещение должно быть выполнено вызовом *NEW(p)*; если тип *T* — *n*-мерный открытый массив, размещение должно быть выполнено вызовом *NEW(p, e0, ..., en-1),* чтобы *T* был размещен с длинами, заданными выражениями *e0, ..., en-1*. В любом случае указатель на размещенную переменную присваивается *p*. Переменная *p* имеет тип *P*. Переменная *p^*, на которую ссылается *p* (*динамическая переменная*) имеет тип *T*. Любая переменная-указатель может принимать значение NIL, которое не указывает ни на какую переменную вообще.

6.5 Процедурные типы

Переменные процедурного типа *T*, имеют значением процедуру (или NIL). Если процедура *P* присваивается переменной типа *T*, списки формальных параметров (см. Гл. 10.1) *P* и *T* должны *совпадать* (см. Прил. А). *P* не должна быть стандартной или связанной с типом процедурой, и не может быть локальной в другой процедуре.

```
ПроцедурныйТип = PROCEDURE [ФормальныеПараметры].
```

7. Объявления переменных

Объявления переменных дают описание переменных, определяя идентификатор и тип данных для них.

```
ОбъявлениеПеременных = СписокИдент ":" Тип.
```

Переменные типа запись и указатель имеют как статический тип (тип, с которым они объявлены — называемый просто их типом), так и динамический тип (тип их значения во время выполнения). Для указателей и параметров-переменных типа запись динамический тип может быть расширением их статического типа. Статический тип определяет какие поля записи доступны.

Динамический тип используется для вызова связанных с типом процедур (см. 10.2).

Примеры объявлений переменных (со ссылками на примеры из Гл. 6):

```
i, j, k: INTEGER
x, y: REAL
p, q: BOOLEAN
s: SET
F: Function
a: ARRAY 100 OF REAL
w: ARRAY 16 OF RECORD
      name: ARRAY 32 OF CHAR;
      count: INTEGER
   END
t, c: Tree
```

8. Выражения

Выражения — конструкции, задающие правила вычисления по значениям констант и текущим значениям переменных других значений путем применения операций и процедур-функций. Выражения состоят из операндов и операций. Круглые скобки могут использоваться для группировки операций и операндов.

8.1 Операнды

За исключением конструкторов множества и литералов (чисел, символьных констант или строк), операнды представлены обозначениями. Обозначение содержит идентификатор константы, переменной или процедуры. Этот идентификатор может быть уточнен именем модуля (см. Гл. 4 и 11) и может сопровождаться селекторами если обозначенный объект — элемент структуры.

```
Обозначение =
      УточнИдент {"." идент | "[" СписокВыражений "]" |
       "^" | "(" УточнИдент ")" }.

СписокВыражений = Выражение {"," Выражение}.
```

Если *a* — обозначение массива, *a[e]* означает элемент *a*, чей индекс — текущее значение выражения *e*. Тип *e* должен быть *целым* типом. Обозначение вида *a[e0, e1, ..., en]* применимо вместо *a[e0] [e1] ... [en]*. Если *r* обозначает запись, то *r.f* означает поле *f* записи *r* или процедуру *f*, связанную с динамическим типом *r* (Гл. 10.2). Если *p* обозначает указатель, *p^* означает переменную, на которую ссылается *p*. Обозначения *p^.f* и *p^[e]* могут быть сокращены до *p.f* и *p[e]*, то есть запись и индекс массива подразумевают разыменование. Если *a* или *r* доступны только для чтения, то *a[e]* и *r.f* также предназначены только для чтения.

Охрана типа v(T) требует, чтобы динамическим типом *v* был *T* (или расширение *T*), то есть выполнение программы прерывается, если динамический тип *v* — не *T* (или расширение *T*). В пределах такого обозначения *v* воспринимается как имеющая статический тип *T*. Охрана применима, если

1. *v* — параметр-переменная типа запись, или *v* — указатель, и если

2. *T* — расширение статического типа *v*

Если обозначенный объект — константа или переменная, то обозначение ссылается на их текущее значение. Если он — процедура, то обозначение ссылается на эту процедуру, если только обозначение не сопровождается (возможно пустым) списком параметров. В последнем случае подразумевается активация процедуры и подстановка значения результата, полученного при ее исполнении. Фактические параметры должны соответствовать формальным параметрам как и при вызовах собственно процедуры (см. 10.1).

Примеры обозначений (со ссылками на примеры из Гл. 7):

```
i                                (INTEGER)
```

```
a[i]                    (REAL)
w[3].name[i]            (CHAR)
t.left.right            (Tree)
t(CenterTree).subnode   (Tree)
```

8.2 Операции

В выражениях синтаксически различаются четыре класса операций с разными приоритетами (порядком выполнения). Операция ~ имеет самый высокий приоритет, далее следуют операции типа умножения, операции типа сложения и отношения. Операции одного приоритета выполняются слева направо. Например, x–y–z означает (x–y) –z.

```
Выражение =
    ПростоеВыражение [Отношение ПростоеВыражение].

ПростоеВыражение =
    ["+" | "-"] Слагаемое {ОперацияСложения Слагаемое}.

Слагаемое =
    Множитель {ОперацияУмножения Множитель}.

Множитель =
    Обозначение [ФактическиеПараметры]|
    число | символ |
    строка | NIL | Множество | "(" Выражение ")" |
    "~" Множитель.

Множество = "{" [Элемент {"," Элемент}] "}".

Элемент = Выражение [".." Выражение].

ФактическиеПараметры = "(" [СписокВыражений] ")".

Отношение = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.

ОперацияСложения = "+" | "-" | OR.
```

ОперацияУмножения = "*" | "/" | DIV | MOD | "&".

Предусмотренные операции перечислены в следующих таблицах. Некоторые операции применимы к операндам различных типов, обозначая разные действия. В этих случаях фактическая операция определяется типом операндов. Операнды должны быть *совместимыми выражениями* для данной операции (см. Прил. А).

8.2.1 Логические операции

OR	Логическая дизъюнкция	p OR q	«если p, то TRUE, иначе q»
&	Логическая конъюнкция	p & q	«если p то q, иначе FALSE»
~	Отрицание	~ p	«не p»

Эти операции применимы к операндам типа BOOLEAN и дают результат типа BOOLEAN.

8.2.2 Арифметические операции

+ сумма

– разность

* произведение

/ вещественное деление

DIV деление нацело

MOD остаток

Операции +, –, *, и / применимы к операндам *числовых типов*. Тип их результата — тип того операнда, который поглощает тип другого операнда, кроме деления (/), чей результат — наименьший *вещественный тип*, который поглощает типы обоих опе-

рандов. При использовании в качестве одноместной операции «-» обозначает перемену знака, а «+» — тождественную операцию. Операции DIV и MOD применимы только к целочисленным операндам. Они связаны следующими формулами, определенными для любого *x* и положительного делителя *y*:

x = (x DIV y) * y + (x MOD y)

0 < = (x MOD y) < y

Примеры:

x	y	x DIV y	x MOD y
5	3	1	2
–5	3	–2	1

8.2.3 Операции над множествами

\+ объединение

– разность (x – y = x * (–y))

* пересечение

/ симметрическая разность множеств (x / y = (x–y) + (y–x))

Эти операции применимы к операндам типа SET и дают результат типа SET. Одноместный «минус» обозначает дополнение *x*, то есть -*x* это множество целых между 0 и MAX(SET), которые не являются элементами *x*. Операции с множествами не ассоциативны *((a+b)–c # a+(b–c))*.

Конструктор множества задает значение множества списком элементов, заключенным в фигурные скобки. Элементы должны быть целыми в диапазоне 0 .. MAX(SET). Диапазон *a..b* обозначает все целые числа в интервале [*a, b*].

8.2.4 Отношения

=	равно
#	не равно
<	меньше
<=	меньшее или равно
>	больше
>=	больше или равно
IN	принадлежность множеству
IS	проверка типа

Отношения дают результат типа BOOLEAN. Отношения =, #, <, <=, > и >= применимы к *числовым типам*, типу CHAR, строкам и символьным массивам, содержащим в конце 0X. Отношения = и # кроме того применимы к типам BOOLEAN и SET, а также к указателям и процедурным типам (включая значение NIL). *x IN s* означает "*x* является элементом *s*". *x* должен быть *целого типа*, а *s* — типа SET. *v IS T* означает «динамический тип *v* есть *T* (или расширение *T*)» и называется проверкой типа. Проверка типа применима, если

1. *v* — параметр-переменная типа запись, или *v* - указатель, и если
2. *T* — расширение статического типа *v*

Примеры выражений (со ссылками на примеры из Гл. 7):

`1991`	INTEGER
`i DIV 3`	INTEGER
`~p OR q`	BOOLEAN
`(i+j) * (i-j)`	INTEGER
`s - {8, 9, 13}`	SET

`i + x`	REAL
`a[i+j] * a[i-j]`	REAL
`(0<=i) & (i<100)`	BOOLEAN
`t.key = 0`	BOOLEAN
`k IN {i..j-1}`	BOOLEAN
`w[i].name <= "John"`	BOOLEAN
`t IS CenterTree`	BOOLEAN

9. Операторы

Операторы обозначают действия. Есть простые и структурные операторы. Простые операторы не содержат в себе никаких частей, которые являются самостоятельными операторами. Простые операторы — присваивание, вызов процедуры, операторы возврата и выхода. Структурные операторы состоят из частей, которые являются самостоятельными операторами. Они используются, чтобы выразить последовательное и условное, выборочное и повторное исполнение. Оператор также может быть пустым, в этом случае он не обозначает никакого действия. Пустой оператор добавлен, чтобы упростить правила пунктуации в последовательности операторов.

Оператор =

[Присваивание | ВызовПроцедуры | ОператорIF | ОператорCASE | ОператорWHILE | ОператорREPEAT | ОператорFOR | ОператорLOOP | ОператорWITH | EXIT | RETURN [Выражение]].

9.1 Присваивания

Присваивание заменяет текущее значение переменной новым значением, определяемым выражением. Выражение должно быть *совместимо по присваиванию* с переменной (см. Приложе-

ние А). Знаком операции присваивания является «:=», который читается «присвоить».

Присваивание = Обозначение ":=" Выражение.

Если выражение *e* типа *Te* присваивается переменной *v* типа *Tv,* имеет место следующее:

Если *Tv* и *Te* — записи, то в присваивании участвуют только те поля *Te*, которые также имеются в *Tv* (проецирование); динамический тип *v* и статический тип *v* должны быть *одинаковы*, и не изменяются присваиванием;

Если *Tv* и *Te* — типы указатель, динамическим типом *v* становится динамический тип *e;*

Если *Tv* это ARRAY *n* OF CHAR, а *e* — строка длины *m* < *n, v [i]* присваиваются значения *ei* для *i = 0 .. m-1*, а *v[m]* получает значение 0X.

Примеры присваиваний (со ссылками на примеры из Гл. 7):

```
i := 0
p := i = j
x := i + 1
k := log2(i+j)
F := log2                    (* см. 10.1 *)
s := {2, 3, 5, 7, 11, 13}
a[i] := (x+y)*(x-y)
t.key := i
w[i+1].name := "John"
t := c
```

9.2 Вызовы процедур

Вызов процедуры активирует процедуру. Он может содержать список фактических параметров, которые заменяют соответствующие формальные параметры, определенные в объявлении

процедуры (см. Гл. 10). Соответствие устанавливается в порядке следования параметров в списках фактических и формальных параметров. Имеются два вида параметров: параметры-переменные и параметры-значения.

Если формальный параметр — параметр-переменная, соответствующий фактический параметр должен быть обозначением переменной. Если фактический параметр обозначает элемент структурной переменной, селекторы компонент вычисляются, когда происходит замена формальных параметров фактическими, то есть перед выполнением процедуры. Если формальный параметр — параметр-значение, соответствующий фактический параметр должен быть выражением. Это выражение вычисляется перед вызовом процедуры, а полученное в результате значение присваивается формальному параметру (см. также 10.1).

ВызовПроцедуры = Обозначение [ФактическиеПараметры].

Примеры:

```
WriteInt(i*2+1) (* см. 10.1 *)
INC(w[k].count)
t.Insert("John")(* см. 11 *)
```

9.3 Последовательность операторов

Последовательность операторов, разделенных точкой с запятой, означает поочередное выполнение действий, заданных составляющими операторами.

ПоследовательностьОператоров = Оператор {";" Оператор}.

9.4 Операторы IF

ОператорIF =

IF Выражение THEN ПоследовательностьОператоров
{ELSIF Выражение THEN ПоследовательностьОператоров}

[ELSE ПоследовательностьОператоров]
END.

Операторы IF задают условное выполнение входящих в них последовательностей операторов. Логическое выражение, предшествующие последовательности операторов, будем называть условием[55]. Условия проверяются последовательно одно за другим, пока очередное не окажется равным TRUE, после чего выполняется связанная с этим условием последовательность операторов. Если ни одно условие не удовлетворено, выполняется последовательность операторов, записанная после слова ELSE, если оно имеется.

Пример:

```
IF (ch >= "A") & (ch <= "Z") THEN ReadIdentifier
ELSIF (ch >= "0") & (ch <= "9") THEN ReadNumber
ELSIF (ch = "'") OR (ch = '"') THEN ReadString
ELSE SpecialCharacter
END
```

9.5 Операторы CASE

Операторы CASE определяют выбор и выполнение последовательности операторов по значению выражения. Сначала вычисляется выбирающее выражение, а затем выполняется та последовательность операторов, чей список меток варианта содержит полученное значение. Выбирающее выражение должно быть такого *целого типа*, который включает типы всех меток вариантов, или и выбирающее выражение и метки вариантов должны иметь тип CHAR. Метки варианта — константы, и ни одно из их значений не должно употребляться больше одного раза. Если значение выражения не совпадает с меткой ни одного из вариантов, выбирается последовательность операторов после слова ELSE, если оно есть, иначе программа прерывается.

[55] В оригинале — guard. *Прим. перев*

ОператорCASE =

CASE Выражение OF Вариант {" | " Вариант}

[ELSE ПоследовательностьОператоров] END.

Вариант =

[СписокМетокВарианта ":"

ПоследовательностьОператоров].

СписокМетокВарианта = МеткиВарианта {"," МеткиВарианта }.

МеткиВарианта =

КонстантноеВыражение [".." КонстантноеВыражение].

Пример:

```
CASE ch OF
 "A".."Z": ReadIdentifier
|   "0".."9": ReadNumber
|   "'", '"': ReadString
ELSE SpecialCharacter
END
```

9.6 Операторы WHILE

Операторы WHILE задают повторное выполнение последовательности операторов, пока логическое выражение (условие) остается равным TRUE. Условие проверяется перед каждым выполнением последовательности операторов.

ОператорWHILE =

WHILE Выражение DO

ПоследовательностьОператоров

END.

Примеры:

```
WHILE i > 0 DO i := i DIV 2; k := k + 1 END
WHILE (t # NIL) & (t.key # i) DO t := t.left END
```

9.7 Операторы REPEAT

Оператор REPEAT определяет повторное выполнение последовательности операторов пока условие, заданное логическим выражением, не удовлетворено. Последовательность операторов выполняется по крайней мере один раз.

ОператорREPEAT =

REPEAT ПоследовательностьОператоров UNTIL Выражение.

9.8 Операторы FOR

Оператор FOR определяет повторное выполнение последовательности операторов фиксированное число раз для прогрессии значений целочисленной переменной, называемой *управляющей переменной* оператора FOR.

ОператорFOR =

FOR идент ":=" Выражение TO Выражение

[BY КонстантноеВыражение]

DO ПоследовательностьОператоров END.

Оператор

```
FOR v := beg TO end BY step DO statements END
```

эквивалентен

```
temp := end; v := beg;
IF step > 0 THEN
 WHILE v <= temp DO statements; v := v + step END
ELSE
 WHILE v >= temp DO statements; v := v + step END
END
```

temp и *v* имеют *одинаковый* тип. Шаг (*step*) должен быть отличным от нуля константным выражением. Если шаг не определен, он принимается равным 1.

Примеры:

```
FOR i := 0 TO 79 DO k := k + a[i] END
FOR i := 79 TO 1 BY -1 DO a[i] := a[i-1] END
```

9.9 Операторы LOOP

Оператор `LOOP` определяет повторное выполнение последовательности операторов. Он завершается после выполнения оператора выхода внутри этой последовательности (см. 9.10).

ОператорLOOP = LOOP ПоследовательностьОператоров END.

Пример:

```
LOOP
   ReadInt(i);
   IF i < 0 THEN EXIT END;
   WriteInt(i)
END
```

Операторы LOOP полезны, чтобы выразить повторения с несколькими точками выхода, или в случаях, когда условие выхода находится в середине повторяемой последовательности операторов.

9.10 Операторы возврата и выхода

Оператор возврата выполняет завершение процедуры. Он обозначается словом RETURN, за которым следует выражение, если процедура является процедурой-функцией. Тип выражения должен быть *совместим по присваиванию* (см. Приложение A) с типом результата, определенным в заголовке процедуры (см. Гл. 10).

Процедуры-функции должны быть завершены оператором возврата, задающим значение результата. В собственно процедурах оператор возврата подразумевается в конце тела процедуры. Любой явный оператор появляется, следовательно, как дополнительная (вероятно, для исключительной ситуации) точка завершения.

Оператор выхода обозначается словом EXIT. Он определяет завершение охватывающего оператора LOOP и продолжение выполнения программы с оператора, следующего за оператором

LOOP. Оператор выхода связан с содержащим его оператором цикла контекстуально, а не синтаксически.

9.11 Операторы WITH

Операторы WITH выполняют последовательность операторов в зависимости от результата проверки типа и применяют охрану типа к каждому вхождению проверяемой переменной внутри этой последовательности операторов.

```
ОператорWITH    =
        WITH Охрана DO ПоследовательностьОператоров
        {"|" Охрана DO ПоследовательностьОператоров}
        [ELSE ПоследовательностьОператоров] END.

Охрана      = УточнИдент ":" УточнИдент.
```

Если *v* — параметр-переменная типа запись или переменная-указатель, и если ее статический тип *T0*, оператор

```
WITH v: T1 DO S1 | v: T2 DO S2 ELSE S3 END
```

имеет следующий смысл: если динамический тип *v* — *T1*, то выполняется последовательность операторов *S1* в которой *v* воспринимается так, будто она имеет статический тип *T1*; иначе, если динамический тип *v* — *T2*, выполняется *S2*, где *v* воспринимается как имеющая статический тип *T2*; иначе выполняется *S3*. *T1* и *T2* должны быть расширениями *T0*. Если ни одна проверка типа не удовлетворена, а ELSE отсутствует, программа прерывается.

Пример:

```
WITH t: CenterTree DO i := t.width; c := t.subnode END
```

10. Объявления процедур

Объявление процедуры состоит из заголовка процедуры и тела процедуры. Заголовок определяет имя процедуры и формальные параметры. Для связанных с типом процедур в объявлении так-

же определяется параметр-приемник. Тело содержит объявления и операторы. Имя процедуры повторяется в конце объявления процедуры.

Имеются два вида процедур: собственно процедуры и процедуры-функции. Последние активизируются обозначением функции как часть выражения и возвращают результат, который является операндом выражения. Собственно процедуры активизируются вызовом процедуры. Процедура является процедурой-функцией, если ее формальные параметры задают тип результата. Тело процедуры-функции должно содержать оператор возврата, который определяет результат.

Все константы, переменные, типы и процедуры, объявленные внутри тела процедуры, локальны в процедуре. Поскольку процедуры тоже могут быть объявлены как локальные объекты, объявления процедур могут быть вложенными. Вызов процедуры изнутри ее объявления подразумевает рекурсивную активацию.

Объекты, объявленные в окружении процедуры, также видимы в тех частях процедуры, в которых они не перекрыты локально объявленным объектом с тем же самым именем.

```
ОбъявлениеПроцедуры =
        ЗаголовокПроцедуры ";" ТелоПроцедуры идент.

ЗаголовокПроцедуры =
        PROCEDURE [Приемник] ИдентОпр
        [ФормальныеПараметры].

ТелоПроцедуры =
        ПоследовательностьОбъявлений
        [BEGIN ПоследовательностьОператоров] END.

ПослОбъявлений =
        {CONST {ОбъявлениеКонстант ";"} |
```

```
    TYPE{ОбъявлениеТипов ";"}
    | VAR {ОбъявлениеПеременных ";"}}
    {ОбъявлениеПроцедуры ";"|ОпережающееОбъявление";"}
ОпережающееОбъявление =
    PROCEDURE"^" [Приемник] ИдентОпр
    [ФормальныеПараметры].
```

Если объявление процедуры содержит параметр-приемник, процедура рассматривается как связанная с типом (см. 10.2). Опережающее объявление служит, чтобы разрешить ссылки на процедуру, чье фактическое объявление появляется в тексте позже. Списки формальных параметров опережающего объявления и фактического объявления должны *совпадать* (см. Прил. А).

10.1 Формальные параметры

Формальные параметры — идентификаторы, объявленные в списке формальных параметров процедуры. Им соответствуют фактические параметры, которые задаются при вызове процедуры. Подстановка фактических параметров вместо формальных происходит при вызове процедуры. Имеются два вида параметров: *параметры-значения* и *параметры-переменные*, обозначаемые в списке формальных параметров отсутствием или наличием ключевого слова VAR. Параметры-значения — это локальные переменные, которым в качестве начального присваивается значение соответствующего фактического параметра. Параметры-переменные соответствуют фактическим параметрам, которые являются переменными, и означают эти переменные. Область действия формального параметра простирается от его объявления до конца блока процедуры, в котором он объявлен. Процедура-функция без параметров должна иметь пустой список параметров. Она должна вызываться обозначением функции, чей список фактических параметров также пуст. Тип результата процедуры не может быть ни записью, ни массивом.

ФормальныеПараметры =

"(" [СекцияФП {";" СекцияФП }] ")" [":" УточненИдент].

СекцияФП =

[VAR] идент {"," идент} ":" Тип.

Пусть *Tf* — тип формального параметра *f* (не открытого массива) и *Ta* — тип соответствующего фактического параметра *a*. Для параметров-переменных *Ta* и *Tf* должны быть *одинаковыми* типами или *Tf* должен быть типом запись, а *Ta* — расширением *Tf*. Для параметров-значений *a* должен быть *совместим по присваиванию* с *f* (см. Прил. А).

Если *Tf* — открытый массив, то *a* должен быть *совместимым массивом* для *f* (см. Прил. А). Длина *f* становится равной длине *a*.

Примеры объявлений процедур:

```
PROCEDURE ReadInt (VAR x: INTEGER);
   VAR i: INTEGER; ch: CHAR;
BEGIN
   i := 0; Read(ch);
   WHILE ("0" <= ch) & (ch <= "9") DO
      i := 10*i + (ORD(ch)-ORD("0")); Read(ch)
   END;
   x := i
END ReadInt

PROCEDURE WriteInt (x: INTEGER); (*0 <= x <100000*)
   VAR i: INTEGER; buf: ARRAY 5 OF INTEGER;
BEGIN
   i := 0;
   REPEAT buf[i] := x MOD 10; x := x DIV 10; INC(i)
   UNTIL x = 0;
   REPEAT DEC(i); Write(CHR(buf[i] + ORD("0")))
   UNTIL i = 0
END WriteInt

PROCEDURE WriteString (s: ARRAY OF CHAR);
   VAR i: INTEGER;
BEGIN i := 0;
   WHILE (i < LEN(s)) & (s[i] # 0X) DO
      Write(s[i]); INC(i)
```

```
   END
END WriteString;

PROCEDURE log2(x: INTEGER): INTEGER;
   VAR y: INTEGER;(*предполагается x>0*)
BEGIN
   y := 0; WHILE x > 1 DO x := x DIV 2; INC(y) END;
   RETURN y
END log2
```

10.2 Процедуры, связанные с типом

Глобально объявленные процедуры могут быть ассоциированы с типом запись, объявленным в том же самом модуле. В этом случае говорится, что процедуры *связаны* с типом запись[56]. Связь выражается типом приемника в заголовке объявления процедуры. Приемник может быть или параметром-переменной типа *T*, если *T* — тип запись, или параметром-значением типа POINTER TO *T* (где *T* - тип запись). Процедура, связанная с типом *T*, рассматривается как локальная для него.

ЗаголовокПроцедуры =

PROCEDURE [Приемник] ИдентОпр

[ФормальныеПараметры].

Приемник = "(" [VAR] имя ":" имя ")".

Если процедура *P* связана с типом *T0*, она неявно также связана с любым типом *T1*, который является расширением *T0*. Однако процедура *P′* (с тем же самым именем, что и *P*) может быть явно связана с *T1*, перекрывая в этом случае связывание с *P*. *P′* рассматривается как переопределение *P* для *T1*. Формальные параметры *P* и *P′* должны *совпадать* (см. Прил. А).

Если *P* и *T1* экспортируются (см. Главу 4), *P′* также должна экспортироваться.

[56] Будем называть их также «связанные процедуры». *Прим. перев.*

Если *v* — обозначение, а *P* — связанная процедура, то *v.P* обозначает процедуру *P*, связанную с динамическим типом *v*. Заметим, что это может быть процедура, отличная от той, что связана со статическим типом *v*. *v* передается приемнику процедуры *P* согласно правилам передачи параметров, определенным в Главе 10.1.

Если *r* — параметр-приемник, объявленный с типом *T*, *r.P^* обозначает (переопределенную) процедуру *P*, связанную с базовым для *T* типом. В опережающем объявлении связанной процедуры и в фактическом объявлении процедуры параметр-приемник должен иметь *одинаковый* тип. Списки формальных параметров в обоих объявлениях должны совпадать (см. Прил. А).

Примеры:

```
PROCEDURE (t: Tree) Insert (node: Tree);
   VAR p, father: Tree;
BEGIN p := t;
   REPEAT father := p;
      IF node.key = p.key THEN RETURN END;
      IF node.key < p.key THEN p := p.left
      ELSE p := p.right END
   UNTIL p = NIL;
   IF node.key < father.key THEN father.left := node
   ELSE father.right := node END;
   node.left := NIL; node.right := NIL
END Insert;

PROCEDURE (t: CenterTree) Insert (node: Tree);
(*переопределение*)
BEGIN
   WriteInt(node(CenterTree).width);
   t.Insert^(node)
      (* вызывает процедуру Insert, связанную с Tree *)
END Insert;
```

10.3 Стандартные процедуры

Следующая таблица содержит список стандартных процедур. Некоторые процедуры — обобщенные, то есть они применимы к операндам нескольких типов. Буква *v* обозначает переменную, *x* и *n* — выражения, *T* — тип.

Процедуры-функции

Название	Тип аргумента	Тип результата	Функция
ABS(x)	числовой тип	совпадает с типом x	абсолютное значение
ASH(x, n)	x, n: целый тип	LONGINT	арифметический сдвиг ($x*2^n$)
CAP(x)	CHAR	CHAR	x — буква: соответствующая заглавная буква
CHR(x)	целый тип	CHAR	символ с порядковым номером x
ENTIER(x)	вещественный тип	LONGINT	наибольшее целое, не превосходящее x
LEN(v, n)	v: массив; n: целая константа.	LONGINT	длина v в измерении n (первое измерение = 0)
LEN(v)	v: массив	LONGINT	равносильно LEN(v, 0)
LONG(x)	SHORTINT	INTEGER	тождество
	INTEGER	LONGINT	
	REAL	LONGREAL	
MAX(T)	T = основной тип	T	наибольшее значение типа T

Название	Тип аргумента	Тип результата	Функция
	T = SET	INTEGER	наибольший элемент множества
MIN(T)	T = основной тип	T	наименьшее значение типа T
	T = SET	INTEGER	0
ODD(x)	целый тип	BOOLEAN	x MOD 2 = 1
ORD(x)	CHAR	INTEGER	порядковый номер x
SHORT(x)	LONGINT	INTEGER	тождество
	INTEGER	SHORTINT	тождество
	LONGREAL	REAL	тождество (возможно усечение)
SIZE(T)	любой тип	целый тип	число байт, занимаемых T

Собственно процедуры

Название	Типы аргументов	Функция
ASSERT(x)	x: логическое выражение	прерывает выполнение программы, если не x
ASSERT(x, n)	x: логическое выражение; n: целая константа	прерывает выполнение программы, если не x

Название	Типы аргументов	Функция
COPY(x, v)	x: символьный массив, строка; v: символьный массив	v := x
DEC(v)	целый тип	v := v – 1
DEC(v, n)	v, n: целый тип	v := v – n
EXCL(v, x)	v: SET; x: целый тип	v := v – {x}
HALT(n)	целая константа	прерывает выполнение программы
INC(v)	целый тип	v := v + 1
INC(v, n)	v, n: целый тип	v := v + n
INCL(v, x)	v: SET; x: целый тип	v := v + {x}
NEW(v)	указатель на запись или массив фиксированной длины	размещает v^
NEW(v, x0, ..., xn)	v: указатель на открытый массив; xi: целый тип	размещает v^ с длинами x0.. xn

COPY разрешает присваивание строки или символьного массива, содержащего ограничитель 0X, другому символьному массиву. В случае необходимости, присвоенное значение усекается до длины получателя минус один. Получатель всегда будет содержать 0X как ограничитель. В ASSERT(x, n) и HALT(n), интерпретация *n* зависит от реализации основной системы.

11. Модули

Модуль — совокупность объявлений констант, типов, переменных и процедур вместе с последовательностью операторов, предназначенных для присваивания начальных значений переменным. Модуль представляет собой текст, который является единицей компиляции.

Модуль =
    MODULE идент ";" [СписокИмпорта]
     ПоследовательностьОбъявлений
    [BEGIN ПоследовательностьОператоров] END идент ".".

СписокИмпорта = IMPORT Импорт {"," Импорт} ";".

Импорт = [идент ":="] идент.

Список импорта определяет имена импортируемых модулей. Если модуль *A* импортируется модулем *M*, и *A* экспортирует идентификатор *x*, то *x* упоминается внутри *M* как *A.x*. Если *A* импортируется как *B:=A*, объект *x* должен вызываться как *B.x*. Это позволяет использовать короткие имена-псевдонимы в уточненных идентификаторах. Модуль не должен импортировать себя. Идентификаторы, которые экспортируются (то есть должны быть видимы в модулях-импортерах) нужно отметить экспортной меткой в их объявлении (см. Главу 4).

Последовательность операторов после символа BEGIN выполняется, когда модуль добавляется к системе (загружается). Это происходит после загрузки импортируемых модулей. Отсюда следует, что циклический импорт модулей запрещен. Отдельные (не имеющие параметров и экспортированные) процедуры могут быть активированы из системы. Эти процедуры служат командами (см. Приложение D1).

```
MODULE Trees;
(* экспорт: Tree, Node, Insert, Search, Write, Init *)
```

```
(* экспорт только для чтения: Node.name *)
 IMPORT Texts, Oberon;
 TYPE
    Tree* = POINTER TO Node;
    Node* = RECORD
       name-: POINTER TO ARRAY OF CHAR;
       left, right: Tree
    END;
 VAR w: Texts.Writer;

 PROCEDURE (t: Tree) Insert* (name: ARRAY OF CHAR);
    VAR p, father: Tree;
 BEGIN p := t;
    REPEAT father := p;
       IF name = p.name^ THEN RETURN END;
       IF name < p.name^ THEN p := p.left
       ELSE p := p.right END
    UNTIL p = NIL;
    NEW(p); p.left := NIL; p.right := NIL;
    NEW(p.name, LEN(name)+1);
    COPY(name, p.name^);
    IF name < father.name^ THEN father.left := p
    ELSE father.right := p END
 END Insert;

 PROCEDURE(t: Tree) Search*(name: ARRAY OF CHAR): Tree;
    VAR p: Tree;
 BEGIN p := t;
    WHILE (p # NIL) & (name # p.name^) DO
       IF name < p.name^ THEN p := p.left
       ELSE p := p.right END
    END;
    RETURN p
 END Search;

 PROCEDURE (t: Tree) Write*;
 BEGIN
    IF t.left # NIL THEN t.left.Write END;
    Texts.WriteString(w, t.name^); Texts.WriteLn(w);
    Texts.Append(Oberon.Log, w.buf);
    IF t.right # NIL THEN t.right.Write END
 END Write;

 PROCEDURE Init* (t: Tree);
 BEGIN NEW(t.name, 1); t.name[0] := 0X;
    t.left := NIL; t.right := NIL
 END Init;
```

```
BEGIN Texts.OpenWriter(w)
END Trees.
```

Приложение A: Определение терминов

Целые типы

SHORTINT, INTEGER, LONGINT

Вещественные типы

REAL, LONGREAL

Числовые типы

Целые типы, вещественные типы

Одинаковые типы

Две переменные *a* и *b* с типами *Ta* и *Tb* имеют *одинаковый* тип, если

1. *Ta* и *Tb* оба обозначены одним и тем же идентификатором типа, или
2. *Ta* объявлен равным *Tb* в объявлении типа вида *Ta = Tb*, или
3. *a* и *b* появляются в одном и том же списке идентификаторов переменных, полей записи или объявлении формальных параметров и не являются открытыми массивами.

Равные типы

Два типа *Ta* и *Tb* равны, если

1. *Ta* и *Tb* — *одинаковые* типы, или
2. *Ta* и *Tb* — типы открытый массив с *равными* типами элементов, или
3. *Ta* и *Tb* — процедурные типы, чьи списки формальных параметров *совпадают*.

Поглощение типов

Числовые типы *поглощают* (значения) меньших числовых типов согласно следующей иерархии:

LONGREAL >= REAL >= LONGINT >= INTEGER >= SHORTINT

Расширение типов (базовый тип)

В объявлении типа *Tb* = RECORD (*Ta*) ... END, *Tb* — непосредственное расширение *Ta*, а *Ta* — непосредственный базовый тип *Tb*. Тип *Tb* есть расширение типа *Ta* (*Ta* есть базовый тип *Tb*), если

1. *Ta* и *Tb* — *одинаковые* типы, или
2. *Tb* — непосредственное расширение типа, являющегося расширением *Ta*

Если *Pa* = POINTER TO Ta и *Pb* = POINTER TO *Tb*, то *Pb* есть расширение *Pa* (*Pa* есть базовый тип *Pb*), если *Tb* есть расширение *Ta*.

Совместимость по присваиванию

Выражение *e* типа *Te* *совместимо по присваиванию* с переменной *v* типа *Tv*, если выполнено одно из следующих условий:

1. *Te* и *Tv* — одинаковые типы;
2. *Te* и *Tv* — числовые типы и *Tv* *поглощает* *Te*;
3. *Te* и *Tv* — типы запись, *Te* есть *расширение* *Tv*, а v имеет динамический тип *Tv*;
4. *Te* и *Tv* — типы указатель и *Te* — расширение *Tv*;
5. *Tv* — тип указатель или процедурный тип, а *e* — NIL;
6. *Tv* — ARRAY n OF CHAR, *e* - строковая константа из *m* символов и *m* < *n*;

7. *Tv* — процедурный тип, а *e* — имя процедуры, чьи формальные параметры *совпадают* с параметрами *Tv*.

Совместимость массивов

Фактический параметр *a* типа *Ta* является *совместимым массивом* для формального параметра *f* типа *Tf*, если

1. *Tf* и *Ta* — *одинаковые* типы, или

2. *Tf* — открытый массив, *Ta* — любой массив, а их элементы — *совместимые массивы*, или

3. *f* — параметр-значение типа ARRAY OF CHAR, а фактический параметр *a* — строка.

Совместимость выражений

Для данной операции операнды являются *совместимыми выражениями*, если их типы соответствуют следующей таблице (в который указан также тип результата выражения). Символьные массивы, которые сравниваются, должны содержать в качестве ограничителя 0X. Тип T1 должен быть расширением типа T0:

Операция	Первый операнд	Второй операнд	Тип результата
+ – *	*числовой*	*числовой*	наименьший *числовой* тип, поглощающий оба операнда
/	*числовой*	*числовой*	наименьший *вещественный* тип, поглощающий оба операнда
+ – * /	SET	SET	SET
DIV MOD	*целый*	*целый*	наименьший *целый* тип, поглощающий оба операнда

Операция	Первый операнд	Второй операнд	Тип результата
OR & ~	BOOLEAN	BOOLEAN	BOOLEAN
= # < <= > >=	*числовой*	*числовой*	BOOLEAN
	CHAR	CHAR	BOOLEAN
	символьный массив, строка	символьный массив, строка	BOOLEAN
= #	BOOLEAN	BOOLEAN	BOOLEAN
	SET	SET	BOOLEAN
	NIL, тип указатель T0 или T1	NIL, тип указатель T0 или T1	BOOLEAN
	процедурный тип T, NIL	процедурный тип T, NIL	BOOLEAN
IN	*целый*	SET	BOOLEAN
IS	тип T0	тип T1	BOOLEAN

Совпадение списков формальных параметров

Два списка формальных параметров *совпадают* если

1. они имеют одинаковое количество параметров, и
2. они имеют или *одинаковый* тип результата функции или не имеют никакого, и
3. параметры в соответствующих позициях имеют *равные* типы, и

4. параметры в соответствующих позициях — оба или параметры-значения или параметры-переменные.

Приложение В: Синтаксис Оберона-2

Модуль	= MODULE идент ";" [СписокИмпорта] ПослОбъявл [BEGIN ПослОператоров] END идент ".".
СписокИмпорта	= IMPORT [идент ":="] идент {"," [идент ":="] идент} ";".
ПослОбъявл	= { CONST {ОбъявлКонст ";" } | TYPE {ОбъявлТипа ";" } | VAR {ОбъявлПерем ";" }} {ОбъявлПроц ";" | ОпережающееОбъявл ";"}.
ОбъявлКонст	= ИдентОпр "=" КонстВыраж.
ОбъявлТипа	= ИдентОпр "=" Тип.
ОбъявлПерем	= СписокИдент ":" Тип.
ОбъявлПроц	= PROCEDURE [Приемник] ИдентОпр [ФормальныеПарам] ";" ПослОбъявл [BEGIN ПослОператоров] END идент.
ОпережающееОбъявл	= PROCEDURE "^" [Приемник] ИдентОпр [ФормальныеПарам].
ФормальныеПарам	= "(" [СекцияФП {";" СекцияФП}] ")" [":" УточнИдент].
СекцияФП	= [VAR] идент {"," идент} ":" Тип.

Приемник	= "(" [VAR] идент ":" идент ")".
Тип	= УточнИдент | ARRAY [КонстВыраж {"," КонстВыраж}] OF Тип | RECORD ["("УточнИдент")"] СписокПолей {";" СписокПолей} END | POINTER TO Тип | PROCEDURE [ФормальныеПарам].
СписокПолей	= [СписокИдент ":" Тип].
ПослОператоров	= Оператор {";" Оператор}.
Оператор	= [Обозначение ":=" Выраж | Обозначение ["(" [СписокВыраж] ")"] | IF Выраж THEN ПослОператоров {ELSIF Выраж THEN ПослОператоров} [ELSE ПослОператоров] END | CASE Выраж OF Вариант {"|" Вариант} [ELSE ПослОператоров] END | WHILE Выраж DO ПослОператоров END | REPEAT ПослОператоров UNTIL Выраж | FOR идент ":=" Выраж TO Выраж [BY КонстВыраж] DO ПослОператоров END | LOOP ПослОператоров END | WITH Охрана DO ПослОператоров {"|" Охрана DO ПослОператоров} [ELSE ПослОператоров] END | EXIT | RETURN [Выраж]].

Вариант	= [МеткиВарианта {"," МеткиВарианта} ":" ПослОператоров].
МеткиВарианта	= КонстВыраж [".." КонстВыраж].
Охрана	= УточнИдент ":" УточнИдент.
КонстВыраж	= Выраж.
Выраж	= ПростоеВыраж [Отношение ПростоеВыраж].
ПростоеВыраж	= ["+" | "-"] Слагаемое {ОперСлож Слагаемое}.
Слагаемое	= Множитель {ОперУмн Множитель}.
Множитель	= Обозначение ["(" [СписокВыраж] ")"] | число | символ | строка | NIL | Множество | "(" Выраж ")" | "~" Множитель.
Множество	= "{" [Элемент {"," Элемент}] "}".
Элемент	= Выраж [".." Выраж].
Отношение	= "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
ОперСлож	= "+" | "-" | OR.
ОперУмн	= " * " | "/" | DIV | MOD | "&".
Обозначение	= УточнИдент {"." идент | "[" СписокВыраж "]" | "^" | "(" УточнИдент ")"}.

СписокВыраж	= Выраж {"," Выраж}.
СписокИдент	= ИдентОпр {"," ИдентОпр}.
УточнИдент	= [идент "."] идент.
ИдентОпр	= идент ["*" | "-"].

Приложение C: Модуль SYSTEM

Модуль SYSTEM содержит некоторые типы и процедуры, которые необходимы для реализации операций низкого уровня, специфичных для данного компьютера и/или реализации. Они включают, например, средства для доступа к устройствам, которые управляются компьютером, и средства, позволяющие обойти правила совместимости типов, наложенные определением языка. Настоятельно рекомендуется ограничить использование этих средств специфическими модулями (модулями низкого уровня). Такие модули непременно являются непереносимыми, но легко распознаются по идентификатору SYSTEM, появляющемуся в их списке импорта. Следующие спецификации действительны для реализации Оберон-2 на компьютере Ceres.

Модуль SYSTEM экспортирует тип BYTE со следующими характеристиками: переменным типа BYTE можно присваивать значения переменных типа CHAR или SHORTINT. Если формальный параметр-переменная имеет тип ARRAY OF BYTE, то соответствующий фактический параметр может иметь любой тип.

Другой тип, экспортируемый модулем SYSTEM, — тип PTR. Переменным типа PTR могут быть присвоены значения переменных-указателей любого типа. Если формальный параметр-переменная имеет тип PTR, фактический параметр может быть указателем любого типа.

Процедуры, содержащиеся в модуле SYSTEM, перечислены в таблицах. Большинство их соответствует одиночным командам и компилируются непосредственно в машинный код. О деталях читатель может справиться в описании процессора. В таблице *v* обозначает переменную, *x, y, a* и *n* — выражения, а *T* — тип.

Процедуры-функции

Название	Типы аргументов	Тип результата	Функция
ADR(v)	любой	LONGINT	адрес переменной v
BIT(a, n)	a: LONGINT n:*целый*	BOOLEAN	n-й бит Память[a]
CC(n)	*целая* константа	BOOLEAN	условие n (0 <= n <= 15)
LSH(x, n)	x: *целый*, CHAR, BYTE n: *целый*	совпадает с типом x	логический сдвиг
ROT(x, n)	x: *целый*, CHAR,BYTE n: *целый*	совпадает с типом x	циклический сдвиг
VAL(T, x)	T, x: любого типа	T	x интерпретируется как значение типа T

Собственно процедуры

Название	Типы аргументов	Функция
GET(a, v)	a: LONGINT; v: любой основной тип, указатель, процедурный тип	v := Память[a]

Название	Типы аргументов	Функция
PUT(a, x)	a: LONGINT; x: любой основной тип, указатель, процедурный тип	Память[a] := x
GETREG(n, v)	n: *целая* константа; v: любой основной тип, указатель, процедурный тип	v := Регистр n
PUTREG(n, x)	n: *целая* константа; x: любой основной тип, указатель, процедурный тип	Регистр n := x
MOVE(a0, a1, n)	a0, a1: LONGINT; n: *целый*	Память[a1..a1+n-1] := Память[a0..a0+n-1]
NEW(v, n)	v: любой указатель; n: *целый*	размещает блок памяти размером n байт; присваивает его адрес переменной v

Приложение D: Среда Оберон

Программы на Обероне-2 обычно выполняются в среде, которая обеспечивает *активацию команд*, *сбор мусора*, *динамическую загрузку* модулей и определенные *структуры данных времени выполнения*. Не являясь частью языка, эта среда способствует увеличению мощности Оберона-2 и до некоторой степени подразумевается при определении языка. В приложении D описаны существенные особенности типичной Оберон-среды и даны советы по реализации. Подробности можно найти в [1], [2], и [3].

D1. Команды

Команда — это любая процедура *P*, которая экспортируется модулем *M* и не имеет параметров. Она обозначается *M.P* и может быть активирована под таким именем из оболочки операционной системы. В Обероне, пользователь вызывает команды вместо программ или модулей. Это дает лучшую структуру управления и предоставляет модули с несколькими точками входа. Когда вызывается команда *M.P*, модуль *M* динамически загружается, если он уже не был в памяти (см. D2) и выполняется процедура *P*. Когда *P* завершается, *M* остается загруженным. Все глобальные переменные и структуры данных, которые могут быть достигнуты через глобальные переменные-указатели в *M*, сохраняют значения. Когда *P* (или другая команда *M*) вызывается снова, она может продолжать использовать эти значения.

Следующий модуль демонстрирует использование команд. Он реализует абстрактную структуру данных *Counter*, которая содержит переменную-счетчик и обеспечивает команды для увеличения и печати его значения.

```
MODULE Counter;
 IMPORT Texts, Oberon;

 VAR
    counter: LONGINT;
    w: Texts.Writer;

 PROCEDURE Add*;
 (*получает числовой аргумент из командной строки*)
    VAR s: Texts.Scanner;
 BEGIN
    Texts.OpenScanner
       (s,   Oberon.Par.text, Oberon.Par.pos);
    Texts.Scan(s);
    IF s.class = Texts.Int THEN INC(counter, s.i) END
 END Add;

 PROCEDURE Write*;
 BEGIN
    Texts.WriteInt(w, counter, 5); Texts.WriteLn(w);
    Texts.Append(Oberon.Log, w.buf)
```

```
   END Write;

BEGIN counter := 0; Texts.OpenWriter(w)
END Counter.
```

Пользователь может выполнить следующие две команды:

`Counter.Add n` Добавляет значение n к переменной *counter*

`Counter.Write` Выводит текущее значение *counter* на экран

Так как команды не содержат параметров, они должны получать свои аргументы из операционной системы. Вообще команды вольны брать параметры отовсюду (например из текста после команды, из текущего выбранного фрагмента или из отмеченного окна просмотра). Команда *Add* использует сканер (тип данных, обеспечиваемый Оберон-системой) чтобы читать значение, которое следует за нею в командной строке.

Когда *Counter.Add* вызывается впервые, модуль *Counter* загружается и выполняется его тело. Каждое обращение *Counter.Add n* увеличивает переменную *counter* на *n*. Каждое обращение *Counter.Write* выводит текущее значение *counter* на экран.

Поскольку модуль остается загруженным после выполнения его команд, должен существовать явный способ выгрузить его (например, когда пользователь хочет заменить загруженную версию перекомпилированной версией). Оберон-система содержит команду, позволяющую это сделать.

D2. Динамическая загрузка модулей

Загруженный модуль может вызывать команду незагруженного модуля, задавая ее имя как строку. Специфицированный модуль при этом динамически загружается и выполняется заданная команда. Динамическая загрузка позволяет пользователю запустить программу как небольшой набор базисных модулей и расширять ее, добавляя последующие модули во время выполнения по мере необходимости.

Модуль *M0* может вызвать динамическую загрузку модуля *M1* без того, чтобы импортировать его. *M1* может, конечно, импортировать и использовать *M0*, но *M0* не должен знать о существовании *M1*. *M1* может быть модулем, который спроектирован и реализован намного позже *M0*.

D3. Сбор мусора

В Обероне-2 стандартная процедура NEW используется, чтобы распределить блоки данных в свободной памяти. Нет, однако, никакого способа явно освободить распределенный блок. Взамен Оберон-среда использует сборщик мусора чтобы найти блоки, которые больше не используются и сделать их снова доступными для распределения. Блок считается используемым только если он может быть достигнут через глобальную переменную-указатель по цепочке указателей. Разрыв этой цепочки (например, установкой указателя в NIL) делает блок утилизируемым.

Сборщик мусора освобождает программиста от нетривиальной задачи правильного освобождения структур данных и таким образом помогает избегать ошибок. Возникает, однако, необходимость иметь информацию о динамических данных во время выполнения (см. D5).

D4. Смотритель

Интерфейс модуля (объявления экспортируемых объектов) извлекается из модуля так называемым смотрителем, который является отдельным инструментом среды Оберон. Например, смотритель производит следующий интерфейс модуля *Trees* из Гл. 11.

```
DEFINITION Trees;
  TYPE
    Tree = POINTER TO Node;
    Node = RECORD
      name: POINTER TO ARRAY OF CHAR;
      PROCEDURE (t: Tree)Insert(name: ARRAY OF CHAR);
      PROCEDURE (t:Tree)Search(name: ARRAY OF CHAR):Tree;
```

```
      PROCEDURE (t: Tree) Write;
    END;
  PROCEDURE Init (VAR t: Tree);
END Trees.
```

Для типа запись смотритель также собирает все процедуры, связанные с этим типом, и показывает их заголовки в объявлении типа запись.

D5. Структуры данных времени выполнения

Некоторая информация о записях должна быть доступна во время выполнения: Динамический тип записей необходим для проверки и охраны типа. Таблица с адресами процедур, связанных с записью, необходима для их вызова. Наконец, сборщик мусора нуждается в информации о расположении указателей в динамически распределенных записях. Вся эта информация сохраняется в так называемых дескрипторах типа. Один дескриптор необходим во время выполнения для каждого типа записи. Ниже показана возможная реализация дескрипторов типа.

Динамический тип записи соответствует адресу дескриптора типа. Для динамически распределенных записей этот адрес сохраняется в так называемом теге типа, который предшествует фактическим данным записи и является невидимым для программиста. Если *t* — переменная типа *CenterTree* (см. пример в Гл. 6), рисунок D5.1 показывает одну из возможных реализаций структур данных времени выполнения.

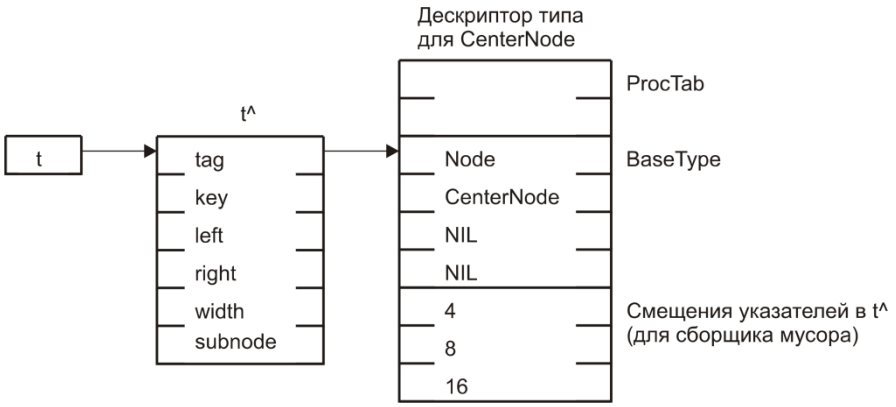


Рис. D5.1. Переменная *t* типа *CenterTree*, запись *t^*, на которую она указывает, и дескриптор типа

Поскольку и таблица адресов процедур и таблица смещений указателей должны иметь фиксированное смещение относительно адреса дескриптора типа, и поскольку они могут расти, когда тип расширяется и добавляются новые процедуры и указатели, то таблицы размещены в противоположных концах дескриптора типа и растут в разных направлениях.

Связанная с типом процедура *t.P* вызывается как *t^.tag^.ProcTab [IndexP]*. Индекс таблицы процедур для каждой связанной с типом процедуры известен во время компиляции. Проверка типа *v IS T* транслируется в *v^.tag^.BaseTypes [ExtensionLevelT] = TypeDescrAdrT*. И уровень расширения типа запись (*ExtensionLevelT*), и адрес описателя типа (*TypeDescrAdrT)* известны во время компиляции. Например, уровень расширения *Node* — 0 (этот тип не имеет базового типа), а уровень расширения *CenterNode* — 1.

[1] N. Wirth, J. Gutknecht: The Oberon System. Software Practice and Experience 19, 9, Sept. 1989

[2] M. Reiser: The Oberon System. User Guide and Programming Manual. Addison-Wesley, 1991

[3] C. Pfister, B. Heeb, J. Templ: Oberon Technical Notes. Report 156, ETH Zürich, March 1991

ПРИЛОЖЕНИЕ 2. ТЕКСТ КОМПИЛЯТОРА ЯЗЫКА «О» НА ПАСКАЛЕ

Приведенные ниже листинги содержат полный исходный код компилятора, разработка которого обсуждалась в главе «Трансляция языков программирования». Программа написана на диалекте языка Паскаль, общем для систем программирования компании Borland.

Листинг П2.1. Основная программа компилятора языка «О»

```
program O;
{Компилятор языка О}

uses
   OText, OScan, OPars, OVM, OGen;

procedure Init;
begin
   ResetText;
   InitScan;
   InitGen;
end;

procedure Done;
begin
   CloseText;
end;

begin
   WriteLn('Компилятор языка О');
   Init;    {Инициализация}
   Compile; {Компиляция}
   Run;     {Выполнение}
   Done;    {Завершение}
end.
```

Реализация драйвера исходного текста не рассматривалась в основной части книги. Ниже приведен вариант, предусматривающий посимвольное чтение из файла, содержащего исходный текст программы на языке «О». Название файла передается компилятору как параметр командной строки. Модуль `OText` обес-

печивает также вывод текста программы в ходе трансляции в стандартный выходной файл (на экран).

Листинг П2.2. Драйвер исходного текста

```
unit  OText;
{Драйвер исходного текста}

interface

const
   chSpace = ' ';     {Пробел      }
   chTab   = chr(9);  {Табуляция   }
   chEOL   = chr(10); {Конец строки}
   chEOT   = chr(0);  {Конец текста}

var
   Ch    : char;    {Очередной символ      }
   Line  : integer; {Номер строки          }
   Pos   : integer; {Номер символа в строке}

procedure ResetText;
procedure CloseText;
procedure NextCh;

{=====================================================}

implementation

uses
   OError;

const
   TabSize = 3;

var
   f : text;

procedure ResetText;
begin
   if ParamCount < 1 then begin
      Writeln('Формат вызова:');
      Writeln('   O <входной файл>');
      Halt;
      end
   else begin
      Assign(f, ParamStr(1));
      {$i-} Reset(f); {$i+}
```

```
      if IOResult <> 0 then
         Error('Входной файл не найден')
      else begin
         Pos := 0;
         Line := 1;
         NextCh;
      end;
   end;
end;

procedure CloseText;
begin
   Close(f);
end;

procedure NextCh;
begin
   if eof(f) then
      Ch := chEOT
   else if eoln(f) then begin
      ReadLn(f);
      WriteLn;
      Line := Line + 1;
      Pos := 0;
      Ch := chEOL;
      end
   else begin
      Read(f, Ch);
      if Ch <> chTab then begin
         Write(Ch);
         Pos := Pos+1;
         end
      else
         repeat
            Write(' ');
            Pos := Pos+1;
         until Pos mod TabSize = 0;
   end;
end;

end.
```

В основном тексте книги реализация модуля обработки ошибок не рассмотрена. Приведенный ниже вариант модуля `OError` предусматривает выдачу сообщений об ошибках с указанием номера строки и номера символа. Место ошибки также отмечается стрелкой, которая указывает на начало лексемы, породившей

ошибку. Перед выдачей сообщения на экран выводится остаток строки, содержащей ошибку.

Листинг П2.3. Модуль обработки ошибок

```
unit  OError;
{Обработка ошибок}

interface

procedure Error(Msg : string);
procedure Expected(Msg : string);
procedure Warning(Msg : string);
{=========================================================}

implementation

uses
   OText, OScan;

procedure Error(Msg : string);
var
   ELine : integer;
begin
   ELine := Line;
   while (Ch <> chEOL) and (Ch <> chEOT) do NextCh;
   if Ch = chEOT then WriteLn;
   WriteLn('^': LexPos);
   Writeln('(Строка ', ELine, ') Ошибка: ', Msg);
   WriteLn;
   WriteLn('Нажмите ВВОД');
   Readln;
   Halt;
end;

procedure Expected(Msg: string);
begin
   Error('Ожидается ' + Msg);
end;

procedure Warning(Msg : string);
begin
   WriteLn;
   Writeln('Предупреждение: ', Msg);
   WriteLn;
end;

end.
```

Листинг П2.4. Лексический анализатор

```
unit  OScan;
{ Сканер }

interface

const
   NameLen = 31; {Наибольшая длина имени}

type
   tName = string[NameLen];
   tLex =  (lexNone,   lexName, lexNum,
            lexMODULE, lexIMPORT, lexBEGIN, lexEND,
            lexCONST,  lexVAR,  lexWHILE, lexDO,
            lexIF, lexTHEN, lexELSIF, lexELSE,
            lexMult, lexDIV, lexMOD, lexPlus, lexMinus,
            lexEQ, lexNE, lexLT, lexLE, lexGT, lexGE,
            lexDot, lexComma, lexColon, lexSemi, lexAss,
            lexLpar, lexRpar,
            lexEOT);

var
   Lex   : tLex;    {Текущая лексема            }
   Name  : tName;   {Строковое значение имени   }
   Num   : integer; {Значение числовых литералов}
   LexPos: integer; {Позиция начала лексемы     }

procedure InitScan;
procedure NextLex;
{=======================================================}
implementation

uses
   OText, OError;

const
   KWNum = 34;

type
   tKeyWord = string[9]; {Длина слова PROCEDURE}

var
   nkw      : integer;
   KWTable  : array [1..KWNum] of
      record
         Word  : tKeyWord;
         Lex   : tLex;
      end;
```

```
procedure EnterKW(Name: tKeyWord; Lex: tLex);
begin
   nkw := nkw + 1;
   KWTable[nkw].Word := Name;
   KWTable[nkw].Lex := Lex;
end;

function TestKW: tLex;
var
   i : integer;
begin
   i := nkw;
   while (i>0) and (Name <> KWTable[i].Word) do
      i := i-1;
   if i>0 then
      TestKW := KWTable[i].Lex
   else
      TestKW := lexName;
end;

procedure Ident;
var
   i : integer;
begin
   i := 0;
   repeat
      if i < NameLen then begin
         i := i + 1;
         Name[i] := Ch;
         end
      else
         Error('Слишком длинное имя');
      NextCh;
   until not (Ch in ['A'..'Z', 'a'..'z', '0'..'9']);
   Name[0] := chr(i); {Длина строки Name теперь равна i}
   Lex := TestKW;     {Проверка на ключевое слово}
end;

procedure Number;
var
   d : integer;
begin
   Lex := lexNum;
   Num := 0;
   repeat
      d := ord(Ch) - ord('0');
      if (Maxint - d) div 10 >= Num then
         Num := 10*Num + d
```

```
      else
         Error('Слишком большое число');
      NextCh;
   until not (Ch in ['0'..'9']);
end;

procedure Comment;
begin
   NextCh;
   repeat
      while (Ch <> '*') and (Ch <> chEOT) do
         if Ch = '(' then begin
            NextCh;
            if Ch = '*' then Comment;
            end
         else
            NextCh;
      if Ch = '*' then
         NextCh;
   until Ch in [')', chEOT];
   if Ch = ')' then
      NextCh
   else begin
      LexPos := Pos;
      Error('Не закончен комментарий');
   end;
end;

(*
procedure Comment;
var
   Level : integer;
begin
   Level := 1;
   NextCh;
   repeat
      if Ch = '*' then begin
         NextCh;
         if Ch = ')' then
            begin Level := Level - 1; NextCh end;
         end
      else if Ch = '(' then begin
         NextCh;
         if Ch = '*' then
            begin Level := Level + 1; NextCh end;
         end
      else {if Ch <> chEOT then}
         NextCh;
   until (Level = 0) or (Ch = chEOT);
```

```
   if Level <> 0 then begin
      LexPos := Pos;
      Error('Не закончен комментарий');
   end;
end;
*)

procedure NextLex;
begin
   while Ch in [chSpace, chTab, chEOL] do NextCh;
   LexPos := Pos;
   case Ch of
   'A'..'Z', 'a'..'z':
      Ident;
   '0'..'9':
      Number;
   ';':
      begin
         NextCh;
         Lex := lexSemi;
      end;
   ':':
      begin
         NextCh;
         if Ch = '=' then begin
            NextCh;
            Lex := lexAss;
            end
         else
            Lex := lexColon;
      end;
   '.':
      begin
         NextCh;
         Lex := lexDot;
      end;
   ',':
      begin
         NextCh;
         Lex := lexComma;
      end;
   '=':
      begin
         NextCh;
         Lex := lexEQ;
      end;
   '#':
      begin
         NextCh;
```

```
      Lex := lexNE;
   end;
'<':
   begin
      NextCh;
      if Ch='=' then begin
         NextCh;
         Lex := lexLE;
         end
      else
         Lex := lexLT;
   end;
'>':
   begin
      NextCh;
      if Ch='=' then begin
         NextCh;
         Lex := lexGE;
         end
      else
         Lex := lexGT;
   end;
'(':
   begin
      NextCh;
      if Ch = '*' then begin
         Comment;
         NextLex;
         end
      else
         Lex := lexLpar;
   end;
')':
   begin
      NextCh;
      Lex := lexRpar;
   end;
'+':
   begin
      NextCh;
      Lex := lexPlus;
   end;
'-':
   begin
      NextCh;
      Lex := lexMinus;
   end;
'*':
   begin
```

```
         NextCh;
         Lex := lexMult;
      end;
   chEOT:
      Lex := lexEOT;
   else
      Error('Недопустимый символ');
   end;
end;

procedure InitScan;
begin
   nkw := 0;

   EnterKW('ARRAY',     lexNone);
   EnterKW('BY',        lexNone);
   EnterKW('BEGIN',     lexBEGIN);
   EnterKW('CASE',      lexNone);
   EnterKW('CONST',     lexCONST);
   EnterKW('DIV',       lexDIV);
   EnterKW('DO',        lexDO);
   EnterKW('ELSE',      lexELSE);
   EnterKW('ELSIF',     lexELSIF);
   EnterKW('END',       lexEND);
   EnterKW('EXIT',      lexNone);
   EnterKW('FOR',       lexNone);
   EnterKW('IF',        lexIF);
   EnterKW('IMPORT',    lexIMPORT);
   EnterKW('IN',        lexNone);
   EnterKW('IS',        lexNone);
   EnterKW('LOOP',      lexNone);
   EnterKW('MOD',       lexMOD);
   EnterKW('MODULE',    lexMODULE);
   EnterKW('NIL',       lexNone);
   EnterKW('OF',        lexNone);
   EnterKW('OR',        lexNone);
   EnterKW('POINTER',   lexNone);
   EnterKW('PROCEDURE', lexNone);
   EnterKW('RECORD',    lexNone);
   EnterKW('REPEAT',    lexNone);
   EnterKW('RETURN',    lexNone);
   EnterKW('THEN',      lexTHEN);
   EnterKW('TO',        lexNone);
   EnterKW('TYPE',      lexNone);
   EnterKW('UNTIL',     lexNone);
   EnterKW('VAR',       lexVAR);
   EnterKW('WHILE',     lexWHILE);
   EnterKW('WITH',      lexNone);
```

```
   NextLex;
end;

end.
```

Листинг П2.5. Синтаксический и контекстный анализатор

```
unit  OPars;
{ Распознаватель }

interface

procedure Compile;

{=====================================================}

implementation

uses
   OScan, OError, OTable, OGen, OVM;

const
   spABS    = 1;
   spMAX    = 2;
   spMIN    = 3;
   spDEC    = 4;
   spODD    = 5;
   spHALT   = 6;
   spINC    = 7;
   spInOpen = 8;
   spInInt  = 9;
   spOutInt = 10;
   spOutLn  = 11;

procedure StatSeq; forward;
procedure Expression(var t: tType); forward;

procedure Check(L: tLex; M: string);
begin
   if Lex <> L then
      Expected(M)
   else
      NextLex;
end;

(* ["+" | "-"] (Число | Имя) *)
procedure ConstExpr(var V: integer);
var
   X  : tObj;
```

```
   Op : tLex;
begin
   Op := lexPlus;
   if Lex in [lexPlus, lexMinus] then begin
      Op := Lex;
      NextLex;
   end;
   if Lex = lexNum then begin
      V := Num;
      NextLex;
      end
   else if Lex = lexName then begin
      Find(Name, X);
      if X^.Cat = catGuard then
         Error('Нельзя определять константу через себя')
      else if X^.Cat <> catConst then
         Expected('имя константы')
      else
         V := X^.Val;
      NextLex;
      end
   else
      Expected('константное выражение');
   if Op = lexMinus then
      V := -V;
end;

(* Имя "=" КонстВыраж *)
procedure ConstDecl;
var
   ConstRef: tObj; {Ссылка на имя в таблице}
begin
   NewName(Name, catGuard, ConstRef);
   NextLex;
   Check(lexEQ, '"="');
   ConstExpr(ConstRef^.Val);
   ConstRef^.Typ := typInt; {Констант других типов нет}
   ConstRef^.Cat := catConst;
end;

procedure ParseType;
var
   TypeRef : tObj;
begin
   if Lex <> lexName then
      Expected('имя')
   else begin
      Find(Name, TypeRef);
      if TypeRef^.Cat <> catType then
```

```
         Expected('имя типа')
      else if TypeRef^.Typ <> typInt then
         Expected('целый тип');
      NextLex;
   end;
end;

(* Имя {"," Имя} ":" Тип *)
procedure VarDecl;
var
   NameRef : tObj;
begin
   if Lex <> lexName then
      Expected('имя')
   else begin
      NewName(Name, catVar, NameRef);
      NameRef^.Typ := typInt;
      NextLex;
   end;
   while Lex = lexComma do begin
      NextLex;
      if Lex <> lexName then
         Expected('имя')
      else begin
         NewName(Name, catVar, NameRef);
         NameRef^.Typ := typInt;
         NextLex;
      end;
   end;
   Check(lexColon, '":"');
   ParseType;
end;

(* {CONST {ОбъявлКонст ";"}
    |VAR {ОбъявлПерем ";"} } *)
procedure DeclSeq;
begin
   while Lex in [lexCONST, lexVAR] do begin
      if Lex = lexCONST then begin
         NextLex;
         while Lex = lexName do begin
            ConstDecl; {Объявление константы}
            Check(lexSemi, '";"');
         end;
         end
      else begin
         NextLex; { VAR }
         while Lex = lexName do begin
            VarDecl;   {Объявление переменных}
```

```
            Check(lexSemi, '";"');
         end;
      end;
   end;
end;

procedure IntExpression;
var
   T : tType;
begin
   Expression(T);
   if T <> typInt then
      Expected('выражение целого типа');
end;

procedure StFunc(F: integer; var T: tType);
begin
   case F of
   spABS:
      begin
         IntExpression;
         GenAbs;
         T := typInt;
      end;
   spMAX:
      begin
         ParseType;
         Gen(MaxInt);
         T := typInt;
      end;
   spMIN:
      begin
         ParseType;
         GenMin;
         T := typInt;
      end;
   spODD:
      begin
         IntExpression;
         GenOdd;
         T := typBool;
      end;
   end;
end;

(* Имя["(" Выраж | Тип ")"] | Число | "("Выраж")" *)
procedure Factor(var T : tType);
var
   X : tObj;
```

```
begin
   if Lex = lexName then begin
      Find(Name, X);
      if X^.Cat = catVar then begin
         GenAddr(X);    {Адрес переменной}
         Gen( cmLoad );
         T := X^.Typ;
         NextLex;
         end
      else if X^.Cat = catConst then begin
         GenConst(X^.Val);
         T := X^.Typ;
         NextLex;
         end
      else if (X^.Cat=catStProc) and (X^.Typ<>typNone)
      then begin
         NextLex;
         Check(lexLPar, '"("');
         StFunc(X^.Val, T);
         Check(lexRPar, '")"');
         end
      else
         Expected(
         'переменная, константа или процедура-функции'
         );
      end
   else if Lex = lexNum then begin
      T := typInt;
      GenConst(Num);
      NextLex
      end
   else if Lex = lexLPar then begin
      NextLex;
      Expression(T);
      Check(lexRPar, '")"');
      end
   else
      Expected('имя, число или "("');
end;

(* Множитель {ОперУмн Множитель} *)
procedure Term(var T: tType);
var
   Op : tLex;
begin
   Factor(T);
   if Lex in [lexMult, lexDIV, lexMOD] then begin
      if T <> typInt then
         Error('Несоответствие операции типу операнда');
```

```
      repeat
         Op := Lex;
         NextLex;
         Factor(T);
         if T <> typInt then
            Expected('выражение целого типа');
         case Op of
         lexMult: Gen(cmMult);
         lexDIV:  Gen(cmDIV);
         lexMOD:  Gen(cmMOD);
         end;
      until not( Lex in [lexMult, lexDIV, lexMOD] );
   end;
end;

(* ["+"|"-"] Слагаемое {ОперСлож Слагаемое} *)
procedure SimpleExpr(var T: tType);
var
   Op : tLex;
begin
   if Lex in [lexPlus, lexMinus] then begin
      Op := Lex;
      NextLex;
      Term(T);
      if T <> typInt then
         Expected('выражение целого типа');
      if Op = lexMinus then
         Gen(cmNeg);
      end
   else
      Term(T);
   if Lex in [lexPlus, lexMinus] then begin
      if T <> typInt then
         Error('Несоответствие операции типу операнда');
      repeat
         Op := Lex;
         NextLex;
         Term(T);
         if T <> typInt then
            Expected('выражение целого типа');
         case Op of
         lexPlus:  Gen(cmAdd);
         lexMinus: Gen(cmSub);
         end;
      until not( Lex in [lexPlus, lexMinus] );
   end;
end;
```

```
(* ПростоеВыраж [Отношение ПростоеВыраж] *)
procedure Expression(var T: tType);
var
   Op : tLex;
begin
   SimpleExpr(T);
   if Lex in [lexEQ, lexNE, lexGT, lexGE, lexLT, lexLE]
   then begin
      Op := Lex;
      if T <> typInt then
         Error('Несоответствие операции типу операнда');
      NextLex;
      SimpleExpr(T); {Правый операнд отношения}
      if T <> typInt then
         Expected('выражение целого типа');
      GenComp(Op);   {Генерация условного перехода}
      T := typBool;
   end; {иначе тип равен типу первого простого выражения}
end;

(* Переменная = Имя. *)
procedure Variable;
var
   X : tObj;
begin
   if Lex <> lexName then
      Expected('имя')
   else begin
      Find(Name, X);
      if X^.Cat <> catVar then
         Expected('имя переменной');
      GenAddr(X);
      NextLex;
   end;
end;

procedure StProc(P: integer);
var
   c : integer;
begin
   case P of
   spDEC:
      begin
         Variable;
         Gen(cmDup);
         Gen(cmLoad);
         if Lex = lexComma then begin
            NextLex;
            IntExpression;
```

```
            end
         else
            Gen(1);
         Gen(cmSub);
         Gen(cmSave);
      end;
   spINC:
      begin
         Variable;
         Gen(cmDup);
         Gen(cmLoad);
         if Lex = lexComma then begin
            NextLex;
            IntExpression;
            end
         else
            Gen(1);
         Gen(cmAdd);
         Gen(cmSave);
      end;
   spInOpen:
      { Пусто };
   spInInt:
      begin
         Variable;
         Gen(cmIn);
         Gen(cmSave);
      end;
   spOutInt:
      begin
         IntExpression;
         Check(lexComma, '","');
         IntExpression;
         Gen(cmOut);
      end;
   spOutLn:
      Gen(cmOutLn);
   spHalt:
      begin
         ConstExpr(c);
         GenConst(c);
         Gen(cmStop);
      end;
   end;
end;

procedure BoolExpression;
var
   T : tType;
```

```
begin
   Expression(T);
   if T <> typBool then
      Expected('логическое выражение');
end;

(* Переменная ":=" Выраж *)
procedure AssStatement;
begin
   Variable;
   if Lex = lexAss then begin
      NextLex;
      IntExpression;
      Gen(cmSave);
      end
   else
      Expected('":="')
end;

(* Имя ["(" { Выраж | Переменная } ")"] *)
procedure CallStatement(P : integer);
begin
   Check(lexName, 'имя процедуры');
   if Lex = lexLPar then begin
      NextLex;
      StProc(P);
      Check(lexRPar, '")"');
      end
   else if P in [spOutLn, spInOpen] then
      StProc(P)
   else
      Expected('"("');
end;

(* IF Выраж THEN   ПослОператоров
   {ELSIF Выраж THEN ПослОператоров}
   [ELSE   ПослОператоров] END *)
procedure IfStatement;
var
   CondPC   : integer;
   LastGOTO : integer;
begin
   Check(lexIF, 'IF');
   LastGOTO := 0;     {Предыдущего перехода нет        }
   BoolExpression;
   CondPC := PC;      {Запомн. положение усл. перехода }
   Check(lexTHEN, 'THEN');
   StatSeq;
   while Lex = lexELSIF do begin
```

```
      Gen(LastGOTO);  {Фиктивный адрес, указывающий    }
      Gen(cmGOTO);    {на место предыдущего перехода.  }
      LastGOTO := PC; {Запомнить место GOTO            }
      NextLex;
      Fixup(CondPC);  {Зафикс. адрес условного перехода}
      BoolExpression;
      CondPC := PC;   {Запомн. положение усл. перехода }
      Check(lexTHEN, 'THEN');
      StatSeq;
   end;
   if Lex = lexELSE then begin
      Gen(LastGOTO);  {Фиктивный адрес, указывающий    }
      Gen(cmGOTO);    {на место предыдущего перехода   }
      LastGOTO := PC; {Запомнить место последнего GOTO }
      NextLex;
      Fixup(CondPC);  {Зафикс. адрес условного перехода}
      StatSeq;
      end
   else
      Fixup(CondPC);   {Если ELSE отсутствует          }
   Check( lexEND, 'END' );
   Fixup(LastGOTO);    {Направить сюда все GOTO       }
end;

(* WHILE Выраж DO ПослОператоров END *)
procedure WhileStatement;
var
   WhilePC  : integer;
   CondPC   : integer;
begin
   WhilePC := PC;
   Check(lexWHILE, 'WHILE');
   BoolExpression;
   CondPC := PC;
   Check(lexDO, 'DO');
   StatSeq;
   Check(lexEND, 'END');
   Gen(WhilePC);
   Gen(cmGOTO);
   Fixup(CondPC);
end;

procedure Statement;
var
   X : tObj;
begin
   if Lex = lexName then begin
      Find(Name, X);
      if X^.Cat = catModule then begin
```

```
            NextLex;
            Check(lexDot, '"."');
            if    (Lex = lexName) and
               (Length(X^.Name)+Length(Name) < NameLen)
            then
               Find(X^.Name+'.'+Name, X)
            else
               Expected('имя из модуля '+ X^.Name);
         end;
         if X^.Cat = catVar then
            AssStatement          {Присваивание}
         else if(X^.Cat=catStProc) and (X^.Typ=typNone)then
            CallStatement(X^.Val) {Вызов процедуры}
         else
            Expected(
               'обозначение переменной или процедуры'
            );
         end
      else if Lex = lexIF then
         IfStatement
      else if Lex = lexWHILE then
         WhileStatement
end;

(* Оператор {";" Оператор} *)
procedure StatSeq;
begin
   Statement;    {Оператор}
   while Lex = lexSemi do begin
      NextLex;
      Statement; {Оператор}
   end;
end;

procedure ImportModule;
var
   ImpRef: tObj;
begin
   if Lex = lexName then begin
      NewName(Name, catModule, ImpRef);
      if Name = 'In' then begin
         Enter('In.Open', catStProc, typNone, spInOpen);
         Enter('In.Int', catStProc, typNone, spInInt);
         end
      else if Name = 'Out' then begin
         Enter('Out.Int', catStProc, typNone, spOutInt);
         Enter('Out.Ln', catStProc, typNone, spOutLn);
         end
      else
```

```
         Error('Неизвестный модуль');
      NextLex;
      end
   else
      Expected('имя импортируемого модуля');
end;

(* IMPORT Имя { "," Имя } ";" *)
procedure Import;
begin
   Check(lexIMPORT, 'IMPORT');
   ImportModule;
   while Lex = lexComma do begin
      NextLex;
      ImportModule;
   end;
   Check(lexSemi, '";"');
end;

(* MODULE Имя ";" [Импорт] ПослОбъявл
   [BEGIN ПослОператоров] END Имя "." *)
procedure Module;
var
   ModRef: tObj; {Ссылка на имя модуля в таблице}
begin
   Check(lexMODULE, 'MODULE');
   if Lex <> lexName then
      Expected('имя модуля')
   else {Имя модуля - в таблицу имен}
      NewName(Name, catModule, ModRef);
   NextLex;
   Check(lexSemi, '";"');
   if Lex = lexIMPORT then
      Import;
   DeclSeq;
   if Lex = lexBEGIN then begin
      NextLex;
      StatSeq;
   end;
   Check(lexEND, 'END');

   {Сравнение имени модуля и имени после END}
      if Lex <> lexName then
         Expected('имя модуля')
      else if Name <> ModRef^.Name then
         Expected('имя модуля "'+ModRef^.Name+'"')
      else
         NextLex;
   if Lex <> lexDot then
```

```
      Expected('"."');
   Gen(0);            {Код возврата}
   Gen(cmStop);       {Команда останова}
   AllocateVariables; {Размещение переменных}
end;

procedure Compile;
begin
   InitNameTable;
   OpenScope; {Блок стандартных имен}
   Enter('ABS', catStProc, typInt, spABS);
   Enter('MAX', catStProc, typInt, spMAX);
   Enter('MIN', catStProc, typInt, spMIN);
   Enter('DEC', catStProc, typNone, spDEC);
   Enter('ODD', catStProc, typBool, spODD);
   Enter('HALT', catStProc, typNone, spHALT);
   Enter('INC', catStProc, typNone, spINC);
   Enter('INTEGER', catType, typInt, 0);
   OpenScope;  {Блок модуля}
   Module;
   CloseScope; {Блок модуля}
   CloseScope; {Блок стандартных имен}
   WriteLn;
   WriteLn('Компиляция завершена');
end;

end.
```

Листинг П2.6. Модуль для работы с таблицей имен

```
unit  OTable;
{ Таблица имен }

interface

uses OScan;

type
   {Категории имён}
      tCat = (catConst, catVar, catType,
              catStProc, catModule, catGuard);

   {Типы}
      tType = (typNone, typInt, typBool);

   tObj = ^tObjRec;     {Тип указателя на запись таблицы}
   tObjRec = record     {Тип записи таблицы имен}
      Name  : tName;    {Ключ поиска            }
      Cat   : tCat;     {Категория имени        }
```

```
      Typ   : tType;    {Тип                    }
      Val   : integer;  {Значение               }
      Prev  : tObj;     {Указатель на пред. имя }
   end;

{Инициализация таблицы}
   procedure InitNameTable;
{Добавление элемента}
   procedure Enter
      (N: tName; C: tCat; T: tType; V: integer);
{Занесение нового имени}
   procedure NewName
      (Name: tName; Cat: tCat; var Obj: tObj);
{Поиск имени}
   procedure Find(Name: tName; var Obj: tObj);
{Открытие области видимости (блока)}
   procedure OpenScope;
{Закрытие области видимости (блока)}
   procedure CloseScope;
{Поиск первой переменной}
   procedure FirstVar(var VRef : tObj);
{Поиск следующей переменной}
   procedure NextVar(var VRef : tObj);
{=========================================================}

implementation

uses
   OError;

var
   Top    : tObj;  {Указатель на вершину списка    }
   Bottom : tObj;  {Указатель на конец (дно) списка}
   CurrObj: tObj;

{Инициализация таблицы имен}
procedure InitNameTable;
begin
   Top := nil;
end;

procedure Enter(N: tName; C: tCat; T: tType; V: integer);
var
  P : tObj;
begin
   New(P);
   P^.Name := N;
   P^.Cat := C;
   P^.Typ := T;
```

```
   P^.Val := V;
   P^.Prev := Top;
   Top := P;
end;

procedure OpenScope;
begin
   Enter( '', catGuard, typNone, 0 );
   if Top^.Prev = nil then
      Bottom := Top;
end;

procedure CloseScope;
var
   P : tObj;
begin
   while Top^.Cat <> catGuard do begin
      P := Top;
      Top := Top^.Prev;
      Dispose(P);
   end;
   P := Top;
   Top := Top^.Prev;
   Dispose(P);
end;

procedure NewName(Name: tName; Cat: tCat; var Obj: tObj);
begin
   Obj := Top;
   while (Obj^.Cat<>catGuard) and (Obj^.Name<>Name) do
      Obj := Obj^.Prev;
   if Obj^.Cat = catGuard then begin
      New(Obj);
      Obj^.Name := Name;
      Obj^.Cat := Cat;
      Obj^.Val := 0;
      Obj^.Prev := Top;
      Top := Obj;
      end
   else
      Error('Повторное объявление имени');
end;

procedure Find(Name: tName; var Obj: tObj);
begin
   Bottom^.Name := Name;
   Obj := Top;
   while Obj^.Name <> Name do
      Obj := Obj^.Prev;
```

```
   if Obj = Bottom then
      Error('Необъявленное имя');
end;

procedure FirstVar(var VRef: tObj);
begin
   CurrObj := Top;
   NextVar(VRef);
end;

procedure NextVar(var VRef: tObj);
begin
   while (CurrObj<>Bottom) and (CurrObj^.Cat<>catVar) do
      CurrObj := CurrObj^.Prev;
   if CurrObj = Bottom then
      VRef := nil
   else begin
      VRef := CurrObj;
      CurrObj := CurrObj^.Prev;
   end
end;

end.
```

Листинг П2.7. Виртуальная О-машина

```
unit  OVM;
{Виртуальная машина}

interface

const
   MemSize = 8*1024;

   cmStop   = -1;

   cmAdd    = -2;
   cmSub    = -3;
   cmMult   = -4;
   cmDiv    = -5;
   cmMod    = -6;
   cmNeg    = -7;

   cmLoad   = -8;
   cmSave   = -9;

   cmDup    = -10;
   cmDrop   = -11;
   cmSwap   = -12;
```

```
   cmOver   = -13;

   cmGOTO   = -14;
   cmIfEQ   = -15;
   cmIfNE   = -16;
   cmIfLE   = -17;
   cmIfLT   = -18;
   cmIfGE   = -19;
   cmIfGT   = -20;

   cmIn     = -21;
   cmOut    = -22;
   cmOutLn  = -23;

var
   M: array [0..MemSize-1] of integer;

procedure Run;

{=====================================================}

implementation

procedure Run;
var
   PC    : integer;
   SP    : integer;
   Cmd   : integer;
   Buf   : integer;
begin
   PC := 0;
   SP := MemSize;
   Cmd := M[PC];
   while Cmd <> cmStop do begin
      PC := PC + 1;
      if Cmd >= 0 then begin
         SP := SP - 1;
         M[SP] := Cmd;
         end
      else
         case Cmd of
         cmAdd:
            begin
               SP := SP + 1;
               M[SP] := M[SP] + M[SP-1];
            end;
         cmSub:
            begin
               SP := SP + 1;
```

```
      M[SP] := M[SP] - M[SP-1];
   end;
cmMult:
   begin
      SP := SP + 1;
      M[SP] := M[SP]*M[SP-1];
   end;
cmDiv:
   begin
      SP := SP + 1;
      M[SP] := M[SP] div M[SP-1];
   end;
cmMod:
   begin
      SP := SP + 1;
      M[SP] := M[SP] mod M[SP-1];
   end;
cmNeg:
   M[SP] := -M[SP];
cmLoad:
   M[SP] := M[M[SP]];
cmSave:
   begin
      M[M[SP+1]] := M[SP];
      SP := SP + 2;
   end;
cmDup:
   begin
      SP := SP - 1;
      M[SP] := M[SP+1];
   end;
cmDrop:
   SP := SP + 1;
cmSwap:
   begin
      Buf := M[SP];
      M[SP] := M[SP+1];
      M[SP+1] := Buf;
   end;
cmOver:
   begin
      SP := SP - 1;
      M[SP] := M[SP+2];
   end;
cmGOTO:
   begin
      PC := M[SP];
      SP := SP + 1;
   end;
```

```
cmIfEQ:
   begin
      if M[SP+2] = M[SP+1] then
         PC := M[SP];
      SP := SP + 3;
   end;
cmIfNE:
   begin
      if M[SP+2] <> M[SP+1] then
         PC := M[SP];
      SP := SP + 3;
   end;
cmIfLE:
   begin
      if M[SP+2] <= M[SP+1] then
         PC := M[SP];
      SP := SP + 3;
   end;
cmIfLT:
   begin
      if M[SP+2] < M[SP+1] then
         PC := M[SP];
      SP := SP + 3;
   end;
cmIfGE:
   begin
      if M[SP+2] >= M[SP+1] then
         PC := M[SP];
      SP := SP + 3;
   end;
cmIfGT:
   begin
      if M[SP+2] > M[SP+1] then
         PC := M[SP];
      SP := SP + 3;
   end;
cmIn:
   begin
      SP := SP - 1;
      Write('?');
      Readln( M[SP] );
   end;
cmOut:
   begin
      Write(M[SP+1]:M[SP]);
      SP := SP + 2;
   end;
cmOutLn:
   WriteLn;
```

```
            else begin
               WriteLn('Недопустимый код операции');
               M[PC] := cmStop;
            end;
            end;
         Cmd := M[PC];
      end;
      WriteLn;
      if SP < MemSize then
         WriteLn('Код возврата ', M[SP]);
      Write('Нажмите ВВОД');
      ReadLn;
   end;

   end.
```

Листинг П2.8. Генератор кода

```
   unit OGen;
   { Генератор кода }

   interface

   uses
      OScan, OTable;

   var
      PC: integer;

   procedure InitGen;

   procedure Gen(Cmd: integer);
   procedure Fixup(A: integer);

   procedure GenAbs;
   procedure GenMin;
   procedure GenOdd;
   procedure GenConst(C: integer);
   procedure GenComp(Op: tLex);
   procedure GenAddr(X: tObj);
   procedure AllocateVariables;

   {=====================================================}

   implementation

   uses
      OVM, OError;
```

```
procedure InitGen;
begin
   PC := 0;
end;

procedure Gen(Cmd: integer);
begin
   M[PC] := Cmd;
   PC := PC+1;
end;

procedure Fixup(A: integer);
var
   temp: integer;
begin
   while A > 0 do begin
      temp := M[A-2];
      M[A-2] := PC;
      A := temp;
   end;
end;

procedure GenAbs;
begin
   Gen(cmDup);
   Gen(0);
   Gen(PC+3);
   Gen(cmIfGE);
   Gen(cmNeg);
end;

procedure GenMin;
begin
   Gen(MaxInt);
   Gen(cmNeg);
   Gen(1);
   Gen(cmSub);
end;

procedure GenOdd;
begin
   Gen(2);
   Gen(cmMod);
   Gen(0);
   Gen(0); { Адрес перехода вперед }
   Gen(cmIfEQ);
end;
```

```
procedure GenConst(C: integer);
begin
   Gen(abs(C));
   if C < 0 then
      Gen(cmNeg);
end;

procedure GenComp(Op: tLex);
begin
   Gen(0); { Переход вперед }
   case Op of
   lexEQ : Gen(cmIfNE);
   lexNE : Gen(cmIfEQ);
   lexLE : Gen(cmIfGT);
   lexLT : Gen(cmIfGE);
   lexGE : Gen(cmIfLT);
   lexGT : Gen(cmIfLE);
   end;
end;

procedure GenAddr(X: tObj);
begin
   Gen(X^.Val); {В текущую ячейку адрес предыдущей + 2}
   X^.Val := PC+1; {Адрес+2 = PC+1}
end;

procedure AllocateVariables;
var
   VRef: tObj; {Ссылка на переменную в таблице имен}
begin
   FirstVar(VRef);         {Найти первую переменную}
   while VRef <> nil do begin
      if VRef^.Val = 0 then
         Warning('Переменная ' + VRef^.Name +
                 ' не используется')
      else begin
         Fixup(VRef^.Val); {Адресная привязка переменной}
         PC := PC + 1;
      end;
      NextVar(VRef);       {Найти следующую переменную}
   end;
end;

end.
```

ПРИЛОЖЕНИЕ 3. ТЕКСТ КОМПИЛЯТОРА ЯЗЫКА «О» НА ОБЕРОНЕ

В этом приложении можно видеть программу-компилятор языка «О», записанную на Обероне-2. Одновременно этот же текст является правильной программой на языке Оберон, поскольку расширения, отличающие Оберон-2 от Оберона (оператор **FOR**, связанные процедуры и др.), не использованы.

Отличия версий для компиляторов JOB и XDS

Модуль драйвера текста, использующий библиотеки ввода-вывода, представлен в двух вариантах: для компилятора JOB, транслирующего в файлы класса виртуальной Ява-машины (JVM), и компилятора XDS[57], позволяющего создавать исполняемые файлы Windows. Компилятор JOB предлагает для работы с файлами модуль, совместимый с «Дубовыми требованиями»; комплект компилятора XDS содержит библиотеки, соответствующие ISO-стандарту Модулы-2.

Компиляторы JOB и XDS не поддерживают концепцию команд, определенную спецификацией Оберона-2 и предполагающую, что выполнение Оберон-программы может начинаться вызовом любой экспортированной процедуры без параметров. JOB использует соглашения Явы, в соответствии с которыми выполнение начинается с процедуры (метода) `main`, XDS требует помечать главный модуль с помощью директивы, предусмотренной ISO-стандартом Модулы-2. По-разному в двух этих системах организуется и получение программой параметров командной

[57] Использована свободно распространяемая для некоммерческого использования персональная версия 2.50 компилятора Модулы-2 и Оберона-2 новосибирской компании Excelsior. Компилятор можно получить на сайте компании: www.excelsior-usa.com

строки. Эти различия привели к необходимости создания дополнительного стартового модуля, вызывающего основной компилятор. Этот модуль называется в одном случае OJOB, в другом — OXDS.

Изменение обозначений

Программа на Обероне написана не заново, а получена «переводом» с Паскаля. Она имеет аналогичную структуру. Основные обозначения сохранены, но некоторые изменились. Обязательное требование Оберона указывать имя модуля при ссылке на любой импортированный идентификатор вынуждает выбирать более естественные имена модулей. Поэтому в их названиях не используется префикс O. Модуль, который в программе на Паскале назывался `OPars`, переименован в `Pars` и т. д. Названия модулей, используемые в уточненных идентификаторах, берут на себя часть смысловой нагрузки. Так, вызовы всех процедур модуля `Gen` начинаются с `Gen`. Записывать обращение к основной процедуре генератора кода в виде `Gen.Gen(...)` было бы нехорошо. Эта процедура, генерирующая команды виртуальной машины, переименована в `Cmd` и вызывается теперь так: `Gen.Cmd(...)` — сгенерировать команду. В других подобных случаях названия также изменены.

Изменения в структуре компилятора

Оберон не допускает циклического импорта. Если один модуль импортирует другой, то второй не должен импортировать первый непосредственно или через другие модули. В используемых диалектах Паскаля циклический импорт не запрещен. Паскаль-версия компилятора содержит кольцевые межмодульные связи (рис. П3.1).

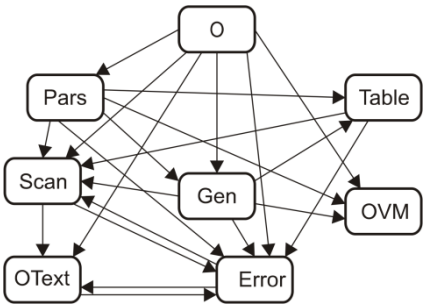


Рис. П3.1. Межмодульные связи в Паскаль-версии компилятора «О»

Циклы связаны с модулями `OText`, `OError` и `Oscan` и обусловлены потребностью модуля, отвечающего за реакцию на ошибки, получать сведения о номере текущей строки (из модуля `OText`) и номере символа, начинающего лексему (из модуля `OScan`). С другой стороны, модули `OScan` и `OText` используют процедуры модуля `OError` для выдачи сообщений об ошибках.

Чтобы избавиться от циклического импорта, в программе на Обероне предусмотрен дополнительный модуль `Location` (местоположение), который экспортирует переменные, определяющие текущее место исходного текста, включая путь к файлу исходного текста (листинг П3.4, рис. П3.2). Модули `Text` и `Scan`, а также `OJOB` заносят данные в переменные модуля `Location`, другие модули могут этой информацией пользоваться.

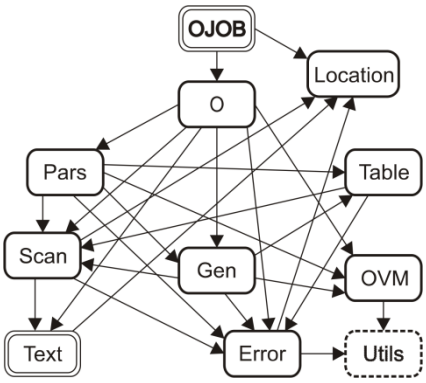


Рис. П3.2. Модульная структура Оберон-версии компилятора «О». Двойным контуром показаны модули, зависящие от версии Оберон-компилятора.

Использованный прием исключения циклического импорта является весьма универсальным: ресурсы, использование которых порождает цикл, выносятся в отдельный модуль. Его применение, хотя и увеличило число модулей, но улучшило структуру программы: `Text` и `Error` стали независимы от модулей верхнего уровня и друг от друга.

Вспомогательный модуль `Utils` (листинг П3.7) экспортирует процедуру `ReadLn`, которой нет в стандартном для Оберон-систем модуле `In`.

Версии компилятора «О», записанные на Си, Яве и Си#, имеют структуру, подобную показанной на рис. П3.2. Может отсутствовать модуль `Utils`, не используется отдельный стартовый модуль.

Листинг П3.1. Стартовый модуль для компилятора JOB

```
MODULE OJob;

IMPORT javalang, O, Location;

PROCEDURE main*(VAR args: ARRAY OF javalang.PString);
VAR
   i: INTEGER;
BEGIN
   IF LEN(args) = 0 THEN
      Location.Path := "";
   ELSE
      i := 0;
      WHILE i<args[0].length() DO
         Location.Path[i] := args[0].charAt(i);
         INC(i);
      END;
      Location.Path[i] := 0X;
   END;
   O.Compiler;
END main;

END OJob.
```

Листинг П3.2. Стартовый модуль для компилятора XDS

```
<*MAIN+*>
MODULE OXDS;
IMPORT O;
BEGIN
   O.Compiler;
END OXDS.
```

Листинг П3.3. Основной модуль компилятора

```
MODULE O;
(* Компилятор языка O *)

IMPORT
   Text, Error, Scan, Pars, OVM, Gen, Out, In;

PROCEDURE Init;
BEGIN
   In.Open; (* Инициализация ввода *)
   Text.Reset;
   IF ~Text.Ok THEN Error.Message(Text.Message) END;
   Scan.Init;
   Gen.Init;
END Init;

PROCEDURE Done;
BEGIN
   Text.Close;
END Done;

PROCEDURE Compiler*;
BEGIN
   Out.String("Компилятор языка O"); Out.Ln;
   Init;          (* Инициализация *)
   Pars.Compile;  (* Компиляция    *)
   OVM.Run;       (* Выполнение    *)
   Done;          (* Завершение    *)
END Compiler;

END O.
```

Листинг П3.4. Вспомогательный интерфейсный модуль `Location`

```
MODULE Location;
VAR
   Line*    : INTEGER; (* Номер строки *)
   Pos*     : INTEGER; (* Номер позиции *)
   LexPos*  : INTEGER; (* Позиция начала лексемы *)
   Path*    : ARRAY 256 OF CHAR;
```

```
BEGIN
   Line := 0;
   Pos := 0;
   LexPos := 0;
   Path := "";
END Location.
```

Листинг П3.5. Модуль драйвера текста для компилятора JOB

```
MODULE  Text;
(*Драйвер исходного текста для компилятора JOB *)

IMPORT Location, Files, Out;

CONST
   chSpace* = " ";  (* Пробел       *)
   chTab*   = 9X;   (* Табуляция    *)
   chEOL*   = 0AX;  (* Конец строки *)
   chEOT*   = 0X;   (* Конец текста *)
   TabSize  = 3;

VAR
   Ch*      : CHAR; (* Очередной символ *)
   Ok*      : BOOLEAN;
   Message* : ARRAY 80 OF CHAR;

   f        : Files.File;
   r        : Files.Rider;

PROCEDURE NextCh*;
BEGIN
   Files.ReadASCII(r, Ch);
   IF r.eof THEN
      Ch := chEOT
   ELSIF Ch = 0AX THEN
      Out.Ln;
      INC(Location.Line);
      Location.Pos := 0;
      Ch := chEOL;
   ELSIF Ch = 0DX THEN
      NextCh
   ELSIF Ch # chTab THEN
      Out.Char(Ch);
      INC(Location.Pos);
   ELSE
      REPEAT
         Out.Char(" ");
         INC(Location.Pos);
      UNTIL Location.Pos MOD TabSize = 0;
```

```
   END;
END NextCh;

PROCEDURE Reset*;
BEGIN
   IF Location.Path = "" THEN
      Out.String("Формат вызова: "); Out.Ln;
      Out.String("   O <входной файл>"); Out.Ln;
      HALT(0);
   ELSE
      f := Files.Old(Location.Path);
      IF f = NIL THEN
         Ok := FALSE;
         Message := "Файл не открыт"
      ELSE
         Files.Set(r, f, 0);
         Ok := TRUE;
         Message := "Ok";
         Location.Pos := 0;
         Location.Line := 1;
         NextCh;
      END;
   END;
END Reset;

PROCEDURE Close*;
BEGIN
   Files.Close(f);
END Close;

BEGIN
   Ok := FALSE;
   Message := "Не инициализирован текст"
END Text.
```

Листинг П3.6. Модуль драйвера текста для компилятора XDS

```
MODULE  Text;
(* Драйвер исходного текста для компилятора XDS *)

IMPORT
   Location, SeqFile, TextIO, IOResult, ProgEnv, Out;

CONST
   chSpace* = " ";  (* Пробел       *)
   chTab*   = 9X;   (* Табуляция    *)
   chEOL*   = 0AX;  (* Конец строки *)
   chEOT*   = 0X;   (* Конец текста *)
   TabSize  = 3;
```

```
VAR
   Ch*      : CHAR;    (* Очередной символ      *)
   Ok*      : BOOLEAN; (* = Файл успешно открыт *)
   Message* : ARRAY 80 OF CHAR;
   f        : SeqFile.ChanId;

PROCEDURE NextCh*;
BEGIN
   TextIO.ReadChar(f, Ch);
   CASE IOResult.ReadResult(f) OF
   |IOResult.endOfInput:
      Ch := chEOT;
   |IOResult.endOfLine:
      TextIO.SkipLine(f);
      Out.Ln;
      INC(Location.Line);
      Location.Pos := 0;
      Ch := chEOL;
   ELSE
      IF Ch # chTab THEN
         Out.Char(Ch);
         INC(Location.Pos);
      ELSE
         REPEAT
            Out.Char(" ");
            INC(Location.Pos);
         UNTIL Location.Pos MOD TabSize = 0;
      END;
   END;
END NextCh;

PROCEDURE Reset*;
CONST
   FLAGS = SeqFile.text + SeqFile.old;
VAR
   res : SeqFile.OpenResults;
BEGIN
   IF ProgEnv.ArgNumber() < 1 THEN
      Out.String("Формат вызова:"); Out.Ln;
      Out.String("   O <входной файл>"); Out.Ln;
      HALT(0);
   ELSE
      ProgEnv.GetArg(0, Location.Path);
      SeqFile.OpenRead(f, Location.Path, FLAGS, res);
      Ok := res = SeqFile.opened;
      IF Ok THEN
         Message := "Ok";
         Location.Pos := 0;
```

```
         Location.Line := 1;
         NextCh;
      ELSE
         Message :="Файл не открыт"
      END;
   END;
END Reset;

PROCEDURE Close*;
BEGIN
   SeqFile.Close(f);
END Close;

BEGIN
   Ok := FALSE;
   Message := "Не инициализирован текст"
END Text.
```

Листинг П3.7. Вспомогательный модуль

```
MODULE Utils;

IMPORT In;

PROCEDURE ReadLn*;
VAR
   Ch : CHAR;
BEGIN
   REPEAT
      In.Char(Ch);
   UNTIL Ch = 0AX;
END ReadLn;

END Utils.
```

Листинг П3.8. Модуль обработки ошибок

```
MODULE  Error; (* Обработка ошибок *)

IMPORT Location, Text, Utils, Out, In, Strings;

PROCEDURE Message*(Msg : ARRAY OF CHAR);
VAR
   ELine : INTEGER;
   i     : INTEGER;
BEGIN
   ELine := Location.Line;
   WHILE (Text.Ch # Text.chEOL)&(Text.Ch # Text.chEOT) DO
      Text.NextCh
   END;
```

```
    IF Text.Ch = Text.chEOT THEN Out.Ln END;
    i := 1;
    WHILE i < Location.LexPos DO
       Out.Char(" ");
       INC(i);
    END;
    Out.String("^"); Out.Ln;
    Out.String("(Строка ");
    Out.Int(ELine, 0);
    Out.String(") Ошибка: ");
    Out.String(Msg); Out.Ln;
    Out.String("Нажмите ВВОД");
    Utils.ReadLn;
    HALT(1);
END Message;

PROCEDURE Expected*(Msg: ARRAY OF CHAR);
VAR
    s : ARRAY 80 OF CHAR;
BEGIN
    s := "Ожидается ";
    Strings.Append(Msg, s);
    Message(s);
END Expected;

PROCEDURE Warning*(Msg : ARRAY OF CHAR);
BEGIN
    Out.Ln;
    Out.String("Предупреждение: ");
    Out.String(Msg);
    Out.Ln;
END Warning;

END Error.
```

В Обероне отсутствуют перечислимые типы, поэтому для обозначения видов лексем, а также категорий и типов имен используются целочисленные константы .

Листинг П3.9. Лексический анализатор

```
MODULE  Scan;
(* Сканер *)

IMPORT Out, Location, Text, Error;

CONST
    NameLen* = 31; (* Наибольшая длина имени *)
```

```
   lexNone*    = 0;  lexName*    = 1;  lexNum*  = 2;

   lexMODULE*  = 3;  lexIMPORT*  = 4;
   lexBEGIN*   = 5;  lexEND*     = 6;
   lexCONST*   = 7;  lexVAR*     = 8;
   lexWHILE*   = 9;  lexDO*      = 10;
   lexIF*      = 11; lexTHEN*    = 12;
   lexELSIF*   = 13; lexELSE*    = 14;
   lexMult*    = 15; lexDIV*     = 16; lexMOD*  = 17;
   lexPlus*    = 18; lexMinus*   = 19;

   lexEQ*   = 20; lexNE*   = 21; lexLT*   = 22;
   lexLE*   = 23; lexGT*   = 24; lexGE*   = 25;

   lexDot*  = 26; lexComma*   = 27; lexColon*   = 28;
   lexSemi* = 29; lexAss*     = 30;
   lexLPar* = 31; lexRPar*    = 32;

   lexEOT*  = 33;

TYPE
   tName* = ARRAY NameLen+1 OF CHAR;

VAR
   Lex*   : INTEGER; (* Текущая лексема             *)
   Name*  : tName;   (* Строковое значение имени    *)
   Num*   : INTEGER; (* Значение числовых литералов *)

CONST
   KWNum = 34;

VAR
   nkw      : INTEGER;
   KWTable  : ARRAY KWNum OF
      RECORD
         Word  : tName;
         Lex   : INTEGER;
      END;

PROCEDURE EnterKW(Name: tName; Lex: INTEGER);
BEGIN
   KWTable[nkw].Word := Name;
   KWTable[nkw].Lex := Lex;
   nkw := nkw + 1;
END EnterKW;
```

```
PROCEDURE TestKW(): INTEGER;
VAR
   i : INTEGER;
BEGIN
   i := nkw-1;
   WHILE (i>=0) & (Name # KWTable[i].Word) DO
      DEC(i);
   END;
   IF i>=0 THEN
      RETURN KWTable[i].Lex
   ELSE
      RETURN lexName;
   END
END TestKW;

PROCEDURE Ident;
VAR
   i : INTEGER;
BEGIN
   i := 0;
   REPEAT
      IF i < NameLen THEN
         Name[i] := Text.Ch;
         INC(i);
      ELSE
         Error.Message("Слишком длинное имя");
      END;
      Text.NextCh;
   UNTIL ((Text.Ch<"A") OR (Text.Ch>"Z")) &
      ((Text.Ch<"a") OR (Text.Ch>"z")) &
      ((Text.Ch<"0") OR (Text.Ch>"9"));
   Name[i] := 0X;  (* Длина строки Name теперь равна i *)
   Lex := TestKW();(* Проверка на ключевое слово       *)
END Ident;

PROCEDURE Number;
VAR
   d : INTEGER;
BEGIN
   Lex := lexNum;
   Num := 0;
   REPEAT
      d := ORD(Text.Ch) - ORD("0");
      IF (MAX(INTEGER) - d) DIV 10 >= Num THEN
         Num := 10*Num + d
      ELSE
         Error.Message("Слишком большое число");
      END;
      Text.NextCh;
```

```
   UNTIL (Text.Ch<"0") OR (Text.Ch>"9");
END Number;

(*
PROCEDURE Comment;
BEGIN
   Text.NextCh;
   REPEAT
      WHILE (Text.Ch # "*") & (Text.Ch # Text.chEOT) DO
         IF Text.Ch = "(" THEN
            Text.NextCh;
            IF Text.Ch = "*" THEN Comment END
         ELSE
            Text.NextCh;
         END;
      END;
      IF Text.Ch = "*" THEN Text.NextCh END
   UNTIL (Text.Ch = ")") OR (Text.Ch = Text.chEOT);
   IF Text.Ch = ")" THEN Text.NextCh
   ELSE Error.Message("Не закончен комментарий");
   END
END Comment;
*)

PROCEDURE Comment;
VAR
   Level : INTEGER;
BEGIN
   Level := 1;
   Text.NextCh;
   REPEAT
      IF Text.Ch = "*" THEN
         Text.NextCh;
         IF Text.Ch = ")" THEN
            DEC(Level); Text.NextCh
         END;
      ELSIF Text.Ch = "(" THEN
         Text.NextCh;
         IF Text.Ch = "*" THEN
            INC(Level); Text.NextCh
         END;
      ELSE (*IF Text.Ch # chEOT THEN*)
         Text.NextCh;
      END;
   UNTIL (Level = 0) OR (Text.Ch = Text.chEOT);
   IF Level # 0 THEN
      Error.Message("Не закончен комментарий");
   END;
END Comment;
```

```
PROCEDURE NextLex*;
BEGIN
   WHILE (Text.Ch = Text.chSpace) OR
      (Text.Ch = Text.chTab) OR
      (Text.Ch = Text.chEOL)
   DO
      Text.NextCh;
   END;
   Location.LexPos := Location.Pos;
   CASE Text.Ch OF
   |"A".."Z", "a"..."z":
      Ident;
   |"0"..."9":
      Number;
   |";":
      Text.NextCh; Lex := lexSemi;
   |":":
      Text.NextCh;
      IF Text.Ch = "=" THEN
         Text.NextCh;   Lex := lexAss;
      ELSE
         Lex := lexColon;
      END;
   |".":
      Text.NextCh; Lex := lexDot;
   |",":
      Text.NextCh; Lex := lexComma;
   |"=":
      Text.NextCh; Lex := lexEQ;
   |"#":
      Text.NextCh; Lex := lexNE;
   |"<":
      Text.NextCh;
      IF Text.Ch="=" THEN
         Text.NextCh;   Lex := lexLE;
      ELSE
         Lex := lexLT;
      END;
   |">":
      Text.NextCh;
      IF Text.Ch="=" THEN
         Text.NextCh;   Lex := lexGE;
      ELSE
         Lex := lexGT;
      END;
   |"(":
      Text.NextCh;
      IF Text.Ch = "*" THEN
```

```
         Comment;
         NextLex;
      ELSE
         Lex := lexLPar;
      END;
   |")":
      Text.NextCh; Lex := lexRPar;
   |"+":
      Text.NextCh; Lex := lexPlus;
   |"-":
      Text.NextCh; Lex := lexMinus;
   |"*":
      Text.NextCh; Lex := lexMult;
   |Text.chEOT:
      Lex := lexEOT;
   ELSE
      Error.Message("Недопустимый символ");
   END;
END NextLex;

PROCEDURE Init*;
BEGIN
   nkw := 0;
   EnterKW("ARRAY",     lexNone);
   EnterKW("BY",        lexNone);
   EnterKW("BEGIN",     lexBEGIN);
   EnterKW("CASE",      lexNone);
   EnterKW("CONST",     lexCONST);
   EnterKW("DIV",       lexDIV);
   EnterKW("DO",        lexDO);
   EnterKW("ELSE",      lexELSE);
   EnterKW("ELSIF",     lexELSIF);
   EnterKW("END",       lexEND);
   EnterKW("EXIT",      lexNone);
   EnterKW("FOR",       lexNone);
   EnterKW("IF",        lexIF);
   EnterKW("IMPORT",    lexIMPORT);
   EnterKW("IN",        lexNone);
   EnterKW("IS",        lexNone);
   EnterKW("LOOP",      lexNone);
   EnterKW("MOD",       lexMOD);
   EnterKW("MODULE",    lexMODULE);
   EnterKW("NIL",       lexNone);
   EnterKW("OF",        lexNone);
   EnterKW("OR",        lexNone);
   EnterKW("POINTER",   lexNone);
   EnterKW("PROCEDURE", lexNone);
   EnterKW("RECORD",    lexNone);
   EnterKW("REPEAT",    lexNone);
```

```
   EnterKW("RETURN",    lexNone);
   EnterKW("THEN",      lexTHEN);
   EnterKW("TO",        lexNone);
   EnterKW("TYPE",      lexNone);
   EnterKW("UNTIL",     lexNone);
   EnterKW("VAR",       lexVAR);
   EnterKW("WHILE",     lexWHILE);
   EnterKW("WITH",      lexNone);
   NextLex;
END Init;

END Scan.
```

Листинг П3.10. Синтаксический и контекстный анализатор

```
MODULE Pars;
(* Распознаватель *)

IMPORT Out, Scan, Error, Table, Gen, OVM, Strings;

CONST
   (* Стандартные процедуры *)
      spABS    = 1;
      spMAX    = 2;
      spMIN    = 3;
      spDEC    = 4;
      spODD    = 5;
      spHALT   = 6;
      spINC    = 7;
      spInOpen = 8;
      spInInt  = 9;
      spOutInt = 10;
      spOutLn  = 11;

PROCEDURE^ StatSeq;
PROCEDURE^ Expression(VAR t: INTEGER);

PROCEDURE Check(L: INTEGER; M: ARRAY OF CHAR);
BEGIN
   IF Scan.Lex # L THEN Error.Expected(M)
   ELSE Scan.NextLex;
   END
END Check;

(* {"+" | "-"} (Число | Имя) *)
PROCEDURE ConstExpr(VAR V: INTEGER);
VAR
   X  : Table.tObj;
   Op : INTEGER;
```

```
BEGIN
   Op := Scan.lexPlus;
   IF Scan.Lex IN {Scan.lexPlus, Scan.lexMinus} THEN
      Op := Scan.Lex;
      Scan.NextLex;
   END;
   IF Scan.Lex = Scan.lexNum THEN
      V := Scan.Num;
      Scan.NextLex;
   ELSIF Scan.Lex = Scan.lexName THEN
      Table.Find(Scan.Name, X);
      IF X^.Cat = Table.catGuard THEN
         Error.Message("Нельзя так определять константу")
      ELSIF X^.Cat # Table.catConst THEN
         Error.Expected("имя константы")
      ELSE
         V := X^.Val;
      END;
      Scan.NextLex;
   ELSE
      Error.Expected("константное выражение");
   END;
   IF Op = Scan.lexMinus THEN
      V := -V;
   END
END ConstExpr;

(* Имя "=" КонстВыраж *)
PROCEDURE ConstDecl;
VAR
   ConstRef: Table.tObj; (* Ссылка на имя в таблице *)
BEGIN
   Table.NewName(Scan.Name, Table.catGuard, ConstRef);
   Scan.NextLex;
   Check(Scan.lexEQ, "'='");
   ConstExpr(ConstRef^.Val);
   ConstRef^.Typ := Table.typInt;
   ConstRef^.Cat := Table.catConst;
END ConstDecl;

(* Тип = Имя *)
PROCEDURE ParseType;
VAR
   TypeRef : Table.tObj;
BEGIN
   IF Scan.Lex # Scan.lexName THEN
      Error.Expected("имя")
   ELSE
      Table.Find(Scan.Name, TypeRef);
```

```
      IF TypeRef^.Cat # Table.catType THEN
         Error.Expected("имя типа")
      ELSIF TypeRef^.Typ # Table.typInt THEN
         Error.Expected("целый тип");
      END;
      Scan.NextLex;
   END;
END ParseType;

(* Имя {"," Имя} ":" Тип *)
PROCEDURE VarDecl;
VAR
   NameRef : Table.tObj;
BEGIN
   NameRef := NIL; (* Однозначное присваивание для JVM *)
   IF Scan.Lex # Scan.lexName THEN
      Error.Expected("имя")
   ELSE
      Table.NewName(Scan.Name, Table.catVar, NameRef);
      NameRef^.Typ := Table.typInt;
      Scan.NextLex;
   END;
   WHILE Scan.Lex = Scan.lexComma DO
      Scan.NextLex;
      IF Scan.Lex # Scan.lexName THEN
         Error.Expected("имя")
      ELSE
         Table.NewName(Scan.Name, Table.catVar, NameRef);
         NameRef^.Typ := Table.typInt;
         Scan.NextLex;
      END;
   END;
   Check(Scan.lexColon, "':'");
   ParseType;
END VarDecl;

(* {CONST {ОбъявлКонст ";"} |VAR {ОбъявлПерем ";"} } *)
PROCEDURE DeclSeq;
BEGIN
   WHILE Scan.Lex IN {Scan.lexCONST, Scan.lexVAR} DO
      IF Scan.Lex = Scan.lexCONST THEN
         Scan.NextLex;
         WHILE Scan.Lex = Scan.lexName DO
            ConstDecl; (* Объявление константы *)
            Check( Scan.lexSemi, "';'" );
         END;
      ELSE
         Scan.NextLex; (* VAR *)
         WHILE Scan.Lex = Scan.lexName DO
```

```
            VarDecl;   (* Объявление переменных *)
            Check( Scan.lexSemi, "';'" );
         END;
      END;
   END;
END DeclSeq;

PROCEDURE IntExpression;
VAR
   T : INTEGER;
BEGIN
   Expression(T);
   IF T # Table.typInt THEN
      Error.Expected("выражение целого типа");
   END
END IntExpression;

PROCEDURE StFunc(F: INTEGER; VAR T: INTEGER);
BEGIN
   CASE F OF
   |spABS:
      IntExpression;
      Gen.Abs;
      T := Table.typInt;
   |spMAX:
      ParseType;
      Gen.Cmd(MAX(INTEGER));
      T := Table.typInt;
   |spMIN:
      ParseType;
      Gen.Min;
      T := Table.typInt;
   |spODD:
      IntExpression;
      Gen.Odd;
      T := Table.typBool;
   END;
END StFunc;

(* Имя{"(" Выраж | Тип ")"} | Число | "("Выраж")" *)
PROCEDURE Factor(VAR T : INTEGER);
VAR
   X : Table.tObj;
BEGIN
   IF Scan.Lex = Scan.lexName THEN
      Table.Find(Scan.Name, X);
      IF X^.Cat = Table.catVar THEN
         Gen.Addr(X);    (* Адрес переменной *)
         Gen.Cmd(OVM.cmLoad);
```

```
         T := X^.Typ;
         Scan.NextLex;
      ELSIF X^.Cat = Table.catConst THEN
         Gen.Const(X^.Val);
         T := X^.Typ;
         Scan.NextLex;
      ELSIF (X^.Cat = Table.catStProc) &
         (X^.Typ # Table.typNone)
      THEN
         Scan.NextLex;
         Check(Scan.lexLPar, "'('");
         StFunc(X^.Val, T);
         Check(Scan.lexRPar, "')'");
      ELSE
         Error.Expected(
         "переменная, константа или процедура-функция"
         );
      END
   ELSIF Scan.Lex = Scan.lexNum THEN
      T := Table.typInt;
      Gen.Const(Scan.Num);
      Scan.NextLex
   ELSIF Scan.Lex = Scan.lexLPar THEN
      Scan.NextLex;
      Expression(T);
      Check(Scan.lexRPar, "')'");
   ELSE
      Error.Expected("имя, число или '('");
   END;
END Factor;

(* Множитель {ОперУмн Множитель} *)
PROCEDURE Term(VAR T: INTEGER);
VAR
   Op : INTEGER;
BEGIN
   Factor(T);
   IF Scan.Lex IN {Scan.lexMult,Scan.lexDIV,Scan.lexMOD}
   THEN
      IF T # Table.typInt THEN
         Error.Message("Неподходящая операция");
      END;
      REPEAT
         Op := Scan.Lex;
         Scan.NextLex;
         Factor(T);
         IF T # Table.typInt THEN
            Error.Expected("выражение целого типа");
         END;
```

```
         CASE Op OF
         |Scan.lexMult: Gen.Cmd(OVM.cmMult);
         |Scan.lexDIV:  Gen.Cmd(OVM.cmDiv);
         |Scan.lexMOD:  Gen.Cmd(OVM.cmMod);
         END;
      UNTIL ~(Scan.Lex IN
            {Scan.lexMult, Scan.lexDIV, Scan.lexMOD});
   END;
END Term;

(* {"+"|"-"} Слагаемое {ОперСлож Слагаемое} *)
PROCEDURE SimpleExpr(VAR T : INTEGER);
VAR
   Op : INTEGER;
BEGIN
   IF Scan.Lex IN {Scan.lexPlus, Scan.lexMinus} THEN
      Op := Scan.Lex;
      Scan.NextLex;
      Term(T);
      IF T # Table.typInt THEN
         Error.Expected("выражение целого типа");
      END;
      IF Op = Scan.lexMinus THEN
         Gen.Cmd(OVM.cmNeg);
      END
   ELSE
      Term(T);
   END;
   IF Scan.Lex IN {Scan.lexPlus, Scan.lexMinus} THEN
      IF T # Table.typInt THEN
         Error.Message("Неподходящая операция");
      END;
      REPEAT
         Op := Scan.Lex;
         Scan.NextLex;
         Term(T);
         IF T # Table.typInt THEN
            Error.Expected("выражение целого типа");
         END;
         CASE Op OF
         |Scan.lexPlus:  Gen.Cmd(OVM.cmAdd);
         |Scan.lexMinus: Gen.Cmd(OVM.cmSub);
         END;
      UNTIL ~(Scan.Lex IN {Scan.lexPlus, Scan.lexMinus});
   END;
END SimpleExpr;
```

```
(* ПростоеВыраж {Отношение ПростоеВыраж} *)
PROCEDURE Expression(VAR T : INTEGER);
VAR
   Op : INTEGER;
BEGIN
   SimpleExpr(T);
   IF Scan.Lex IN
      {Scan.lexEQ, Scan.lexNE, Scan.lexGT,
      Scan.lexGE, Scan.lexLT, Scan.lexLE}
   THEN
      Op := Scan.Lex;
      IF T # Table.typInt THEN
         Error.Message("Сравнение здесь не разрешено");
      END;
      Scan.NextLex;
      SimpleExpr(T);  (* Правый операнд отношения     *)
      IF T # Table.typInt THEN
         Error.Expected("выражение целого типа");
      END;
      Gen.Comp(Op);   (* Генерация условного перехода *)
      T := Table.typBool;
   END;
END Expression;

(* Переменная = Имя. *)
PROCEDURE Variable;
VAR
   X : Table.tObj;
BEGIN
   IF Scan.Lex # Scan.lexName THEN
      Error.Expected("имя")
   ELSE
      Table.Find(Scan.Name, X);
      IF X^.Cat # Table.catVar THEN
         Error.Expected("имя переменной");
      END;
      Gen.Addr(X);
      Scan.NextLex;
   END;
END Variable;

(* Стандартные процедуры *)
PROCEDURE StProc(P: INTEGER);
VAR
   c : INTEGER;
BEGIN
   CASE P OF
   |spDEC:
      Variable;
```

```
      Gen.Cmd(OVM.cmDup);
      Gen.Cmd(OVM.cmLoad);
      IF Scan.Lex = Scan.lexComma THEN
         Scan.NextLex;
         IntExpression;
      ELSE
         Gen.Cmd(1);
      END;
      Gen.Cmd(OVM.cmSub);
      Gen.Cmd(OVM.cmSave);
   |spINC:
      Variable;
      Gen.Cmd(OVM.cmDup);
      Gen.Cmd(OVM.cmLoad);
      IF Scan.Lex = Scan.lexComma THEN
         Scan.NextLex;
         IntExpression;
      ELSE
         Gen.Cmd(1);
      END;
      Gen.Cmd(OVM.cmAdd);
      Gen.Cmd(OVM.cmSave);
   |spInOpen:
      (* Пусто *);
   |spInInt:
      Variable;
      Gen.Cmd(OVM.cmIn);
      Gen.Cmd(OVM.cmSave);
   |spOutInt:
      IntExpression;
      Check(Scan.lexComma , "',");
      IntExpression;
      Gen.Cmd(OVM.cmOut);
   |spOutLn:
      Gen.Cmd(OVM.cmOutLn);
   |spHALT:
      ConstExpr(c);
      Gen.Const(c);
      Gen.Cmd(OVM.cmStop);
   END;
END StProc;

PROCEDURE BoolExpression;
VAR
   T : INTEGER;
BEGIN
   Expression(T);
   IF T # Table.typBool THEN
      Error.Expected("логическое выражение");
```

```
   END
END BoolExpression;

(* Переменная ":=" Выраж *)
PROCEDURE AssStatement;
BEGIN
   Variable;
   IF Scan.Lex = Scan.lexAss THEN
      Scan.NextLex;
      IntExpression;
      Gen.Cmd(OVM.cmSave);
   ELSE
      Error.Expected("':='")
   END;
END AssStatement;

(* Имя { "(" { Выраж | Переменная } ")" } *)
PROCEDURE CallStatement(sp : INTEGER);
BEGIN
   Check(Scan.lexName, "имя процедуры");
   IF Scan.Lex = Scan.lexLPar THEN
      Scan.NextLex;
      StProc(sp);
      Check( Scan.lexRPar, "')'" );
   ELSIF sp IN {spOutLn, spInOpen} THEN
      StProc(sp)
   ELSE
      Error.Expected("'('");
   END;
END CallStatement;

(* IF Выраж THEN ПослОператоров
   {ELSIF Выраж THEN ПослОператоров}
   {ELSE ПослОператоров} END *)
PROCEDURE IfStatement;
VAR
   CondPC   : INTEGER;
   LastGOTO : INTEGER;
BEGIN
   Check(Scan.lexIF, "IF");
   LastGOTO := 0;     (* Предыдущего перехода нет    *)
   BoolExpression;
   CondPC := Gen.PC;  (* Запомн. полож. усл.перехода *)
   Check(Scan.lexTHEN, "THEN");
   StatSeq;
   WHILE Scan.Lex = Scan.lexELSIF DO
      Gen.Cmd(LastGOTO);  (* Фикт. адрес, указывающий *)
      Gen.Cmd(OVM.cmGOTO);(* на место пред. перехода. *)
      LastGOTO := Gen.PC; (* Запомнить место GOTO     *)
```

```
      Scan.NextLex;
      Gen.Fixup(CondPC);  (* Фикс. адр. усл. перехода *)
      BoolExpression;
      CondPC := Gen.PC; (* Запомн. полож. усл. перех. *)
      Check(Scan.lexTHEN, "THEN");
      StatSeq;
   END;
   IF Scan.Lex = Scan.lexELSE THEN
      Gen.Cmd(LastGOTO);  (* Фикт. адрес, указывающий *)
      Gen.Cmd(OVM.cmGOTO);(* на место пред. перехода  *)
      LastGOTO := Gen.PC; (* Запомн. место посл. GOTO *)
      Scan.NextLex;
      Gen.Fixup(CondPC);(* Зафикс. адрес услов. перех.*)
      StatSeq;
   ELSE
      Gen.Fixup(CondPC);  (* Если ELSE отсутствует     *)
   END;
   Check( Scan.lexEND, "END" );
   Gen.Fixup(LastGOTO);   (* Направить сюда все GOTO   *)
END IfStatement;

(* WHILE Выраж DO ПослОператоров END *)
PROCEDURE WhileStatement;
VAR
   WhilePC  : INTEGER;
   CondPC   : INTEGER;
BEGIN
   WhilePC := Gen.PC;
   Check(Scan.lexWHILE, "WHILE");
   BoolExpression;
   CondPC := Gen.PC;
   Check(Scan.lexDO, "DO");
   StatSeq;
   Check(Scan.lexEND, "END");
   Gen.Cmd(WhilePC);
   Gen.Cmd(OVM.cmGOTO);
   Gen.Fixup(CondPC);
END WhileStatement;

PROCEDURE Statement;
VAR
   X           : Table.tObj;
   Designator  : Scan.tName;
   msg         : ARRAY 80 OF CHAR;
BEGIN
   IF Scan.Lex = Scan.lexName THEN
      Table.Find(Scan.Name, X);
      IF X^.Cat = Table.catModule THEN
         Scan.NextLex;
```

```
         Check(Scan.lexDot, "'.'");
         IF (Scan.Lex = Scan.lexName)&
            (Strings.Length(X^.Name) +
               Strings.Length(Scan.Name) < Scan.NameLen)
         THEN
            Designator := X^.Name;
            Strings.Append(".", Designator);
            Strings.Append(Scan.Name, Designator);
            Table.Find(Designator, X)
         ELSE
            msg := "имя из модуля ";
            Strings.Append(X^.Name, msg);
            Error.Expected(msg);
         END;
      END;
      IF X^.Cat = Table.catVar THEN
         AssStatement          (* Присваивание *)
      ELSIF (X^.Cat = Table.catStProc) &
            (X^.Typ = Table.typNone)
      THEN
         CallStatement(X^.Val) (* Вызов процедуры *)
      ELSE
         Error.Expected(
            "обозначение переменной или процедуры"
         );
      END
   ELSIF Scan.Lex = Scan.lexIF THEN
      IfStatement
   ELSIF Scan.Lex = Scan.lexWHILE THEN
      WhileStatement
   END;
END Statement;

(* Оператор {";" Оператор} *)
PROCEDURE StatSeq;
BEGIN
   Statement;    (* Оператор *)
   WHILE Scan.Lex = Scan.lexSemi DO
      Scan.NextLex;
      Statement; (* Оператор *)
   END;
END StatSeq;

PROCEDURE ImportModule;
VAR
   ImpRef: Table.tObj;
BEGIN
   IF Scan.Lex = Scan.lexName THEN
      Table.NewName(Scan.Name, Table.catModule, ImpRef);
```

```
      IF Scan.Name = "In" THEN
         Table.Enter("In.Open",
            Table.catStProc, Table.typNone, spInOpen);
         Table.Enter("In.Int",
            Table.catStProc, Table.typNone, spInInt);
      ELSIF Scan.Name = "Out" THEN
         Table.Enter("Out.Int",
            Table.catStProc, Table.typNone, spOutInt);
         Table.Enter("Out.Ln",
            Table.catStProc, Table.typNone, spOutLn);
      ELSE
         Error.Message("Неизвестный модуль");
      END;
      Scan.NextLex;
   ELSE
      Error.Expected("имя импортируемого модуля");
   END;
END ImportModule;

(* IMPORT Имя { "," Имя } ";" *)
PROCEDURE Import;
BEGIN
   Check(Scan.lexIMPORT, "IMPORT");
   LOOP
      ImportModule; (* Обработка имени импорт. модуля *)
   IF Scan.Lex # Scan.lexComma THEN EXIT END;
      Scan.NextLex;
   END;
   Check(Scan.lexSemi, "';'");
END Import;


(* MODULE Имя ";" {Импорт} ПослОбъявл
   {BEGIN ПослОператоров} END Имя "." *)
PROCEDURE Module;
VAR
   ModRef: Table.tObj; (* Ссылка на имя модуля в табл. *)
   msg   : ARRAY 80 OF CHAR;
BEGIN
   ModRef := NIL;
   Check(Scan.lexMODULE, "MODULE");
   IF Scan.Lex # Scan.lexName THEN
      Error.Expected("имя модуля")
   ELSE (* Имя модуля - в таблицу имен *)
      Table.NewName(Scan.Name, Table.catModule, ModRef);
   END;
   Scan.NextLex;
   Check(Scan.lexSemi, "';'");
   IF Scan.Lex = Scan.lexIMPORT THEN
```

```
      Import;
   END;
   DeclSeq;
   IF Scan.Lex = Scan.lexBEGIN THEN
      Scan.NextLex;
      StatSeq;
   END;
   Check(Scan.lexEND, "END");

   (*Сравнение имени модуля и имени после END*)
      IF Scan.Lex # Scan.lexName THEN
         Error.Expected("имя модуля")
      ELSIF Scan.Name # ModRef^.Name THEN
         msg := "имя модуля '";
         Strings.Append(ModRef^.Name, msg);
         Strings.Append("'", msg);
         Error.Expected(msg)
      ELSE
         Scan.NextLex;
      END;
   IF Scan.Lex # Scan.lexDot THEN
      Error.Expected("'.'");
   END;
   Gen.Cmd(0);            (* Код возврата *)
   Gen.Cmd(OVM.cmStop);   (* Команда останова *)
   Gen.AllocateVariables; (* Размещение переменных *)
END Module;

PROCEDURE Compile*;
BEGIN
   Table.Init;
   Table.OpenScope; (* Блок стандартных имен *)
   Table.Enter("ABS",
      Table.catStProc, Table.typInt, spABS);
   Table.Enter("MAX",
      Table.catStProc, Table.typInt, spMAX);
   Table.Enter("MIN",
      Table.catStProc, Table.typInt, spMIN);
   Table.Enter("DEC",
      Table.catStProc, Table.typNone, spDEC);
   Table.Enter("ODD",
      Table.catStProc, Table.typBool, spODD);
   Table.Enter("HALT",
      Table.catStProc, Table.typNone, spHALT);
   Table.Enter("INC",
      Table.catStProc, Table.typNone, spINC);
   Table.Enter("INTEGER",
      Table.catType, Table.typInt, 0);
   Table.OpenScope;  (* Блок модуля *)
```

```
   Module;
   Table.CloseScope; (* Блок модуля *)
   Table.CloseScope; (* Блок стандартных имен *)
   Out.Ln;
   Out.String("Компиляция завершена");
   Out.Ln;
END Compile;

END Pars.
```

Обратите внимание, что наличие сборщика мусора позволило упростить процедуру `CloseScope`, удаляющую блок из таблицы имен (листинг П3.11).

Листинг П3.11. Модуль для работы с таблицей имен

```
MODULE  Table;
(* Таблица имен *)

IMPORT Scan, Error;

CONST
   (* Категории имён *)
      catConst*   = 1;  catVar*     = 2;
      catType*    = 3;  catStProc*  = 4;
      catModule*  = 5;  catGuard*   = 0;

   (* Типы *)
      typNone* = 0; typInt* = 1; typBool* = 2;

TYPE
   (*Тип указателя на запись таблицы*)
      tObj* = POINTER TO tObjRec;

   tObjRec* = RECORD      (* Тип записи таблицы имен *)
      Name* : Scan.tName; (* Ключ поиска             *)
      Cat*  : INTEGER;    (* Категория имени         *)
      Typ*  : INTEGER;    (* Тип                     *)
      Val*  : INTEGER;    (* Значение                *)
      Prev  : tObj;       (* Указатель на пред. имя  *)
   END;


VAR
   Top    : tObj;  (* Указатель на вершину списка     *)
   Bottom : tObj;  (* Указатель на конец (дно) списка *)
   CurrObj: tObj;  (* Очередной объект таблицы имен   *)
```

```
(* Инициализация таблицы имен *)
PROCEDURE Init*;
BEGIN
   Top := NIL;
END Init;

(* Добавление элемента *)
PROCEDURE Enter*(N: Scan.tName; C, T, V: INTEGER);
VAR
  P : tObj;
BEGIN
   NEW(P); P.Name := N;
   P.Cat := C; P.Typ := T;
   P.Val := V; P.Prev := Top;
   Top := P;
END Enter;

(* Открытие области видимости (блока) *)
PROCEDURE OpenScope*;
BEGIN
   Enter("", catGuard, 0, 0);
   IF Top.Prev = NIL THEN
      Bottom := Top;
   END;
END OpenScope;

(* Закрытие области видимости (блока) *)
PROCEDURE CloseScope*;
BEGIN
   WHILE Top.Cat # catGuard DO
      Top := Top.Prev;
   END;
   Top := Top.Prev;
END CloseScope;

(* Занесение нового имени *)
PROCEDURE NewName*
   (N: Scan.tName; C: INTEGER; VAR Obj: tObj);
BEGIN
   Obj := Top;
   WHILE (Obj.Cat # catGuard) & (Obj.Name # N) DO
      Obj := Obj.Prev;
   END;
   IF Obj.Cat = catGuard THEN
      NEW(Obj); Obj.Name := N;
      Obj.Cat := C; Obj.Val := 0;
      Obj.Prev := Top; Top := Obj;
   ELSE
      Error.Message("Повторное объявление имени");
```

```
   END
END NewName;

(* Поиск имени *)
PROCEDURE Find*(Name: Scan.tName; VAR Obj: tObj);
BEGIN
   Bottom.Name := Name;
   Obj := Top;
   WHILE Obj.Name # Name DO
      Obj := Obj.Prev;
   END;
   IF Obj=Bottom THEN
      Error.Message("Необъявленное имя");
   END
END Find;

(* Поиск следующей переменной *)
PROCEDURE NextVar*(VAR VRef : tObj);
BEGIN
   WHILE (CurrObj # Bottom) & (CurrObj.Cat # catVar) DO
      CurrObj := CurrObj.Prev;
   END;
   IF CurrObj = Bottom THEN
      VRef := NIL
   ELSE
      VRef := CurrObj;
      CurrObj := CurrObj.Prev;
   END
END NextVar;

(* Поиск первой переменной *)
PROCEDURE FirstVar*(VAR VRef : tObj);
BEGIN
   CurrObj := Top;
   NextVar(VRef);
END FirstVar;

END Table.
```

При реализации основного цикла виртуальной машины использован оператор **LOOP**. Команда STOP обрабатывается наравне с другими. Это уменьшает число проверок, выполняемых в каждом цикле.

Листинг П3.12. Виртуальная О-машина

```
MODULE  OVM;
(* Виртуальная машина *)

IMPORT In, Out, Utils;

CONST
   MemSize = 8*1024;

   cmStop*   = -1;

   cmAdd*    = -2;
   cmSub*    = -3;
   cmMult*   = -4;
   cmDiv*    = -5;
   cmMod*    = -6;
   cmNeg*    = -7;

   cmLoad*   = -8;
   cmSave*   = -9;

   cmDup*    = -10;
   cmDrop*   = -11;
   cmSwap*   = -12;
   cmOver*   = -13;

   cmGOTO*   = -14;
   cmIfEQ*   = -15;
   cmIfNE*   = -16;
   cmIfLE*   = -17;
   cmIfLT*   = -18;
   cmIfGE*   = -19;
   cmIfGT*   = -20;

   cmIn*     = -21;
   cmOut*    = -22;
   cmOutLn*  = -23;

VAR
   M*: ARRAY MemSize OF INTEGER;

PROCEDURE Run*;
VAR
   PC    : INTEGER;
   SP    : INTEGER;
   Cmd   : INTEGER;
   Buf   : INTEGER;
```

```
BEGIN
   PC := 0;
   SP := MemSize;
   LOOP
      Cmd := M[PC];
      INC(PC);
      IF Cmd >= 0 THEN
         DEC(SP);
         M[SP] := Cmd;
      ELSE
         CASE Cmd OF
         |cmAdd:
            INC(SP); M[SP] := M[SP] + M[SP-1];
         |cmSub:
            INC(SP); M[SP] := M[SP] - M[SP-1];
         |cmMult:
            INC(SP); M[SP] := M[SP]*M[SP-1];
         |cmDiv:
            INC(SP); M[SP] := M[SP] DIV M[SP-1];
         |cmMod:
            INC(SP); M[SP] := M[SP] MOD M[SP-1];
         |cmNeg:
            M[SP] := -M[SP];
         |cmLoad:
            M[SP] := M[M[SP]];
         |cmSave:
            M[M[SP+1]] := M[SP]; INC(SP, 2);
         |cmDup:
            DEC(SP); M[SP] := M[SP+1];
         |cmDrop:
            INC(SP);
         |cmSwap:
            Buf := M[SP]; M[SP] := M[SP+1];
            M[SP+1] := Buf;
         |cmOver:
            DEC(SP); M[SP] := M[SP+2];
         |cmGOTO:
            PC := M[SP]; INC(SP);
         |cmIfEQ:
            IF M[SP+2] = M[SP+1] THEN
               PC := M[SP];
            END;
            INC(SP, 3);
         |cmIfNE:
            IF M[SP+2] # M[SP+1] THEN
               PC := M[SP];
            END;
            INC(SP, 3);
         |cmIfLE:
```

```
            IF M[SP+2] <= M[SP+1] THEN
               PC := M[SP];
            END;
            INC(SP, 3);
         |cmIfLT:
            IF M[SP+2] < M[SP+1] THEN
               PC := M[SP];
            END;
            INC(SP, 3);
         |cmIfGE:
            IF M[SP+2] >= M[SP+1] THEN
               PC := M[SP];
            END;
            INC(SP, 3);
         |cmIfGT:
            IF M[SP+2] > M[SP+1] THEN
               PC := M[SP];
            END;
            INC(SP, 3);
         |cmIn:
            DEC(SP);
            Out.Char("?");
            In.Int( M[SP] );
            Utils.ReadLn;
         |cmOut:
            Out.Int(M[SP+1], M[SP]);
            INC(SP, 2);
         |cmOutLn:
            Out.Ln;
         |cmStop:
            EXIT;
         ELSE
            Out.String("Недопустимый код операции");
            Out.Ln;
            EXIT;
         END;
      END;
   END;
   Out.Ln;
   IF SP<MemSize THEN
      Out.String("Код возврата ");
      Out.Int(M[SP], 0);  Out.Ln;
   END;
   Out.String("Нажмите ВВОД");
   Utils.ReadLn;
END Run;

END OVM.
```

Листинг П3.13. Модуль генератора кода

```
MODULE  Gen;
(* Генератор кода *)

IMPORT Scan, Table,  OVM, Error, Strings, Out;

VAR
   PC* : INTEGER;

PROCEDURE Init*;
BEGIN
   PC := 0;
END Init;

PROCEDURE Cmd*(Cmd: INTEGER);
BEGIN
   OVM.M[PC] := Cmd;
   PC := PC+1;
END Cmd;

PROCEDURE Fixup*(A: INTEGER);
VAR
   temp: INTEGER;
BEGIN
   WHILE A > 0 DO
      temp := OVM.M[A-2];
      OVM.M[A-2] := PC;
      A := temp;
   END;
END Fixup;

PROCEDURE Abs*;
BEGIN
   Cmd(OVM.cmDup);
   Cmd(0);
   Cmd(PC+3);
   Cmd(OVM.cmIfGE);
   Cmd(OVM.cmNeg);
END Abs;

PROCEDURE Min*;
BEGIN
   Cmd(MAX(INTEGER));
   Cmd(OVM.cmNeg);
   Cmd(1);
   Cmd(OVM.cmSub);
END Min;
```

```
PROCEDURE Odd*;
BEGIN
   Cmd(2);
   Cmd(OVM.cmMod);
   Cmd(0);
   Cmd(0); (* Адрес перехода вперед *)
   Cmd(OVM.cmIfEQ);
END Odd;

PROCEDURE Const*(C: INTEGER);
BEGIN
   Cmd(ABS(C));
   IF C < 0 THEN
      Cmd(OVM.cmNeg);
   END
END Const;

PROCEDURE Comp*(Op : INTEGER);
BEGIN
   Cmd(0); (* Переход вперед *)
   CASE Op OF
   |Scan.lexEQ : Cmd(OVM.cmIfNE);
   |Scan.lexNE : Cmd(OVM.cmIfEQ);
   |Scan.lexLE : Cmd(OVM.cmIfGT);
   |Scan.lexLT : Cmd(OVM.cmIfGE);
   |Scan.lexGE : Cmd(OVM.cmIfLT);
   |Scan.lexGT : Cmd(OVM.cmIfLE);
   END;
END Comp;

PROCEDURE Addr*(X: Table.tObj);
BEGIN
   Cmd(X^.Val); (* В текущую ячейку адрес предыдущей+2 *)
   X^.Val := PC+1; (* Адрес+2 = PC+1 *)
END Addr;

PROCEDURE AllocateVariables*;
VAR
   VRef: Table.tObj;  (* Ссылка на перем. в табл. имен *)
   msg : ARRAY 80 OF CHAR;
BEGIN
   Table.FirstVar(VRef);    (* Найти первую переменную *)
   WHILE VRef # NIL DO
      IF VRef^.Val = 0 THEN
         msg := "Переменная ";
         Strings.Append(VRef^.Name, msg);
         Strings.Append(" не используется", msg);
         Error.Warning(msg)
      ELSE
```

```
         Fixup(VRef^.Val);        (* Адресная привязка *)
         PC := PC + 1;
       END;
       Table.NextVar(VRef);(* Найти следующую переменную*)
    END;
END AllocateVariables;

END Gen.
```

ПРИЛОЖЕНИЕ 4. ТЕКСТ КОМПИЛЯТОРА ЯЗЫКА «О» НА СИ/СИ++

Приведенная здесь программа написана на ANSI Си — языке, определенном стандартом ISO/IEC 9899:1990. Используются только библиотеки, предусмотренные этим стандартом. Программа может быть откомпилирована в любой системе, поддерживающей указанный стандарт. Одновременно приведенный текст является и правильной программой на Си++.

Для контроля межмодульных связей[58] на стадии компиляции в Си используется механизм заголовочных файлов, которые включаются во входной поток компилятора с помощью директив `#include`. Сами заголовочные файлы также могут содержать директивы `#include` (см., например, листинг П4.12). Для избежания повторного включения заголовочных файлов при их определении использованы директивы условной компиляции.

Традиция публикации программ на Си и происходящих от него языках не требует шрифтового выделения ключевых слов, хотя большинство современных систем программирования делают это. В приведенных листингах служебные слова выделены жирным шрифтом, что должно облегчить чтение программы.

Листинг П4.1. Основной файл компилятора

```
/* Компилятор языка O (o.c) */

#include <string.h>
#include <stdio.h>

#include "text.h"
#include "scan.h"
#include "ovm.h"
```

[58] Официального термина «модуль» в языке Си нет. Здесь это слово используется в общем смысле и относится к одному файлу, который и является в Си единицей компиляции.

```
#include "scan.h"
#include "gen.h"
#include "location.h"
#include "pars.h"

void Init(void) {
   ResetText();
   if( ResetError )
      Error(Message);
   InitScan();
   InitGen();
}

void Done(void) {
   CloseText();
}

int main(int argc, char *argv[]) {
   puts("\nКомпилятор языка O");
   if( argc <= 1 )
      Path = NULL;
   else
      Path = argv[1];
   Init();    /* Инициализация */
   Compile(); /* Компиляция    */
   Run();     /* Выполнение    */
   Done();    /* Завершение    */
   return 0;
}
```

Листинг П4.2. Заголовочный файл драйвера текста

```
/* Драйвер исходного текста (text.h) */
#ifndef TEXT
#define TEXT

#define  chSpace ' '    /* Пробел       */
#define  chTab   '\t'   /* Табуляция    */
#define  chEOL   '\n'   /* Конец строки */
#define  chEOT   '\0'   /* Конец текста */

extern char ResetError;
extern char* Message;
extern int Ch;

void ResetText(void);
void CloseText(void);
void NextCh(void);
```

```
#endif
```

Листинг П4.3. Драйвер исходного текста

```
/* Драйвер исходного текста (text.c) */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "text.h"
#include "location.h"

#define TABSIZE   3
#define TRUE      1
#define FALSE     0

char ResetError = TRUE;
char* Message = "Файл не открыт";
int Ch = chEOT;

static FILE *f;

void NextCh() {
   if( (Ch = fgetc(f)) == EOF )
      Ch = chEOT;
   else if( Ch == '\n' ) {
      puts("");
      Line++; Pos = 0; Ch = chEOL;
      }
   else if( Ch == '\r' )
      NextCh();
   else if( Ch != '\t' ) {
      putchar(Ch); Pos++;
      }
   else
      do
         putchar(' ');
      while( ++Pos % TABSIZE );
}

void ResetText() {
   if( Path == NULL ) {
      puts("Формат вызова:\n   O <входной файл>");
      exit(1);
      }
   else if( (f = fopen(Path, "r")) == NULL ){
      ResetError = TRUE;
      Message = "Входной файл не найден";
```

```
      }
   else {
      ResetError = FALSE; Message = "Ok";
      Pos = 0; Line = 1;
      NextCh();
   }
}

void CloseText() {
   fclose(f);
}
```

Листинг П4.4. Заголовочный файл обработчика ошибок

```
/* Обработка ошибок (error.h) */
#ifndef ERROR
#define ERROR

void Error(char* Msg);
void Expected(char* Msg);
void Warning(char* Msg);

#endif
```

Листинг П4.5. Файл обработчика ошибок

```
/* Обработка ошибок (error.c) */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "location.h"
#include "text.h"

void Error(char* Msg) {
   int i, ELine = Line;

   while( Ch != chEOL && Ch != chEOT ) NextCh();
   if( Ch == chEOT ) puts("");
   for(i = 1; i < LexPos; i++) putchar(' ');
   printf("^\n(Строка %i) Ошибка: %s\n\nНажмите ВВОД",
      ELine, Msg);
   while( getchar() != '\n' );
   exit(EXIT_SUCCESS);
}

void Expected(char* Msg) {
   char message[80];
   strcpy(message, "Ожидается ");
```

```
   Error(strcat(message,  Msg));
}

void Warning(char* Msg) {
   printf("\nПредупреждение: %s\n", Msg);
}
```

Листинг П4.6. Заголовочный файл location.h

```
/* Текущая позиция в исходном тексте (location.h) */
#ifndef LOCATION
#define LOCATION

extern int Line;     /* Номер строки           */
extern int Pos;      /* Номер символа в строке */
extern int LexPos;   /* Позиция начала лексемы */
extern char* Path;   /* Путь к файлу           */

#endif
```

Листинг П4.7. Вспомогательный модуль location.c

```
/* Текущая позиция в исходном тексте (location.c) */
#include <stdlib.h>

int Line = 0;      /* Номер строки           */
int Pos = 0;       /* Номер символа в строке */
int LexPos = 0;    /* Позиция начала лексемы */
char* Path = NULL; /* Путь к файлу           */
```

Листинг П4.8. Заголовочный файл лексического анализатора

```
/* Сканер (scan.h) */
#ifndef SCAN
#define SCAN

#define NAMELEN 31 /* Наибольшая длина имени */

typedef char tName[NAMELEN+1];

typedef enum {
   lexNone, lexName, lexNum,
   lexMODULE, lexIMPORT, lexBEGIN, lexEND,
   lexCONST,  lexVAR,  lexWHILE, lexDO,
   lexIF, lexTHEN, lexELSIF, lexELSE,
   lexMult, lexDIV, lexMOD, lexPlus, lexMinus,
   lexEQ, lexNE, lexLT, lexLE, lexGT, lexGE,
   lexDot, lexComma, lexColon, lexSemi, lexAss,
   lexLPar, lexRPar,
   lexEOT
```

```
} tLex;

extern tLex Lex;     /* Текущая лексема             */
extern char  Name[]; /* Строковое значение имени    */
extern int   Num;    /* Значение числовых литералов */
extern int   LexPos; /* Позиция начала лексемы      */

void InitScan(void);
void NextLex(void);
void NextLex(void);
void InitScan(void);

#endif
```

Листинг П4.9. Файл лексического анализатора

```
/* Лексический анализатор (scan.c) */

#include <string.h>
#include <ctype.h>
#include <limits.h>

#include "scan.h"
#include "text.h"
#include "error.h"
#include "location.h"

#define KWNUM 34

tLex  Lex;  /* Текущая лексема             */
tName Name; /* Строковое значение имени    */
int   Num;  /* Значение числовых литералов */

typedef char tKeyWord[9]; /* Длина слова PROCEDURE */

static int nkw = 0;

static struct {
   tKeyWord Word;
   tLex Lex;
} KWTable[KWNUM];

static void EnterKW(tKeyWord Name, tLex Lex) {
   strcpy(KWTable[nkw].Word, Name);
   KWTable[nkw++].Lex = Lex;
}

static tLex TestKW(void) {
   int i;
```

```
   for( i = nkw - 1; i >= 0; i-- )
      if( strcmp(Name, KWTable[i].Word) == 0 )
         return KWTable[i].Lex;
   return lexName;
}

static void Ident(void) {
   int i = 0;
   do {
      if ( i < NAMELEN )
         Name[i++] = Ch;
      else
         Error("Слишком длинное имя");
      NextCh();
   } while( isalnum(Ch) );
   Name[i] = '\0';
   Lex = TestKW(); /* Проверка на ключевое слово */
}

static void Number(void) {
   Lex = lexNum;
   Num = 0;
   do {
      int d = Ch - '0';
      if( (INT_MAX - d)/10 >= Num )
         Num = 10*Num + d;
      else
         Error("Слишком большое число");
      NextCh();
   } while( isdigit(Ch) );
}

/*
static void Comment(void) {
   NextCh();
   do {
      while( Ch != '*' && Ch != chEOT )
         if( Ch == '(' ) {
            NextCh();
            if( Ch == '*' )
               Comment();
            }
         else
            NextCh();
      if ( Ch == '*' )
         NextCh();
   } while( Ch != ')' && Ch != chEOT );
   if ( Ch == ')' )
      NextCh();
```

```
   else {
      LexPos = Pos;
      Error("Не закончен комментарий");
   }
}
*/

static void Comment(void) {
   int Level = 1;
   NextCh();
   do
      if( Ch == '*' ) {
         NextCh();
         if( Ch == ')' )
            { Level--; NextCh(); }
         }
      else if( Ch == '(' ) {
         NextCh();
         if( Ch == '*' )
            { Level++; NextCh(); }
         }
      else /* if( Ch <> chEOT ) */
         NextCh();
   while( Level && Ch != chEOT );
   if( Level ) {
      LexPos = Pos;
      Error("Не закончен комментарий");
   }
}

void NextLex(void) {
   while( Ch == chSpace || Ch == chTab || Ch == chEOL )
      NextCh();
   LexPos = Pos;
   if( isalpha(Ch) )
      Ident();
   else if( isdigit(Ch) )
      Number();
   else
      switch( Ch ) {
      case ';':
         NextCh(); Lex = lexSemi;
         break;
      case ':':
         NextCh();
         if ( Ch == '=' )
            { NextCh(); Lex = lexAss; }
         else
            Lex = lexColon;
```

```
      break;
   case '.':
      NextCh(); Lex = lexDot;
      break;
   case ',':
      NextCh(); Lex = lexComma;
      break;
   case '=':
      NextCh(); Lex = lexEQ;
      break;
   case '#':
      NextCh(); Lex = lexNE;
      break;
   case '<':
      NextCh();
      if ( Ch == '=' )
         { NextCh(); Lex = lexLE; }
      else
         Lex = lexLT;
      break;
   case '>':
      NextCh();
      if ( Ch == '=' )
         { NextCh(); Lex = lexGE; }
      else
         Lex = lexGT;
      break;
   case '(':
      NextCh();
      if ( Ch == '*' )
         { Comment();   NextLex(); }
      else
         Lex = lexLPar;
      break;
   case ')':
      NextCh(); Lex = lexRPar;
      break;
   case '+':
      NextCh(); Lex = lexPlus;
      break;
   case '-':
      NextCh(); Lex = lexMinus;
      break;
   case '*':
      NextCh(); Lex = lexMult;
      break;
   case chEOT:
      Lex = lexEOT;
      break;
```

```
      default:
         Error("Недопустимый символ");
      }
}

void InitScan(void) {
   EnterKW("ARRAY",     lexNone);
   EnterKW("BY",        lexNone);
   EnterKW("BEGIN",     lexBEGIN);
   EnterKW("CASE",      lexNone);
   EnterKW("CONST",     lexCONST);
   EnterKW("DIV",       lexDIV);
   EnterKW("DO",        lexDO);
   EnterKW("ELSE",      lexELSE);
   EnterKW("ELSIF",     lexELSIF);
   EnterKW("END",       lexEND);
   EnterKW("EXIT",      lexNone);
   EnterKW("FOR",       lexNone);
   EnterKW("IF",        lexIF);
   EnterKW("IMPORT",    lexIMPORT);
   EnterKW("IN",        lexNone);
   EnterKW("IS",        lexNone);
   EnterKW("LOOP",      lexNone);
   EnterKW("MOD",       lexMOD);
   EnterKW("MODULE",    lexMODULE);
   EnterKW("NIL",       lexNone);
   EnterKW("OF",        lexNone);
   EnterKW("OR",        lexNone);
   EnterKW("POINTER",   lexNone);
   EnterKW("PROCEDURE", lexNone);
   EnterKW("RECORD",    lexNone);
   EnterKW("REPEAT",    lexNone);
   EnterKW("RETURN",    lexNone);
   EnterKW("THEN",      lexTHEN);
   EnterKW("TO",        lexNone);
   EnterKW("TYPE",      lexNone);
   EnterKW("UNTIL",     lexNone);
   EnterKW("VAR",       lexVAR);
   EnterKW("WHILE",     lexWHILE);
   EnterKW("WITH",      lexNone);

   NextLex();
}
```

Листинг П4.10. Заголовочный файл синтаксического и контекстного анализатора

```
/* Распознаватель (pars.h) */
#ifndef PARS
```

```
#define PARS

void Compile(void);

#endif
```

Листинг П4.11. Файл синтаксического и контекстного анализатора

```
/* Распознаватель (pars.c) */

#include <limits.h>
#include <string.h>
#include <stdio.h>

#include "scan.h"
#include "table.h"
#include "gen.h"
#include "error.h"
#include "ovm.h"

#define spABS    1
#define spMAX    2
#define spMIN    3
#define spDEC    4
#define spODD    5
#define spHALT   6
#define spINC    7
#define spInOpen 8
#define spInInt  9
#define spOutInt 10
#define spOutLn  11

static void StatSeq(void);
static tType Expression(void);

static void Check(tLex L, char* M) {
   if( Lex != L )
      Expected(M);
   else
      NextLex();
}

/* ["+" | "-"] (Число | Имя) */
static int ConstExpr(void) {
   int v;
   tObj* X;
   tLex Op;

   Op = lexPlus;
```

```
   if( Lex == lexPlus || Lex == lexMinus ) {
      Op = Lex;
      NextLex();
   }
   if( Lex == lexNum ) {
      v = Num;
      NextLex();
      }
   else if( Lex == lexName ) {
      X = Find(Name);
      if( X->Cat == catGuard )
         Error("Нельзя определять константу через себя");
      else if( X->Cat != catConst )
         Expected( "имя константы" );
      else {
         v = X->Val;
         NextLex();
      }
      }
   else
      Expected( "константное выражение" );
   if( Op == lexMinus )
      return -v;
   return v;
}

/* Имя "=" КонстВыраж */
static void ConstDecl(void) {
   tObj* ConstRef; /* Ссылка на имя в таблице */
   ConstRef = NewName(Name, catGuard);
   NextLex();
   Check(lexEQ, "\"=\"");
   ConstRef->Val = ConstExpr();
   ConstRef->Typ = typInt;/* Констант других типов нет */
   ConstRef->Cat = catConst;
}

static void ParseType(void) {
   tObj* TypeRef;
   if( Lex != lexName )
      Expected("имя");
   else {
      TypeRef = Find(Name);
      if( TypeRef->Cat != catType )
         Expected("имя типа");
      else if( TypeRef->Typ != typInt )
         Expected("целый тип");
      NextLex();
   }
```

```
}

/* Имя {"," Имя} ":" Тип */
static void VarDecl(void) {
   tObj* NameRef;

   if( Lex != lexName )
      Expected("имя");
   else {
      NameRef = NewName(Name, catVar);
      NameRef->Typ = typInt;
      NextLex();
   }
   while( Lex == lexComma ) {
      NextLex();
      if( Lex != lexName )
         Expected("имя");
      else {
         NameRef = NewName(Name, catVar );
         NameRef->Typ = typInt;
         NextLex();
      }
   }
   Check(lexColon, "\":\"");
   ParseType();
}

/* {CONST {ОбъявлКонст ";"}
    |VAR {ОбъявлПерем ";"} } */
static void DeclSeq(void) {
   while( Lex == lexCONST || Lex == lexVAR ) {
      if( Lex == lexCONST ) {
         NextLex();
         while( Lex == lexName ) {
            ConstDecl(); /* Объявление константы */
            Check(lexSemi, "\";\"");
         }
         }
      else {
         NextLex(); /* VAR */
         while( Lex == lexName ) {
            VarDecl();   /* Объявление переменных */
            Check(lexSemi, "\";\"");
         }
      }
   }
}

static void IntExpression(void) {
```

```
   if( Expression() != typInt )
      Expected("выражение целого типа");
}

static tType StFunc(int F) {
   switch( F ) {
   case spABS:
      IntExpression();
      GenAbs();
      return typInt;
   case spMAX:
      ParseType();
      Gen(INT_MAX);
      return typInt;
   case  spMIN:
      ParseType();
      GenMin();
      return typInt;
   case spODD:
      IntExpression();
      GenOdd();
      return typBool;
   }
   return typNone; /* Чтоб не было предупреждений */
}

static tType Factor(void) {
   tObj* X;
   tType T;

   if( Lex == lexName ) {
      if( (X = Find(Name))->Cat == catVar ) {
         GenAddr(X);    /* Адрес переменной */
         Gen(cmLoad);
         NextLex();
         return X->Typ;
         }
      else if( X->Cat == catConst ) {
         GenConst(X->Val);
         NextLex();
         return X->Typ;
         }
      else if( X->Cat == catStProc && X->Typ != typNone )
         {
         NextLex();
         Check(lexLPar, "\"(\"");
         T = StFunc(X->Val);
         Check(lexRPar, "\")\"");
         }
```

```
      else
         Expected(
         "переменная, константа или процедура-функция"
         );
      }
   else if( Lex == lexNum ) {
      GenConst(Num);
      NextLex();
      return typInt;
      }
   else if( Lex == lexLPar ) {
      NextLex();
      T = Expression();
      Check(lexRPar, "\")\"");
      }
   else
      Expected("имя, число или \"(\"");
   return T;
}

static tType Term(void) {
   tLex Op;
   tType T = Factor();
   if( Lex == lexMult || Lex == lexDIV || Lex == lexMOD )
   {
      if( T != typInt )
         Error("Несоответствие операции типу операнда");
      do {
         Op = Lex;
         NextLex();
         if( (T = Factor()) != typInt )
            Expected("выражение целого типа");
         switch(Op) {
         case lexMult: Gen(cmMult); break;
         case lexDIV:  Gen(cmDiv);  break;
         case lexMOD:  Gen(cmMod);  break;
         }
      } while( Lex == lexMult ||
               Lex == lexDIV ||
               Lex == lexMOD );
   }
   return T;
}

/* ["+"|"-"] Слагаемое {ОперСлож Слагаемое} */
static tType SimpleExpr(void) {
   tType T;
   tLex Op;
```

```
   if( Lex == lexPlus || Lex == lexMinus ) {
      Op = Lex;
      NextLex();
      if( (T = Term()) != typInt )
         Expected("выражение целого типа");
      if( Op == lexMinus )
         Gen(cmNeg);
      }
   else
      T = Term();
   if( Lex == lexPlus || Lex == lexMinus ) {
      if( T != typInt )
         Error("Несоответствие операции типу операнда");
      do {
         Op = Lex;
         NextLex();
         if( (T = Term()) != typInt )
            Expected("выражение целого типа");
         switch(Op) {
         case lexPlus:  Gen(cmAdd); break;
         case lexMinus: Gen(cmSub); break;
         }
      } while( Lex == lexPlus || Lex == lexMinus );
   }
   return T;
}

/* ПростоеВыраж [Отношение ПростоеВыраж] */
static tType Expression(void) {
   tLex  Op;
   tType T = SimpleExpr();
   if( Lex == lexEQ || Lex == lexNE || Lex == lexGT ||
       Lex == lexGE || Lex == lexLT || Lex == lexLE )
   {
      Op = Lex;
      if( T != typInt )
         Error("Несоответствие операции типу операнда");
      NextLex();
      if( (T = SimpleExpr()) != typInt )
         Expected("выражение целого типа");
      GenComp(Op);   /* Генерация условного перехода */
      T = typBool;
   } /* иначе тип равен типу первого простого выражения*/
   return T;
}

/* Переменная = Имя */
static void Variable(void) {
   tObj* X;
```

```
   if( Lex != lexName )
      Expected("имя");
   else {
      if( (X = Find(Name))->Cat != catVar )
         Expected("имя переменной");
      GenAddr(X);
      NextLex();
   }
}

static void StProc(int P) {
   switch( P ) {
   case spDEC:
      Variable();
      Gen(cmDup);
      Gen(cmLoad);
      if( Lex == lexComma ) {
         NextLex();
         IntExpression();
         }
      else
         Gen(1);
      Gen(cmSub);
      Gen(cmSave);
      return;
   case spINC:
      Variable();
      Gen(cmDup);
      Gen(cmLoad);
      if( Lex == lexComma ) {
         NextLex();
         IntExpression();
         }
      else
         Gen(1);
      Gen(cmAdd);
      Gen(cmSave);
      return;
   case spInOpen:
      /* Пусто */;
      return;
   case spInInt:
      Variable();
      Gen(cmIn);
      Gen(cmSave);
      return;
   case spOutInt:
      IntExpression();
```

```
      Check(lexComma , "\",\"");
      IntExpression();
      Gen(cmOut);
      return;
   case spOutLn:
      Gen(cmOutLn);
      return;
   case spHALT:
      GenConst(ConstExpr());
      Gen(cmStop);
      return;
   }
}

static void BoolExpression(void) {
   if( Expression() != typBool )
      Expected("логическое выражение");
}

/* Переменная "=" Выраж */
static void AssStatement(void) {
   Variable();
   if( Lex == lexAss ) {
      NextLex();
      IntExpression();
      Gen(cmSave);
      }
   else
      Expected("\":=\"");
}

/* Имя ["(" { Выраж | Переменная } ")"] */
static void CallStatement(int sp) {
   Check(lexName, "имя процедуры");
   if( Lex == lexLPar ) {
      NextLex();
      StProc(sp);
      Check( lexRPar, "\")\"" );
      }
   else if( sp == spOutLn || sp == spInOpen )
      StProc(sp);
   else
      Expected("\"(\"");
}

static void IfStatement(void) {
   int CondPC;
   int LastGOTO;
```

```
   Check(lexIF, "IF");
   LastGOTO = 0;    /* Предыдущего перехода нет        */
   BoolExpression();
   CondPC = PC;     /* Запомн. положение усл. перехода */
   Check(lexTHEN, "THEN");
   StatSeq();
   while( Lex == lexELSIF ) {
      Gen(LastGOTO); /* Фиктивный адрес, указывающий   */
      Gen(cmGOTO);   /* на место предыдущего перехода. */
      LastGOTO = PC; /* Запомнить место GOTO           */
      NextLex();
      Fixup(CondPC); /* Зафикс. адрес условн. перехода */
      BoolExpression();
      CondPC = PC;   /* Запомн. полож. усл. перехода   */
      Check(lexTHEN, "THEN");
      StatSeq();
   }
   if( Lex == lexELSE ) {
      Gen(LastGOTO); /* Фиктивный адрес, указывающий  */
      Gen(cmGOTO);   /* на место предыдущего перехода */
      LastGOTO = PC; /* Запомн. место последнего GOTO */
      NextLex();
      Fixup(CondPC); /* Зафикс. адрес усл. перехода   */
      StatSeq();
      }
   else
      Fixup(CondPC); /* Если ELSE отсутствует         */
   Check( lexEND, "END" );
   Fixup(LastGOTO);  /* Направить сюда все GOTO       */
}

static void WhileStatement(void) {
   int CondPC;
   int WhilePC = PC;
   Check(lexWHILE, "WHILE");
   BoolExpression();
   CondPC = PC;
   Check(lexDO, "DO");
   StatSeq();
   Check(lexEND, "END");
   Gen(WhilePC);
   Gen(cmGOTO);
   Fixup(CondPC);
}

static void Statement(void) {
   tObj* X;
   char designator[NAMELEN+1];
   char msg[80];
```

```
   if( Lex == lexName ) {
      if( (X=Find(Name))->Cat == catModule ) {
         NextLex();
         Check(lexDot, "\".\"");
         if( Lex == lexName &&
               strlen(X->Name) +
                  strlen(Name) <= NAMELEN )
         {
            strcpy(designator, X->Name);
            strcat(designator, ".");
            X = Find(strcat(designator, Name));
            }
         else {
            strcpy(msg, "имя из модуля ");
            Expected(strcat(msg, X->Name));
         }
      }
      if( X->Cat == catVar )
         AssStatement();        /* Присваивание    */
      else if( X->Cat == catStProc && X->Typ == typNone )
         CallStatement(X->Val); /* Вызов процедуры */
      else
         Expected(
            "обозначение переменной или процедуры"
         );
      }
   else if( Lex == lexIF )
      IfStatement();
   else if( Lex == lexWHILE )
      WhileStatement();
   /* иначе пустой оператор */
}

/* Оператор {";" Оператор} */
static void StatSeq(void) {
   Statement();    /* Оператор */
   while( Lex == lexSemi ) {
      NextLex();
      Statement(); /* Оператор */
   }
}

static void ImportModule(void) {
   if( Lex == lexName ) {
      NewName(Name, catModule);
      if( strcmp(Name, "In") ) {
         Enter("In.Open", catStProc, typNone, spInOpen);
         Enter("In.Int", catStProc, typNone, spInInt);
```

```
         }
      else if( strcmp(Name, "Out") ) {
         Enter("Out.Int", catStProc, typNone, spOutInt);
         Enter("Out.Ln", catStProc, typNone, spOutLn);
         }
      else
         Error("Неизвестный модуль");
      NextLex();
      }
   else
      Expected("имя импортируемого модуля");
}

/* IMPORT Имя { "," Имя } ";" */
static void Import(void) {
   Check(lexIMPORT, "IMPORT");
   ImportModule();
   while( Lex == lexComma ) {
      NextLex();
      ImportModule();
   }
   Check(lexSemi, "\";\"");
}

/* MODULE Имя ";" [Импорт] ПослОбъявл
   [BEGIN ПослОператоров] END Имя "." */
static void Module(void) {
   tObj* ModRef; /* Ссылка на имя модуля в таблице */
   char msg[80];

   Check(lexMODULE, "MODULE");
   if( Lex != lexName )
      Expected("имя модуля");
   else /* Имя модуля - в таблицу имен */
      ModRef = NewName(Name, catModule);
   NextLex();
   Check(lexSemi, "\";\"");
   if( Lex == lexIMPORT )
      Import();
   DeclSeq();
   if( Lex == lexBEGIN ) {
      NextLex();
      StatSeq();
   }
   Check(lexEND, "END");

   /* Сравнение имени модуля и имени после END */
      if( Lex != lexName )
         Expected("имя модуля");
```

```
      else if( strcmp(Name, ModRef->Name) ) {
         strcpy(msg, "имя модуля \"");
         strcat(msg, ModRef->Name);
         Expected(strcat(msg, "\""));
         }
      else
         NextLex();
   if( Lex != lexDot )
      Expected("\".\"");
   Gen(0);              /* Код возврата */
   Gen(cmStop);         /* Команда останова */
   AllocateVariables(); /* Размещение переменных */
}

void Compile(void) {
   InitNameTable();
   OpenScope(); /* Блок стандартных имен */
   Enter("ABS", catStProc, typInt, spABS);
   Enter("MAX", catStProc, typInt, spMAX);
   Enter("MIN", catStProc, typInt, spMIN);
   Enter("DEC", catStProc, typNone, spDEC);
   Enter("ODD", catStProc, typBool, spODD);
   Enter("HALT", catStProc, typNone, spHALT);
   Enter("INC", catStProc, typNone, spINC);
   Enter("INTEGER", catType, typInt, 0);
   OpenScope();  /* Блок модуля */
   Module();
   CloseScope(); /* Блок модуля */
   CloseScope(); /* Блок стандартных имен */
   puts("\nКомпиляция завершена");
}
```

Листинг П4.12. Заголовочный файл модуля таблицы имен

```
/* Таблица имен (table.h) */
#ifndef TABLE
#define TABLE

#include "scan.h"

/* Категории имён */
typedef enum {
   catConst, catVar, catType,
   catStProc, catModule, catGuard
}tCat ;

/* Типы */
typedef enum {
   typNone, typInt, typBool
```

```
}tType ;

typedef struct tObjDesc{   /* Тип записи таблицы имен */
   char Name[NAMELEN+1];   /* Ключ поиска             */
   tCat Cat;               /* Категория имени         */
   tType Typ;              /* Тип                     */
   int Val;                /* Значение                */
   struct tObjDesc* Prev;  /* Указатель на пред. имя  */
} tObj;

/* Инициализация таблицы */
   void InitNameTable(void);
/* Добавление элемента */
   void Enter(char* N, tCat C, tType T, int V);
/* Занесение нового имени */
   tObj* NewName(char* Name, tCat Cat);
/* Поиск имени */
   tObj* Find(char* Name);
/* Открытие области видимости (блока) */
   void OpenScope(void);
/* Закрытие области видимости (блока) */
   void CloseScope(void);
/* Поиск первой переменной */
   tObj* FirstVar(void);
/* Поиск следующей переменной */
   tObj* NextVar(void);

#endif
```

Листинг П4.13. Модуль таблицы имен

```
/* Таблица имен (table.c) */

#include <stdlib.h>
#include <string.h>

#include "table.h"
#include "scan.h"
#include "error.h"

static tObj*   Top;     /* Указатель на вершину списка */
static tObj*   Bottom;  /* Указатель на дно списка     */
static tObj*   CurrObj;

/* Инициализация таблицы имен */
void InitNameTable(void) {
   Top = NULL;
}
```

```
void Enter(char* N, tCat C, tType T, int V) {
   tObj* P = (tObj*) malloc(sizeof(*P));
   strcpy(P->Name, N);
   P->Cat = C;
   P->Typ = T;
   P->Val = V;
   P->Prev = Top;
   Top = P;
}

void OpenScope(void) {
   Enter("", catGuard, typNone, 0);
   if ( Top->Prev == NULL )
      Bottom = Top;
}

void CloseScope(void) {
   tObj* P;

   while( (P = Top)->Cat != catGuard ){
      Top = Top->Prev;
      free(P);
   }
   Top = Top->Prev;
   free(P);
}

tObj* NewName(char* Name, tCat Cat) {
   tObj* Obj = Top;

   while(Obj->Cat != catGuard && strcmp(Obj->Name, Name))
      Obj = Obj->Prev;
   if ( Obj->Cat == catGuard ) {
      Obj = (tObj*) malloc(sizeof(*Obj));
      strcpy(Obj->Name, Name);
      Obj->Cat = Cat;
      Obj->Val = 0;
      Obj->Prev = Top;
      Top = Obj;
      }
   else
      Error("Повторное объявление имени");
   return Obj;
}

tObj* Find(char* Name) {
   tObj* Obj;

   strcpy(Bottom->Name, Name);
```

```
   for( Obj = Top;
         strcmp(Obj->Name, Name);
                     Obj = Obj->Prev );
   if( Obj == Bottom )
      Error("Необъявленное имя");
   return Obj;
}

tObj* FirstVar(void) {
   CurrObj = Top;
   return NextVar();
}

tObj* NextVar(void) {
   tObj* VRef;

   while( CurrObj != Bottom && CurrObj->Cat != catVar )
      CurrObj = CurrObj->Prev;
   if( CurrObj == Bottom )
      return NULL;
   else {
      VRef = CurrObj;
      CurrObj = CurrObj->Prev;
      return VRef;
   }
}
```

Листинг П4.14. Заголовочный файл модуля виртуальной машины

```
/* Виртуальная машина (ovm.h) */
#ifndef OVM
#define OVM

#define MEMSIZE  8*1024

#define  cmStop    -1

#define  cmAdd     -2
#define  cmSub     -3
#define  cmMult    -4
#define  cmDiv     -5
#define  cmMod     -6
#define  cmNeg     -7

#define  cmLoad    -8
#define  cmSave    -9

#define  cmDup     -10
#define  cmDrop    -11
```

```
#define  cmSwap    -12
#define  cmOver    -13

#define  cmGOTO    -14
#define  cmIfEQ    -15
#define  cmIfNE    -16
#define  cmIfLE    -17
#define  cmIfLT    -18
#define  cmIfGE    -19
#define  cmIfGT    -20

#define  cmIn      -21
#define  cmOut     -22
#define  cmOutLn   -23

extern int M[MEMSIZE];

void Run(void);

#endif
```

Листинг П4.15. Файл виртуальной машины

```
/* Виртуальная машина (ovm.c) */

#include <stdio.h>
#include "ovm.h"

#define readln while( getchar() != '\n' )

int M[MEMSIZE];

void Run(void) {
   register int PC = 0;
   register int SP = MEMSIZE;
   register int Cmd;
   int Buf;

   while( (Cmd = M[PC++]) != cmStop )
      if ( Cmd >= 0 )
         M[--SP] = Cmd;
      else
         switch( Cmd ) {
         case cmAdd:
            SP++; M[SP] += M[SP-1];
            break;
         case cmSub:
            SP++; M[SP] -= M[SP-1];
            break;
```

```
case cmMult:
   SP++; M[SP] *= M[SP-1];
   break;
case cmDiv:
   SP++; M[SP] /= M[SP-1];
   break;
case cmMod:
   SP++; M[SP] %= M[SP-1];
   break;
case cmNeg:
   M[SP] = -M[SP];
   break;
case cmLoad:
   M[SP] = M[M[SP]];
   break;
case cmSave:
   M[M[SP+1]] = M[SP];
   SP += 2;
   break;
case cmDup:
   SP--; M[SP] = M[SP+1];
   break;
case cmDrop:
   SP++;
   break;
case cmSwap:
   Buf = M[SP]; M[SP] = M[SP+1]; M[SP+1] = Buf;
   break;
case cmOver:
   SP--; M[SP] = M[SP+2];
   break;
case cmGOTO:
   PC = M[SP++];
   break;
case cmIfEQ:
   if ( M[SP+2] == M[SP+1] )
      PC = M[SP];
   SP += 3;
   break;
case cmIfNE:
   if ( M[SP+2] != M[SP+1] )
      PC = M[SP];
   SP += 3;
   break;
case cmIfLE:
   if ( M[SP+2] <= M[SP+1] )
      PC = M[SP];
   SP += 3;
   break;
```

```
            case cmIfLT:
               if ( M[SP+2] < M[SP+1] )
                  PC = M[SP];
               SP += 3;
               break;
            case cmIfGE:
               if ( M[SP+2] >= M[SP+1] )
                  PC = M[SP];
               SP += 3;
               break;
            case cmIfGT:
               if ( M[SP+2] > M[SP+1] )
                  PC = M[SP];
               SP += 3;
               break;
            case cmIn:
               putchar('?');
               scanf("%d", &(M[--SP]));
               readln;
               break;
            case cmOut:
               printf("%*d", M[SP], M[SP+1]);
               SP += 2;
               break;
            case cmOutLn:
               puts("");
               break;
            default:
               puts("Недопустимый код операции");
               M[PC] = cmStop;
            }
      puts("");
      if( SP < MEMSIZE )
         printf("Код возврата %d\n", M[SP]);
      printf("Нажмите ВВОД");
      readln;
   }
```

Листинг П4.16. Заголовочный файл генератора кода

```
/* Генератор кода (gen.h) */
#ifndef GEN
#define GEN

#include "table.h"

extern int PC;

void InitGen(void);
```

```
void Gen(int Cmd);
void Fixup(int A);

void GenAbs(void);
void GenMin(void);
void GenOdd(void);
void GenConst(int C);
void GenComp(tLex Op);
void GenAddr(tObj *X);
void AllocateVariables(void);

#endif
```

Листинг П4.17. Файл генератора кода

```
/* Генератор кода (gen.c) */

#include <limits.h>
#include <stdlib.h>
#include <string.h>

#include "scan.h"
#include "table.h"
#include "ovm.h"
#include "error.h"

int PC;

void InitGen(void) {
   PC = 0;
}

void Gen(int Cmd) {
   M[PC++] = Cmd;
}

void Fixup(int A) {
   while( A > 0 ) {
      int temp = M[A-2];
      M[A-2] = PC;
      A = temp;
   }
}

void GenAbs(void) {
   Gen(cmDup);
   Gen(0);
   Gen(PC+3);
```

```
   Gen(cmIfGE);
   Gen(cmNeg);
}

void GenMin(void) {
   Gen(INT_MAX);
   Gen(cmNeg);
   Gen(1);
   Gen(cmSub);
}

void GenOdd(void) {
   Gen(2);
   Gen(cmMod);
   Gen(0);
   Gen(0); /* Адрес перехода вперед */
   Gen(cmIfEQ);
}

void GenConst(int C) {
   Gen(abs(C));
   if ( C < 0 )
      Gen(cmNeg);
}

void GenComp(tLex Lex) {
   Gen(0); /* Адрес перехода вперед */
   switch( Lex ) {
   case lexEQ : Gen(cmIfNE); break;
   case lexNE : Gen(cmIfEQ); break;
   case lexLE : Gen(cmIfGT); break;
   case lexLT : Gen(cmIfGE); break;
   case lexGE : Gen(cmIfLT); break;
   case lexGT : Gen(cmIfLE); break;
   }
}

void GenAddr(tObj* X) {
   Gen(X->Val);/* В текущую ячейку адрес предыдущей+2 */
   X->Val = PC+1; /* Адрес+2 = PC+1 */
}

void AllocateVariables(void) {
   char msg[80];
   tObj* VRef; /* Ссылка на переменную в таблице имен */
   VRef = FirstVar(); /* Найти первую переменную */
   while( VRef != NULL ) {
      if( VRef->Val == 0 ) {
         strcpy(msg, "Переменная ");
```

```
            strcat(msg, VRef->Name);
             strcat(msg,  " не используется");
            Warning(msg);
            }
         else {
            Fixup(VRef->Val); /* Адресная привязка */
            PC++;
         }
         VRef = NextVar(); /* Найти следующую переменную */
      }
   }
```

ПРИЛОЖЕНИЕ 5. ТЕКСТ КОМПИЛЯТОРА «О» НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ ЯВА

Приведенная в этом приложении Ява-версия компилятора языка «О» соответствует спецификациям Явы 2. В классе сканера использован внутренний класс, то есть возможность, отличающая язык Ява 2 от первоначальной версии. Программа может быть откомпилирована в любой системе, поддерживающей спецификацию Ява 2.

Устройство Ява-версии компилятора подобно вариантам, написанным на Си и Обероне. Общее с программой на Обероне — явное указание имен классов при ссылках на статические поля и методы; с программой на Си — синтаксис управляющих конструкций.

Можно обратить внимание на характерные для языка Ява приемы работы со строками, которые являются объектами, а не массивами символов. Специфика Ява-технологии проявляется также в использовании библиотек ввода-вывода и обработке исключений.

Из-за отсутствия в Яве перечислений, для идентификации видов лексем языка «О», категорий и типов идентификаторов в таблице имен использован тип `int`.

Листинг П5.1. Основной класс приложения

```
// Компилятор языка "О"
public class O {

static void Init() {
   Text.Reset();
   if( !Text.Ok )
      Error.Message(Text.Message);
   Scan.Init();
   Gen.Init();
}
```

```
  static void Done() {
     Text.Close();
  }

  public static void main(String[] args) {
     System.out.println("\nКомпилятор языка O");
     if( args.length == 0 )
        Location.Path = null;
     else
        Location.Path = args[0];
     Init();          // Инициализация
     Pars.Compile(); // Компиляция
     OVM.Run();       // Выполнение
     Done();          // Завершение
  }

  }
```

Листинг П5.2. Класс, определяющий текущую позицию в тексте

```
// Текущая позиция в исходном тексте
class Location {
   static int Line;    // Номер строки
   static int Pos;     // Номер символа в строке
   static int LexPos;  // Позиция начала лексемы
   static String Path; // Путь к файлу
}
```

Листинг П5.3. Класс драйвера текста

```
// Драйвер исходного текста
import java.io.*;

class Text {

   static final int  TABSIZE = 3;
   static final char chSPACE = ' ';    // Пробел
   static final char chTAB   = '\t';   // Табуляция
   static final char chEOL   = '\n';   // Конец строки
   static final char chEOT   = '\0';   // Конец текста

   static boolean Ok = false;
   static String Message = "Файл не открыт";
   static int Ch = chEOT;

   private static InputStream f;

   static void NextCh() {
      try {
         if( (Ch = f.read()) == -1 )
```

```
            Ch = chEOT;
         else if( Ch == '\n' ) {
            System.out.println();
            Location.Line++;
            Location.Pos = 0;
            Ch = chEOL;
            }
         else if( Ch == '\r' )
            NextCh();
         else if( Ch != '\t' ) {
            System.out.write(Ch);
               // write, не print! Почему?
            Location.Pos++;
            }
         else
            do
               System.out.print(' ');
            while( ++Location.Pos % TABSIZE != 0 );
      } catch (IOException e) {};
   }

   static void Reset() {
      if( Location.Path == null ) {
         System.out.println(
            "Формат вызова:\n   O <входной файл>");
         System.exit(1);
         }
      else
         try {
            f = new FileInputStream(Location.Path);
            Ok = true; Message = "Ok";
            Location.Pos = 0; Location.Line = 1;
            NextCh();
         } catch (IOException e) {
            Ok = false;
            Message = "Входной файл не найден";
         }
   }

   static void Close() {
      try {
         f.close();
      } catch (IOException e) {};
   }

}
```

Листинг П5.4. Класс обработчика ошибок

```
// Обработка ошибок
import java.io.*;

class Error {

static void Message(String Msg) {
   int ELine = Location.Line;
   while(Text.Ch != Text.chEOL && Text.Ch != Text.chEOT)
      Text.NextCh();
   if( Text.Ch == Text.chEOT ) System.out.println();
   for( int i = 1; i < Location.LexPos; i++ )
      System.out.print(' ');
   System.out.println("^");
   System.out.println(
      "(Строка " + ELine + ") Ошибка: " + Msg
   );
   System.out.println();
   System.out.print("Нажмите ВВОД");
   try{ while( System.in.read() != '\n' ); }
   catch (IOException e) {};
   System.exit(0);
}

static void Expected(String Msg) {
   Message("Ожидается " + Msg);
}

static void Warning(String Msg) {
   System.out.println();
   System.out.println("Предупреждение: " + Msg);
}

}
```

Обратите внимание в листинге П5.5 на использование класса `StringBuffer` для сканирования имени.

Листинг П5.5. Класс лексического анализатора

```
// Лексический анализатор
class Scan {

static int NAMELEN = 31; // Наибольшая длина имени

final static int
   lexNone   = 0,
   lexName   = 1,   lexNum    = 2,
```

```
   lexMODULE = 3,   lexIMPORT = 4,
   lexBEGIN  = 5,   lexEND    = 6,
   lexCONST  = 7,   lexVAR    = 8,
   lexWHILE  = 9,   lexDO     = 10,
   lexIF     = 11,  lexTHEN   = 12,
   lexELSIF  = 13,  lexELSE   = 14,
   lexMult   = 15,  lexDIV    = 16,  lexMOD    = 17,
   lexPlus   = 18,  lexMinus  = 19,
   lexEQ     = 20,  lexNE     = 21,
   lexLT     = 22,  lexLE     = 23,
   lexGT     = 24,  lexGE     = 25,
   lexDot    = 26,  lexComma  = 27,  lexColon  = 28,
   lexSemi   = 29,  lexAss    = 30,
   lexLPar   = 31,  lexRPar   = 32,
   lexEOT    = 33;

// Текущая лексема
   static int Lex;
// Строковое значение имени
   private static StringBuffer Buf =
      new StringBuffer(NAMELEN);
   static String Name;

// Значение числовых литералов
   static int Num;

private static int KWNUM = 34;
private static int nkw = 0;

static private class Item {
   String Word;
   int Lex;
}

private static Item[] KWTable = new Item[KWNUM];

private static void EnterKW(String Name, int Lex) {
   // Обратите внимание на следующую строку!!!
   (KWTable[nkw] = new Item()).Word = new String(Name);
   KWTable[nkw++].Lex = Lex;
}

private static int TestKW() {
   for( int i = nkw - 1; i >= 0; i-- )
      if( KWTable[i].Word.compareTo(Name) == 0 )
         return KWTable[i].Lex;
   return lexName;
}
```

```
private static void Ident() {
   int i = 0;
   Buf.setLength(0);
   do {
      if ( i++ < NAMELEN )
         Buf.append((char)Text.Ch);
      else
         Error.Message("Слишком длинное имя");
      Text.NextCh();
   } while( Character.isLetterOrDigit((char)Text.Ch) );
   Name = Buf.toString();
   Lex = TestKW(); // Проверка на ключевое слово
}

private static void Number() {
   Lex = lexNum;
   Num = 0;
   do {
      int d = Text.Ch - '0';
      if( (Integer.MAX_VALUE - d)/10 >= Num )
         Num = 10*Num + d;
      else
         Error.Message("Слишком большое число");
      Text.NextCh();
   } while( Character.isDigit((char)Text.Ch) );
}

private static void Comment() {
   Text.NextCh();
   do {
      while( Text.Ch != '*' && Text.Ch != Text.chEOT )
         if( Text.Ch == '(' ) {
            Text.NextCh();
            if( Text.Ch == '*' )
               Comment();
            }
         else
            Text.NextCh();
      if ( Text.Ch == '*' )
         Text.NextCh();
   } while( Text.Ch != ')' && Text.Ch != Text.chEOT );
   if ( Text.Ch == ')' )
      Text.NextCh();
   else {
      Location.LexPos = Location.Pos;
      Error.Message("Не закончен комментарий");
   }
}
```

```
/*
private static void Comment() {
   int Level = 1;
   Text.NextCh();
   do
      if( Text.Ch == '*' ) {
         Text.NextCh();
         if( Text.Ch == ')' )
            { Level--; Text.NextCh(); }
         }
      else if( Text.Ch == '(' ) {
         Text.NextCh();
         if( Text.Ch == '*' )
            { Level++; Text.NextCh(); }
         }
      else //if ( Text.Ch <> chEOT )
         Text.NextCh();
   while( Level != 0 && Text.Ch != Text.chEOT );
   if( Level != 0 ) {
      Location.LexPos = Location.Pos;
      Error.Message("Не закончен комментарий");
   }
}
*/

static void NextLex() {
   while(
      Text.Ch == Text.chSPACE ||
      Text.Ch == Text.chTAB ||
      Text.Ch == Text.chEOL
   )
      Text.NextCh();
   Location.LexPos = Location.Pos;
   if( Character.isLetter((char)Text.Ch) )
      Ident();
   else if( Character.isDigit((char)Text.Ch) )
      Number();
   else
      switch( Text.Ch ) {
      case ';':
         Text.NextCh(); Lex = lexSemi;
         break;
      case ':':
         Text.NextCh();
         if( Text.Ch == '=' )
            { Text.NextCh(); Lex = lexAss; }
         else
            Lex = lexColon;
         break;
```

```
case '.':
   Text.NextCh(); Lex = lexDot;
   break;
case ',':
    Text.NextCh(); Lex = lexComma;
   break;
case '=':
   Text.NextCh(); Lex = lexEQ;
   break;
case '#':
   Text.NextCh(); Lex = lexNE;
   break;
case '<':
   Text.NextCh();
   if( Text.Ch == '=' )
      { Text.NextCh(); Lex = lexLE; }
   else
      Lex = lexLT;
   break;
case '>':
   Text.NextCh();
   if ( Text.Ch == '=' )
      { Text.NextCh(); Lex = lexGE; }
   else
       Lex = lexGT;
   break;
case '(':
   Text.NextCh();
   if( Text.Ch == '*' )
      { Comment();   NextLex(); }
   else
      Lex = lexLPar;
   break;
case ')':
   Text.NextCh(); Lex = lexRPar;
   break;
case '+':
   Text.NextCh(); Lex = lexPlus;
   break;
case '-':
   Text.NextCh(); Lex = lexMinus;
   break;
case '*':
   Text.NextCh(); Lex = lexMult;
   break;
case Text.chEOT:
   Lex = lexEOT;
   break;
default:
```

```
            Error.Message("Недопустимый символ");
         }
   }

   static void Init() {
      EnterKW("ARRAY",     lexNone);
      EnterKW("BY",        lexNone);
      EnterKW("BEGIN",     lexBEGIN);
      EnterKW("CASE",      lexNone);
      EnterKW("CONST",     lexCONST);
      EnterKW("DIV",       lexDIV);
      EnterKW("DO",        lexDO);
      EnterKW("ELSE",      lexELSE);
      EnterKW("ELSIF",     lexELSIF);
      EnterKW("END",       lexEND);
      EnterKW("EXIT",      lexNone);
      EnterKW("FOR",       lexNone);
      EnterKW("IF",        lexIF);
      EnterKW("IMPORT",    lexIMPORT);
      EnterKW("IN",        lexNone);
      EnterKW("IS",        lexNone);
      EnterKW("LOOP",      lexNone);
      EnterKW("MOD",       lexMOD);
      EnterKW("MODULE",    lexMODULE);
      EnterKW("NIL",       lexNone);
      EnterKW("OF",        lexNone);
      EnterKW("OR",        lexNone);
      EnterKW("POINTER",   lexNone);
      EnterKW("PROCEDURE", lexNone);
      EnterKW("RECORD",    lexNone);
      EnterKW("REPEAT",    lexNone);
      EnterKW("RETURN",    lexNone);
      EnterKW("THEN",      lexTHEN);
      EnterKW("TO",        lexNone);
      EnterKW("TYPE",      lexNone);
      EnterKW("UNTIL",     lexNone);
      EnterKW("VAR",       lexVAR);
      EnterKW("WHILE",     lexWHILE);
      EnterKW("WITH",      lexNone);

      NextLex();
   }

   }
```

Листинг П5.6. Класс синтаксического и контекстного анализатора

```
   // Распознаватель
   class Pars {
```

```
static final int
   spABS    = 1;
   spMAX    = 2;
   spMIN    = 3;
   spDEC    = 4;
   spODD    = 5;
   spHALT   = 6;
   spINC    = 7;
   spInOpen = 8;
   spInInt  = 9;
   spOutInt = 10;
   spOutLn  = 11;

static void Check(int L, String M) {
   if( Scan.Lex != L )
      Error.Expected(M);
   else
      Scan.NextLex();
}

// ["+" | "-"] (Число | Имя)
static int ConstExpr() {
   int v = 0;
   Obj X;
   int Op;

   Op = Scan.lexPlus;
   if( Scan.Lex == Scan.lexPlus ||
      Scan.Lex == Scan.lexMinus )
   {
      Op = Scan.Lex;
      Scan.NextLex();
   }
   if( Scan.Lex == Scan.lexNum ) {
      v = Scan.Num;
      Scan.NextLex();
      }
   else if( Scan.Lex == Scan.lexName ) {
      X = Table.Find(Scan.Name);
      if( X.Cat == Table.catGuard )
         Error.Message(
            "Нельзя определять константу через себя"
         );
      else if( X.Cat != Table.catConst )
         Error.Expected( "имя константы" );
      else {
         v = X.Val;
         Scan.NextLex();
```

```
      }
      }
   else
      Error.Expected( "константное выражение" );
   if( Op == Scan.lexMinus )
      return -v;
   return v;
}

// Имя "=" КонстВыраж
static void ConstDecl() {
   Obj ConstRef; // Ссылка на имя в таблице

   ConstRef = Table.NewName(Scan.Name, Table.catGuard);
   Scan.NextLex();
   Check(Scan.lexEQ, "\"=\"");
   ConstRef.Val = ConstExpr();
   ConstRef.Typ = Table.typInt;
   ConstRef.Cat = Table.catConst;
}

static void ParseType() {
   Obj TypeRef;

   if( Scan.Lex != Scan.lexName )
      Error.Expected("имя");
   else {
      TypeRef = Table.Find(Scan.Name);
      if( TypeRef.Cat != Table.catType )
         Error.Expected("имя типа");
      else if( TypeRef.Typ != Table.typInt )
         Error.Expected("целый тип");
      Scan.NextLex();
   }
}

// Имя {"," Имя} ":" Тип
static void VarDecl() {
   Obj NameRef;

   if( Scan.Lex != Scan.lexName )
      Error.Expected("имя");
   else {
      NameRef = Table.NewName(Scan.Name, Table.catVar);
      NameRef.Typ = Table.typInt;
      Scan.NextLex();
   }
   while( Scan.Lex == Scan.lexComma ) {
      Scan.NextLex();
```

```
      if( Scan.Lex != Scan.lexName )
         Error.Expected("имя");
      else {
         NameRef =
            Table.NewName(Scan.Name, Table.catVar);
         NameRef.Typ = Table.typInt;
         Scan.NextLex();
      }
   }
   Check(Scan.lexColon, "\":\"");
   ParseType();
}

// { CONST {ОбъявлКонст ";"} | VAR {ОбъявлПерем ";"} }
static void DeclSeq() {
   while( Scan.Lex == Scan.lexCONST ||
      Scan.Lex == Scan.lexVAR )
   {
      if( Scan.Lex == Scan.lexCONST ) {
         Scan.NextLex();
         while( Scan.Lex == Scan.lexName ) {
            ConstDecl(); //Объявление константы
            Check(Scan.lexSemi, "\";\"");
         }
         }
      else {
         Scan.NextLex(); // VAR
         while( Scan.Lex == Scan.lexName ) {
            VarDecl();   //Объявление переменных
            Check(Scan.lexSemi, "\";\"");
         }
      }
   }
}

static void IntExpression() {
   if(Expression() != Table.typInt )
      Error.Expected("выражение целого типа");
}

static int StFunc(int F) {
   switch( F ) {
   case spABS:
      IntExpression();
      Gen.Abs();
      return Table.typInt;
   case spMAX:
      ParseType();
      Gen.Cmd(Integer.MAX_VALUE);
```

```
      return Table.typInt;
   case spMIN:
      ParseType();
      Gen.Min();
      return Table.typInt;
   case spODD:
      IntExpression();
      Gen.Odd();
      return Table.typBool;
   }
   return Table.typNone; // Чтоб не было предупреждений
}

static int Factor() {
   Obj X;
   int T = 0; // Чтоб не было предупреждений

   if( Scan.Lex == Scan.lexName ) {
      if( (X = Table.Find(Scan.Name)).Cat ==
            Table.catVar )
      {
         Gen.Addr(X);    // Адрес переменной
         Gen.Cmd(OVM.cmLoad);
         Scan.NextLex();
         return X.Typ;
         }
      else if( X.Cat == Table.catConst ) {
         Gen.Const(X.Val);
         Scan.NextLex();
         return X.Typ;
         }
      else if( X.Cat == Table.catStProc &&
         X.Typ != Table.typNone )
      {
         Scan.NextLex();
         Check(Scan.lexLPar, "\"(\"");
         T = StFunc(X.Val);
         Check(Scan.lexRPar, "\")\"");
         }
      else
         Error.Expected(
         "переменная, константа или процедура-функция"
         );
      }
   else if( Scan.Lex == Scan.lexNum ) {
      Gen.Const(Scan.Num);
      Scan.NextLex();
      return Table.typInt;
      }
```

```
   else if( Scan.Lex == Scan.lexLPar ) {
      Scan.NextLex();
      T = Expression();
      Check(Scan.lexRPar, "\")\"");
      }
   else
      Error.Expected("имя, число или \"(\"");
   return T;
}

static int Term() {
   int Op;
   int T = Factor();
   if(      Scan.Lex == Scan.lexMult ||
         Scan.Lex == Scan.lexDIV  ||
         Scan.Lex == Scan.lexMOD )
   {
      if( T != Table.typInt )
         Error.Message(
            "Несоответствие операции типу операнда"
         );
      do {
         Op = Scan.Lex;
         Scan.NextLex();
         if( (T = Factor()) != Table.typInt )
            Error.Expected("выражение целого типа");
         switch( Op ) {
         case Scan.lexMult: Gen.Cmd(OVM.cmMult); break;
         case Scan.lexDIV:  Gen.Cmd(OVM.cmDiv); break;
         case Scan.lexMOD:  Gen.Cmd(OVM.cmMod); break;
         }
      } while( Scan.Lex == Scan.lexMult ||
         Scan.Lex == Scan.lexDIV ||
         Scan.Lex == Scan.lexMOD );
   }
   return T;
}

// ["+"|"-"] Слагаемое {ОперСлож Слагаемое}
static int SimpleExpr() {
   int T;
   int Op;

   if( Scan.Lex == Scan.lexPlus ||
      Scan.Lex == Scan.lexMinus )
   {
      Op = Scan.Lex;
      Scan.NextLex();
      if( (T = Term()) != Table.typInt )
```

```
            Error.Expected("выражение целого типа");
         if( Op == Scan.lexMinus )
            Gen.Cmd(OVM.cmNeg);
         }
      else
         T = Term();
      if( Scan.Lex == Scan.lexPlus ||
         Scan.Lex == Scan.lexMinus )
      {
         if( T != Table.typInt )
            Error.Message(
               "Несоответствие операции типу операнда"
            );
         do {
            Op = Scan.Lex;
            Scan.NextLex();
            if( (T = Term()) != Table.typInt )
               Error.Expected("выражение целого типа");
            switch(Op) {
            case Scan.lexPlus:  Gen.Cmd(OVM.cmAdd); break;
            case Scan.lexMinus: Gen.Cmd(OVM.cmSub); break;
            }
         } while( Scan.Lex == Scan.lexPlus ||
            Scan.Lex == Scan.lexMinus );
      }
      return T;
   }

   // ПростоеВыраж [Отношение ПростоеВыраж]
   static int Expression() {
      int Op;

      int T = SimpleExpr();
      if(   Scan.Lex == Scan.lexEQ ||
         Scan.Lex == Scan.lexNE ||
         Scan.Lex == Scan.lexGT ||
         Scan.Lex == Scan.lexGE ||
         Scan.Lex == Scan.lexLT ||
         Scan.Lex == Scan.lexLE )
      {
         Op = Scan.Lex;
         if( T != Table.typInt )
            Error.Message(
               "Несоответствие операции типу операнда"
            );
         Scan.NextLex();
         if( (T = SimpleExpr()) != Table.typInt )
            Error.Expected("выражение целого типа");
         Gen.Comp(Op);   //Генерация условного перехода
```

```
      T = Table.typBool;
   } //иначе тип равен типу первого простого выражения
   return T;
}

// Переменная = Имя
static void Variable() {
   Obj X;

   if( Scan.Lex != Scan.lexName )
      Error.Expected("имя");
   else {
      if(
         (X = Table.Find(Scan.Name)).Cat !=
         Table.catVar
      )
         Error.Expected("имя переменной");
      Gen.Addr(X);
      Scan.NextLex();
   }
}

static void StProc(int sp) {
   switch( sp ) {
   case spDEC:
      Variable();
      Gen.Cmd(OVM.cmDup);
      Gen.Cmd(OVM.cmLoad);
      if( Scan.Lex == Scan.lexComma ) {
         Scan.NextLex();
         IntExpression();
         }
      else
         Gen.Cmd(1);
      Gen.Cmd(OVM.cmSub);
      Gen.Cmd(OVM.cmSave);
      return;
   case spINC:
      Variable();
      Gen.Cmd(OVM.cmDup);
      Gen.Cmd(OVM.cmLoad);
      if( Scan.Lex == Scan.lexComma ) {
         Scan.NextLex();
         IntExpression();
         }
      else
         Gen.Cmd(1);
      Gen.Cmd(OVM.cmAdd);
      Gen.Cmd(OVM.cmSave);
```

```
      return;
   case spInOpen:
      // Пусто ;
      return;
   case spInInt:
      Variable();
      Gen.Cmd(OVM.cmIn);
      Gen.Cmd(OVM.cmSave);
      return;
   case spOutInt:
      IntExpression();
      Check(Scan.lexComma, "\",\"");
      IntExpression();
      Gen.Cmd(OVM.cmOut);
      return;
   case spOutLn:
      Gen.Cmd(OVM.cmOutLn);
      return;
   case spHALT:
      Gen.Const(ConstExpr());
      Gen.Cmd(OVM.cmStop);
      return;
   }
}

static void BoolExpression() {
   if( Expression() != Table.typBool )
      Error.Expected("логическое выражение");
}

// Переменная "=" Выраж
static void AssStatement() {
   Variable();
   if( Scan.Lex == Scan.lexAss ) {
      Scan.NextLex();
      IntExpression();
      Gen.Cmd(OVM.cmSave);
      }
   else
      Error.Expected("\":=\"");
}

// Имя ["(" // Выраж | Переменная  ")"]
static void CallStatement(int sp) {
   Check(Scan.lexName, "имя процедуры");
   if( Scan.Lex == Scan.lexLPar ) {
      Scan.NextLex();
      StProc(sp);
      Check( Scan.lexRPar, "\")\"" );
```

```
      }
   else if( sp == spOutLn || sp == spInOpen )
      StProc(sp);
   else
      Error.Expected("\"(\"");
}

static void IfStatement() {
   int CondPC;
   int LastGOTO;

   Check(Scan.lexIF, "IF");
   LastGOTO = 0;      // Предыдущего перехода нет
   BoolExpression();
   CondPC = Gen.PC;   // Запомн. положение усл. перехода
   Check(Scan.lexTHEN, "THEN");
   StatSeq();
   while( Scan.Lex == Scan.lexELSIF ) {
      Gen.Cmd(LastGOTO);  // Фиктивн. адрес, указывающий
      Gen.Cmd(OVM.cmGOTO);// на место предыдущ. перехода
      LastGOTO = Gen.PC;  // Запомнить место GOTO
      Scan.NextLex();
      Gen.Fixup(CondPC);  // Зафикс. адрес усл. перехода
      BoolExpression();
      CondPC = Gen.PC;    // Запомн. полож. усл. перехода
      Check(Scan.lexTHEN, "THEN");
      StatSeq();
   }
   if( Scan.Lex == Scan.lexELSE ) {
      Gen.Cmd(LastGOTO);  // Фиктивн. адрес, указывающий
      Gen.Cmd(OVM.cmGOTO);// на место предыдущ. перехода
      LastGOTO = Gen.PC;  // Запомн. место последн. GOTO
      Scan.NextLex();
      Gen.Fixup(CondPC);  // Зафикс. адрес усл. перехода
      StatSeq();
      }
   else
      Gen.Fixup(CondPC);  //Если ELSE отсутствует
   Check(Scan.lexEND, "END");
   Gen.Fixup(LastGOTO);   //Направить сюда все GOTO
}

static void WhileStatement() {
   int WhilePC = Gen.PC;
   Check(Scan.lexWHILE, "WHILE");
   BoolExpression();
   int CondPC = Gen.PC;
   Check(Scan.lexDO, "DO");
   StatSeq();
```

```
   Check(Scan.lexEND, "END");
   Gen.Cmd(WhilePC);
   Gen.Cmd(OVM.cmGOTO);
   Gen.Fixup(CondPC);
}

static void Statement() {
   Obj X;

   if( Scan.Lex == Scan.lexName ) {
      if( (X=Table.Find(Scan.Name)).Cat ==
          Table.catModule )
      {
         Scan.NextLex();
         if( Scan.Lex != Scan.lexDot )
            Error.Expected("\".\"");
         if(
            Scan.Lex == Scan.lexName &&
            X.Name.length() + Scan.Name.length() <=
            Scan.NAMELEN
         )
            X = Table.Find( X.Name + "." + Scan.Name);
         else
            Error.Expected("имя из модуля " + X.Name);
      }
      if( X.Cat == Table.catVar )
         AssStatement();        // Присваивание
      else if( X.Cat == Table.catStProc &&
         X.Typ == Table.typNone
      )
         CallStatement(X.Val); // Вызов процедуры
      else
         Error.Expected(
            "обозначение переменной или процедуры"
         );
      }
   else if( Scan.Lex == Scan.lexIF )
      IfStatement();
   else if( Scan.Lex == Scan.lexWHILE )
      WhileStatement();
   // иначе пустой оператор
}

// Оператор {";" Оператор}
static void StatSeq() {
   Statement();    //Оператор
   while( Scan.Lex == Scan.lexSemi ) {
      Scan.NextLex();
      Statement(); //Оператор
```

```
   }
}

static void ImportModule() {
   if( Scan.Lex == Scan.lexName ) {
      Table.NewName(Scan.Name, Table.catModule);
      if( Scan.Name.compareTo("In") == 0 ) {
         Table.Enter("In.Open",
            Table.catStProc, Table.typNone, spInOpen);
         Table.Enter("In.Int",
            Table.catStProc, Table.typNone, spInInt);
         }
      else if( Scan.Name.compareTo("Out") == 0 ) {
         Table.Enter("Out.Int",
            Table.catStProc, Table.typNone, spOutInt);
         Table.Enter("Out.Ln",
            Table.catStProc, Table.typNone, spOutLn);
         }
      else
         Error.Message("Неизвестный модуль");
      Scan.NextLex();
      }
   else
      Error.Expected("имя импортируемого модуля");
}

// IMPORT Имя { "," Имя } ";"
static void Import() {
   Check(Scan.lexIMPORT, "IMPORT");
   ImportModule();
   while( Scan.Lex == Scan.lexComma ) {
      Scan.NextLex();
      ImportModule();
   }
   Check(Scan.lexSemi, "\";\"");
}

// MODULE Имя ";" [Импорт] ПослОбъявл
// [BEGIN ПослОператоров]
// END Имя "."
static void Module() {
   Obj ModRef; //Ссылка на имя модуля в таблице

   Check(Scan.lexMODULE, "MODULE");
   if( Scan.Lex != Scan.lexName )
      Error.Expected("имя модуля");
   //Имя модуля - в таблицу имен
      ModRef = Table.NewName(Scan.Name, Table.catModule);
   Scan.NextLex();
```

```
   Check(Scan.lexSemi, "\";\"");
   if( Scan.Lex == Scan.lexIMPORT )
      Import();
   DeclSeq();
   if( Scan.Lex == Scan.lexBEGIN ) {
      Scan.NextLex();
      StatSeq();
   }
   Check(Scan.lexEND, "END");

   //Сравнение имени модуля и имени после END
      if( Scan.Lex != Scan.lexName )
         Error.Expected("имя модуля");
      else if( Scan.Name.compareTo(ModRef.Name) != 0 )
         Error.Expected(
            "имя модуля \"" + ModRef.Name + "\""
         );
      else
         Scan.NextLex();
   Check(Scan.lexDot, "\".\"");
   Gen.Cmd(0);              // Код возврата
   Gen.Cmd(OVM.cmStop);     // Команда останова
   Gen.AllocateVariables(); // Размещение переменных
}

static void Compile() {
   Table.Init();
   Table.OpenScope(); //Блок стандартных имен
   Table.Enter("ABS",
      Table.catStProc, Table.typInt, spABS);
   Table.Enter("MAX",
      Table.catStProc, Table.typInt, spMAX);
   Table.Enter("MIN",
      Table.catStProc, Table.typInt, spMIN);
   Table.Enter("DEC",
      Table.catStProc, Table.typNone, spDEC);
   Table.Enter("ODD",
      Table.catStProc, Table.typBool, spODD);
   Table.Enter("HALT",
      Table.catStProc, Table.typNone, spHALT);
   Table.Enter("INC",
      Table.catStProc, Table.typNone, spINC);
   Table.Enter("INTEGER",
      Table.catType, Table.typInt, 0);
   Table.OpenScope();  //Блок модуля
   Module();
   Table.CloseScope(); //Блок модуля
   Table.CloseScope(); //Блок стандартных имен
```

```
      System.out.println("\nКомпиляция завершена");
   }

   }
```

Листинг П5.7. Классы таблицы имен

```
   // Элемент таблицы имен
   class Obj {        // Тип записи таблицы имен
      String Name;    // Ключ поиска
      int Cat;        // Категория имени
      int Typ;        // Тип
      int Val;        // Значение
      Obj Prev;       // Указатель на пред. имя
   }

   // Таблица имен
   class Table {

   // Категории имён
      static final int
         catConst  = 1,  catVar    = 2,
         catType   = 3,  catStProc = 4,
         catModule = 5,  catGuard  = 6;

   // Типы
      static final int
         typNone = 0;  typInt = 1;  typBool = 2;

   private static Obj  Top;    // Указатель на вершину
   списка
   private static Obj  Bottom; // Указатель на конец списка
   private static Obj  CurrObj;

   // Инициализация таблицы
   static void Init() {
      Top = null;
   }

   // Добавление элемента
   static void Enter(String N, int C, int T, int V) {
      Obj P = new Obj();
      P.Name = new String(N);
      P.Cat = C;
      P.Typ = T;
      P.Val = V;
      P.Prev = Top;
      Top = P;
   }
```

```
static void OpenScope() {
   Enter("", catGuard, typNone, 0);
   if ( Top.Prev == null )
      Bottom = Top;
}

static void CloseScope() {
   while( Top.Cat != catGuard ){
      Top = Top.Prev;
   }
   Top = Top.Prev;
}

static Obj NewName(String Name, int Cat) {
   Obj obj = Top;
   while(
      obj.Cat != catGuard &&
      obj.Name.compareTo(Name) != 0
   )
      obj = obj.Prev;
   if ( obj.Cat == catGuard ) {
      obj = new Obj();
      obj.Name = new String(Name);
      obj.Cat = Cat;
      obj.Val = 0;
      obj.Prev = Top;
      Top = obj;
      }
   else
      Error.Message("Повторное объявление имени");
   return obj;
}

static Obj Find(String Name) {
   Obj obj;

   Bottom.Name = new String(Name);
   for( obj=Top;
         obj.Name.compareTo(Name)!=0;
            obj=obj.Prev );
   if( obj == Bottom )
      Error.Message("Необъявленное имя");
   return obj;
}
```

```
  static Obj FirstVar() {
     CurrObj = Top;
     return NextVar();
  }

  static Obj NextVar() {
     Obj VRef;

     while( CurrObj != Bottom && CurrObj.Cat != catVar )
        CurrObj = CurrObj.Prev;
     if( CurrObj == Bottom )
        return null;
     else {
        VRef = CurrObj;
        CurrObj = CurrObj.Prev;
        return VRef;
     }
  }

  }
```

Листинг П5.8. Класс виртуальной машины

```
  // Виртуальная машина
  import java.io.*;

  class OVM {

  static final int  MEMSIZE = 8*1024;

  static final int
     cmStop   = -1,

     cmAdd    = -2,
     cmSub    = -3,
     cmMult   = -4,
     cmDiv    = -5,
     cmMod    = -6,
     cmNeg    = -7,

     cmLoad   = -8,
     cmSave   = -9,

     cmDup    = -10,
     cmDrop   = -11,
     cmSwap   = -12,
     cmOver   = -13,

     cmGOTO   = -14,
```

```
   cmIfEQ   = -15,
   cmIfNE   = -16,
   cmIfLE   = -17,
   cmIfLT   = -18,
   cmIfGE   = -19,
   cmIfGT   = -20,

   cmIn     = -21,
   cmOut    = -22,
   cmOutLn  = -23;

static int M[] = new int[MEMSIZE];

static void readln() {
   try { while( System.in.read() != '\n' ); }
   catch (IOException e) {};
}

private static StreamTokenizer input =
   new StreamTokenizer(new InputStreamReader(System.in));

static int ReadInt() {
   try{ input.nextToken(); } catch (IOException e) {};
   return (int)input.nval;
}

static void Run() {
   int PC = 0;
   int SP = MEMSIZE;
   int Cmd;
   int Buf;

   loop: for (;;)
      if ( (Cmd = M[PC++]) >= 0 )
         M[--SP] = Cmd;
      else
         switch( Cmd ) {
         case cmAdd:
            SP++; M[SP] += M[SP-1];
            break;
         case cmSub:
            SP++; M[SP] -= M[SP-1];
            break;
         case cmMult:
            SP++; M[SP] *= M[SP-1];
            break;
         case cmDiv:
            SP++; M[SP] /= M[SP-1];
            break;
```

```
case cmMod:
   SP++; M[SP] %= M[SP-1];
   break;
case cmNeg:
   M[SP] = -M[SP];
   break;
case cmLoad:
   M[SP] = M[M[SP]];
   break;
case cmSave:
   M[M[SP+1]] = M[SP];
   SP += 2;
   break;
case cmDup:
   SP--; M[SP] = M[SP+1];
   break;
case cmDrop:
   SP++;
   break;
case cmSwap:
   Buf = M[SP]; M[SP] = M[SP+1]; M[SP+1] = Buf;
   break;
case cmOver:
   SP--; M[SP] = M[SP+2];
   break;
case cmGOTO:
   PC = M[SP++];
   break;
case cmIfEQ:
   if ( M[SP+2] == M[SP+1] )
      PC = M[SP];
   SP += 3;
   break;
case cmIfNE:
   if ( M[SP+2] != M[SP+1] )
      PC = M[SP];
   SP += 3;
   break;
case cmIfLE:
   if ( M[SP+2] <= M[SP+1] )
      PC = M[SP];
   SP += 3;
   break;
case cmIfLT:
   if ( M[SP+2] < M[SP+1] )
      PC = M[SP];
   SP += 3;
   break;
case cmIfGE:
```

```
            if ( M[SP+2] >= M[SP+1] )
               PC = M[SP];
            SP += 3;
            break;
         case cmIfGT:
            if ( M[SP+2] > M[SP+1] )
               PC = M[SP];
            SP += 3;
            break;
         case cmIn:
            System.out.print('?');
            M[--SP] = ReadInt();
            break;
         case cmOut:
            int w = M[SP] - (new Integer(M[SP+1])).
               toString().length();
            for( int i = 1; i <= w; i++ )
               System.out.print(" ");
            System.out.print(M[SP+1]);
            SP += 2;
            break;
         case cmOutLn:
            System.out.println();
            break;
         case cmStop:
            break loop;
         default:
            System.out.println(
               "Недопустимый код операции"
            );
            break loop;
         }
      System.out.println();
      if( SP < MEMSIZE )
         System.out.println("Код возврата " + M[SP]);
      System.out.print("Нажмите ВВОД");
      readln();
   }

   }
```

Листинг П5.9. Класс генератора кода

```
// Генератор кода
class Gen {

static int PC;

static void Init() {
```

```
   PC = 0;
}

static void Cmd(int Cmd) {
   OVM.M[PC++] = Cmd;
}

static void Fixup(int A) {
   while( A > 0 ) {
      int temp = OVM.M[A-2];
      OVM.M[A-2] = PC;
      A = temp;
   }
}

static void Abs() {
   Cmd(OVM.cmDup);
   Cmd(0);
   Cmd(PC+3);
   Cmd(OVM.cmIfGE);
   Cmd(OVM.cmNeg);
}

static void Min() {
   Cmd(Integer.MAX_VALUE);
   Cmd(OVM.cmNeg);
   Cmd(1);
   Cmd(OVM.cmSub);
}

static void Odd() {
   Cmd(2);
   Cmd(OVM.cmMod);
   Cmd(0);
   Cmd(0); // Адрес перехода вперед
   Cmd(OVM.cmIfEQ);
}

static void Const(int C) {
   Cmd(Math.abs(C));
   if ( C < 0 )
      Cmd(OVM.cmNeg);
}

static void Comp(int Lex) {
   Cmd(0); // Адрес перехода вперед
   switch( Lex ) {
   case Scan.lexEQ : Cmd(OVM.cmIfNE); break;
   case Scan.lexNE : Cmd(OVM.cmIfEQ); break;
```

```
   case Scan.lexLE : Cmd(OVM.cmIfGT); break;
   case Scan.lexLT : Cmd(OVM.cmIfGE); break;
   case Scan.lexGE : Cmd(OVM.cmIfLT); break;
   case Scan.lexGT : Cmd(OVM.cmIfLE); break;
   }
}

static void Addr(Obj X) {
   Cmd(X.Val);   // В текущую ячейку адрес предыдущей + 2
   X.Val = PC+1; // Адрес+2 = PC+1
}

static void AllocateVariables() {
   Obj VRef; // Ссылка на переменную в таблице имен

   VRef = Table.FirstVar();   // Найти первую переменную
   while( VRef != null ) {
      if ( VRef.Val == 0 )
         Error.Warning(
         "Переменная " + VRef.Name + " не используется"
         );
      else {
         Fixup(VRef.Val);    // Адресная привязка
         PC++;
      }
      VRef = Table.NextVar();// Найти след. переменную
   }
}

}
```

ПРИЛОЖЕНИЕ 6. ТЕКСТ КОМПИЛЯТОРА «О» НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ СИ#

Приведенный текст написан в соответствии со спецификацией языка Си#, определенной стандартом ECMA-334. Для компиляции программы в ОС Windows с установленным .NET Framework можно использовать командную строку:

```
>csc /recurse:*.cs
```

Особенностями Си#, которые можно увидеть в приведенном тексте компилятора «О» и которые отличают Си#-версию от варианта на Яве, являются:

- Способ работы со строками, которые являются объектами, но допускают традиционную нотацию для сравнения значений с помощью «==» и «!=», и обращение к отдельному символу с помощью индексатора.
- Использование перечислений и структур.
- Использование библиотек для консольного ввода-вывода и чтения файлов.

Листинг П6.1. Основной класс приложения. Файл `O.cs`

```
// Компилятор языка "O"
using System;

public class O {

static void Init() {
   Text.Reset();
   if( !Text.Ok )
      Error.Message(Text.Message);
   Scan.Init();
   Gen.Init();
}

static void Done() {
   Text.Close();
}
```

```
   static void Main(string[] args) {
      Console.WriteLine("\nКомпилятор языка O");
      if( args.Length == 0 )
         Location.Path = null;
      else
         Location.Path = args[0];
      Init();         // Инициализация
      Pars.Compile(); // Компиляция
      OVM.Run();      // Выполнение
      Done();         // Завершение
   }

   }
```

Листинг П6.2. Файл `Location.cs`

```
// Текущая позиция в исходном тексте
class Location {
   public static int Line;    // Номер строки
   public static int Pos;     // Номер символа в строке
   public static int LexPos;  // Позиция начала лексемы
   public static string Path; // Путь к файлу
}
```

Обратите внимание в листинге П6.3 на использование класса `Encoding` для вывода на консоль в правильной кодировке символов кириллицы, прочитанных из ASCII-файла.

Листинг П6.3. Файл `Text.cs`

```
// Драйвер исходного текста
using System;
using System.IO;
using System.Text;

class Text {

const int  TABSIZE = 3;

public const char chSPACE = ' ';    // Пробел
public const char chTAB   = '\t';   // Табуляция
public const char chEOL   = '\n';   // Конец строки
public const char chEOT   = '\0';   // Конец текста

public static bool Ok = false;
public static string Message = "Ok";
public static int Ch;

private static StreamReader f;
```

```
  public static void NextCh() {
     if( (Ch = f.Read()) == -1 )
        Ch = chEOT;
     else if( Ch == '\n' ) {
        Console.WriteLine();
        Location.Line++; Location.Pos = 0; Ch = chEOL;
        }
     else if( Ch == '\r' )
        NextCh();
     else if( Ch != '\t' ) {
        Console.Write((char)Ch);
        Location.Pos++;
        }
     else
        do
           Console.Write(' ');
        while( ++Location.Pos % TABSIZE != 0 );
  }

  public static void Reset() {
     if( Location.Path == null ) {
        Console.WriteLine(
           "Формат вызова:\n   O <входной файл>");
        Environment.Exit(1);
        }
     else if( File.Exists(Location.Path) ) {
        f = new StreamReader(Location.Path,
           Encoding.GetEncoding(866)); // Кириллица DOS
        Ok = true; Message = "Ok";
        Location.Pos = 0; Location.Line = 1;
        NextCh();
        }
     else {
        Ok = false;
        Message = "Входной файл не найден";
     }
  }

  public static void Close() {
     f.Close();
  }

  }
```

Листинг П6.4. Файл Error.cs

```
  // Обработка ошибок
  using System;
```

```
class Error {

public static void Message(string Msg) {
   int ELine = Location.Line;
   while(Text.Ch != Text.chEOL && Text.Ch != Text.chEOT)
      Text.NextCh();
   if( Text.Ch == Text.chEOT ) Console.WriteLine();
   for( int i = 1; i < Location.LexPos; i++ )
      Console.Write(' ');
   Console.WriteLine("^");
   Console.WriteLine(
      "(Строка " + ELine + ") Ошибка: " + Msg
   );
   Console.WriteLine();
   Console.Write("Нажмите ВВОД");
   Console.ReadLine();
   Environment.Exit(0);
}

public static void Expected(string Msg) {
   Message("Ожидается " + Msg);
}

public static void Warning(string Msg) {
   Console.WriteLine();
   Console.WriteLine("Предупреждение: " + Msg);
}

}
```

Листинг П6.5. Файл Scan.cs

```
// Лексический анализатор
using System;

public enum tLex {
   lexNone, lexName, lexNum,
   lexMODULE, lexIMPORT, lexBEGIN, lexEND,
   lexCONST,  lexVAR,  lexWHILE, lexDO,
   lexIF, lexTHEN, lexELSIF, lexELSE,
   lexMult, lexDIV, lexMOD, lexPlus, lexMinus,
   lexEQ, lexNE, lexLT, lexLE, lexGT, lexGE,
   lexDot, lexComma, lexColon, lexSemi, lexAss,
   lexLPar, lexRPar,
   lexEOT
};
```

```
class Scan {

public static int NAMELEN = 31; // Наибольшая длина имени

// Текущая лексема
   public static tLex Lex;
// Строковое значение имени
   public static string Name;

   private static System.Text.StringBuilder Buf =
      new System.Text.StringBuilder(NAMELEN);

// Значение числовых литералов
   public static int Num;

private static int KWNUM = 34;
private static int nkw = 0;

struct Item {
   public string Word;
   public tLex Lex;
}

private static Item[] KWTable = new Item[KWNUM];

private static void EnterKW(string Name, tLex Lex) {
   KWTable[nkw].Word = String.Copy(Name);
   KWTable[nkw++].Lex = Lex;
}

private static tLex TestKW() {
   for( int i = nkw - 1; i >= 0; i-- )
      if( KWTable[i].Word == Name )
         return KWTable[i].Lex;
   return tLex.lexName;
}

private static void Ident() {
   int i = 0;

   Buf.Length = NAMELEN;
   do {
      if ( i < NAMELEN )
         Buf[i++] = (char)Text.Ch;
      else
         Error.Message("Слишком длинное имя");
      Text.NextCh();
   } while( char.IsLetterOrDigit((char)Text.Ch) );
   Buf.Length = i;
```

```
   Name = Buf.ToString();
   Lex = TestKW(); // Проверка на ключевое слово
}

private static void Number() {
   Lex = tLex.lexNum;
   Num = 0;
   do {
      int d = Text.Ch - '0';
      if( (int.MaxValue - d)/10 >= Num )
         Num = 10*Num + d;
      else
         Error.Message("Слишком большое число");
      Text.NextCh();
   } while( char.IsDigit((char)Text.Ch) );
}

/*
private static void Comment() {
   Text.NextCh();
   do {
      while( Text.Ch != '*' && Text.Ch != Text.chEOT )
         if( Text.Ch == '(' ) {
            Text.NextCh();
            if( Text.Ch == '*' )
               Comment();
            }
         else
            Text.NextCh();
      if ( Text.Ch == '*' )
         Text.NextCh();
   } while( Text.Ch != ')' && Text.Ch != Text.chEOT );
   if( Text.Ch == ')' )
      Text.NextCh();
   else {
      Location.LexPos = Location.Pos;
      Error.Message("Не закончен комментарий");
   }
}
*/

private static void Comment() {
   int Level = 1;
   Text.NextCh();
   do
      if( Text.Ch == '*' ) {
         Text.NextCh();
         if( Text.Ch == ')' )
            { Level--; Text.NextCh(); }
```

```
         }
      else if( Text.Ch == '(' ) {
         Text.NextCh();
         if( Text.Ch == '*' )
            { Level++; Text.NextCh(); }
         }
      else // if ( Text.Ch <> chEOT )
         Text.NextCh();
   while( Level != 0 && Text.Ch != Text.chEOT );
   if( Level != 0 ) {
      Location.LexPos = Location.Pos;
      Error.Message("Не закончен комментарий");
   }
}

public static void NextLex() {
   while(
      Text.Ch == Text.chSPACE ||
      Text.Ch == Text.chTAB ||
      Text.Ch == Text.chEOL
   )
   Text.NextCh();
   Location.LexPos = Location.Pos;
   if( char.IsLetter((char)Text.Ch) )
      Ident();
   else if( char.IsDigit((char)Text.Ch) )
      Number();
   else
      switch( Text.Ch ) {
      case ';':
         Text.NextCh(); Lex = tLex.lexSemi;
         break;
      case ':':
         Text.NextCh();
         if( Text.Ch == '=' )
            { Text.NextCh(); Lex = tLex.lexAss; }
         else
            Lex = tLex.lexColon;
         break;
      case '.':
         Text.NextCh(); Lex = tLex.lexDot;
         break;
      case ',':
         Text.NextCh(); Lex = tLex.lexComma;
         break;
      case '=':
         Text.NextCh(); Lex = tLex.lexEQ;
         break;
      case '#':
```

```
            Text.NextCh(); Lex = tLex.lexNE;
            break;
         case '<':
            Text.NextCh();
            if( Text.Ch == '=' )
               { Text.NextCh(); Lex = tLex.lexLE; }
            else
               Lex = tLex.lexLT;
            break;
         case '>':
            Text.NextCh();
            if ( Text.Ch == '=' )
               { Text.NextCh(); Lex = tLex.lexGE; }
            else
               Lex = tLex.lexGT;
            break;
         case '(':
            Text.NextCh();
            if( Text.Ch == '*' )
               { Comment();   NextLex(); }
            else
               Lex = tLex.lexLPar;
            break;
         case ')':
            Text.NextCh(); Lex = tLex.lexRPar;
            break;
         case '+':
            Text.NextCh(); Lex = tLex.lexPlus;
            break;
         case '-':
            Text.NextCh(); Lex = tLex.lexMinus;
            break;
         case '*':
            Text.NextCh(); Lex = tLex.lexMult;
            break;
         case Text.chEOT:
            Lex = tLex.lexEOT;
            break;
         default:
            Error.Message("Недопустимый символ");
            break;
         }
   }

   public static void Init() {
      EnterKW("ARRAY",     tLex.lexNone);
      EnterKW("BY",        tLex.lexNone);
      EnterKW("BEGIN",     tLex.lexBEGIN);
      EnterKW("CASE",      tLex.lexNone);
```

```
   EnterKW("CONST",     tLex.lexCONST);
   EnterKW("DIV",       tLex.lexDIV);
   EnterKW("DO",        tLex.lexDO);
   EnterKW("ELSE",      tLex.lexELSE);
   EnterKW("ELSIF",     tLex.lexELSIF);
   EnterKW("END",       tLex.lexEND);
   EnterKW("EXIT",      tLex.lexNone);
   EnterKW("FOR",       tLex.lexNone);
   EnterKW("IF",        tLex.lexIF);
   EnterKW("IMPORT",    tLex.lexIMPORT);
   EnterKW("IN",        tLex.lexNone);
   EnterKW("IS",        tLex.lexNone);
   EnterKW("LOOP",      tLex.lexNone);
   EnterKW("MOD",       tLex.lexMOD);
   EnterKW("MODULE",    tLex.lexMODULE);
   EnterKW("NIL",       tLex.lexNone);
   EnterKW("OF",        tLex.lexNone);
   EnterKW("OR",        tLex.lexNone);
   EnterKW("POINTER",   tLex.lexNone);
   EnterKW("PROCEDURE", tLex.lexNone);
   EnterKW("RECORD",    tLex.lexNone);
   EnterKW("REPEAT",    tLex.lexNone);
   EnterKW("RETURN",    tLex.lexNone);
   EnterKW("THEN",      tLex.lexTHEN);
   EnterKW("TO",        tLex.lexNone);
   EnterKW("TYPE",      tLex.lexNone);
   EnterKW("UNTIL",     tLex.lexNone);
   EnterKW("VAR",       tLex.lexVAR);
   EnterKW("WHILE",     tLex.lexWHILE);
   EnterKW("WITH",      tLex.lexNone);

   NextLex();
}

}
```

Листинг П6.5. Файл Pars.cs

```
// Распознаватель
class Pars {

const int
   spABS    = 1,
   spMAX    = 2,
   spMIN    = 3,
   spDEC    = 4,
   spODD    = 5,
   spHALT   = 6,
   spINC    = 7,
```

```
   spInOpen = 8,
   spInInt  = 9,
   spOutInt = 10,
   spOutLn  = 11;

static void Check(tLex L, string M) {
   if( Scan.Lex != L )
      Error.Expected(M);
   else
      Scan.NextLex();
}

// ["+" | "-"] (Число | Имя)
static int ConstExpr() {
   int v = 0;
   Obj X;
   tLex Op;

   Op = tLex.lexPlus;
   if( Scan.Lex == tLex.lexPlus ||
      Scan.Lex == tLex.lexMinus )
   {
      Op = Scan.Lex;
      Scan.NextLex();
   }
   if( Scan.Lex == tLex.lexNum ) {
      v = Scan.Num;
      Scan.NextLex();
      }
   else if( Scan.Lex == tLex.lexName ) {
      X = Table.Find(Scan.Name);
      if( X.Cat == tCat.Guard )
         Error.Message(
            "Нельзя определять константу через себя"
         );
      else if( X.Cat != tCat.Const )
         Error.Expected( "имя константы" );
      else {
         v = X.Val;
         Scan.NextLex();
      }
      }
   else
      Error.Expected( "константное выражение" );
   if( Op == tLex.lexMinus )
      return -v;
   return v;
}
```

```
// Имя "=" КонстВыраж
static void ConstDecl() {
   Obj ConstRef = Table.NewName(Scan.Name, tCat.Guard);
   Scan.NextLex();
   Check(tLex.lexEQ, "\"=\"");
   ConstRef.Val = ConstExpr();
   ConstRef.Type = tType.Int;
   ConstRef.Cat = tCat.Const;
}

static void ParseType() {
   Obj TypeRef;

   if( Scan.Lex != tLex.lexName )
      Error.Expected("имя");
   else {
      TypeRef = Table.Find(Scan.Name);
      if( TypeRef.Cat != tCat.Type )
         Error.Expected("имя типа");
      else if( TypeRef.Type != tType.Int )
         Error.Expected("целый тип");
      Scan.NextLex();
   }
}

// Имя {"," Имя} ":" Тип
static void VarDecl() {
   Obj NameRef;

   if( Scan.Lex != tLex.lexName )
      Error.Expected("имя");
   else {
      NameRef = Table.NewName(Scan.Name, tCat.Var);
      NameRef.Type = tType.Int;
      Scan.NextLex();
   }
   while( Scan.Lex == tLex.lexComma ) {
      Scan.NextLex();
      if( Scan.Lex != tLex.lexName )
         Error.Expected("имя");
      else {
         NameRef = Table.NewName(Scan.Name, tCat.Var );
         NameRef.Type = tType.Int;
         Scan.NextLex();
      }
   }
   Check(tLex.lexColon, "\":\"");
   ParseType();
}
```

```
// { CONST {ОбъявлКонст ";"} | VAR {ОбъявлПерем ";"} }
static void DeclSeq() {
   while( Scan.Lex == tLex.lexCONST ||
      Scan.Lex == tLex.lexVAR )
   {
      if( Scan.Lex == tLex.lexCONST ) {
         Scan.NextLex();
         while( Scan.Lex == tLex.lexName ) {
            ConstDecl(); // Объявление константы
            Check(tLex.lexSemi, "\";\"");
         }}
      else {
         Scan.NextLex(); // VAR
         while( Scan.Lex == tLex.lexName ) {
            VarDecl();   // Объявление переменных
            Check(tLex.lexSemi, "\";\"");
         }
      }
   }
}

static void IntExpression() {
   tType T = Expression();
   if( T != tType.Int )
      Error.Expected("выражение целого типа");
}

static tType StFunc(int F) {
   switch( F ) {
   case spABS:
      IntExpression();
      Gen.Abs();
      return tType.Int;
   case spMAX:
      ParseType();
      Gen.Cmd(int.MaxValue);
      return tType.Int;
   case spMIN:
      ParseType();
      Gen.Min();
      return tType.Int;
   case spODD:
      IntExpression();
      Gen.Odd();
      return tType.Bool;
   }
   return tType.None; // Чтоб не было предупреждений
}
```

```
// Имя {"(" Выраж | Тип ")"} | Число | "(" Выраж ")"
static tType Factor() {
   Obj X;
   tType T = tType.None;

   if( Scan.Lex == tLex.lexName ) {
      if( (X = Table.Find(Scan.Name)).Cat == tCat.Var ) {
         Gen.Addr(X);    // Адрес переменной
         Gen.Cmd(OVM.cmLoad);
         Scan.NextLex();
         return X.Type;
         }
      else if( X.Cat == tCat.Const ) {
         Gen.Const(X.Val);
         Scan.NextLex();
         return X.Type;
         }
      else if( X.Cat == tCat.StProc &&
         X.Type != tType.None )
      {
         Scan.NextLex();
         Check(tLex.lexLPar, "\"(\"");
         T = StFunc(X.Val);
         Check(tLex.lexRPar, "\")\"");
         }
      else
         Error.Expected(
         "переменная, константа или процедура-функция"
         );
      }
   else if( Scan.Lex == tLex.lexNum ) {
      Gen.Const(Scan.Num);
      Scan.NextLex();
      return tType.Int;
      }
   else if( Scan.Lex == tLex.lexLPar ) {
      Scan.NextLex();
      T = Expression();
      Check(tLex.lexRPar, "\")\"");
      }
   else
      Error.Expected("имя, число или \"(\"");
   return T;
}
```

```
// Множитель {ОперУмн Множитель}
static tType Term() {
   tLex Op;
   tType T = Factor();
   if( Scan.Lex == tLex.lexMult ||
      Scan.Lex == tLex.lexDIV ||
      Scan.Lex == tLex.lexMOD )
   {
      if( T != tType.Int )
         Error.Message(
            "Несоответствие операции типу операнда"
         );
      do {
         Op = Scan.Lex;
         Scan.NextLex();
         if( (T = Factor()) != tType.Int )
            Error.Expected("выражение целого типа");
         switch( Op ) {
         case tLex.lexMult: Gen.Cmd(OVM.cmMult); break;
         case tLex.lexDIV:  Gen.Cmd(OVM.cmDiv); break;
         case tLex.lexMOD:  Gen.Cmd(OVM.cmMod); break;
         }
      } while( Scan.Lex == tLex.lexMult ||
         Scan.Lex == tLex.lexDIV ||
         Scan.Lex == tLex.lexMOD );
   }
   return T;
}

// ["+"|"-"] Слагаемое {ОперСлож Слагаемое}
static tType SimpleExpr() {
   tType T;
   tLex Op;

   if( Scan.Lex == tLex.lexPlus ||
      Scan.Lex == tLex.lexMinus )
   {
      Op = Scan.Lex;
      Scan.NextLex();
      if( (T = Term()) != tType.Int )
         Error.Expected("выражение целого типа");
      if( Op == tLex.lexMinus )
         Gen.Cmd(OVM.cmNeg);
      }
   else
      T = Term();
   if( Scan.Lex == tLex.lexPlus ||
      Scan.Lex == tLex.lexMinus )
   {
```

```
      if( T != tType.Int )
         Error.Message(
            "Несоответствие операции типу операнда"
         );
      do {
         Op = Scan.Lex;
         Scan.NextLex();
         if( (T = Term()) != tType.Int )
            Error.Expected("выражение целого типа");
         switch(Op) {
         case tLex.lexPlus:  Gen.Cmd(OVM.cmAdd); break;
         case tLex.lexMinus: Gen.Cmd(OVM.cmSub); break;
         }
      } while( Scan.Lex == tLex.lexPlus ||
         Scan.Lex == tLex.lexMinus );
   }
   return T;
}

// ПростоеВыраж [Отношение ПростоеВыраж]
static tType Expression() {
   tLex Op;

   tType T = SimpleExpr();
   if(
      Scan.Lex == tLex.lexEQ || Scan.Lex == tLex.lexNE ||
      Scan.Lex == tLex.lexGT || Scan.Lex == tLex.lexGE ||
      Scan.Lex == tLex.lexLT || Scan.Lex == tLex.lexLE
   ){
      Op = Scan.Lex;
      if( T != tType.Int )
         Error.Message(
            "Несоответствие операции типу операнда"
         );
      Scan.NextLex();
      if( (T = SimpleExpr()) != tType.Int )
         Error.Expected("выражение целого типа");
      Gen.Comp(Op);   // Генерация условного перехода
      T = tType.Bool;
   } // иначе тип равен типу первого простого выражения
   return T;
}

// Переменная = Имя
static void Variable() {
   Obj X;

   if( Scan.Lex != tLex.lexName )
      Error.Expected("имя");
```

```
   else {
      if( (X = Table.Find(Scan.Name)).Cat != tCat.Var )
         Error.Expected("имя переменной");
      Gen.Addr(X);
      Scan.NextLex();
   }
}

static void StProc(int sp) {
   switch( sp ) {
   case spDEC:
      Variable();
      Gen.Cmd(OVM.cmDup);
      Gen.Cmd(OVM.cmLoad);
      if( Scan.Lex == tLex.lexComma ) {
         Scan.NextLex();
         IntExpression();
         }
      else
         Gen.Cmd(1);
      Gen.Cmd(OVM.cmSub);
      Gen.Cmd(OVM.cmSave);
      return;
   case spINC:
      Variable();
      Gen.Cmd(OVM.cmDup);
      Gen.Cmd(OVM.cmLoad);
      if( Scan.Lex == tLex.lexComma ) {
         Scan.NextLex();
         IntExpression();
         }
      else
         Gen.Cmd(1);
      Gen.Cmd(OVM.cmAdd);
      Gen.Cmd(OVM.cmSave);
      return;
   case spInOpen:
      // Пусто
      return;
   case spInInt:
      Variable();
      Gen.Cmd(OVM.cmIn);
      Gen.Cmd(OVM.cmSave);
      return;
   case spOutInt:
      IntExpression();
      Check(tLex.lexComma , "\",\"");
      IntExpression();
      Gen.Cmd(OVM.cmOut);
```

```
      return;
   case spOutLn:
      Gen.Cmd(OVM.cmOutLn);
      return;
   case spHALT:
      Gen.Const(ConstExpr());
      Gen.Cmd(OVM.cmStop);
      return;
   }
}

static void BoolExpression() {
   if( Expression() != tType.Bool )
      Error.Expected("логическое выражение");
}

// Переменная "=" Выраж
static void AssStatement() {
   Variable();
   if( Scan.Lex == tLex.lexAss ) {
      Scan.NextLex();
      IntExpression();
      Gen.Cmd(OVM.cmSave);
      }
   else
      Error.Expected("\":=\"");
}

// Имя ["(" // Выраж | Переменная  ")"]
static void CallStatement(int sp) {
   Check(tLex.lexName, "имя процедуры");
   if( Scan.Lex == tLex.lexLPar ) {
      Scan.NextLex();
      StProc(sp);
      Check( tLex.lexRPar, "\")\"" );
      }
   else if( sp == spOutLn || sp == spInOpen )
      StProc(sp);
   else
      Error.Expected("\"(\"");
}

static void IfStatement() {
   int CondPC;
   int LastGOTO;

   Check(tLex.lexIF, "IF");
   LastGOTO = 0;      // Предыдущего перехода нет
   BoolExpression();
```

```
   CondPC = Gen.PC;  // Запомн. положение усл. перехода
   Check(tLex.lexTHEN, "THEN");
   StatSeq();
   while( Scan.Lex == tLex.lexELSIF ) {
      Gen.Cmd(LastGOTO);  // Фикт. адрес, указывающий
      Gen.Cmd(OVM.cmGOTO);// на место предыдущ. перехода
      LastGOTO = Gen.PC;  // Запомнить место GOTO
      Scan.NextLex();
      Gen.Fixup(CondPC);  // Зафикс. адрес усл. перехода
      BoolExpression();
      CondPC = Gen.PC;    // Запомн. полож. усл. перехода
      Check(tLex.lexTHEN, "THEN");
      StatSeq();
   }
   if( Scan.Lex == tLex.lexELSE ) {
      Gen.Cmd(LastGOTO);  // Фиктивный адрес, указывающий
      Gen.Cmd(OVM.cmGOTO);// на место предыдущ. перехода
      LastGOTO = Gen.PC;  // Запомн. место последн. GOTO
      Scan.NextLex();
      Gen.Fixup(CondPC);  // Зафикс. адрес усл. перехода
      StatSeq();
      }
   else
      Gen.Fixup(CondPC);  // Если ELSE отсутствует
   Check( tLex.lexEND, "END" );
   Gen.Fixup(LastGOTO);   // Направить сюда все GOTO
}

static void WhileStatement() {
   int WhilePC, CondPC;

   WhilePC = Gen.PC;
   Check(tLex.lexWHILE, "WHILE");
   BoolExpression();
   CondPC = Gen.PC;
   Check(tLex.lexDO, "DO");
   StatSeq();
   Check(tLex.lexEND, "END");
   Gen.Cmd(WhilePC);
   Gen.Cmd(OVM.cmGOTO);
   Gen.Fixup(CondPC);
}

static void Statement() {
   Obj X;

   if( Scan.Lex == tLex.lexName ) {
```

```
      if( (X=Table.Find(Scan.Name)).Cat == tCat.Module )
      {
         Scan.NextLex();
         Check(tLex.lexDot, "\".\"");
         if(
            Scan.Lex == tLex.lexName &&
               X.Name.Length + Scan.Name.Length <=
                  Scan.NAMELEN
         )
            X = Table.Find( X.Name + "." + Scan.Name);
         else
            Error.Expected("имя из модуля " + X.Name);
      }
      if( X.Cat == tCat.Var )
         AssStatement();        // Присваивание
      else if(
         X.Cat == tCat.StProc &&
         X.Type == tType.None
      )
         CallStatement(X.Val); // Вызов процедуры
      else
         Error.Expected(
            "обозначение переменной или процедуры"
         );
      }
   else if( Scan.Lex == tLex.lexIF )
      IfStatement();
   else if( Scan.Lex == tLex.lexWHILE )
      WhileStatement();
   // иначе пустой оператор
}

// Оператор {";" Оператор}
static void StatSeq() {
   Statement();    // Оператор
   while( Scan.Lex == tLex.lexSemi ) {
      Scan.NextLex();
      Statement(); // Оператор
   }
}

static void ImportModule() {
   if( Scan.Lex == tLex.lexName ) {
      Table.NewName(Scan.Name, tCat.Module);
      if( Scan.Name == "In" ) {
         Table.Enter("In.Open",
            tCat.StProc, tType.None, spInOpen);
         Table.Enter("In.Int",
            tCat.StProc, tType.None, spInInt);
```

```
         }
      else if( Scan.Name == "Out" ) {
         Table.Enter("Out.Int",
            tCat.StProc, tType.None, spOutInt);
         Table.Enter("Out.Ln",
            tCat.StProc, tType.None, spOutLn);
         }
      else
         Error.Message("Неизвестный модуль");
      Scan.NextLex();
      }
   else
      Error.Expected("имя импортируемого модуля");
}

// IMPORT Имя { "," Имя } ";"
static void Import() {
   Check(tLex.lexIMPORT, "IMPORT");
   ImportModule();
   while( Scan.Lex == tLex.lexComma ) {
      Scan.NextLex();
      ImportModule();
   }
   Check(tLex.lexSemi, "\";\"");
}

// MODULE Имя ";" [Импорт] ПослОбъявл
// [BEGIN ПослОператоров]
// END Имя "."
static void Module() {
   Obj ModRef; // Ссылка на имя модуля в таблице

   Check(tLex.lexMODULE, "MODULE");
   if( Scan.Lex != tLex.lexName )
      Error.Expected("имя модуля");
   // Имя модуля - в таблицу имен
      ModRef = Table.NewName(Scan.Name, tCat.Module);
   Scan.NextLex();
   Check(tLex.lexSemi, "\";\"");
   if( Scan.Lex == tLex.lexIMPORT )
      Import();
   DeclSeq();
   if( Scan.Lex == tLex.lexBEGIN ) {
      Scan.NextLex();
      StatSeq();
   }
   Check(tLex.lexEND, "END");
```

```
      // Сравнение имени модуля и имени после END
         if( Scan.Lex != tLex.lexName )
            Error.Expected("имя модуля");
         else if( Scan.Name != ModRef.Name )
            Error.Expected(
               "имя модуля \"" + ModRef.Name + "\""
            );
         else
            Scan.NextLex();
      if( Scan.Lex != tLex.lexDot )
         Error.Expected("\".\"");
      Gen.Cmd(0);              // Код возврата
      Gen.Cmd(OVM.cmStop);     // Команда останова
      Gen.AllocateVariables(); // Размещение переменных
   }

   public static void Compile() {
      Table.Init();
      Table.OpenScope(); // Блок стандартных имен
      Table.Enter("ABS", tCat.StProc, tType.Int, spABS);
      Table.Enter("MAX", tCat.StProc, tType.Int, spMAX);
      Table.Enter("MIN", tCat.StProc, tType.Int, spMIN);
      Table.Enter("DEC", tCat.StProc, tType.None, spDEC);
      Table.Enter("ODD", tCat.StProc, tType.Bool, spODD);
      Table.Enter("HALT", tCat.StProc, tType.None, spHALT);
      Table.Enter("INC", tCat.StProc, tType.None, spINC);
      Table.Enter("INTEGER", tCat.Type, tType.Int, 0);
      Table.OpenScope();  // Блок модуля
      Module();
      Table.CloseScope(); // Блок модуля
      Table.CloseScope(); // Блок стандартных имен
      System.Console.WriteLine("\nКомпиляция завершена");
   }

   }
```

Листинг П6.6. Файл Table.cs

```
   using System;

   // Элемент таблицы имен
   public class Obj {      // Тип записи таблицы имен
      public string  Name; // Ключ поиска
      public tCat    Cat;  // Категория имени
      public tType   Type; // Тип
      public int     Val;  // Значение
      public Obj     Prev; // Указатель на пред. имя
   }
```

```
// Категории имён
   public enum tCat {
      Const, Var, Type, StProc, Module, Guard
   };

// Типы
   public enum tType {
      None, Int, Bool
   }

// Таблица имен
class Table {

private static Obj Top;    // Указатель на вершину списка
private static Obj Bottom; // Указатель на дно списка
private static Obj CurrObj;

// Инициализация таблицы
public static void Init() {
   Top = null;
}

// Добавление элемента
public static void Enter(string N, tCat C, tType T, int
V) {
   Obj P = new Obj();
   P.Name = String.Copy(N);
   P.Cat = C;
   P.Type = T;
   P.Val = V;
   P.Prev = Top;
   Top = P;
}

public static void OpenScope() {
   Enter("", tCat.Guard, tType.None, 0);
   if ( Top.Prev == null )
      Bottom = Top;
}

public static void CloseScope() {
   while( Top.Cat != tCat.Guard ){
      Top = Top.Prev;
   }
   Top = Top.Prev;
}

public static Obj NewName(string Name, tCat cat) {
   Obj obj = Top;
```

```
   while( obj.Cat != tCat.Guard && obj.Name != Name )
      obj = obj.Prev;
   if ( obj.Cat == tCat.Guard ) {
      obj = new Obj();
      obj.Name = String.Copy(Name);
      obj.Cat = cat;
      obj.Val = 0;
      obj.Prev = Top;
      Top = obj;
      }
   else
      Error.Message("Повторное объявление имени");
   return obj;
}

public static Obj Find(string Name) {
   Obj obj;

   Bottom.Name = String.Copy(Name);
   for( obj=Top; obj.Name != Name; obj=obj.Prev );
   if( obj == Bottom )
      Error.Message("Необъявленное имя");
   return obj;
}

public static Obj FirstVar() {
   CurrObj = Top;
   return NextVar();
}

public static Obj NextVar() {
   Obj VRef;

   while( CurrObj != Bottom && CurrObj.Cat != tCat.Var )
      CurrObj = CurrObj.Prev;
   if( CurrObj == Bottom )
      return null;
   else {
      VRef = CurrObj;
      CurrObj = CurrObj.Prev;
      return VRef;
   }
}

}
```

Листинг П6.7. Файл OVM.cs

```
// Виртуальная машина
using System;

class OVM {

const int  MEMSIZE = 8*1024;

public const int
   cmStop    = -1,
   cmAdd     = -2,
   cmSub     = -3,
   cmMult    = -4,
   cmDiv     = -5,
   cmMod     = -6,
   cmNeg     = -7,

   cmLoad    = -8,
   cmSave    = -9,

   cmDup     = -10,
   cmDrop    = -11,
   cmSwap    = -12,
   cmOver    = -13,

   cmGOTO    = -14,
   cmIfEQ    = -15,
   cmIfNE    = -16,
   cmIfLE    = -17,
   cmIfLT    = -18,
   cmIfGE    = -19,
   cmIfGT    = -20,

   cmIn      = -21,
   cmOut     = -22,
   cmOutLn   = -23;

public static int[] M = new int[MEMSIZE];

public static void Run() {
   int PC = 0;
   int SP = MEMSIZE;
   int Cmd;
   int Buf;

   while( (Cmd = M[PC++]) != cmStop )
      if ( Cmd >= 0 )
         M[--SP] = Cmd;
```

```
else
   switch( Cmd ) {
   case cmAdd:
      SP++; M[SP] += M[SP-1];
      break;
   case cmSub:
      SP++; M[SP] -= M[SP-1];
      break;
   case cmMult:
      SP++; M[SP] *= M[SP-1];
      break;
   case cmDiv:
      SP++; M[SP] /= M[SP-1];
      break;
   case cmMod:
      SP++; M[SP] %= M[SP-1];
      break;
   case cmNeg:
      M[SP] = -M[SP];
      break;
   case cmLoad:
      M[SP] = M[M[SP]];
      break;
   case cmSave:
      M[M[SP+1]] = M[SP];
      SP += 2;
      break;
   case cmDup:
      SP--; M[SP] = M[SP+1];
      break;
   case cmDrop:
      SP++;
      break;
   case cmSwap:
      Buf = M[SP]; M[SP] = M[SP+1]; M[SP+1] = Buf;
      break;
   case cmOver:
      SP--; M[SP] = M[SP+2];
      break;
   case cmGOTO:
      PC = M[SP++];
      break;
   case cmIfEQ:
      if ( M[SP+2] == M[SP+1] )
         PC = M[SP];
      SP += 3;
      break;
   case cmIfNE:
      if ( M[SP+2] != M[SP+1] )
```

```
            PC = M[SP];
         SP += 3;
         break;
      case cmIfLE:
         if ( M[SP+2] <= M[SP+1] )
            PC = M[SP];
         SP += 3;
         break;
      case cmIfLT:
         if ( M[SP+2] < M[SP+1] )
            PC = M[SP];
         SP += 3;
         break;
      case cmIfGE:
         if ( M[SP+2] >= M[SP+1] )
            PC = M[SP];
         SP += 3;
         break;
      case cmIfGT:
         if ( M[SP+2] > M[SP+1] )
            PC = M[SP];
         SP += 3;
         break;
      case cmIn:
         Console.Write('?');
         M[--SP] = int.Parse(Console.ReadLine());
         break;
      case cmOut:
         int w = M[SP] - M[SP+1].ToString().Length;
         for( int i = 1; i <= w; i++ )
            Console.Write(" ");
         Console.Write(M[SP+1]);
         SP += 2;
         break;
      case cmOutLn:
         Console.WriteLine();
         break;
      default:
         Console.WriteLine(
            "Недопустимый код операции"
         );
         M[PC] = cmStop;
         break;
      }
Console.WriteLine();
```

```
      if( SP < MEMSIZE )
         Console.WriteLine("Код возврата " + M[SP]);
      Console.Write("Нажмите ВВОД");
      Console.ReadLine();
   }

   }
```

Листинг П6.8. Файл Gen.cs

```
   // Генератор кода
   using System;

   class Gen {

   public static int PC;

   public static void Init() {
      PC = 0;
   }

   public static void Cmd(int Cmd) {
      OVM.M[PC++] = Cmd;
   }

   public static void Fixup(int A) {
      while( A > 0 ) {
         int temp = OVM.M[A-2];
         OVM.M[A-2] = PC;
         A = temp;
      }
   }

   public static void Abs() {
      Cmd(OVM.cmDup);
      Cmd(0);
      Cmd(PC+3);
      Cmd(OVM.cmIfGE);
      Cmd(OVM.cmNeg);
   }

   public static void Min() {
      Cmd(int.MaxValue);
      Cmd(OVM.cmNeg);
      Cmd(1);
      Cmd(OVM.cmSub);
   }
```

```
public static void Odd() {
   Cmd(2);
   Cmd(OVM.cmMod);
   Cmd(0);
   Cmd(0); // Адрес перехода вперед
   Cmd(OVM.cmIfEQ);
}

public static void Const(int C) {
   Cmd(Math.Abs(C));
   if ( C < 0 )
      Cmd(OVM.cmNeg);
}

public static void Comp(tLex Lex) {
   Cmd(0); // Адрес перехода вперед
   switch(Lex) {
   case tLex.lexEQ : Cmd(OVM.cmIfNE); break;
   case tLex.lexNE : Cmd(OVM.cmIfEQ); break;
   case tLex.lexLE : Cmd(OVM.cmIfGT); break;
   case tLex.lexLT : Cmd(OVM.cmIfGE); break;
   case tLex.lexGE : Cmd(OVM.cmIfLT); break;
   case tLex.lexGT : Cmd(OVM.cmIfLE); break;
   }
}

public static void Addr(Obj X) {
   Cmd(X.Val);   // В текущую ячейку адрес предыдущей + 2
   X.Val = PC+1; // Адрес+2 = PC+1
}

public static void AllocateVariables() {
   Obj VRef; // Ссылка на переменную в таблице имен

   VRef = Table.FirstVar(); // Найти первую переменную
   while( VRef != null ) {
      if ( VRef.Val == 0 )
         Error.Warning(
         "Переменная " + VRef.Name + " не используется"
         );
      else {
         Fixup(VRef.Val);     // Адресная привязка
         PC++;
      }
      VRef = Table.NextVar(); // Найти след. переменную
   }
}

}
```

ЛИТЕРАТУРА

1. Wirth N. Compiler Construction. Addison Wesley, 1996. — 176 с.
2. Ахо А., Сети Р., Ульман Д. Компиляторы: принципы, технологии и инструменты.: Пер. с англ. — М.: Издательский дом «Вильямс», 2001. — 768 с.: ил.
3. Ахо А., Сети Р., Лам М., Ульман Д. Компиляторы: принципы, технологии и инструментарий.: Пер. с англ. — М.: ООО «И. Д. Вильямс», 2008. — 1184 с.: ил.
4. Баранов С. Н., Ноздрунов Н. Р. Язык Форт и его реализации. — Л.: Машиностроение. Ленингр. от-ние, 1988. — 157 с., ил.
5. Бек Л. Введение в системное программирование: Пер. с англ. — М.: Мир, 1988. — 448 с., ил.
6. Богданов В. В. и др. Программирование на языке АЛМО. Под общ. ред. С. С. Камынина и Э. З. Любимского. М., «Статистика», 1976. 118 с. с ил.
7. Вирт Н. Алгоритмы + структуры данных = программы./ Пер. с англ. — М.: Мир, 1985.
8. Вирт Н. Построение компиляторов/ Пер. с англ. Борисов Е. В., Чернышов Л. Н. – М.: ДМК Пресс, 2010. – 192 с.: ил.
9. Глушков В. М., Цейтлин Г. Е., Ющенко Е. Л. Алгебра, языки, программирование. Киев, «Наукова думка», 1974.
10. Гордеев А. В. Молчанов А. Ю. Системное программное обеспечение. СПб.: Питер, 2001. — 736 с. ил.
11. Грис Д. Конструирование компиляторов для цифровых вычислительных машин.: Пер. с англ. — М.: Мир, 1975.
12. Залогова Л. А. Разработка Паскаль-компилятора: Учебное пособие по спецкурсу / Перм. ун-т. Пермь, 1993. 120 с.

13. Зелковиц М. Шоу А., Гэннон Дж. Принципы разработки программного обеспечения: Пер. с англ. — М.: Мир, 1982 — 368 с., ил.

14. Карпов Ю. Г. Основы построения компиляторов. Учебное пособие. — Л., изд. ЛПИ, 1982.

15. Карпов Ю. Г. Теория автоматов. — СПб.: Питер, 2002. — 224 с.,: ил.

16. Касьянов В. Н., Поттосин И. В. Методы построения трансляторов. — Новосибирск: Наука, 1986. — 344 с.

17. Касьянов В. П. Оптимизирующие преобразования программ. — М.: наука. Гл. ред. физ.-мат. лит., 1988. — 336 с.

18. Керниган Б., Ритчи Д. Язык програмирования Си.\ Пер. с англ., 3-е изд., испр. — СПб.: «Невский Диалект», 2001. — 352 с.: ил.

19. Компаниец Р. И., Маньков Е. В., Филатов Н. Е. Системное программирование. Основы построения трансляторов. / Учебное пособие для высших и средних учебных заведений. — СПб.: КОРОНА принт, 2000. — 256 с.

20. Костельцев А. В. Построение интерпретаторов и компиляторов. СПб: Наука и Техника, 2001. — 224 стр. с ил.

21. Лавров С. С. Программирование. Математические основы, средства, теория. СПб.: БХВ-Петербург, 2001. — 320 с.: ил.

22. Лебедев В. Н. Введение в системы программирования. М., «Статистика», 1975. 312 с. ил.

23. Матиясевич Ю. В., Терехов А. Н. 16-разрядная виртуальная ЭВМ, ориентированная на АЯВУ. Программирование микропроцессорной техники. Таллин, 1984 г., стр. 68-72

24. Матиясевич Ю. В., Терехов А. Н., Федотов Б. А. Унификация программногообеспечения микроЭВМ на базе виртуальной машины. Автоматика и телемеханика №5, М., 1990 г.

25. Молчанов А. Ю. Системное программное обеспечение. Учебник для вузов / СПб.: Питер, 2003. — 396 с.: ил.

26. Пратт Т. Языки программирования: разработка и реализация. Пер. с англ. — М.: Мир, 1979.

27. Пратт Т., Зелковиц М. Языки программирования: разработка и реализация / Под общей редакцией А. Матросова. — СПб.: Питер, 2002. — 688 с.: ил.

28. Рейуорд-Смит В. Дж. Теория формальных языков. Вводный курс: Пер. с англ. — М.: радио и связь, 1988. — 128 с.: ил.

29. Сафонов В. О. Языки и методы программирования в системе «Эльбрус»/ Под ред. С. С. Лаврова. — М.: Наука. Гл. ред. физ.-мат. лит., 1989. — 392 с.

30. Свердлов С. З. Арифметика синтаксиса. PC Week/RE №42-43, 1998.

31. Свердлов С. З. Бейсик для микро-ЭВМ "Искра-1256", Микропроцессорные средства и системы, №2, 1988.

32. Свердлов С. З. Введение в методы трансляции. Учебное пособие. Вологда, "Русь", 1994.

33. Свердлов С. З. Оберон — воплощение мечты Никлауса Вирта. Компьютерра, №46 (173) 25 ноября 1996.

34. Свердлов С. З. Маленький большой язык Оберон. PC Week/RE, №35 (109) от 9/9/1997.

35. Свердлов С. З. Язык программирования Си#: критическая оценка. PC Week/RE №20, 22, 2001.

36. Свердлов С. З. Языки программирования и методы трансляции: Учебное пособие. - СПб.: Питер, 2007. - 638 с.: ил.

37. Свердлов С. З., Хивина А. А. О структурировании синтаксических диаграмм // Вестник Вологодского государственного педагогического университета, №3, 2008. Серия «Физико-математические и естественные науки».

38. Серебряков В. А., Галочкин М. П. Основы конструирования компиляторов. — М.: Эдиториал УРСС, 2001. — 224 с.

39. Соколов А. П. Системы программирования: теория, методы, алгоритмы: Учеб. пособие. — М.: Финансы и статистика, 2004. — 330 с.: ил.

40. Хамахер К. Врашевич З., Заки С. Организация ЭВМ. 5-е изд. — СПб.: Питер; Киев: Издательская группа BHV, 2003. — 848 с.: ил.

41. Хантер Р. Основные концепции компиляторов.: Пер. с англ. — М.: Издательский дом «Вильямс», 2002. — 256 с.: ил.

42. Хантер Р. Проектирование и конструирование компиляторов/ Пер. с англ.: Предисл. В. М. Савинкова. — М.: Финансы и статистика, 1984. — 232 с., ил.

43. Хендрикс Д. Компилятор языка Си для микроЭВМ: Пер. с англ. — М.: радио и связь, 1989. — 240 с.: ил.

44. Хопкрофт Д., Мотвани Р., Ульман Д. Введение в теорию автоматов, языков и вычислений, 2-е изд., : Пер. с англ. — М., Издательский дом «Вильямс», 2002. — 528 с. : ил.