Учебник для вузов

ПИТЕР

А. Ю. Молчанов

Системное программное обеспечение

3-е издание

Допущено
Министерством образования и науки РФ

А. Ю. Молчанов

Системное программное обеспечение

3-е издание

Допущено Министерством образования и науки Российской Федерации в качестве учебника для студентов высших учебных заведений, обучающихся по специальностям «Вычислительные машины, комплексы, системы и сети» и «Автоматизированные системы обработки информации и управления» направления подготовки дипломированных специалистов «Информатика и вычислительная техника»

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2010

ББК 32.973-018я7
УДК 004.45(075)
М76

Рецензенты:

Немолочнов О. Ф., завкафедрой информатики и прикладной математики Санкт-Петербургского университета информационных технологий, механики и оптики, доктор технических наук, профессор;

Яшин А. И., доктор технических наук, профессор кафедры автоматизированных систем обработки информации и управления СПбГЭТУ «ЛЭТИ»

Молчанов А. Ю.

М76 Системное программное обеспечение: Учебник для вузов. 3-е изд. — СПб.: Питер, 2010. — 400 с.: ил.

ISBN 978-5-49807-153-4

В книге рассматриваются основные теоретические принципы и реализующие их технологии, лежащие в основе современных средств разработки программного обеспечения. В ней содержится вся необходимая информация о трансляторах, компиляторах, интерпретаторах, а также о других составляющих систем программирования, от базовых теоретических сведений до современных технологий разработки распределенных программ. Третье издание полностью отвечает содержанию дисциплины «Системное программное обеспечение» по новой версии стандарта. Автор постарался придать материалу практическую направленность по сравнению с предыдущей редакцией книги. С одной стороны, это позволило сократить теоретические разделы книги, не отклоняясь от требований образовательного стандарта, что, по мнению автора, должно способствовать лучшему усвоению учебного материала, а с другой стороны, книга может оказаться полезной не только студентам, но и специалистам, чья деятельность напрямую связана с созданием средств обработки текстов и структурированных текстовых команд.

Допущено Министерством образования и науки Российской Федерации в качестве учебника для студентов высших учебных заведений, обучающихся по специальностям «Вычислительные машины, комплексы, системы и сети» и «Автоматизированные системы обработки информации и управления» направления подготовки дипломированных специалистов «Информатика и вычислительная техника».

ББК 32.973-018я7
УДК 004.45(075)

ISBN 978-5-49807-153-4

Краткое содержание

Содержание

Предисловие

Эта книга является новым изданием учебника «Системное программное обеспечение», который вышел в свет в 2003 году, а первая версия этого учебника была выпущена еще в 2001 году. Главной целевой аудиторией книги «Системное программное обеспечение» были и остаются студенты технических вузов, обучающиеся по специальности «Вычислительные машины, комплексы, системы и сети» и родственным с ней направлениям. Поэтому материал данной книги подобран исходя из требований стандарта этой специальности.

После выхода в свет издания 2003 года автором данной книги был подготовлен и опубликован в издательстве «Питер» лабораторный практикум по курсу «Системное программное обеспечение»[42], в котором уделено особое внимание практическим вопросам, связанным с реализацией компиляторов и их отдельных частей. Поэтому в данном издании книги автор счел возможным сократить некоторую часть практического материала, оставив только ссылки на лабораторный практикум.

Кроме того, в этом издании автор упростил ряд теоретических выкладок и алгоритмов, оставив для интересующихся ссылки на соответствующую литературу. Это должно облегчить восприятие материала книги.

В остальном же первые 4 главы данного издания остались без существенных изменений по сравнению с изданием 2003 года.

Однако в главу 5 были внесены изменения, а главу 6, посвященную современным системам программирования, по сути, пришлось написать заново, дополнив ее новым материалом. Это вызвано тем, что технологии создания программных систем, а особенно систем, связанных с распределенными вычислениями, претерпели изменения по сравнению с 2003 годом. Поэтому автор счел необходимым уделить в книге больше внимания современным системам программирования, несколько сократив объем материала, в котором излагаются теоретические основы построения компиляторов. Это стало возможным еще и по той причине, что сейчас появилось достаточное количество изданий, связанных с теорией компиляции, и дефицит литературы по этому вопросу в настоящее время не такой острый, как был в 2003 и, тем более, в 2001 году.

Данный учебник, как и предыдущие издания, посвящен компиляторам и системам программирования, что полностью отвечает содержанию дисциплины «Системное программное обеспечение» по последней редакции образовательного стандарта специальности «Вычислительные машины, комплексы, системы и сети». Эта книга может оказаться полезной не только студентам, но и специалистам, чья деятельность напрямую связана с созданием средств обработки текстов и структурированных текстовых команд.

Введение

Традиционная архитектура компьютера (архитектура фон Неймана) остается неизменной и преобладает в современных вычислительных системах. Столь же неизменными остаются и базовые принципы, на основе которых строятся средства разработки программного обеспечения для компьютеров — трансляторы, компиляторы и интерпретаторы. Но современные средства разработки, оставаясь на тех же базовых принципах, что и компьютеры традиционной архитектуры, прошли долгий путь совершенствования и развития от командных систем до интегрированных сред и систем программирования. И это обстоятельство нашло отражение в предлагаемом учебнике.

С одной стороны, компьютеры традиционной архитектуры умеют понимать только коды машинных команд. С другой стороны, разработчики не имеют возможности создавать прикладные и системные программы на уровне машинных кодов — слишком велик процент ошибок и непомерно велика трудоемкость такой работы. Поэтому давно возникла потребность в появлении «переводчиков» с различных языков программирования (языков ассемблера и языков высокого уровня) на язык машинных кодов. Т акими переводчиками стали трансляторы . Само слово «транслятор» (translator) в переводе с английского означает не что иное, как «переводчик». Наряду с термином «транслятор» часто употребляется еще термин «компилятор» (compiler), имеющий почти то же значение. Разница между ними объясняется в этой книге, в той главе, где даются точные определения этих понятий; здесь же можно только сказать, что «транслятор» — понятие более широкое, а «компилятор» — более узкое (любой компилятор является транслятором, но не наоборот).

Ныне существует огромное количество разнообразных языков программирования. Все они имеют свою историю, свою область применения, и перечислять даже наиболее известные из них здесь не имеет смысла. По каждому из этих языков программирования существует масса литературы, посвященной именно этому языку и средствам разработки, основанным на нем. Но в этой книге излагаются принципы и технологии, лежащие в основе всех современных языков программирования, поскольку все эти языки построены на одном фундаментальном базисе, который составляет теория формальных языков и грамматик. На этих принципах и технологиях построены все средства разработки, которые в настоящее время являются не просто трансляторами и компиляторами, а комплексами, представляющими собой системы программирования.

Автор надеется, что эта книга будет полезна не только студентам, изучающим данную дисциплину, но также тем, кто создает компиляторы или работает с ними, то есть

подавляющему большинству практикующих программистов. Немаловажно знать инструмент, которым пользуешься, а потому в книге представлено много подсказок и советов, ориентированных прежде всего на разработчиков программ.

Автор выражает благодарность и признательность своей жене Олесе за огромную выдержку и моральную поддержку, благодаря которой он смог закончить работу над новой редакцией книги в самый короткий срок.

От издательства

Подробную информацию о наших книгах вы найдете на веб-сайте издательства http://www.piter.com.

Там же вы можете оставлять ваши отзывы и пожелания. Мы будем рады узнать ваше мнение!

ГЛАВА 1 Формальные языки и грамматики

Языки и цепочки символов. Способы задания языков

Цепочки символов. Операции над цепочками символов

Цепочкой символов (или строкой) называют произвольную упорядоченную конечную последовательность символов, записанных один за другим. Понятие*символа* (или буквы) является базовым в теории формальных языков и не нуждается в определении.

Далее цепочки символов будут обозначаться греческими буквами: α, β, γ.

Цепочка символов — это последовательность, в которую могут входить любые символы. Строка, которую вы сейчас читаете, является примером цепочки, символы в которой — строчные и заглавные русские буквы, знаки препинания и символ пробела. Но цепочка — это необязательно некоторая осмысленная последовательность символов. Последовательность « *аввв..аагрьь,..лл*» — тоже пример цепочки символов.

Цепочка символов — это упорядоченная последовательность символов. Это значит , что для цепочки символов имеют значение три фактора: состав входящих в цепочку символов, их количество, а также порядок символов в цепочке. Поэтому цепочки «*а*» и «*аа*», а также «*аб*» и «*ба*» — это различные цепочки символов.

Цепочки символов α и β *равны* (совпадают), $\alpha = \beta$, если они имеют один и тот же состав символов, одно и то же их количество и одинаковый порядок следования символов в цепочке.

Количество символов в цепочке называют *длиной цепочки*. Длина цепочки символов α обозначается как $|\alpha|$. Очевидно, что если $\alpha = \beta$, то и $|\alpha| = |\beta|$.

Основной операцией над цепочками символов является операция конкатенации (объединения или сложения) цепочек.

Конкатенация (сложение, объединение) двух цепочек символов — это дописывание второй цепочки в конец первой. Конкатенация цепочек α и β обозначается как $\alpha\beta$. Выполнить конкатенацию цепочек просто: например, если α = *аб*, а β = *вг*, то $\alpha\beta$ = *абвг*.

ВНИМАНИЕ

Так как в цепочке важен порядок символов, то очевидно, что для операции конкатенации двух цепочек символов важно, в каком порядке записаны цепочки. Иными словами, конкатенация цепочек символов не обладает свойством коммутативности, то есть в общем случае $\exists\ \alpha$ и β такие что: $\alpha\beta \neq \beta\alpha$ (например, для $\alpha =$ *аб* и $\beta =$ вг: $\alpha\beta =$ *абвг*, а $\beta\alpha =$ *вгаб* и $\alpha\beta \neq \beta\alpha$).

Также очевидно, что конкатенация обладает свойством ассоциативности, то есть $(\alpha\beta)\gamma = \alpha(\beta\gamma)$.

Любую цепочку символов можно представить как конкатенацию составляющих ее частей — разбить цепочку на несколько подцепочек. Такое разбиение можно выполнить несколькими способами произвольным образом. Например, цепочку $\gamma =$ *абвг* можно представить в виде конкатенации цепочек $\alpha =$ *аб* и $\beta =$ *вг* ($\gamma = \alpha\beta$), а можно — в виде конкатенации цепочек $\upsilon =$ *а* и $\omega =$ *бвг* ($\gamma = \upsilon\omega$). Чем длиннее исходная цепочка, тем больше вариантов разбиения ее на составляющие подцепочки.

Если некоторую цепочку символов разбить на составляющие ее подцепочки, а затем заменить одну из подцепочек на любую произвольную цепочку символов, то в результате получится новая цепочка символов. Такое действие называется *заменой*, или *подстановкой,* цепочки. Например, возьмем все ту же цепочку $\gamma =$ *абвг*, разобьем ее на три подцепочки, $\alpha =$ *а*, $\omega =$ *б* и $\beta =$ *вг* ($\gamma = \alpha\omega\beta$), и выполним подстановку цепочки $\upsilon =$ *аба* вместо подцепочки ω. Получим новую цепочку $\gamma' =$ *ааабавг* ($\gamma' = \alpha\upsilon\beta$). Любая подстановка выполняется с помощью разбиения исходной цепочки на подцепочки и операции конкатенации.

Можно выделить еще две операции над цепочками.

Обращение цепочки — это запись символов цепочки в обратном порядке. Обращение цепочки α обозначается как α^R. Если $\alpha =$ «*абвг*», то $\alpha^R =$ «*гвба*». Для операции обращения справедливо следующее равенство $\forall\ \alpha,\beta$: $(\alpha\beta)^R = \beta^R\alpha^R$.

Итерация (повторение) цепочки n раз, где $n \in \mathbf{N}$, $n > 0$ — это конкатенация цепочки самой с собой n раз. Итерация цепочки α n раз обозначается как α^n. Для операции повторения справедливы следующие равенства $\forall\ \alpha$: $\alpha_1 = \alpha$, $\alpha^2 = \alpha\alpha$, $\alpha^3 = \alpha\alpha\alpha$, ... и т. д. Итерация цепочки символов определена и для $n = 0$ — в этом случае результатом итерации будет пустая цепочка символов.

Пустая цепочка символов — это цепочка, не содержащая ни одного символа. Пустая цепочка здесь везде будет обозначаться греческой буквой λ (в литературе ее иногда обозначают латинской буквой e или греческой ε).

Для пустой цепочки справедливы следующие равенства.

1. $|\lambda| = 0$.
2. $\forall\alpha\ \ \lambda\alpha = \alpha\lambda = \alpha$.
3. $\lambda^R = \lambda$.
4. $\forall n \geq 0$: $\lambda^n = \lambda$.
5. $\forall\alpha\ \ \alpha^0 = \lambda$.

Понятие языка. Формальное определение языка

В общем случае язык — это заданный набор символов и правил, устанавливающих способы комбинации этих символов между собой для записи осмысленных текстов.

Основой любого естественного или искусственного языка является алфавит, определяющий набор допустимых символов языка.

Алфавит — это счетное множество допустимых символов языка. Будем обозначать это множество символом **V**. Интересно, что, согласно формальному определению, алфавит необязательно должен быть конечным множеством, но реально все существующие языки строятся на основе конечных алфавитов.

Цепочка символов α является *цепочкой над алфавитом* **V**: $\alpha(\mathbf{V})$, если в нее входят только символы, принадлежащие множеству символов **V**. Для любого алфавита **V** пустая цепочка λ может как являться, так и не являться цепочкой $\lambda(\mathbf{V})$. Это условие оговаривается дополнительно.

Если **V** — некоторый алфавит, то

$\mathbf{V}^+$ — множество всех цепочек над алфавитом **V** без λ;

$\mathbf{V}^*$ — множество всех цепочек над алфавитом **V**, включая λ.

Справедливо равенство: $\mathbf{V}^* = \mathbf{V}^+ \cup \{\lambda\}$.

Языком **L** над алфавитом **V**: $L(\mathbf{V})$ называется некоторое счетное подмножество цепочек конечной длины из множества всех цепочек над алфавитом **V**. Из этого определения следует два вывода: во-первых, множество цепочек языка не обязано быть конечным; во-вторых, хотя каждая цепочка символов, входящая в язык, обязана иметь конечную длину, эта длина может быть сколь угодно большой и формально ничем не ограничена.

Все существующие языки подпадают под это определение. Большинство реальных естественных и искусственных языков содержат бесконечное множество цепочек. Также в большинстве языков длина цепочки ничем не ограничена (например, этот длинный текст — пример цепочки символов русского языка). Цепочку символов, принадлежащую заданному языку, часто называют *предложением* языка, а множество цепочек символов некоторого языка $L(\mathbf{V})$ — множеством предложений этого языка.

Для любого языка $L(\mathbf{V})$ справедливо: $L(\mathbf{V}) \subseteq \mathbf{V}^*$.

Язык $L(\mathbf{V})$ *включает в себя* язык $L'(\mathbf{V})$: $L'(\mathbf{V}) \subseteq L(\mathbf{V})$, если $\forall \alpha \in L(\mathbf{V})$: $\alpha \in L'(\mathbf{V})$. Множество цепочек языка $L'(\mathbf{V})$ является подмножеством множества цепочек языка $L(\mathbf{V})$ (или эти множества совпадают). Очевидно, что оба языка должны строиться на основе одного и того же алфавита.

Два языка, $L(\mathbf{V})$ и $L'(\mathbf{V})$, *совпадают* (эквивалентны): $L'(\mathbf{V}) = L(\mathbf{V})$, если $L'(\mathbf{V}) \subseteq L(\mathbf{V})$ и $L(\mathbf{V}) \subseteq L'(\mathbf{V})$; или, что то же самое, $\forall \alpha \in L'(\mathbf{V})$: $\alpha \in L(\mathbf{V})$ и $\forall \beta \in L(\mathbf{V})$: $\beta \in L'(\mathbf{V})$. Множества допустимых цепочек символов для эквивалентных языков равны.

Два языка, $L(\mathbf{V})$ и $L'(\mathbf{V})$, *почти эквивалентны*: $L'(\mathbf{V}) \cong L(\mathbf{V})$, если $L'(\mathbf{V}) \cup \{\lambda\} = L(\mathbf{V}) \cup \{\lambda\}$. Множества допустимых цепочек символов почти эквивалентных языков могут различаться только на пустую цепочку символов.

Способы задания языков. Синтаксис и семантика языка

Итак, каждый язык — это множество цепочек символов над некоторым алфавитом. Но кроме алфавита язык предусматривает также правила построения допустимых цепочек, поскольку обычно далеко не все цепочки над заданным алфавитом принадлежат языку. Символы могут объединяться в слова или лексемы — элементарные

конструкции языка, на их основе строятся предложения — более сложные конструкции. И те и другие в общем виде являются цепочками символов, но предусматривают некоторые правила построения. Таким образом, необходимо указать эти правила, или, строго говоря, задать язык.

В общем случае язык можно определить тремя способами:

1) перечислением всех допустимых цепочек языка;
2) указанием способа порождения цепочек языка (заданием грамматики языка);
3) определением метода распознавания цепочек языка.

Первый из методов является чисто формальным и на практике не применяется, так как большинство языков содержат бесконечное число допустимых цепочек и перечислить их просто невозможно. Трудно себе представить, чтобы появилась возможность перечислить, например, множество всех правильных текстов на русском языке или всех правильных программ на языке P ascal. Иногда для чисто формальных языков можно перечислить множество входящих в них цепочек, прибегнув к математическим определениям множеств. Однако этот подход уже стоит ближе ко второму способу.

Например, запись $L(\{0,1\}) = \{0^n1^n, n > 0\}$ задает язык над алфавитом $\mathbf{V} = \{0,1\}$, содержащий все последовательности с чередующимися символами 0 и 1, начинающиеся с 0 и заканчивающиеся 1. Видно, что пустая цепочка символов в этот язык не входит Если изменить условие в этом определении с $n > 0$ на $n \geq 0$, то получим почти эквивалентный язык $L'(\{0,1\})$, содержащий пустую цепочку.

Второй способ предусматривает некоторое описание правил, с помощью которых строятся цепочки языка. Т огда любая цепочка, построенная с помощью этих правил из символов алфавита языка, будет принадлежать заданному языку . Например, с правилами построения цепочек символов русского языка вы долго и упорно знакомились в средней школе.

Третий способ предусматривает построение некоторого логического устройства (распознавателя) — автомата, который на входе получает цепочку символов, а на выходе выдает ответ, принадлежит или нет эта цепочка заданному языку . Например, читая этот текст, вы сейчас в некотором роде выступаете в роли распознавателя (надеюсь, что ответ на вопрос о принадлежности текста русскому языку будет положительным).

Говоря о любом языке, можно выделить его синтаксис и семантику . Кроме того, трансляторы имеют дело также с лексическими конструкциями (лексемами), которые задаются лексикой языка. Ниже даны определения всех этих понятий.

Синтаксис языка — это набор правил, определяющий допустимые конструкции языка. Синтаксис определяет «форму языка» — задает набор цепочек символов, которые принадлежат языку. Чаще всего синтаксис языка можно задать в виде строгого набора правил, но полностью это утверждение справедливо только для чисто формальных языков. Даже для большинства языков программирования набор заданных синтаксических конструкций нуждается в дополнительных пояснениях, а синтаксис языков естественного общения вполне соответствует общепринятому мнению о том, что «исключения только подтверждают правило».

Например, любой окончивший среднюю школу может сказать, что строка «3 + 2» является арифметическим выражением, а «3 2 +» — не является. Правда, не каждый задумается при этом, что он оперирует синтаксисом алгебры.

Семантика языка — это раздел языка, определяющий значение предложений языка. Семантика определяет «содержание языка» — задает смысл для всех допустимых цепочек языка. Семантика для большинства языков определяется неформальными методами (отношения между знаками и тем, что они обозначают, изучаются семиотикой). Чисто формальные языки лишены какого-либо смысла. Возвращаясь к примеру, приведенному выше, и используя семантику алгебры, мы можем сказать, что строка «3 + 2» есть сумма чисел 3 и 2, а также что «3 + 2 = 5» — это истинное выражение. Однако изложить любому ученику синтаксис алгебры гораздо проще, чем ее семантику, хотя в случае алгебры семантику как раз можно определить формально.

Лексика — это совокупность слов (словарный запас) языка. Слово, или лексическая единица (лексема) языка, — это конструкция, которая состоит из элементов алфавита языка и не содержит в себе других конструкций. Иначе говоря, лексическая единица может содержать только элементарные символы и не может содержать других лексических единиц.

Лексическими единицами (лексемами) русского языка являются слова русского языка, а знаки препинания и пробелы представляют собой разделители, не образующие лексем. Лексическими единицами алгебры являются числа, знаки математических операций, обозначения функций и неизвестных величин. В языках программирования лексическими единицами являются ключевые слова, идентификаторы, константы, метки, знаки операций; в них также существуют и разделители (запятые, скобки, точки с запятой и т. д.).

Особенности языков программирования

Языки программирования занимают некоторое промежуточное положение между формальными и естественными языками. С формальными языками их объединяют строгие синтаксические правила, на основе которых строятся предложения языка. От языков естественного общения в языки программирования перешли лексические единицы, представляющие собой основные ключевые слова (чаще всего это слова английского языка, но существуют языки программирования, чьи ключевые слова заимствованы из русского и других языков). Кроме того, из алгебры языки программирования переняли основные обозначения математических операций, что также делает их более понятными человеку.

Для задания языка программирования необходимо решить три вопроса:

- определить множество допустимых символов языка;
- определить множество правильных программ языка;
- задать смысл для каждой правильной программы.

Только первые два вопроса полностью или частично удается решить с помощью теории формальных языков. Для решения остальных вопросов приходится прибегать к другим, неформальным методам.

Первый вопрос решается легко. Определяя алфавит языка, мы автоматически определяем множество допустимых символов. Для языков программирования алфавит — это чаще всего тот набор символов, которые можно ввести с клавиатуры. Основу его составляет младшая половина таблицы международной кодировки символов (таблицы ASCII), к которой добавляются символы национальных алфавитов.

Второй вопрос решается в теории формальных языков только частично. Для всех языков программирования существуют правила, определяющие синтаксис языка, но, как уже было сказано, их недостаточно для того, чтобы строго определить все допустимые предложения языков программирования. Дополнительные ограничения накладываются семантикой языка. Эти ограничения оговариваются в неформальном виде для каждого отдельного языка программирования. К таким ограничениям можно отнести необходимость предварительного описания переменных и функций, необходимость соответствия типов переменных и констант в выражениях, формальных и фактических параметров в вызовах функций и др.

Отсюда следует, что практически все языки программирования, строго говоря, не являются формальными языками. И именно поэтому во всех трансляторах, кроме синтаксического разбора и анализа предложений языка, дополнительно предусмотрен семантический анализ.

Третий вопрос в принципе не относится к теории формальных языков, поскольку , как уже было сказано, такие языки лишены какого-либо смысла. Для ответа на него нужно использовать другие подходы.

В следующей главе, посвященной основным принципам построения трансляторов и компиляторов, будут более подробно указаны отличия языков программирования от формальных языков, которые нужно принимать во внимание при создании трансляторов и компиляторов для языков программирования.

Грамматики и распознаватели

Формальное определение грамматики. Форма Бэкуса—Наура

Грамматика — это описание способа построения предложений некоторого языка. Иными словами, грамматика — это математическая система, определяющая язык.

Фактически, определив грамматику языка, мы указываем правила порождения цепочек символов, принадлежащих этому языку . Таким образом, грамматика — это генератор цепочек языка. Она относится ко второму способу определения языков — порождению цепочек символов.

Грамматику языка можно описать различными способами. Например, грамматика русского языка описывается довольно сложным набором правил, которые изучают в начальной школе. Для некоторых языков (в том числе для синтаксических конструкций языков программирования) можно использовать формальное описание грамматики, построенное на основе системы правил (или продукций).

Правило (или продукция) — это упорядоченная пара цепочек символов (α,β). В правилах важен порядок цепочек, поэтому их чаще записывают в виде $\alpha \to \beta$ (или $\alpha ::= \beta$). Такая запись читается как «α порождает β» или «α по определению есть β».

Грамматика языка программирования содержит правила двух типов: первые (определяющие синтаксические конструкции языка) довольно легко поддаются формальному описанию; вторые (определяющие семантические ограничения языка) обычно излагаются в неформальной форме. Поэтому любое описание (или стандарт) языка программирования обычно состоит из двух частей: вначале формально излагаются

правила построения синтаксических конструкций, а потом на естественном языке дается описание семантических правил.

ВНИМАНИЕ

Далее, говоря о грамматиках языков программирования, будем иметь в виду только правила построения синтаксических конструкций языка. Однако следует помнить, что грамматика любого языка программирования в общем случае не ограничивается только этими правилами.

Язык, заданный грамматикой **G**, обозначается как L(**G**).

Две грамматики, **G** и **G'**, называются эквивалентными, если они определяют один и тот же язык: L(**G**) = L(**G'**). Две грамматики, **G** и **G'**, называются почти эквивалентными, если заданные ими языки различаются не более чем на пустую цепочку символов: L(**G**)∪{λ} = L(**G'**)∪{λ}.

Формально грамматика **G** определяется как четверка G(**VT**, **VN**, **P**, S), где

VT — множество терминальных символов, или алфавит терминальных символов;

VN — множество нетерминальных символов, или алфавит нетерминальных символов;

P — множество правил (продукций) грамматики вида $\alpha \to \beta$, где $\alpha \in (\mathbf{VN} \cup \mathbf{VT})^+$, $\beta \in (\mathbf{VN} \cup \mathbf{VT})^*$;

S — целевой (начальный) символ грамматики $S \in \mathbf{VN}$.

Алфавиты терминальных и нетерминальных символов грамматики не пересекаются: $\mathbf{VN} \cap \mathbf{VT} = \varnothing$. Это значит, что каждый символ в грамматике может быть либо терминальным, либо нетерминальным, но не может быть терминальным и нетерминальным одновременно. Целевой символ грамматики — это всегда нетерминальный символ. Множество $\mathbf{V} = \mathbf{VN} \cup \mathbf{VT}$ называют полным алфавитом грамматики **G**.

Далее будут даны строгие формальные описания того, как связаны различные элементы грамматики и порождаемый ею язык. А пока предварительно опишем смысл множеств **VN** и **VT**. Множество терминальных символов **VT** содержит символы, которые входят в алфавит языка, порождаемого грамматикой. Как правило, символы из множества **VT** встречаются только в цепочках правых частей правил. Множество нетерминальных символов **VN** содержит символы, которые определяют слова, понятия, конструкции языка. Каждый символ этого множества может встречаться в цепочках как левой, так и правой частей правил грамматики, но он обязан хотя бы один раз быть в левой части хотя бы одного правила. Правила грамматики строятся так, чтобы в левой части каждого правила был хотя бы один нетерминальный символ.

Во множестве правил грамматики может быть несколько правил, имеющих одинаковые левые части вида, $\alpha \to \beta_1$, $\alpha \to \beta_2$, ... $\alpha \to \beta_n$. Тогда эти правила объединяют вместе и записывают в виде, $\alpha \to \beta_1 \mid \beta_2 \mid ... \mid \beta_n$. Одной строке в такой записи соответствует сразу n правил.

Такую форму записи правил грамматики называют формой Бэкуса—Наура . Форма Бэкуса—Наура, как правило, предусматривает также, что нетерминальные символы берутся в угловые скобки: < >. Иногда знак → в правилах грамматики заменяют на знак ::= (что характерно для старых монографий), но это всего лишь незначительные модификации формы записи, не влияющие на ее суть.

Ниже приведен пример грамматики, которая определяет язык целых десятичных чисел со знаком:

```
G({0,1,2,3,4,5,6,7,8,9,–,+},{<число>,<чс>,<цифра>},P,<число>)
P:
<число> → <чс> | +<чс> | –<чс>
<чс> → <цифра> | <чс><цифра>
<цифра> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Рассмотрим составляющие элементы грамматики **G**:

- множество терминальных символов **VT** содержит двенадцать элементов: десять десятичных цифр и два знака;
- множество нетерминальных символов **VN** содержит три элемента: символы <число>, <чс> и <цифра>;
- множество правил содержит 15 правил, которые записаны в три строки (то есть имеется только три различные правые части правил);
- целевым символом грамматики является символ <число>.

Следует отметить, что символ <чс> — это бессмысленное сочетание букв русского языка, но это обычный нетерминальный символ грамматики, такой же, как и два других. Названия нетерминальных символов не обязаны быть осмысленными, это сделано просто для удобства понимания правил грамматики человеком. В принципе в любой грамматике можно полностью изменить имена всех нетерминальных символов, не меняя при этом языка, заданного грамматикой, — точно так же, например, в программе на языке Pascal можно изменить имена идентификаторов, и при этом не изменится смысл программы.

Для терминальных символов это неверно. Набор терминальных символов всегда строго соответствует алфавиту языка, определяемого грамматикой.

Вот, например, та же самая грамматика для языка целых десятичных чисел со знаком, в которой нетерминальные символы обозначены большими латинскими буквами (далее это будет часто применяться в примерах):

```
G'({0,1,2,3,4,5,6,7,8,9,–,+},{S,T,F},P,S)
P:
S → T | +T | –T
T → F | TF
F → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Здесь изменилось только множество нетерминальных символов. Теперь **VN** = {S,T,F}. Язык, заданный грамматикой, не изменился — можно сказать, что грамматики **G** и **G'** эквивалентны.

Принцип рекурсии в правилах грамматики

Особенность рассмотренных выше формальных грамматик в том, что они позволяют определить бесконечное множество цепочек языка с помощью конечного набора правил (разумеется, множество цепочек языка тоже может быть конечным, но даже для простых реальных языков это условие обычно не выполняется). Приведенная

выше в примере грамматика для целых десятичных чисел со знаком определяет бесконечное множество целых чисел с помощью 15 правил.

В такой форме записи грамматики возможность пользоваться конечным набором правил достигается за счет рекурсивных правил. Рекурсия в правилах грамматики выражается в том, что один из нетерминальных символов определяется сам через себя. Рекурсия может быть непосредственной (явной) — тогда символ определяется сам через себя в одном правиле, либо косвенной (неявной) — тогда то же самое происходит через цепочку правил.

В рассмотренной выше грамматике **G** непосредственная рекурсия присутствует в правиле `<чс>` → `<чс><цифра>`, а в эквивалентной ей грамматике **G'** — в правиле `T` → `TF`.

Чтобы рекурсия не была бесконечной, для участвующего в ней нетерминального символа грамматики должны существовать также и другие правила, которые определяют его, минуя самого себя, и позволяют избежать бесконечного рекурсивного определения (в противном случае этот символ в грамматике был бы просто не нужен). Такими правилами являются `<чс>` → `<цифра>` — в грамматике **G** и `T` → `F` — в грамматике **G'**.

В теории формальных языков более ничего сказать о рекурсии нельзя. Но чтобы полнее понять смысл рекурсии, можно прибегнуть к семантике языка — в рассмотренном выше примере это язык целых десятичных чисел со знаком. Рассмотрим его смысл.

Если попытаться дать определение тому, что же является числом, то начать можно с того, что любая цифра сама по себе есть число. Далее можно заметить, что любые две цифры — это тоже число, затем — три цифры и т. д. Если строить определение числа таким методом, то оно никогда не будет закончено (в математике разрядность числа ничем не ограничена). Однако можно заметить, что каждый раз, порождая новое число, мы просто дописываем цифру справа (поскольку привыкли писать слева направо) к уже написанному ряду цифр. А этот ряд цифр, начиная от одной цифры, тоже, в свою очередь, является числом. Таким образом определение понятия «число» можно построить таким образом: «число — это любая цифра либо другое число, к которому справа дописана любая цифра». Именно это и составляет основу правил грамматик **G** и **G'** и отражено в правилах `<чс>` → `<цифра>` | `<чс><цифра>` и `T` → `F` | `TF` (вторая строка правил). Другие правила в этих грамматиках позволяют добавить к числу знак (первая строка правил) и дают определение понятию «цифра» (третья строка правил). Они элементарны и не требуют пояснений.

Принцип рекурсии (иногда его называют «принцип итерации», что не меняет сути) — важное понятие в представлении о формальных грамматиках. Так или иначе, явно или неявно рекурсия всегда присутствует в грамматиках любых реальных языков программирования. Именно она позволяет строить бесконечное множество цепочек языка, и говорить об их порождении невозможно без понимания принципа рекурсии. Как правило, в грамматике реального языка программирования содержится не одно, а целое множество правил, построенных с помощью рекурсии.

Другие способы задания грамматик

Форма Бэкуса—Наура — удобный с формальной точки зрения, но не всегда доступный для понимания способ записи формальных грамматик. Рекурсивные определе-

ния хороши для формального анализа цепочек языка, но неудобны с точки зрения человека. Например, то, что правила `<чс> → <цифра> | <чс><цифра>` отражают возможность для построения числа дописывать справа любое количество цифр, начиная от одной, неочевидно и требует дополнительного пояснения.

Но при создании языка программирования важно, чтобы его грамматику понимали не только те, кому предстоит создавать компиляторы для этого языка, но и пользователи языка — будущие разработчики программ. Поэтому существуют другие способы описания правил формальных грамматик, которые ориентированы на бóльшую понятность для человека.

Далее рассмотрим два наиболее распространенных из этих способов: запись правил грамматик с использованием метасимволов и запись правил грамматик в графическом виде.

Запись правил грамматик с использованием метасимволов

Запись правил грамматик с использованием метасимволов предполагает, что в строке правила грамматики могут встречаться специальные символы — метасимволы, которые имеют особый смысл и трактуются специальным образом. В качестве таких метасимволов чаще всего используются следующие символы: `( )` (круглые скобки), `[ ]` (квадратные скобки), `{ }` (фигурные скобки), `" "` (кавычки) и `,` (запятая).

Эти метасимволы имеют следующий смысл:

- круглые скобки означают, что из всех перечисленных внутри них цепочек символов в данном месте правила грамматики может стоять только одна цепочка;
- квадратные скобки означают, что указанная в них цепочка может встречаться, а может и не встречаться в данном месте правила грамматики (то есть может быть в нем один раз или ни одного раза);
- фигурные скобки означают, что указанная внутри них цепочка может не встречаться в данном месте правила грамматики ни одного раза, встречаться один раз или сколь угодно много раз[1];
- кавычки используются в тех случаях, когда один из метасимволов нужно включить в цепочку обычным образом — то есть когда одна из скобок или запятая должны присутствовать в цепочке символов языка (если саму кавычку нужно включить в цепочку символов, то ее надо повторить дважды — этот принцип знаком разработчикам программ);
- запятая служит для того, чтобы разделять цепочки символов внутри круглых скобок.

Вот как должны выглядеть правила рассмотренной выше грамматики **G**, если их записать с использованием метасимволов:

```
<число> → [(+,−)]<цифра>{<цифра>}
<цифра> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

[1] Следует заметить, что фигурные скобки в математике также используются для обозначения множеств. Иногда это может вызвать некоторую путаницу.

Вторая строка правил не нуждается в комментариях, а первое правило читается так: «число есть цепочка символов, которая может начинаться с символов + или –, должна содержать дальше одну цифру , за которой может следовать последовательность из любого количества цифр». В отличие от формы Бэкуса—Наура, в форме записи с помощью метасимволов, как видно, во-первых, убран из грамматики малопонятный нетерминальный символ <чс>, а во-вторых — удалось полностью исключить рекурсию. Грамматика в итоге стала более понятной.

Форма записи правил с использованием метасимволов — это удобный и понятный способ представления правил грамматик. Она во многих случаях позволяет полностью избавиться от рекурсии, заменив ее символом итерации{ }. Как будет понятно из дальнейшего материала, эта форма наиболее употребительна для одного из типов грамматик — регулярных грамматик.

Кроме указанных выше метасимволов в целях удобства записи в описаниях грамматик иногда используют и другие метасимволы, при этом предварительно дается разъяснение их смысла. Принцип записи от этого не меняется. Т акже иногда дополняют смысл уже существующих метасимволов. Например, для метасимвола { } существует удобная форма записи, позволяющая ограничить число повторений цепочки символов, заключенной внутри них: $\{\ \}^n$, где $n \in \mathbf{N}$ и $n > 0$. Такая запись означает, что цепочка символов, стоящая в фигурных скобках, может быть повторена от 0 до n раз (не более n раз). Это очень удобный метод наложения ограничений на длину цепочки.

Для рассмотренной выше грамматики **G** таким способом можно, например, записать правила, если предположить, что она должна порождать целые десятичные числа, содержащие не более 15 цифр:

```
<число> → [(+,–)]<цифра>{<цифра>}14
<цифра> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Для записи того же самого ограничения в форме Бэкуса—Наура или в форме с метасимволами потребовалось бы 15 правил.

Запись правил грамматик в графическом виде

При записи правил в графическом виде вся грамматика представляется в форме набора специальным образом построенных диаграмм. Эта форма была предложена при описании грамматики языка Pascal, а затем она получила широкое распространение в литературе. Она доступна не для всех типов грамматик, а только для тех типов, где в левой части правил присутствует не более одного символа, но этого достаточно, чтобы ее можно было использовать для описания грамматик известных языков программирования.

В такой форме записи каждому нетерминальному символу грамматики соответствует диаграмма, построенная в виде направленного графа. Граф имеет следующие типы вершин:

- точка входа (на диаграмме никак не обозначена, из нее просто начинается входная дуга графа);
- нетерминальный символ (на диаграмме обозначается прямоугольником, в который вписано обозначение символа);

- цепочка терминальных символов (на диаграмме обозначается овалом, кругом или прямоугольником с закругленными краями, внутрь которого вписана цепочка);
- узловая точка (на диаграмме обозначается жирной точкой или закрашенным кружком);
- точка выхода (никак не обозначена, в нее просто входит выходная дуга графа).

Каждая диаграмма имеет только одну точку входа и одну точку выхода, но сколько угодно вершин других трех типов. Вершины соединяются между собой направленными дугами графа (линиями со стрелками). Из выходной точки дуги могут только выходить, а во входную точку — только входить. В остальные вершины дуги могут как входить, так и выходить (в правильно построенной грамматике каждая вершина должна иметь как минимум один вход и как минимум один выход).

Чтобы построить цепочку символов, соответствующую какому-либо нетерминальному символу грамматики, надо рассмотреть диаграмму для этого символа. Т огда, начав движение от точки входа, надо двигаться по дугам графа диаграммы через любые вершины вплоть до точки выхода. При этом, проходя через вершину, обозначенную нетерминальным символом, этот символ следует поместить в результирующую цепочку. При прохождении через вершину , обозначенную цепочкой терминальных символов, эти символы также следует поместить в резуль тирующую цепочку. При прохождении через узловые точки диаграммы над результирующей цепочкой никаких действий выполнять не надо. Через любую вершину графа диаграммы, в зависимости от возможного пути движения, можно пройти один раз, ни разу или сколь угодно много раз. Как только мы попадем в точку выхода диаграммы, построение результирующей цепочки закончено.

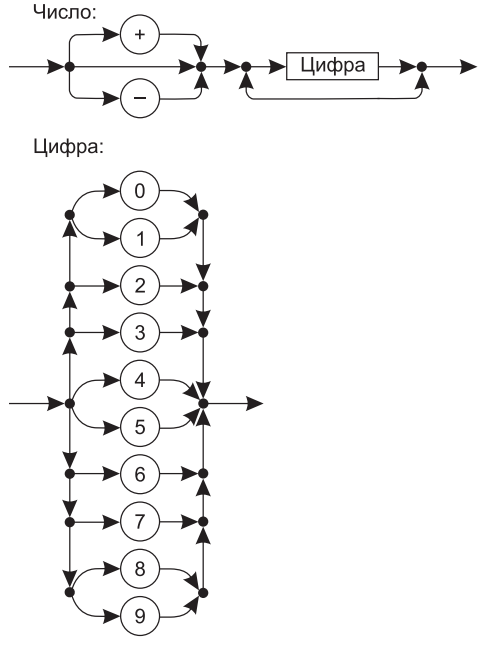

Рис. 1.1. Графическое представление грамматики целых десятичных чисел со знаком (диаграммы для нетерминальных символов <Число> и <Цифра>)

Результирующая цепочка, в свою очередь, может содержать нетерминальные символы. Чтобы заменить их на цепочки терминальных символов, нужно, опять же, рассматривать соответствующие им диаграммы. И так до тех пор, пока в цепочке не останутся только терминальные символы. Очевидно, что для того, чтобы построить цепочку символов заданного языка, надо начать рассмотрение с диаграммы целевого символа грамматики.

Это удобный способ описания правил грамматик, оперирующий образами, а потому ориентированный исключительно на людей. Даже простое изложение его основных принципов здесь оказалось довольно громоздким, в то время как суть способа довольно проста. Это можно легко заметить, если посмотреть на описание понятия «число» из грамматики **G** с помощью диаграмм на рис. 1.1.

Как уже было сказано выше, данный способ в основном применяется в литературе при изложении грамматик языков программирования. Для пользователей — разработчиков программ он удобен, но практического применения в компиляторах пока не имеет.

Существуют и другие способы описания грамматик, но, поскольку они не так часто встречаются в литературе, как два описанных выше способа, в данном учебном пособии они не рассматриваются.

Распознаватели. Общая схема распознавателя

Распознаватель (или разборщик) — это специальный автомат, который позволяет определить принадлежность цепочки символов некоторому языку. Задача распознавателя заключается в том, чтобы на основании исходной цепочки дать ответ на вопрос, принадлежит ли она заданному языку или нет . Распознаватели, как было сказано выше, представляют собой один из способов определения языка.

В общем виде распознаватель можно отобразить в виде условной схемы, представленной на рис. 1.2.

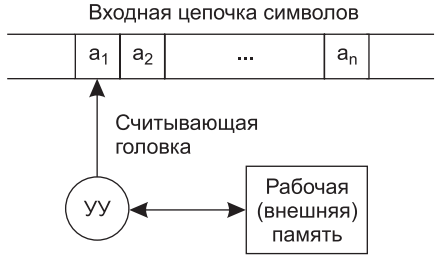

Рис. 1.2. Условная схема распознавателя

Следует подчеркнуть, что представленный рисунок — всего лишь условная схема, отображающая работу алгоритма распознавателя. Ни в коем случае не стоит искать подобного устройства в составе компьютера. Распознаватель, являющийся частью компилятора, представляет собой часть программного обеспечения компьютера.

Как видно из рис. 1.2, распознаватель состоит из следующих основных компонентов:

- входной цепочки символов и считывающей головки (считывающего устройства), обозревающей очередной символ в этой цепочке;

- устройства управления (УУ), которое координирует работу распознавателя, имеет некоторый набор состояний и конечную память (для хранения своего состояния и некоторой промежуточной информации);
- внешней (рабочей) памяти, которая может хранить некоторую информацию в процессе работы распознавателя и, в отличие от памяти УУ, имеет неограниченный объем.

Распознаватель работает с символами своего алфавита — алфавита распознавателя. Алфавит распознавателя конечен. Он включает в себя все допустимые символы входных цепочек, а также некоторый дополнительный алфавит символов, которые могут обрабатываться УУ и храниться в рабочей памяти распознавателя.

В процессе своей работы распознаватель может выполнять некоторые элементарные операции:

- чтение очередного символа из входной цепочки;
- сдвиг входной цепочки на заданное количество символов (вправо или влево);
- доступ к рабочей памяти для чтения или записи информации;
- преобразование информации в памяти УУ, изменение состояния УУ.

То, какие конкретно операции должны выполняться в процессе работы распознавателя, определяется в УУ.

Распознаватель работает по шагам, или тактам. В начале такта, как правило, считывается очередной символ из входной цепочки, и в зависимости от этого символа УУ определяет, какие действия необходимо выполнить. Вся работа распознавателя состоит из последовательности тактов. В начале каждого такта состояние распознавателя определяется его конфигурацией. В процессе работы конфигурация распознавателя меняется.

Конфигурация распознавателя определяется следующими параметрами:

- содержимым входной цепочки символов и положением считывающей головки в ней;
- состоянием УУ;
- содержимым внешней памяти.

Для распознавателя всегда задается определенная конфигурация, которая считается начальной конфигурацией. В начальной конфигурации считывающая головка обозревает первый символ входной цепочки, УУ находится в заданном начальном состоянии, а внешняя память либо пуста, либо содержит строго определенную информацию.

Кроме начального состояния для распознавателя задается одна или несколько конечных конфигураций. В конечной конфигурации считывающая головка, как правило, находится за концом исходной цепочки (часто для распознавателей вводят специальный символ, обозначающий конец входной цепочки).

Распознаватель *допускает входную цепочку символов* α, если, находясь в начальной конфигурации и получив на вход эту цепочку , он может проделать последовательность шагов, заканчивающуюся одной из его конечных конфигураций.

Формулировка «может проделать последовательность шагов» более точна, чем прямое указание «проделает последовательность шагов», так как для многих распозна-

вателей при одной и той же входной цепочке символов из начальной конфигурации могут быть допустимы различные последовательности шагов, не все из которых ведут к конечной конфигурации.

Язык, определяемый распознавателем , — это множество всех цепочек, которые допускает распознаватель.

Далее в этой книге рассмотрены конкретные типы распознавателей для различных языков. Но все, что было сказано здесь, относится ко всем без исключения типам распознавателей для всех типов языков.

Классификация распознавателей по структуре

Распознаватели можно классифицировать в зависимости от вида составляющих их компонентов: считывающего устройства, УУ и внешней памяти.

По видам считывающего устройства распознаватели могут быть двусторонние и односторонние.

Односторонние распознаватели допускают перемещение считывающей головки по ленте входных символов только в одном направлении. Это значит , что на каждом шаге работы распознавателя считывающая головка может либо переместиться по ленте символов на некоторое число позиций в заданном направлении, либо остаться на месте. Поскольку все языки программирования подразумевают нотацию чтения исходной программы «слева направо», то так же работают и все распознаватели. Поэтому, когда говорят об односторонних распознавателях, прежде всего имеют в виду левосторонние, которые читают входную цепочку слева направо и не возвращаются назад к уже прочитанной части цепочки.

Двусторонние распознаватели допускают, что считывающая головка может перемещаться относительно ленты входных символов в обоих направлениях: как вперед, от начала ленты к концу , так и назад, возвращаясь к уже прочитанным символам.

По видам УУ распознаватели бывают детерминированные и недетерминированные.

Распознаватель называется *детерминированным* в том случае, если для каждой допустимой конфигурации распознавателя, которая возникла на некотором шаге его работы, существует единственно возможная конфигурация, в которую распознаватель перейдет на следующем шаге работы.

В противном случае распознаватель называется *недетерминированным*. Недетерминированный распознаватель может иметь такую допустимую конфигурацию, для которой существует некоторое конечное множество конфигураций, возможных на следующем шаге работы. Достаточно иметь хотя бы одну такую конфигурацию, чтобы распознаватель был недетерминированным.

По видам внешней памяти распознаватели бывают следующих типов:

- распознаватели без внешней памяти;
- распознаватели с ограниченной внешней памятью;
- распознаватели с неограниченной внешней памятью.

У *распознавателей без внешней памяти* внешняя память полностью отсутствует . В процессе их работы используется только конечная память УУ.

Для *распознавателей с ограниченной внешней памятью* размер внешней памяти ограничен в зависимости от длины входной цепочки символов. Эти ограничения могут налагаться некоторой зависимостью объема памяти от длины цепочки — линейной, полиномиальной, экспоненциальной и т. д. Кроме того, для таких распознавателей может быть указан способ организации внешней памяти — стек, очередь, список и т. п.

Распознаватели с неограниченной внешней памятью предполагают, что для их работы может потребоваться внешняя память неограниченного объема (вне зависимости от длины входной цепочки). У таких распознавателей предполагается память с произвольным методом доступа.

Вместе эти три составляющие позволяют организовать общую классификацию распознавателей. Например, в этой классификации возможен такой тип: «двусторонний недетерминированный распознаватель с линейно ограниченной стековой памятью».

Тип распознавателя в классификации определяет сложность создания такого распознавателя, а следовательно, сложность разработки соответствующего программного обеспечения для компилятора. Чем выше в классификации стоит распознаватель, тем сложнее создавать алгоритм, обеспечивающий его работу . Разрабатывать двусторонние распознаватели сложнее, чем односторонние. Можно заметить, что недетерминированные распознаватели по сложности выше детерминированных. Зависимость затрат на создание алгоритма от типа внешней памяти также очевидна.

Классификация языков и грамматик

Выше уже упоминались различные типы грамматик, но не было указано, как и по какому принципу они подразделяются на типы. Для человека языки бывают простые и сложные, но это сугубо субъективное мнение, которое в первую очередь зависит от того, какой язык является для человека родным, а также от личности человека.

Для компиляторов языки также можно разделить на простые и сложные, но в данном случае существуют жесткие критерии для такого подразделения. Как будет показано далее, от того, к какому типу относится тот или иной язык программирования, зависит сложность компилятора для этого языка. Чем сложнее язык, тем выше вычислительные затраты компилятора на анализ цепочек исходной программы, написанной на этом языке, а следовательно, сложнее сам компилятор и его структура. Для некоторых типов языков в принципе невозможно построить компилятор, который анализировал бы исходные тексты на этих языках за приемлемое время на основе ограниченных вычислительных ресурсов (именно поэтому до сих пор невозможно создавать программы на естественных языках, например на русском или английском).

Классификация грамматик. Четыре типа грамматик по Хомскому

Согласно классификации, предложенной американским лингвистом Ноамом Хомским, профессором Массачусетского технологического института, формальные грамматики классифицируются по структуре их правил. Если все без исключения правила грамматики удовлетворяют некоторой заданной структуре, то такую грамматику относят к определенному типу. Достаточно иметь в грамматике одно правило, не удовлетворяющее требованиям структуры правил, и она уже не попадает в заданный тип.

По классификации Хомского выделяют четыре типа грамматик.

Тип 0: грамматики с фразовой структурой

На структуру их правил не накладывается никаких ограничений: для грамматики вида G(**VT**,**VN**,**P**,S), **V** = **VN**∪**VT**, правила имеют вид $\alpha \to \beta$, где $\alpha \in V^+$, $\beta \in V^*$.

Это самый общий тип грамматик. В него попадают все без исключения формальные грамматики, но часть из них, к общей радости, может быть также отнесена и к другим классификационным типам. Дело в том, что грамматики, которые относятся только к типу 0 и не могут быть отнесены к другим типам, являются самыми сложными по структуре.

Практического применения грамматики, относящиеся только к типу 0, не имеют.

Тип 1: контекстно-зависимые (КЗ) и неукорачивающие грамматики

В этот тип входят два основных класса грамматик.

Контекстно-зависимые грамматики G(**VT**,**VN**,**P**,S), **V** = **VN**∪**VT**, имеют правила вида $\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$, где $\alpha_1, \alpha_2 \in \mathbf{V}^*$, $A \in \mathbf{VN}$, $\beta \in \mathbf{V}^+$.

Неукорачивающие грамматики G(**VT**,**VN**,**P**,S), **V** = **VN**∪**VT**, имеют правила вида: $\alpha \to \beta$, где $\alpha, \beta \in \mathbf{V}^+$, $|\beta| \geq |\alpha|$.

Структура правил КЗ-грамматик такова, что при построении предложений заданного ими языка один и тот же нетерминальный символ может быть заменен на ту или иную цепочку символов в зависимости от того контекста, в котором он встречается. Именно поэтому эти грамматики называют «контекстно-зависимыми» (КЗ). Цепочки α_1 и α_2 в правилах грамматики обозначают контекст (α_1 — левый контекст, а α_2 — правый контекст), в общем случае любая из них (или даже обе) может быть пустой. Говоря иными словами, значение одного и того же символа может быть различным в зависимости от того, в каком контексте он встречается.

Неукорачивающие грамматики имеют такую структуру правил, что при построении предложений языка, заданного грамматикой, любая цепочка символов может быть заменена на цепочку символов не меньшей длины. Отсюда и название «неукорачивающие».

Доказано, что эти два класса грамматик эквивалентны. Это значит , что для любого языка, заданного КЗ-грамматикой, можно построить неукорачивающую грамматику, которая будет задавать эквивалентный язык, и наоборот: для любого языка, заданного неукорачивающей грамматикой, можно построить КЗ-грамматику, которая будет задавать эквивалентный язык.

При построении компиляторов такие грамматики не применяются, поскольку синтаксические конструкции языков программирования, рассматриваемые компиляторами, имеют более простую структуру и могут быть построены с помощью грамматик других типов. Что касается семантических ограничений языков программирования, то с точки зрения затрат вычислительных ресурсов их выгоднее проверять другими методами, а не с помощью КЗ-грамматик.

Тип 2: контекстно-свободные (КС) грамматики

Неукорачивающие контекстно-свободные (НКС) грамматики G(**VT**,**VN**,**P**,S), **V**=**VN**∪**VT**, имеют правила вида $A \to \beta$, где $A \in \mathbf{VN}$, $\beta \in \mathbf{V}^+$. Такие грамматики называют НКС-грамматиками, поскольку видно, что в правой части правил у них должен всегда стоять как минимум один символ.

Существует также почти эквивалентный им класс грамматик — укорачивающие контекстно-свободные (УКС) грамматики G(**VT**,**VN**,**P**,S), **V** = **VN**∪**VT**, правила которых могут иметь вид: A → β, где A∈ **VN**, β∈ **V***.

Эти два класса составляют тип контекстно-свободных (КС) грамматик. Разница между ними заключается лишь в том, что в УКС-грамматиках в правой части правил может присутствовать пустая цепочка (λ), а в НКС-грамматиках — нет. Отсюда ясно, что язык, заданный НКС-грамматикой, не может содержать пустой цепочки. Доказано, что эти два класса грамматик почти эквивалентны. В дальнейшем, когда речь будет идти о КС-грамматиках, уже не будет уточняться, какой класс грамматики (УКС или НКС) имеется в виду , если возможность наличия в языке пустой цепочки не имеет принципиального значения.

КС-грамматики широко используются при описании синтаксических конструкций языков программирования. Синтаксис большинства известных языков программирования основан именно на КС-грамматиках, поэтому в данном учебнике им уделяется большое внимание.

Внутри типа КС-грамматик кроме классов НКС и УКС выделяют еще целое множество различных классов грамматик, и все они относятся к типу 2. Далее, когда КС-грамматики будут рассматриваться более подробно, на некоторые из этих классов грамматик и их характерные особенности будет обращено особое внимание.

Тип 3: регулярные грамматики

К типу регулярных относятся два эквивалентных класса грамматик: леволинейные и праволинейные .

Леволинейные грамматики G(**VT**,**VN**,**P**,S), **V** = **VN**∪**VT**, могут иметь правила двух видов: A → Bγ или A → γ, где A,B∈ **VN**, γ∈ **VT***.

В свою очередь, праволинейные грамматики G(**VT**,**VN**,**P**,S), **V** = **VN**∪**VT**, могут иметь правила тоже двух видов: A → γB или A → γ, где A,B∈ **VN**, γ∈ **VT***.

Эти два класса грамматик эквивалентны и относятся к типу регулярных грамматик.

Регулярные грамматики используются при описании простейших конструкций языков программирования: идентификаторов, констант, строк, комментариев и т. д. Эти грамматики исключительно просты и удобны в использовании, поэтому в компиляторах на их основе строятся функции лексического анализа входного языка (принципы их построения будут рассмотрены далее).

Классификация языков

Языки классифицируются в соответствии с типами грамматик, с помощью которых они заданы. Причем, поскольку один и тот же язык в общем случае может быть задан сколь угодно большим количеством грамматик, которые могут относиться к различным классификационным типам, для классификации самого языка среди всех его грамматик выбирается грамматика с максимально возможным классификационным типом. Например, если язык **L** может быть задан грамматиками $\mathbf{G}_1$ и $\mathbf{G}_2$, относящимися к типу 1 (КЗ), грамматикой $\mathbf{G}_3$, относящейся к типу 2 (КС), и грамматикой $\mathbf{G}_4$, относящейся к типу 3 (регулярные), сам язык должен быть отнесен к типу 3 и является регулярным языком.

От классификационного типа языка зависит не только то, с помощью какой грамматики можно построить предложения этого языка, но также и то, насколько сложно распознать эти предложения. Распознать предложения — значит построить распознаватель для языка (третий способ задания языка). Классификация распознавателей рассмотрена далее, здесь же можно указать, что сложность распознавателя языка напрямую зависит от классификационного типа, к которому относится язык.

Сложность языка убывает с возрастанием номера классификационного типа языка. Самыми сложными являются языки типа 0, самыми простыми — языки типа 3.

Согласно классификации грамматик, также существует четыре типа языков.

Тип 0: языки с фразовой структурой

Это самые сложные языки, которые могут быть заданы только грамматикой, относящейся к типу 0. Для распознавания цепочек таких языков требуются вычислители, равномощные машине Тьюринга. Поэтому можно сказать, что, если язык относится к типу 0, для него невозможно построить компилятор, который гарантированно выполнял бы разбор предложений языка за ограниченное время на основе ограниченных вычислительных ресурсов.

К сожалению, практически все естественные языки общения между людьми, строго говоря, относятся именно к этому типу языков. Дело в том, что структура и значение фразы естественного языка может зависеть не только от контекста данной фразы, но и от содержания того текста, где эта фраза встречается. Одно и то же слово в естественном языке может не только иметь разный смысл в зависимости от контекста, но и играть различную роль в предложении. Именно поэтому столь велики сложности в автоматизации перевода текстов, написанных на естественных языках, а также отсутствуют (и, видимо, никогда не появятся) компиляторы, которые воспринимали бы программы на основе таких языков.

Далее языки с фразовой структурой рассматриваться не будут.

Тип 1: контекстно-зависимые (КЗ) языки

Тип 1 — второй по сложности тип языков. В общем случае время на распознавание предложений языка, относящегося к типу 1, экспоненциально зависит от длины исходной цепочки символов.

Языки и грамматики, относящиеся к типу 1, применяются в анализе и переводе текстов на естественных языках. Распознаватели, построенные на их основе, позволяют анализировать тексты с учетом контекстной зависимости в предложениях входного языка (но они не учитывают содержание текста, поэтому в общем случае для точного перевода с естественного языка требуется вмешательство человека). На основе таких грамматик может выполняться автоматизированный перевод с одного естественного языка на другой, ими могут пользоваться сервисные функции проверки орфографии и правописания в языковых процессорах.

В компиляторах КЗ-языки не используются, поскольку языки программирования имеют более простую структуру, поэтому здесь они подробно не рассматриваются.

Тип 2: контекстно-свободные (КС) языки

КС-языки лежат в основе синтаксических конструкций большинства современных языков программирования, на их основе функционируют некоторые довольно сложные командные процессоры.

В общем случае время на распознавание предложений языка, относящегося к типу 2, полиномиально зависит от длины входной цепочки символов (в зависимости от класса языка это либо кубическая, либо квадратичная зависимость). Однако среди КС-языков существует много классов языков, для которых эта зависимость линейна. Практически все языки программирования можно отнести к одному из таких классов.

КС-языки подробно рассматриваются в главе «Синтаксические анализаторы» данного учебника.

Тип 3: регулярные языки

Регулярные языки — самый простой тип языков. Поэтому они являются самым широко используемым типом языков в области вычислительных систем. Время на распознавание предложений регулярного языка линейно зависит от длины входной цепочки символов.

Как уже было сказано выше, регулярные языки лежат в основе простейших конструкций языков программирования (идентификаторов, констант и т. п.), кроме того, на их основе строятся многие мнемокоды машинных команд (языки ассемблеров), а также простейшие командные процессоры, символьные управляющие команды и другие подобные структуры.

Регулярные языки — очень удобное средство. Для работы с ними можно использовать регулярные множества и выражения, конечные автоматы. Регулярные языки подробно рассматриваются в главе «Лексические анализаторы».

Задача разбора. Классификация распознавателей по типам

Для каждого языка программирования важно уметь не только построить текст программы на этом языке, но и определить принадлежность имеющегося текста к данному языку. Именно эту задачу решают компиляторы в числе прочих задач (компилятор должен не только распознать исходную программу, но и построить эквивалентную ей результирующую программу). В отношении исходной программы компилятор выступает как распознаватель, а человек, создавший программу на некотором языке программирования, выступает в роли генератора цепочек этого языка.

Грамматики и распознаватели — два независимых метода, которые реально могут быть использованы для определения какого-либо языка. Однако при создании компилятора для некоторого языка программирования возникает задача, которая требует связать между собой эти методы задания языков. Разработчики компилятора всегда имеют дело с уже определенным языком программирования. Грамматика для синтаксических конструкций этого языка известна. Задача разработчиков заключается в том, чтобы построить распознаватель для заданного языка, который затем будет основой синтаксического анализатора в компиляторе.

Таким образом, *задача разбора* в общем виде заключается в следующем: на основе имеющейся грамматики некоторого языка построить распознаватель для этого языка. Заданная грамматика и распознаватель должны быть эквивалентны, то есть определять один и тот же язык (часто допускается, чтобы они были почти эквивалентны, поскольку пустая цепочка во внимание обычно не принимается).

Задача разбора в общем виде может быть решена не для всех языков. Разработчиков компиляторов интересуют прежде всего синтаксические конструкции языков программирования. Для этих конструкций доказано, что задача разбора для них разрешима. Более того, для них найдены формальные методы ее решения. Описанию и обоснованию именно методов решения задачи разбора будет посвящена большая часть материала последующих глав данной книги.

Поскольку языки программирования не являются чисто формальными языками и несут в себе некоторый смысл (семантику), то задача разбора для создания реальных компиляторов понимается несколько шире, чем она формулируется для чисто формальных языков. Компилятор должен не просто установить принадлежность входной цепочки символов заданному языку, но и определить ее смысловую нагрузку. Для этого необходимо выявить те правила грамматики, на основании которых цепочка была построена.

Если же входная цепочка символов не принадлежит заданному языку — исходная программа содержит ошибку — разработчику программы не интересно просто узнать сам факт наличия ошибки. В данном случае задача разбора также расширяется: распознаватель в составе компилятора должен не только установить факт присутствия ошибки во входной программе, но и по возможности определить тип ошибки и то место во входной цепочке символов, где она встречается.

Как было показано ранее, классификация распознавателей определяет сложность алгоритма работы распознавателя. Но сложность распознавателя также напрямую связана с типом языка, входные цепочки которого может принимать (допускать) распознаватель. Выше были определены четыре основных типа языков. Доказано, что для каждого из этих типов языков существует свой тип распознавателя.

Для *языков с фразовой структурой* (тип 0) необходим распознаватель, равномощный машине Тьюринга, — недетерминированный двусторонний автомат , имеющий неограниченную внешнюю память. Поэтому для языков данного типа нельзя гарантировать, что за ограниченное время на ограниченных вычислительных ресурсах распознаватель завершит работу. Отсюда можно заключить, что практического применения языки с фразовой структурой не имеют.

Для *контекстно-зависимых языков* (тип 1) распознавателями являются двусторонние недетерминированные автоматы с линейно ограниченной внешней памятью. Алгоритм работы такого автомата в общем случае имеет экспоненциальную сложность — количество шагов (тактов), необходимых автомату для распознавания входной цепочки, экспоненциально зависит от длины этой цепочки.

Такой алгоритм распознавателя уже может быть реализован в программном обеспечении компьютера — зная длину входной цепочки, всегда можно сказать, за какое максимально возможное время будет принято решение о принадлежности цепочки данному языку и какие вычислительные ресурсы для этого потребуются. Однако экспоненциальная зависимость времени разбора от длины цепочки существенно

ограничивает применение распознавателей для КЗ-языков. Как правило, такие распознаватели применяются для автоматизированного перевода и анализа текстов на естественных языках (следует также напомнить, что, поскольку естественные языки более сложны, чем КЗ-языки, после такой обработки часто требуется вмешательство человека).

В рамках этого учебника КЗ-языки не рассматриваются.

Для *контекстно-свободных языков* (тип 2) распознавателями являются односторонние недетерминированные автоматы с магазинной (стековой) внешней памятью — МП-автоматы. При простейшей реализации алгоритма работы такого автомата он имеет экспоненциальную сложность, однако путем некоторых усовершенствований алгоритма можно добиться полиномиальной (кубической) зависимости времени, необходимого на разбор входной цепочки, от длины этой цепочки. Следовательно, можно говорить о полиномиальной сложности распознавателя для КС-языков.

Среди всех КС-языков можно выделить класс детерминированных КС-языков (ДКС), распознавателями для которых являются детерминированные МП-автоматы — ДМП-автоматы. Для таких ДКС-языков существует алгоритм работы распознавателя с квадратичной сложностью.

Среди всех ДКС-языков существуют такие классы языков, для которых можно построить линейный распознаватель — распознаватель, у которого время разбора цепочки имеет линейную зависимость от длины цепочки. Именно эти языки представляют интерес при построении компиляторов. Синтаксические конструкции всех существующих языков программирования могут быть отнесены к одному из таких классов языков. Поэтому в главе, посвященной синтаксическим анализаторам, в первую очередь будет уделено внимание именно этим классам языков.

Тем не менее следует помнить, что только синтаксические конструкции языков программирования допускают разбор с помощью распознавателей КС-языков. Сами языки программирования, как уже было сказано, не могут быть полностью отнесены к типу КС-языков, поскольку предполагают контекстную зависимость в тексте исходной программы (например, такую, как необходимость предварительного описания переменных). Поэтому кроме синтаксического разбора все компиляторы предполагают дополнительный семантический анализ текста исходной программы. Этого можно было бы избежать, если построить компилятор на основе КЗ-распознавателя, но скорость работы такого компилятора была бы недопустимо низка, поскольку время разбора в таком варианте будет экспоненциально зависеть от длины исходной программы. Комбинация из распознавателя КС-языка и дополнительного семантического анализатора является более эффективной с точки зрения скорости разбора исходной программы.

Для *регулярных языков* (тип 3) распознавателями являются односторонние недетерминированные автоматы без внешней памяти — конечные автоматы(КА). Это очень простой тип распознавателя, который предполагает линейную зависимость времени разбора входной цепочки от ее длины. Кроме того, КА имеют важную особенность: любой недетерминированный КА всегда может быть преобразован в детерминированный. Это обстоятельство существенно упрощает разработку программного обеспечения, обеспечивающего функционирование распознавателя.

Простота и высокая скорость работы распознавателей определяют широкую область применения регулярных языков.

В компиляторах распознаватели на основе регулярных языков используются для лексического анализа текста исходной программы — выделения в нем простейших конструкций языка (лексем), таких как идентификаторы, строки, константы и т. п. Это позволяет существенно сократить объем исходной информации и упрощает синтаксический разбор программы. Более подробно взаимодействие лексического и синтаксического анализаторов текста программы рассмотрено дальше, в главе, посвященной лексическим анализаторам.

Кроме компиляторов, регулярные языки находят применение еще во многих областях, связанных с разработкой программного обеспечения. На их основе функционируют многие командные процессоры как в системном, так и в прикладном программном обеспечении. Для регулярных языков существуют развитые математически обоснованные методы, которые позволяют облегчить создание распознавателей. Они положены в основу существующих программных средств, которые позволяют автоматизировать этот процесс.

Регулярные языки и связанные с ними математические методы рассматриваются в главе «Лексические анализаторы» данного учебника.

Примеры классификации языков и грамматик

Классификация языков идет от простого к сложному Если мы имеем дело с регулярным языком, то можно утверждать, что он также является и КС, и КЗ, и даже языком с фразовой структурой. В то же время известно, что существуют КС-языки, которые не являются регулярными, и существуют КЗ-языки, которые не являются ни регулярными, ни контекстно-свободными.

Далее приводятся примеры некоторых языков указанных типов.

Рассмотрим в качестве первого примера ту же грамматику для целых десятичных чисел со знаком $\mathbf{G}_1$({0,1,2,3,4,5,6,7,8,9,-,+},{S,T,F},$\mathbf{P}_1$,S):

$\mathbf{P}_1$:

S → T | +T | −T

T → F | TF

F → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

По структуре своих правил данная грамматика $\mathbf{G}_1$ относится к КС-грамматикам (тип 2). Конечно, ее можно отнести к типу 0 и к типу 1, но максимально возможным является тип 2, поскольку к типу 3 эту грамматику отнести нельзя: строка T → F | TF содержит правило T → TF, которое недопустимо для типа 3. И хотя все остальные правила типу 3 соответствуют, одного несоответствия достаточно.

Для того же самого языка (целых десятичных чисел со знаком) можно построить и другую грамматику $\mathbf{G}_1$'({0,1,2,3,4,5,6,7,8,9,−,+},{S,T},$\mathbf{P}_1$',S):

$\mathbf{P}_1$':

S → T | +T | −T

T → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0T | 1T | 2T | 3T | 4T | 5T | 6T | 7T | 8T | 9T

По структуре своих правил эта грамматика $\mathbf{G}_1$' является праволинейной и может быть отнесена к регулярным грамматикам (тип 3).

Для этого же языка можно построить эквивалентную леволинейную грамматику (тип 3) $G_1''(\{0,1,2,3,4,5,6,7,8,9,-,+\},\{S,T\},P_1'',S)$:

P_1'':

$T \to + \mid - \mid \lambda$

$S \to T0 \mid T1 \mid T2 \mid T3 \mid T4 \mid T5 \mid T6 \mid T7 \mid T8 \mid T9 \mid S0 \mid S1 \mid S2 \mid S3 \mid S4 \mid S5 \mid S6 \mid S7 \mid S8 \mid S9$

Следовательно, язык L_1 целых десятичных чисел со знаком, заданный грамматиками G_1, G_1' и G_1'', относится к регулярным языкам (тип 3).

В качестве второго примера возьмем грамматику $G_2(\{0,1\},\{A,S\},P_2,S)$ с правилами P_2:

$S \to 0A1$

$0A \to 00A1$

$A \to \lambda$

Эта грамматика относится к типу 0. Она определяет язык, множество предложений которого можно было бы записать так: $L(G_2) = \{0^n1^n \mid n>0\}$.

Для того же самого языка можно построить также КЗ-грамматику $G_2'(\{0,1\},\{A,S\},P_2',S)$ с правилами P_2':

$S \to 0A1 \mid 01$

$0A \to 00A1 \mid 001$

Однако для того же самого языка можно использовать и КС-грамматику $G_2''(\{0,1\},\{S\},P_2'',S)$ с правилами P_2'':

$S \to 0S1 \mid 01$

Следовательно, язык $L_2 = \{0^n1^n \mid n>0\}$ является КС-языком (тип 2).

В третьем примере рассмотрим грамматику $G_3(\{a,b,c\},\{B,C,D,S\},P_3,S)$ с правилами P_3:

$S \to BD$

$B \to aBbC \mid ab$

$Cb \to bC$

$CD \to Dc$

$bDc \to bcc$

$abD \to abc$

Эта грамматика относится к типу 1. Очевидно, что она является неукорачивающей . Она определяет язык, множество предложений которого можно было бы записать так: $L(G_3) = \{a^nb^nc^n \mid n>0\}$. Известно, что этот язык не является КС-языком, поэтому для него нельзя построить грамматики типа 2 или 3.

Но для того же самого языка можно построить КЗ-грамматику $G_3'(\{a,b,c\},\{B,C,D,E,F,S\},P_3', S)$ с правилами P_3':

$S \to abc \mid AE$

$A \to aABC \mid aBC$

$CBC \to CDC$

$CDC \to BDC$

```
BDC → BCC
CCE → CFE
CFE → CFc
CFc → CEc
aB → ab
bB → bb
bCE → bCc
bCc → bc
```

Язык $L_3 = \{a^n b^n c^n \mid n>0\}$ является контекстно-зависимым (тип 1).

Конечно, для произвольного языка, заданного некоторой грамматикой, в общем случае довольно сложно определить его тип. Не всегда можно так просто построить грамматику максимально возможного типа для произвольного языка. К тому же требуется еще доказать, что две грамматики (первоначально имеющаяся и вновь построенная) эквивалентны, а также, что для того же языка не существует грамматики с бóльшим по номеру типом.

Для многих языков, и в частности для КС-языков и регулярных языков, существуют специальным образом сформулированные утверждения, которые позволяют проверить принадлежность языка к указанному типу. Такие утверждения (леммы) можно найти в [4 т .1, 15, 29]. Т огда для произвольного языка достаточно лишь доказать нужную лемму и после этого можно утверждать, что данный язык относится к тому или иному типу. Преобразование грамматик в этом случае не требуется.

Тем не менее иногда возникает задача построения для имеющегося языка грамматики более простого типа, чем данная. И даже в том случае, когда тип языка уже известен, эта задача в общем случае не имеет формального решения (проблема преобразования грамматик рассматривается далее).

Цепочки вывода. Сентенциальная форма

Вывод. Цепочки вывода

Выводом называется процесс порождения предложения языка на основе правил определяющей язык грамматики. Чтобы дать формальное определение процессу вывода, необходимо ввести еще несколько дополнительных понятий.

Цепочка $\beta = \delta_1\gamma\delta_2$ называется *непосредственно выводимой* из цепочки $\alpha = \delta_1\omega\delta_2$ в грамматике G(**VT**,**VN**,**P**,S), **V** = **VT**∪**VN**, δ_1, γ, $\delta_2 \in$ **V***, $\omega \in$ **V**$^+$, если в грамматике **G** существует правило: $\omega \to \gamma \in$ **P**. Непосредственная выводимость цепочки β из цепочки α обозначается так: $\alpha \Rightarrow \beta$. Согласно определению при выводе $\alpha \Rightarrow \beta$ выполняется подстановка подцепочки γ вместо подцепочки ω. Иными словами, цепочка β выводима из цепочки α в том случае, если можно взять несколько символов в цепочке α, поменять их на другие символы, согласно некоторому правилу грамматики, и получить цепочку β.

В формальном определении непосредственной выводимости любая из цепочек δ_1 или δ_2, (а равно и обе эти цепочки), может быть пустой. В предельном случае вся цепочка α может быть заменена цепочкой β, тогда в грамматике **G** должно существовать правило: $\alpha \to \beta \in$ **P**.

Цепочка β называется *выводимой* из цепочки α (обозначается $\alpha \Rightarrow^* \beta$) в том случае, если выполняется одно из двух условий:

- β непосредственно выводима из α ($\alpha \Rightarrow \beta$);
- $\exists\ \gamma$ такая, что γ выводима из α и β непосредственно выводима из γ ($\alpha \Rightarrow^* \gamma$ и $\gamma \Rightarrow \beta$).

Это рекурсивное определение выводимости цепочки. Суть его заключается в том, что цепочка β выводима из цепочки α, если $\alpha \Rightarrow \beta$ или же если можно построить последовательность непосредственно выводимых цепочек от α к β следующего вида: $\alpha \Rightarrow \gamma_1 \Rightarrow ... \Rightarrow \gamma_i \Rightarrow ... \Rightarrow \gamma_n \Rightarrow \beta$, $n \geq 1$. В этой последовательности каждая последующая цепочка γ_i непосредственно выводима из предыдущей цепочки γ_{i-1}.

Такая последовательность непосредственно выводимых цепочек называется выводом, или *цепочкой вывода*. Каждый переход от одной непосредственно выводимой цепочки к следующей в цепочке вывода называется *шагом вывода*. Очевидно, что шагов вывода в цепочке вывода всегда на один больше, чем промежуточных цепочек. Если цепочка β непосредственно выводима из цепочки α: $\alpha \Rightarrow \beta$, то имеется всего один шаг вывода.

Если цепочка вывода из α к β содержит одну или более промежуточных цепочек (два или более шага вывода), то она имеет специальное обозначение $\alpha \Rightarrow^+ \beta$ (говорят, что цепочка β *нетривиально выводима* из цепочки α). Если количество шагов вывода известно, то его можно указать непосредственно у знака выводимости цепочек. Например, запись $\alpha \Rightarrow^4 \beta$ означает, что цепочка β выводится из цепочки α за 4 шага вывода[1].

Возьмем в качестве примера ту же грамматику для целых десятичных чисел со знаком `G({0,1,2,3,4,5,6,7,8,9,–,+},{S,T,F},P,S)`:

P:

```
S → T | +T | –T
T → F | TF
F → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Построим в ней несколько произвольных цепочек вывода (для понимания каждого шага вывода подцепочка, для которой выполняется подстановка, выделена жирным шрифтом):

1. **S** ⇒ –**T** ⇒ –**T**F ⇒ –**TFF** ⇒ –**FFF** ⇒ –4**FF** ⇒ –47**F** ⇒ –479.
2. **S** ⇒ **T** ⇒ T**F** ⇒ **T**8 ⇒ **F**8 ⇒ 18.
3. **T** ⇒ T**F** ⇒ **T**0 ⇒ T**F**0 ⇒ **T**50 ⇒ **F**50 ⇒ 350.
4. **T**FT ⇒ TFF**T** ⇒ **T**FFF ⇒ **F**FFF ⇒ 1FF**F** ⇒ 1**F**F4 ⇒ 10**F**4 ⇒ 1004.
5. **F** ⇒ 5.

Получим следующие выводы:

1. S ⇒ * –479, или S ⇒ + –479, или S ⇒ ^{7} –479.
2. S ⇒ * 18, или S ⇒ + 18, или S ⇒ ^{5} 18.

[1] В литературе встречается также обозначение: $\alpha \Rightarrow^0 \beta$, которое означает, что цепочка α выводима из цепочки β за 0 шагов — иными словами, в таком случае эти цепочки равны: $\alpha = \beta$. Подразумевается, что обозначение вывода $\alpha \Rightarrow^* \beta$ допускает и такое толкование — включает в себя вариант $\alpha \Rightarrow^0 \beta$.

3. T ⇒ * 350, или T ⇒ + 350, или T ⇒ ^{6} 350.

4. TFT ⇒ * 1004, или TFT ⇒ + 1004, или TFT ⇒ ^{7} 1004.

5. F ⇒ * 5 или F ⇒ ^{1} 5 (утверждение F ⇒ + 5 неверно!).

Все эти выводы построены на основе грамматики **G**. В принципе, в этой грамматике (как практически и в любой другой грамматике реального языка) можно построить сколь угодно много цепочек вывода.

Возьмем в качестве второго примера грамматику $\mathbf{G}_3(\{a,b,c\},\{B,C,D,S\},\mathbf{P}_3,S)$ с правилами $\mathbf{P}_3$, которая уже рассматривалась выше:

S → BD

B → aBbC | ab

Cb → bC

CD → Dc

bDc → bcc

abD → abc

Как было сказано ранее, она задает язык $L(\mathbf{G}_3) = \{a^n b^n c^n \mid n>0\}$. Рассмотрим пример вывода предложения aaaabbbbcccc языка $L(\mathbf{G}_3)$ на основе грамматики $\mathbf{G}_3$:

S ⇒ **B**D ⇒ a**B**bCD ⇒ aa**B**bCbCD ⇒ aaa**B**bCbCbCD ⇒ aaaabb**Cb**CbCD ⇒
aaaabbbC**Cb**CD ⇒ aaaabbb**Cb**CCD ⇒ aaaabbbbCC**CD** ⇒ aaaabbbbC**CD**c ⇒
aaaabbbb**CD**cc ⇒ aaaabbb**bDc**cc ⇒ aaaabbbbcccc.

Тогда для грамматики $\mathbf{G}_3$ получаем вывод: S ⇒* aaaabbbbcccc.

Сентенциальная форма грамматики. Язык, заданный грамматикой

Вывод называется *законченным* (или *конечным*), если на основе цепочки β, полученной в результате этого вывода, нельзя больше сделать ни одного шага вывода. Иначе говоря, вывод называется законченным, если цепочка β, полученная в результате этого вывода, пустая или содержит только терминальные символы грамматики G(**VT**,**VN**,**P**,S): β∈ **VT***. Цепочка β, полученная в результате законченного вывода, называется *конечной* цепочкой вывода.

В рассмотренном выше примере все построенные выводы являются законченными, а, например, вывод S ⇒* –4FF (из первой цепочки в примере) будет незаконченным.

Цепочка символов α∈ **V*** называется *сентенциальной формой* грамматики G(**VT**,**VN**,**P**,S), **V** = **VT**∪**VN**, если она выводима из целевого символа грамматики S: S ⇒* α. Если цепочка α∈ **VT*** получена в результате законченного вывода, то она называется *конечной сентенциальной формой*.

Из рассмотренного выше примера можно заключить, что цепочки символов –479 и 18 являются конечными сентенциальными формами грамматики целых десятичных чисел со знаком, так как существуют выводы S ⇒* –479 и S ⇒* 18 (выводы 1 и 2). Цепочка F8 из вывода 2, например, тоже является сентенциальной формой, поскольку справедливо S ⇒* F8, но она не является конечной цепочкой вывода. В то же время в выводах 3, 4 и 5 примера явно не присутствуют сентенциальные формы. На самом деле цепочки 350, 1004 и 5 тоже являются конечными сентенциальными формами. Чтобы доказать это, необходимо просто построить другие цепочки вывода

(например, для цепочки 5 строим S ⇒ T ⇒ F ⇒ 5 и получаем S ⇒* 5). А вот цепочка TFT (вывод 4) не выводима из целевого символа грамматики S, а потому сентенциальной формой не является.

Язык **L**, заданный грамматикой G(**VT**,**VN**,**P**,S), — это множество всех конечных сентенциальных форм грамматики **G**. Язык **L**, заданный грамматикой **G**, обозначается как L(**G**). Очевидно, что алфавитом такого языка L(**G**) будет множество терминальных символов грамматики **VT**, поскольку все конечные сентенциальные формы грамматики — это цепочки над алфавитом **VT**.

Следует помнить, что две грамматики, G(**VT**,**VN**,**P**,S) и G'(**VT**',**VN**',**P**',S'), называются эквивалентными, если эквивалентны заданные ими языки: L(**G**) = L(**G**'). Очевидно, что эквивалентные грамматики должны иметь, по крайней мере, пересекающиеся множества терминальных символов **VT**∩**VT**' ≠ ∅ (как правило, эти множества даже совпадают: **VT**=**VT**'), а вот множества нетерминальных символов, правила грамматики и целевой символ у них могут кардинально отличаться.

Левосторонний и правосторонний выводы

Вывод называется *левосторонним*, если в нем на каждом шаге вывода правило грамматики применяется всегда к крайнему левому нетерминальному символу в цепочке. Другими словами, вывод называется левосторонним, если на каждом шаге вывода происходит подстановка цепочки символов на основании правила грамматики вместо крайнего левого нетерминального символа в исходной цепочке.

Аналогично, вывод называется *правосторонним*, если в нем на каждом шаге вывода правило грамматики применяется всегда к крайнему правому нетерминальному символу в цепочке.

Если рассмотреть цепочки вывода из того же примера, то в нем выводы 1 и 5 являются левосторонними, выводы 2, 3 и 5 — правосторонними (вывод 5 одновременно является и лево- и правосторонним), а вот вывод 4 не является ни левосторонним, ни правосторонним.

Для грамматик типов 2 и 3 (КС-грамматик и регулярных грамматик) для любой сентенциальной формы всегда можно построить левосторонний или правосторонний вывод. Для грамматик других типов это не всегда возможно, так как по структуре их правил не всегда можно выполнить замену крайнего левого или крайнего правого нетерминального символа в цепочке.

А вот рассмотренный выше вывод S ⇒* aaaabbbbcccc для грамматики $\mathbf{G}_3$, задающей язык $L(\mathbf{G}_3) = \{a^n b^n c^n \mid n>0\}$, не является ни левосторонним, ни правосторонним. Грамматика относится к типу 1, и в данном случае для нее нельзя построить такой вывод, на каждом шаге которого только один нетерминальный символ заменялся бы на цепочку символов.

Дерево вывода. Методы построения дерева вывода

Деревом вывода грамматики G(**VT**,**VN**,**P**,S) называется дерево (граф), которое соответствует некоторой цепочке вывода и удовлетворяет следующим условиям:

- каждая вершина дерева обозначается символом грамматики A∈(**VT**∪**VN**∪{λ});
- корнем дерева является вершина, обозначенная целевым символом грамматики — S;

- листьями дерева (концевыми вершинами) являются вершины, обозначенные терминальными символами грамматики или символом пустой цепочки λ;
- если некоторый узел дерева обозначен нетерминальным символом A∈ **VN**, а связанные с ним узлы — символами $b_1, b_2, ..., b_n$; $n > 0$, $\forall n \geq i > 0$: $b_i \in (\mathbf{VT} \cup \mathbf{VN} \cup \{\lambda\})$, то в грамматике G(**VT**,**VN**,**P**,S) существует правило $A \rightarrow b_1 b_2 ... b_n \in \mathbf{P}$.

Из определения видно, что по структуре правил дерево вывода в указанном виде всегда можно построить только для грамматик типов 2 и 3 (контекстно-свободных и регулярных). Для грамматик других типов дерево вывода в таком виде можно построить не всегда (либо же оно будет иметь несколько иной вид).

На основе рассмотренного выше примера построим деревья вывода для цепочек выводов 1 и 2. Эти деревья приведены на рис. 1.3.

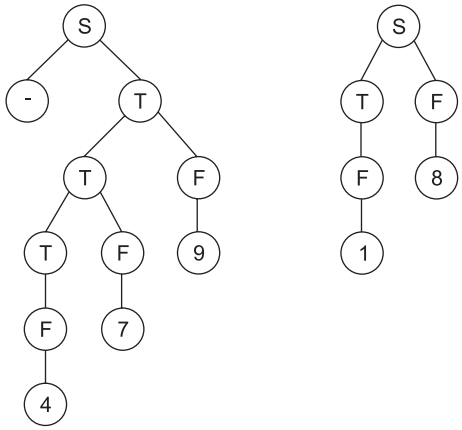

Рис. 1.3. Примеры деревьев вывода для грамматики целых десятичных чисел со знаком

Для того чтобы построить дерево вывода, достаточно иметь только цепочку вывода. Дерево вывода можно построить двумя способами: сверху вниз и снизу вверх. Для строго формализованного построения дерева вывода всегда удобнее пользоваться строго определенным выводом: либо левосторонним, либо правосторонним.

При построении дерева вывода сверху вниз построение начинается с целевого символа грамматики, который помещается в корень дерева. Затем в грамматике выбирается необходимое правило, и на первом шаге вывода корневой символ раскрывается на несколько символов первого уровня. На втором шаге среди всех концевых вершин дерева выбирается крайняя (крайняя левая — для левостороннего вывода, крайняя правая — для правостороннего) вершина, обозначенная нетерминальным символом, для этой вершины выбирается нужное правило грамматики, и она раскрывается на несколько вершин следующего уровня. Построение дерева заканчивается, когда все концевые вершины обозначены терминальными символами, в противном случае надо вернуться ко второму шагу и продолжить построение.

Построение дерева вывода снизу вверх начинается с листьев дерева. В качестве листьев выбираются терминальные символы конечной цепочки вывода, которые на первом шаге построения образуют последний уровень (слой) дерева. Построение дерева идет по слоям. На втором шаге построения в грамматике выбирается правило,

правая часть которого соответствует крайним символам в слое дерева (крайним правым символам при правостороннем выводе и крайним левым — при левостороннем). Выбранные вершины слоя соединяются с новой вершиной, которая выбирается из левой части правила. Новая вершина попадает в слой дерева вместо выбранных вершин. Построение дерева закончено, если достигнута корневая вершина (обозначенная целевым символом), а иначе надо вернуться ко второму шагу и повторить его над полученным слоем дерева.

Поскольку все известные языки программирования имеют нотацию записи «слева — направо», компилятор также всегда читает входную программу слева направо (и сверху вниз, если программа разбита на несколько строк). Поэтому для построения дерева вывода методом «сверху вниз», как правило, используется левосторонний вывод, а для построения «снизу вверх» — правосторонний вывод. На эту особенность компиляторов стоит обратить внимание. Нотация чтения программ «слева направо» влияет не только на порядок разбора программы компилятором (для пользователя это, как правило, не имеет значения), но и на порядок выполнения операций — при отсутствии скобок большинство равноправных операций выполняется в порядке слева направо, а это уже имеет существенное значение.

Проблемы однозначности и эквивалентности грамматик

Однозначные и неоднозначные грамматики

Рассмотрим некоторую грамматику `G({+,–,*,/,(,),a,b},{S},P,S)`:

`P:`

`S → S+S | S–S | S*S | S/S | (S) | a | b`

Видно, что представленная грамматика определяет язык арифметических выражений с четырьмя основными операциями (сложение, вычитание, умножение и деление) и скобками над операндами a и b. Примерами предложений этого языка могут служить: `a*b+a`, `a*(a+b)`, `a*b+a*a` и т. д.

Возьмем цепочку `a*b+a` и построим для нее левосторонний вывод . Получится два варианта:

`S ⇒ S+S ⇒ S*S+S ⇒ a*S+S ⇒ a*b+S ⇒ a*b+a,`

`S ⇒ S*S ⇒ a*S ⇒ a*S+S ⇒ a*b+S ⇒ a*b+a.`

Каждому из этих вариантов будет соответствовать свое дерево вывода Два варианта дерева вывода для цепочки `a*b+a` приведены на рис. 1.4.

С точки зрения формального языка, заданного грамматикой, не имеет значения, какая цепочка вывода и какое дерево вывода из возможных вариантов будут построены. Однако в реальных языках структура предложения и его значение (смысл) взаимосвязаны. Это справедливо как для естественных языков, так и для языков программирования. Дерево вывода (или цепочка вывода) является формой представления структуры предложения языка. Поэтому для языков программирования, которые несут смысловую нагрузку, имеет принципиальное значение то, какая цепочка вывода будет построена для предложения языка.

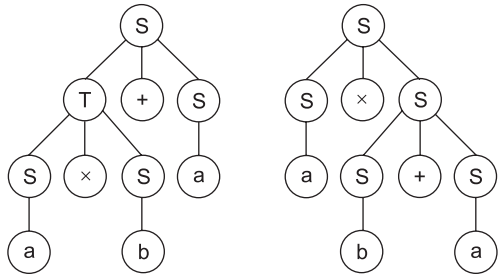

Рис. 1.4. Два варианта дерева вывода цепочки a*b+a вывода для неоднозначной грамматики арифметических выражений

Например, если принять во внимание, что рассмотренная здесь грамматика определяет язык арифметических выражений, то с точки зрения семантики арифметических выражений порядок построения дерева вывода соответствует порядку выполнения арифметических действий. В арифметике, как известно, при отсутствии скобок умножение всегда выполняется раньше сложения (умножение имеет более высокий приоритет), но в рассмотренной выше грамматике это ниоткуда не следует — в ней все операции равноправны. Поэтому с точки зрения арифметических операций приведенная грамматика имеет неверную семантику — в ней нет приоритета операций, а, кроме того, для равноправных операций не определен порядок выполнения (в арифметике принят порядок выполнения действий «слева направо»), хотя синтаксическая структура построенных с ее помощью выражений будет правильной.

Такая ситуация называется неоднозначностью в грамматике. Естественно, для построения компиляторов и языков программирования нельзя использовать грамматики, допускающие неоднозначности. Дадим более точное определение однозначной грамматики.

Грамматика называется *однозначной*, если для каждой цепочки символов языка, заданного этой грамматикой, можно построить единственный левосторонний (и единственный правосторонний) вывод. Или, что то же самое, грамматика называется однозначной, если для каждой цепочки символов языка, заданного этой грамматикой, существует единственное дерево вывода. В противном случае грамматика называется *неоднозначной*.

Рассмотренная в примере грамматика арифметических выражений, очевидно, является неоднозначной.

Проверка однозначности и эквивалентности грамматик

Поскольку грамматика языка программирования, по сути, всегда должна быть однозначной, то возникают два вопроса, которые необходимо в любом случае решить:

- как проверить, является ли данная грамматика однозначной?
- если заданная грамматика является неоднозначной, то как преобразовать ее к однозначному виду?

Однозначность — это свойство грамматики, а не языка. Для некоторых языков, заданных неоднозначными грамматиками, иногда удается построить эквивалентную однозначную грамматику (однозначную грамматику, задающую тот же язык).

Чтобы убедиться в том, что некоторая грамматика не является однозначной (является неоднозначной), согласно определению, достаточно найти в заданном ею языке хотя бы одну цепочку, которая допускала бы более чем один левосторонний или правосторонний вывод (как это было в рассмотренном примере). Однако не всегда удается легко обнаружить такую цепочку символов. Кроме того, если такая цепочка не найдена, мы не можем утверждать, что данная грамматика является однозначной, поскольку перебрать все цепочки языка невозможно — как правило, их бесконечное множество. Следовательно, нужны другие способы проверки однозначности грамматики.

Если грамматика все же является неоднозначной, то необходимо попытаться преобразовать ее в однозначный вид. Например, для рассмотренной выше неоднозначной грамматики арифметических выражений над операндами a и b существует эквивалентная ей однозначная грамматика вида `G'({+,−,*,/,(,),a,b},{S,T,E},P',S)`:

`P'`:

`S → S+T | S−T | T`

`T → T*E | T/E | E`

`E → (S) | a | b`

В этой грамматике для рассмотренной выше цепочки символов языка `a*b+a` возможен только один левосторонний вывод:

`S ⇒ S+T ⇒ T+T ⇒ T*E+T ⇒ E*E+T ⇒ a*E+T ⇒ a*b+T ⇒ a*b+E ⇒ a*b+a`

Этому выводу соответствует единственно возможное дерево вывода. Оно приведено на рис. 1.5. Видно, что, хотя цепочка вывода несколько удлинилась, приоритет операций в данном случае единственно возможный и соответствует их порядку в арифметике.

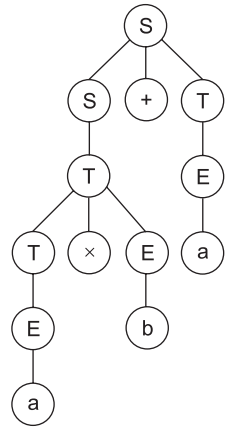

Рис. 1.5. Дерево вывода для однозначной грамматики арифметических выражений

В таком случае необходимо решить две проблемы: во-первых, доказать, что две имеющиеся грамматики эквивалентны (задают один и тот же язык); во-вторых, иметь возможность проверить, что вновь построенная грамматика является однозначной.

Проблема эквивалентности грамматик в общем виде формулируется следующим образом: имеется две грамматики, **G** и **G'**, необходимо построить алгоритм, который бы позволял проверить, являются ли эти две грамматики эквивалентными. То есть надо проверить утверждение L(**G**) = L(**G'**).

К сожалению, доказано, что проблема эквивалентности грамматик в общем случае алгоритмически неразрешима. Это значит , что не только до сих пор не существует алгоритма, который бы позволял проверить, являются ли две заданные грамматики эквивалентными, но и доказано, что такой алгоритм в принципе не существует , а значит, он никогда не будет создан.

Точно так же неразрешима в общем виде и проблема однозначности грамматик. Это значит, что не существует (и никогда не будет существовать) алгоритм, который бы позволял проверить, является ли произвольно заданная грамматика **G** однозначной или нет. Аналогично, не существует алгоритма, который бы позволял преобразовать заведомо неоднозначную грамматику **G** в эквивалентную ей однозначную грамматику **G'**.

В общем случае вопрос об алгоритмической неразрешимости проблем однозначности и эквивалентности грамматик сводится к вопросу об алгоритмической неразрешимости проблемы, известной как «проблема соответствий Поста» [4 т. 1, 5].

Неразрешимость проблем эквивалентности и однозначности грамматик в общем случае совсем не означает, что они не разрешимы вообще. Для многих частных случаев — например, для определенных типов и классов грамматик (в частности для регулярных грамматик) — эти проблемы решены. Например, приведенная выше грамматика **G'** для арифметических выражений над операндами a и b относится к классу грамматик операторного предшествования из типа КС-грамматик, который будет рассмотрен далее. На основе этой грамматики можно построить распознаватель в виде детерминированного расширенного МП-автомата, а потому можно утверждать, что она является однозначной (см. раздел «Восходящие синтаксические распознаватели без возвратов» главы 4).

Правила, задающие неоднозначность в грамматиках

В общем виде невозможно проверить, является ли заданная грамматика однозначной или нет . Однако для КС-грамматик существуют определенного вида правила , по наличию которых во всем множестве правил грамматики G(**VT**,**VN**,**P**,S) можно утверждать, что она является неоднозначной. Эти правила имеют следующий вид:

1. $A \to AA \mid \alpha$.
2. $A \to A\alpha A \mid \beta$.
3. $A \to \alpha A \mid A\beta \mid \gamma$.
4. $A \to \alpha A \mid \alpha A\beta A \mid \gamma$,

здесь $A \in$ **VN**; $\alpha,\beta,\gamma \in$ (**VN**$\cup$**VT**)*.

Если в заданной грамматике встречается хотя бы одно правило подобного вида (любого из приведенных вариантов), то доказано, что такая грамматика точно будет неоднозначной. Однако, если подобных правил во всем множестве правил грамматики нет, это совсем не означает, что грамматика является однозначной. Такая грамматика может быть однозначной, а может и не быть. То есть отсутствие правил указанного

вида (всех вариантов) — это необходимое, но не достаточное условие однозначности грамматики.

С другой стороны, установлены условия, при удовлетворении которым грамматика заведомо является однозначной. Они справедливы для всех регулярных и многих классов КС-грамматик. Однако известно, что эти условия, напротив, являются достаточными, но не необходимыми для однозначности грамматик.

Например, в рассмотренном выше примере грамматики арифметических выражений с операндами a и b — `G({+,–,*,/,(,),a,b},{S},P,S)` — во множестве правил `P: S → S+S | S–S | S*S | S/S | (S) | a | b` встречаются правила 2-го типа (например, два правила: `S → S+S` и `S → a`). Поэтому данная грамматика является неоднозначной, что и было показано выше.

Контрольные вопросы и задачи

Вопросы

1. Какие операции можно выполнять над цепочками символов?
2. Какие из перечисленных ниже тождеств являются истинными для двух произвольных цепочек символов α и β, а какие нет:

   $|\alpha\beta| = |\alpha| + |\beta| = |\beta\alpha|$,

   $\alpha\beta = \beta\alpha$,

   $|\alpha^R| = |\alpha|$,

   $(\alpha^2\beta^2)^R = (\beta^R\alpha^R)^2$,

   $(\alpha^2\beta^2)^R = (\beta^R)^2(\alpha^R)^2$?
3. Какие существуют методы задания языков? Почему метод перечисления всех допустимых цепочек языка не находит практического применения?
4. Какие дополнительные вопросы необходимо решить при задании языка программирования? Какие из них могут быть решены в рамках теории формальных языков?
5. Кто (или что) для любого языка программирования выступает в роли генератора цепочек языка? Кто (или что) выступает в роли распознавателя цепочек?
6. Как формулируется задача разбора? Всегда ли она разрешима?
7. Что такое грамматика языка? Дайте определения грамматики.
8. Как выглядит описание грамматики в форме Бэкуса—Наура? Какие еще формы описания грамматик существуют?
9. Почему в форме Бэкуса—Наура практически невозможно построить грамматику для реального языка так, чтобы она не содержала рекурсивных правил?
10. Что такое распознаватель?
11. Как классифицируются распознаватели? Как их классификация соотносится с классификацией языков и грамматик?
12. На основе какого принципа классифицируются грамматики в классификации Хомского?

13. Какие типы грамматик выделяют по классификации Хомского? Как они между собой соотносятся?
14. Какие типы языков выделяют по классификации Хомского? Как классификация языков соотносится с классификацией грамматик?
15. Что такое сентенциальная форма грамматики?
16. Что такое левосторонний и правосторонний выводы? Можно ли построить еще какие-нибудь варианты цепочек вывода?

Задачи

1. Ниже даны различные варианты грамматик, определяющие язык десятичных чисел с фиксированной точкой. У кажите, к какому типу относится каждая из этих грамматик. К какому типу относится сам язык десятичных чисел с фиксированной точкой?

G_1({".",+,-,0,1,2,3,4,5,6,7,8,9},{<число>,<цел>,<дроб>,<цифра>,<осн>,<знак>},$\mathbf{P}_1$,<число>):

$\mathbf{P}_1$: <число> → <знак><осн>

<знак> → λ | + | -

<осн> → <цел>.<дроб> | <цел>

<цел> → <цифра> | <цифра><цифра>

<дроб> → λ | <цел>

<цифра><цифра> → <цифра><цифра><цифра>

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

G_2({".",+,-,0,1,2,3,4,5,6,7,8,9},{<число>,<часть>,<цифра>,<осн>},$\mathbf{P}_2$,<число>):

$\mathbf{P}_2$: <число> → +<осн> | -<осн> | <осн>

<осн> → <часть>.<часть> | <часть>. | <часть>

<часть> → <цифра> | <цифра><цифра>

<цифра><цифра> → <цифра><цифра><цифра>

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

G_3({".",+,-,0,1,2,3,4,5,6,7,8,9},{<число>,<часть>,<осн>},$\mathbf{P}_3$,<число>):

$\mathbf{P}_3$: <число> → +<осн> | -<осн> | <осн>

<осн> → <часть>. | <часть> | <осн>0 | <осн>1 | <осн>2 | <осн>3 | <осн>4 | <осн>5 | <осн>6 | <осн>7 | <осн>8 | <осн>9

<часть> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | <часть>0 | <часть>1 | <часть>2 | <часть>3 | <часть>4 | <часть>5 | <часть>6 | <часть>7 | <часть>8 | <часть>9

G_4({".",+,-,0,1,2,3,4,5,6,7,8,9},{<знак>,<число>,<часть>,<осн>},$\mathbf{P}_4$,<число>):

P_4: <число> → <часть> | <часть>. | <число>0 | <число>1 | <число>2 | <число>3 | <число>4 | <число>5 | <число>6 | <число>7 | <число>8 | <число>9

<часть> → <знак>0 | <знак>1 | <знак>2 | <знак>3 | <знак>4 | <знак>5 | <знак>6 | <знак>7 | <знак>8 | <знак>9 | <часть>0 | <часть>1 | <часть>2 | <часть>3 | <часть>4 | <часть>5 | <часть>6 | <часть>7 | <часть>8 | <часть>9

<знак> → λ | + | -

G_5({".",+,-,0,1,2,3,4,5,6,7,8,9},{<знак>,<число>,<часть>,<осн>},**P_5**, <число>):

P_5: <число> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | <часть>. | <осн>0 | <осн>1 | <осн>2 | <осн>3 | <осн>4 | <осн>5 | <осн>6 | <осн>7 | <осн>8 | <осн>9 | <часть>0 | <часть>1 | <часть>2 | <часть>3 | <часть>4 | <часть>5 | <часть>6 | <часть>7 | <часть>8 | <часть>9

<осн> → <часть>. | <осн>0 | <осн>1 | <осн>2 | <осн>3 | <осн>4 | <осн>5 | <осн>6 | <осн>7 | <осн>8 | <осн>9

<часть> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | <знак>0 | <знак>1 | <знак>2 | <знак>3 | <знак>4 | <знак>5 | <знак>6 | <знак>7 | <знак>8 | <знак>9 | <часть>0 | <часть>1 | <часть>2 | <часть>3 | <часть>4 | <часть>5 | <часть>6 | <часть>7 | <часть>8 | <часть>9

<знак> → + | -

2. Язык десятичных чисел с фиксированной точкой, рассмотренный в задаче № 1, содержит цепочки `-83`, `239`, `10.4`. Постройте цепочки вывода для каждой из этих трех цепочек символов на основе предложенных в задаче № 1 грамматик.
3. Язык десятичных чисел с фиксированной точкой, рассмотренный выше в задаче № 1, не содержит цепочек вида `.29` или `.104`. Перестройте любую из предложенных в задаче № 1 пяти грамматик так, чтобы заданный ею новый язык допускал такого рода цепочки. Постройте цепочки вывода для этих цепочек.
4. Что можно сказать о типе языка, если
   - язык задан регулярной грамматикой и КС-грамматикой;
   - язык задан КС-грамматикой и КЗ-грамматикой;
   - язык задан КЗ-грамматикой и регулярной грамматикой.

   Является ли в каждом случае ответ окончательным?
5. Поездом называется произвольная последовательность локомотивов и вагонов, начинающаяся с локомотива. Постройте грамматику для понятия <поезд> в форме Бэкуса—Наура, считая, что понятия *локомотив* и *вагон* являются терминальными символами. Модернизируйте грамматику для любого из следующих условий:
   - все локомотивы должны быть сосредоточены в начале поезда;
   - поезд начинается с локомотива и заканчивается локомотивом (попробуйте построить регулярную грамматику);
   - поезд не должен содержать два локомотива либо два вагона подряд.
6. Насколько сложно построить в форме Бэкуса—Наура определение поезда (согласно задаче номер № 5), содержащего не более 60 вагонов или локомо-

тивов? Постройте такое определение в форме грамматики с метасимволами. Распространите его на одно из предложенных в задаче № 5 дополнительных условий.

7. Постройте вывод для цепочки `aaaabbbbcccc` в грамматике `G({a,b,c},{B,C,D,S},P,S)` с правилами **P**:

   `S → abc | AE`

   `A → aABC | aBC`

   `CBC → CDC`

   `CDC → BDC`

   `BDC → BCC`

   `CCE → CFE`

   `CFE → CFc`

   `CFc → CEc`

   `aB → ab`

   `bB → bb`

   `bCE → bCc`

   `bCc → bc`

8. Дана грамматика условных выражений `G({if,then,else,b,a},{E},P,E)` с правилами **P**: `E → if b then E else E | if b then E | a`.

   Не строя цепочек вывода, покажите, что эта грамматика является неоднозначной. Проверьте это, построив некоторую цепочку вывода. Попробуйте построить эквивалентную ей однозначную грамматику.

ГЛАВА 2 Основные принципы построения трансляторов

Выше были рассмотрены теоретические вопросы, связанные с теорией формальных языков и грамматик. В этой главе пойдет речь о том, как эти теоретические аспекты применяются на практике при построении трансляторов.

Трансляторы, компиляторы и интерпретаторы — общая схема работы

Определение транслятора, компилятора, интерпретатора

Для начала дадим несколько определений — что же все-таки такое есть уже многократно упоминавшиеся трансляторы и компиляторы.

Формальное определение транслятора

Транслятор — это программа, которая переводит программу на исходном (входном) языке в эквивалентную ей программу на результирующем (выходном) языке.

В этом определении слово «программа» встречается три раза, и это не ошибка и не тавтология. В работе транслятора действительно участвуют три программы.

Во-первых, сам транслятор является программой[1]. То есть транслятор — это часть программного обеспечения (ПО), он представляет собой набор машинных команд и данных и выполняется компьютером, как и все прочие программы в рамках операционной системы (ОС). Все составные части транслятора представляют собой динамически загружаемые библиотеки или модули этой программы со своими входными и выходными данными.

Во-вторых, исходными данными для работы транслятора служит программа на исходном языке программирования — некоторая последовательность предложений

[1] Теоретически возможна реализация транслятора с помощью аппаратных средств. Автору встречались такого рода разработки, однако широкое практическое применение их не известно. В таком случае и все составные части транслятора могут быть реализованы в виде аппаратных средств и их фрагментов — вот тогда схема распознавателя, который является составной частью транслятора, может получить вполне практическое воплощение!

входного языка. Эта программа называется *входной*, или *исходной программой*. Обычно это символьный файл, но этот файл должен содержать текст программы, удовлетворяющий синтаксическим и семантическим требованиям входного языка. Кроме того, этот файл несет в себе некоторый смысл, определяемый семантикой входного языка. Часто файл, содержащий текст исходной программы, называют исходным файлом .

В-третьих, выходными данными транслятора является программа на результирующем языке. Эта программа называется *результирующей программой*. Результирующая программа строится по синтаксическим правилам выходного языка транслятора, а ее смысл определяется семантикой выходного языка.

Важным пунктом в определении транслятора является эквивалентность исходной и результирующей программ. Эквивалентность этих двух программ означает совпадение их смысла с точки зрения семантики входного языка (для исходной программы) и семантики выходного языка (для результирующей программы). Без выполнения этого требования сам транслятор теряет всякий практический смысл.

Итак, чтобы создать транслятор, необходимо прежде всего выбрать входной и выходной языки. С точки зрения преобразования предложений входного языка в эквивалентные им предложения выходного языка транслятор выступает как переводчик. Например, трансляция программы с языка C в язык ассемблера, по сути, ничем не отличается от перевода, скажем, с русского языка на английский, с той только разницей, что сложность языков несколько иная (о том, почему не существует трансляторов с естественных языков, сказано в предыдущей главе). Поэтому и само слово «транслятор» (английское «translator») означает «переводчик».

Результатом работы транслятора будет результирующая программа, но только в том случае, если текст исходной программы является правильным — не содержит ошибок с точки зрения синтаксиса и семантики входного языка. Если исходная программа неправильная (содержит хотя бы одну ошибку), то результатом работы транслятора будет сообщение об ошибке (с дополнительными пояснениями и указанием места ошибки в исходной программе). В этом смысле транслятор сродни переводчику, например, с английского, которому подсунули неверный текст.

Определение компилятора. Отличие компилятора от транслятора

Кроме понятия «транслятор» широко употребляется также близкое ему по смыслу понятие «компилятор».

Компилятор — это транслятор, который осуществляет перевод исходной программы в эквивалентную ей результирующую программу на языке машинных команд или на языке ассемблера.

Таким образом, компилятор отличается от транслятора лишь тем, что его результирующая программа всегда должна быть написана на языке машинных кодов или на языке ассемблера. Результирующая программа транслятора в общем случае может быть написана на любом языке — возможен, например, транслятор программ с языка Pascal на язык C.

ВНИМАНИЕ

Всякий компилятор является транслятором, но не наоборот — не всякий транслятор будет компилятором. Например, упомянутый выше транслятор с языка Pascal на C компилятором не является[1].

Результирующая программа компилятора называется *объектной программой*, или *объектным кодом*, а исходную программу в этом случае часто называют *исходным кодом*. Файл, в который записана объектная программа, обычно называется *объектным файлом*. Даже в том случае, когда результирующая программа порождается на языке машинных команд, между объектной программой (объектным файлом) и исполняемой программой (исполняемым файлом) есть существенная разница. Порожденная компилятором программа не может непосредственно выполняться на компьютере (более подробно об этом рассказано в главе «Современные системы программирования»).

Само слово «компилятор» происходит от английского термина «compiler» («составитель», «компоновщик»). Термин, вероятно, обязан своему происхождению способности компиляторов составлять объектную программу из фрагментов машинных кодов, соответствующих синтаксическим конструкциям исходной программы (то, как это происходит, описано в главе «Генерация и оптимизация кода»).

Результирующая программа, созданная компилятором, строится на языке машинных кодов или ассемблера, то есть на языках, которые обязательно ориентированы на определенную вычислительную систему. Следовательно, такая результирующая программа всегда предназначена для выполнения на вычислительной системе с определенной архитектурой.

ПРИМЕЧАНИЕ

Следует упомянуть, что в современных системах программирования существуют компиляторы, в которых результирующая программа создается не на языке машинных команд и не на языке ассемблера, а на некотором промежуточном языке. Сам по себе этот промежуточный язык не может непосредственно исполняться на компьютере, а требует специального промежуточного интерпретатора для выполнения написанных на нем программ. Хотя в данном случае термин «транслятор» был бы, наверное, более правильным, в литературе употребляется понятие «компилятор», поскольку промежуточный язык является языком низкого уровня, будучи родственным машинным командам и языкам ассемблера.

Вычислительная система, на которой должна выполняться результирующая (объектная) программа, созданная компилятором, называется *целевой вычислительной системой*.

В понятие целевой вычислительной системы входит не только архитектура аппаратных средств компьютера, но и операционная система, а зачастую также и набор динамически подключаемых библиотек, которые необходимы для выполнения объектной программы. При этом следует помнить, что объектная программа ориентирована на целевую вычислительную систему, но не может быть непосредственно выполнена на ней без дополнительной обработки.

[1] В некоторых литературных источниках эти два понятия не разделяют между собой, хотя разница между ними все-таки существует.

Целевая вычислительная система не всегда является той же вычислительной системой, на которой работает сам компилятор. Часто они совпадают , но бывает так, что компилятор работает под управлением вычислительной системы одного типа, а строит объектные программы, предназначенные для выполнения на вычислительных системах совсем другого типа.

ПРИМЕЧАНИЕ

Здесь есть один «подводный камень», связанный с терминологической путаницей. Термины «объектный код» и «объектный файл» возникли достаточно давно. Однако сейчас появилось понятие «объектно-ориентированное программирование», где под термином «объект» подразумевается совершенно иное, нежели в «объектном коде». Следует помнить, что «объектный код» никакого отношения к «объекту» с точки зрения объектно-ориентированного программирования не имеет! И хотя в результате компиляции программ, написанных на объектно-ориентированных языках, тоже получаются объектный код и объектные файлы, это никак не связывает между собой эти различные по смыслу термины.

Компиляторы, безусловно, самый распространенный вид трансляторов (многие считают их вообще единственным видом трансляторов, хотя это и не так). Они имеют самое широкое практическое применение, которым обязаны широкому распространению всевозможных языков программирования. Далее всегда будем говорить о компиляторах, подразумевая, что резуль тирующая программа порождается на языке машинных кодов или языке ассемблера (если это не так, то это будет специально указываться отдельно).

Естественно, трансляторы и компиляторы, как и все прочие программы, разрабатывают люди — обычно это группа разработчиков. В принципе они могли бы создавать его непосредственно на языке машинных команд, однако объем кода и данных современных компиляторов таков, что их создание на языке машинных команд практически невозможно в разумные сроки при разумных трудозатратах. Поэтому практически все современные компиляторы также создаются с помощью компиляторов (чаще всего в этой роли выступают предыдущие версии компиляторов той же фирмы-производителя). И в этом качестве компилятор сам является результирующей программой для другого компилятора, которая ничем не отличается от всех прочих порождаемых результирующих программ[1].

Определение интерпретатора. Разница между интерпретаторами и трансляторами

Кроме схожих между собой понятий «транслятор» и «компилятор», существует принципиально отличное от них понятие интерпретатора.

Интерпретатор — это программа, которая воспринимает исходную программу на входном (исходном) языке и выполняет ее.

Интерпретатор, так же как и транслятор, анализирует текст исходной программы. Однако он не порождает результирующую программу, а сразу же выполняет исходную

[1] Здесь возникает извечный вопрос «о курице и яйце». Конечно, в первом поколении самые первые компиляторы писались непосредственно на машинных командах, но потом, с появлением компиляторов, от этой практики отошли. Даже самые ответственные части компиляторов создаются как минимум с применением языка ассемблера — а он тоже обрабатывается компилятором.

программу в соответствии с ее смыслом, заданным семантикой входного языка. Таким образом, результатом работы интерпретатора будет результат, определенный смыслом исходной программы, в том случае, если эта программа синтаксически и семантически правильная с точки зрения входного языка, или сообщение об ошибке в противном случае.

ВНИМАНИЕ

В отличие от трансляторов, интерпретаторы не порождают результирующую программу — в этом принципиальная разница между ними.

Чтобы исполнить исходную программу, интерпретатор так или иначе должен преобразовать ее в язык машинных кодов, поскольку в противном случае выполнение программ на компьютере невозможно. Он, конечно же, делает это, однако эти машинные коды недоступны — их не видит пользователь интерпретатора. Машинные коды порождаются интерпретатором, исполняются и уничтожаются по мере надобности — так, как того требует конкретная реализация интерпретатора. Пользователь же видит только результат выполнения этих кодов — то есть результат выполнения исходной программы (требование об эквивалентности исходной программы и порожденных машинных кодов и в этом случае, безусловно, также должно выполняться).

Более подробно вопросы, связанные с реализацией интерпретаторов и их отличием от компиляторов, рассмотрены далее в соответствующем разделе.

Этапы компиляции. Общая схема работы компилятора

На рис. 2.1 представлена общая схема работы компилятора. Из нее видно, что в целом процесс компиляции состоит из двух основных этапов: анализа и синтеза.

На этапе анализа выполняется распознавание текста исходной программы, создание и заполнение таблиц идентификаторов . Результатом его работы служит некое внутреннее представление программы, понятное компилятору.

На этапе синтеза на основании внутреннего представления программы и информации, содержащейся в таблице идентификаторов, порождается текст результирующей программы. Результатом этого этапа является объектный код.

Кроме того, в составе компилятора присутствует часть, ответственная за анализ и исправление ошибок, которая при наличии ошибки в тексте исходной программы должна максимально полно информировать пользователя о типе ошибки и месте ее возникновения. В лучшем случае компилятор может предложить пользователю вариант исправления ошибки.

Эти этапы, в свою очередь, состоят из более мелких этапов, называемых фазами компиляции. Состав фаз компиляции на рис. 2.1 приведен в самом общем виде, их конкретная реализация и процесс взаимодействия могут, конечно, различаться в зависимости от версии компилятора. Однако в том или ином виде все представленные фазы практически всегда присутствуют в каждом конкретном компиляторе [5, 15, 21, 33, 34, 58].

Компилятор в целом с точки зрения теории формальных языков выступает в «двух ипостасях», выполняет две основные функции.

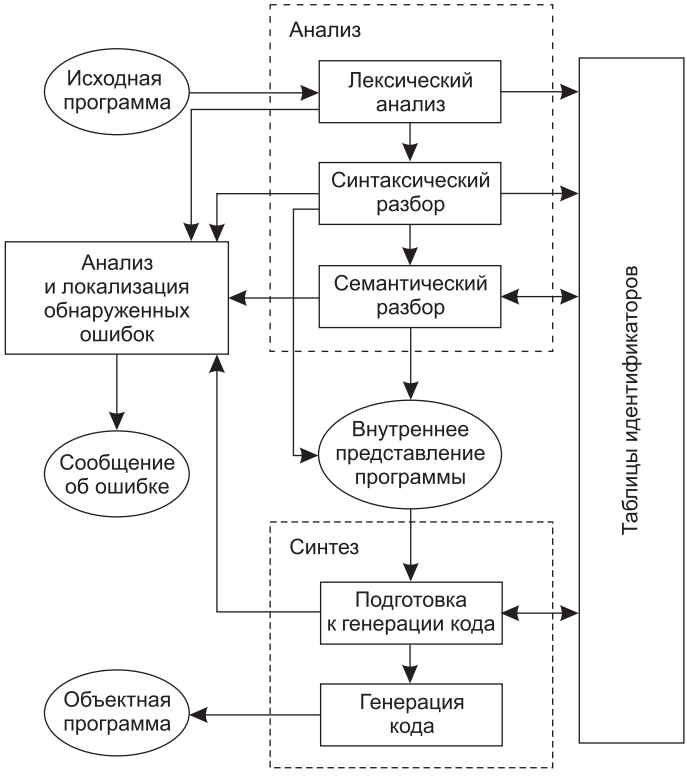

Рис. 2.1. Общая схема работы компилятора

Во-первых, он является распознавателем для языка исходной программы. То есть он должен получить на вход цепочку символов входного языка, проверить ее принадлежность языку и, более того, выявить правила, по которым эта цепочка построена (поскольку сам ответ на вопрос о принадлежности «да» или «нет» представляет мало интереса). Интересно, что генератором цепочек входного языка выступает пользователь — автор исходной программы.

Во-вторых, компилятор является генератором для языка результирующей программы. Он должен построить на выходе цепочку выходного языка по определенным правилам, предполагаемым языком машинных команд или языком ассемблера. В случае машинных команд распознавателем этой цепочки будет выступать целевая вычислительная система, под которую создается результирующая программа.

Далее дается перечень основных фаз (частей) компиляции и краткое описание их функций. Более подробная информация дана в главах учебного пособия, соответствующих этим фазам.

Лексический анализ (сканер) — это часть компилятора, которая читает литеры программы на исходном языке и строит из них слова (лексемы) исходного языка. На вход лексического анализатора поступает текст исходной программы, а выходная информация передается для дальнейшей обработки компилятором на этапе синтаксического разбора. С теоретической точки зрения лексический анализатор не яв-

ляется обязательной частью компилятора. Однако существуют причины, которые определяют его присутствие практически во всех компиляторах (более подробно они описаны в главе «Лексические анализаторы»).

Синтаксический разбор — это основная часть компилятора на этапе анализа. Она выполняет выделение синтаксических конструкций в тексте исходной программы, обработанном лексическим анализатором. На этой же фазе компиляции проверяется синтаксическая правильность программы. Синтаксический разбор играет главную роль — роль распознавателя текста входного языка программирования(этот процесс описан в главе «Синтаксические анализаторы»).

Семантический анализ — это часть компилятора, проверяющая правильность текста исходной программы с точки зрения семантики входного языка. Кроме непосредственно проверки, семантический анализ должен выполнять преобразования текста, требуемые семантикой входного языка (например, такие как добавление функций неявного преобразования типов). В различных реализациях компиляторов семантический анализ может частично входить в фазу синтаксического разбора, частично — в фазу подготовки к генерации кода (в данной книге семантический анализ более подробно рассмотрен в главе «Генерация и оптимизация кода»).

Подготовка к генерации кода — это фаза, на которой компилятором выполняются предварительные действия, необходимые для синтеза результирующей программы, но не ведущие к порождению текста на выходном языке. Обычно в эту фазу входят действия, связанные с идентификацией элементов языка, распределением памяти и т. п. (они рассмотрены в главе «Генерация и оптимизация кода»).

Генерация кода — это фаза, непосредственно связанная с порождением текста результирующей программы. Это основная фаза на этапе синтеза результирующей программы. Кроме непосредственно порождения текста результирующей программы генерация обычно включает в себя также оптимизацию — процесс, связанный с обработкой уже порожденного текста. Иногда оптимизацию выделяют в отдельную фазу компиляции, так как она оказывает существенное влияние на качество и эффективность результирующей программы (подробности вы найдете в главе «Генерация и оптимизация кода»).

Таблицы идентификаторов (иногда «таблицы символов») — это специальным образом организованные структуры данных, служащие для хранения информации об элементах исходной программы, которые затем используются для порождения текста результирующей программы. В конкретной реализации компилятора может быть как одна, так и несколько таблиц идентификаторов. Элементами исходной программы, информацию о которых необходимо хранить в процессе компиляции, являются переменные, константы, функции и т. п. — конкретный состав этих элементов зависит от входного языка. Термин «таблицы» вовсе не предполагает, что это хранилище данных должно быть организовано именно в виде таблиц или массивов информации — возможные методы их организации подробно рассмотрены далее, в разделе «Таблицы идентификаторов. Организация таблиц идентификаторов».

Представленное на рис. 2.1 деление процесса компиляции на фазы служит скорее методическим целям и на практике может не соблюдаться столь строго. Далее, в главах и разделах этого учебного пособия рассматриваются различные варианты технической организации представленных фаз компиляции. При этом указано, как они

могут быть связаны между собой. Здесь рассмотрим только общие аспекты такого рода взаимосвязи.

Во-первых, в фазе лексического анализа лексемы выделяются из текста входной программы постольку, поскольку они необходимы для фазы синтаксического разбора. Во-вторых, как будет показано ниже, синтаксический разбор и генерация кода могут выполняться одновременно. Таким образом, эти три фазы компиляции могут работать комбинированно, а вместе с ними может выполняться и подготовка к генерации кода. Далее рассмотрены технические вопросы реализации основных фаз компиляции, которые тесно связаны с понятием *прохода*.

Понятие прохода. Многопроходные и однопроходные компиляторы

Как уже было сказано, процесс компиляции программ состоит из нескольких фаз . В реальных компиляторах состав этих фаз может несколько отличаться от рассмотренного выше — некоторые из них могут быть разбиты на составляющие, другие, напротив, объединены в одну фазу. Порядок выполнения фаз компиляции также может меняться в разных вариантах компиляторов. В одном случае компилятор просматривает текст исходной программы, сразу выполняет все фазы компиляции и получает результат — объектный код. В другом варианте он выполняет над исходным текстом только некоторые из фаз компиляции и получает не конечный резуль тат, а набор некоторых промежуточных данных. Эти данные затем снова подвергаются обработке, причем этот процесс может повторяться несколько раз.

Реальные компиляторы, как правило, выполняют трансляцию текста исходной программы за несколько проходов.

Проход — это процесс последовательного чтения компилятором данных из внешней памяти, их обработки и помещения резуль тата работы во внешнюю память. Чаще всего один проход включает в себя выполнение одной или нескольких фаз компиляции. Результатом промежуточных проходов является внутреннее представление исходной программы, результатом последнего прохода — объектная программа.

В качестве внешней памяти могут выступать любые носители информации — оперативная память компьютера, накопители на магнитных дисках, магнитных лентах и т. п. Современные компиляторы, как правило, стремятся максимально использовать для хранения данных оперативную память компьютера, и только при недостатке объема доступной оперативной памяти используются накопители на жестких магнитных дисках. Другие носители информации в современных компиляторах не используются из-за невысокой скорости обмена данными.

При выполнении каждого прохода компилятору доступна информация, полученная в результате всех предыдущих проходов. Как правило, он стремится использовать в первую очередь только информацию, полученную на проходе, непосредственно предшествовавшем текущему, но в принципе может обращаться и к данным от более ранних проходов вплоть до исходного текста программы. Информация, получаемая компилятором при выполнении проходов, недоступна пользователю. Она либо хранится в оперативной памяти, которая освобождается компилятором после завершения процесса трансляции, либо оформляется в виде временных файлов на диске, которые также уничтожаются после завершения работы компилятора. Поэтому

человек, работающий с компилятором, может даже не знать, сколько проходов выполняет компилятор, — он всегда видит только текст исходной программы и результирующую объектную программу. Но количество выполняемых проходов — это важная техническая характеристика компилятора, солидные фирмы-разработчики компиляторов обычно указывают ее в описании своего продукта.

Понятно, что разработчики стремятся максимально сократить количество проходов, выполняемых компиляторами. При этом увеличивается скорость работы компилятора, сокращается объем необходимой ему памяти. Однопроходный компилятор, получающий на вход исходную программу и сразу же порождающий результирующую объектную программу, — это идеальный вариант.

Однако сократить число проходов не всегда удается. Количество необходимых проходов определяется прежде всего грамматикой и семантическими правилами исходного языка. Чем сложнее грамматика языка и чем больше вариантов предполагают семантические правила — тем больше проходов будет выполнять компилятор (конечно, играет свою роль и квалификация разработчиков компилятора). Например, именно поэтому обычно компиляторы с языка Pascal работают быстрее, чем компиляторы с языка C — грамматика Pascal более проста, а семантические правила более жесткие.

Однопроходные компиляторы [58] — редкость, они возможны только для очень простых языков. Реальные компиляторы выполняют, как правило, от двух до пяти проходов. Таким образом, реальные компиляторы являются многопроходными. Наиболее распространены двух- и трехпроходные компиляторы, например: первый проход — лексический анализ, второй — синтаксический разбор и семантический анализ, третий — генерация и оптимизация кода (варианты исполнения, конечно, зависят от разработчика). В современных системах программирования нередко первый проход компилятора (лексический анализ кода) выполняется параллельно с редактированием кода исходной программы (такой вариант построения компиляторов рассмотрен далее в главе «Современные системы программирования»).

Современные компиляторы и интерпретаторы

Принципы работы современных компиляторов

Как уже было сказано ранее, для задания языка программирования необходимо ответить на три вопроса:

1) определить множество допустимых символов языка;
2) определить множество правильных программ языка;
3) задать смысл для каждой правильной программы.

Главную проблему при этом представляет третий вопрос, поскольку он в принципе не решается в рамках теории формальных языков. Чисто формальные языки лишены всякого смысла, а потому для ответа на этот вопрос нужно использовать другие подходы.

В качестве таких подходов можно указать следующие:

- изложить смысл программы, написанной на языке программирования, на другом языке, который может воспринимать пользователь программы или компьютер;
- использовать для проверки смысла некоторую «идеальную машину», которая предназначена для выполнения программ, написанных на данном языке.

Все, кто писал программы, так или иначе обязательно использовали первый подход. Например, комментарии в хорошо написанной программе — это и есть изложение ее смысла. Построение блок-схемы, а равно любое другое описание алгоритма исходной программы — это тоже способ изложить смысл программы на другом языке (например, языке графических символов блок-схем алгоритмов, смысл которого, в свою очередь, изложен в соответствующем ГОСТе). Но все эти способы ориентированы на человека, которому они, конечно же, более понятны. Однако не существует способа проверить, насколько описание соответствует программе.

Компьютер умеет воспринимать только машинные команды. С точки зрения человека, можно сказать, что компьютер понимает только язык машинных команд. Поэтому изложить для компьютера смысл исходной программы, написанной на любом языке программирования, значит перевести эту программу на язык машинных команд. Для человека выполнение этой операции — слишком трудоемкая задача. Именно таким переводом занимаются компиляторы.

Следовательно, первый подход, а именно изложение смысла исходной программы на языке машинных команд, составляет основу функционирования компиляторов.

Далее в учебном пособии будут рассмотрены принципы, на основе которых компиляторы выполняют перевод исходной программы с языка программирования на язык машинных команд. Здесь можно только отметить, что главная проблема такого перевода заключается в том, что, в отличие от человека, ни один компилятор не способен понять смысл всей исходной программы в целом. Отсюда проистекают многие проблемы, связанные с созданием программ на любом языке программирования и работой компиляторов. Эти проблемы остаются нерешенными по сей день и, по всей видимости, никогда не будут решены, поскольку нет и не будет теоретической основы для их решения. Главная проблема заключается в том, что компилятор способен обнаруживать только самые простейшие семантические (смысловые) ошибки в исходной программе, а бóльшую часть такого рода ошибок должен обнаруживать человек (разработчик программы или ее пользователь).

Второй подход используется при отладке программы. Чаще всего «идеальной машиной» является компьютер, на котором выполняется откомпилированная программа. Выполнение программы может происходить непосредственно в программно-аппаратной среде этого компьютера либо под управлением специализированного программного обеспечения, предназначенного для отладки и тестирования программ — отладчика. При этом предполагается, что компилятор переводит программу с языка программирования на язык машинных команд без изменения ее смысла (исключаются из рассмотрения ошибки компиляции), а также не рассматриваются ошибки и сбои функционирования самого компьютера — целевой вычислительной системы. Но оценку результатов выполнения программы при отладке выполняет человек. Любые попытки поручить это дело машине лишены смысла вне контекста решаемой задачи.

Например, предложение в программе на языке Pascal вида `i:=0; while i=0 do i:=0;` может быть легко оценено любой машиной как бессмысленное. Но если в задачу входит обеспечить взаимодействие с другой параллельно выполняемой программой или, например, просто проверить надежность процессора или ячейки памяти, то это же предложение покажется уже не лишенным смысла.

Многие современные компиляторы позволяют выявить сомнительные с точки зрения смысла места в исходной программе. Такими подозрительными на наличие семантической (смысловой) ошибки местами являются недостижимые операторы, неиспользуемые переменные, неопределенные результаты функций и т. п. Компилятор указывает такие места в виде предупреждений, которые разработчик может принимать или не принимать во внимание. Для достижения этой цели компилятор должен иметь представление о том, как программа будет выполняться, и во время компиляции отследить пути выполнения отдельных фрагментов исходной программы — там, где это возможно. То есть компилятор использует второй подход для поиска потенциальных мест возможных семантических ошибок.

Возможности компиляторов по проверке осмысленности предложений входного языка существенно ограничены. Именно поэтому большинство из них в лучшем случае ограничиваются только рекомендациями по тем местам исходного текста программ, которые вызывают сомнения с точки зрения семантики. Компиляторы обнаруживают только незначительный процент от общего числа смысловых (семантических) ошибок, а следовательно, подавляющее число такого рода ошибок всегда, к большому сожалению, остается на совести автора программы.

Некоторые успехи в процессе проверки осмысленности программ достигнуты в рамках систем автоматизации разработки программного обеспечения (CASE-системы). Их подход основан на проектировании сверху вниз — от описания задачи на языке, понятном человеку, до перевода ее в операторы языка программирования. Но такой подход выходит за рамки возможностей компиляторов, поэтому здесь рассматриваться не будет.

С тех пор как большинство теоретических аспектов в области компиляторов получило свою практическую реализацию, развитие компиляторов пошло по пути повышения их дружественности человеку — пользователю, разработчику программ на языках высокого уровня. Логичным завершением этого процесса стало создание систем программирования — программных комплексов, объединяющих в себе, кроме непосредственно компиляторов, множество связанных с ними компонентов программного обеспечения. О том, что представляют собой и как организованы современные системы программирования см. в главе «Современные системы программирования».

Желающие ознакомиться с историей компиляторов и систем программирования могут обратиться к литературе [11, 15, 18, 19, 38, 47, 50, 56, 57].

Интерпретаторы. Особенности построения интерпретаторов

Интерпретатор — это программа, которая воспринимает исходную программу на входном (исходном) языке и выполняет ее. Как уже было сказано выше, основное отличие интерпретаторов от трансляторов и компиляторов заключается в том, что интерпретатор не порождает результирующую программу, а просто выполняет исходную программу.

Термин «интерпретатор» (interpreter), как и «транслятор», означает «переводчик». С точки зрения терминологии эти понятия схожи, но с точки зрения теории формальных языков и компиляции между ними существует принципиальная разница. Если понятия «транслятор» и «компилятор» почти неразличимы, то с понятием «интерпретатор» их путать никак нельзя.

Простейшим способом реализации интерпретатора можно было бы считать вариант , когда исходная программа сначала полностью транслируется в машинные команды, а затем сразу же выполняется. В такой реализации интерпретатор, по сути, мало бы чем отличался от компилятора с той лишь разницей, что результирующая программа в нем недоступна пользователю. Недостатком такого интерпретатора является то, что пользователь должен ждать компиляции всей исходной программы прежде, чем начнется ее выполнение. Поэтому в таком интерпретаторе не было бы никакого особого смысла — он не давал бы никаких преимуществ по сравнению с аналогичным компилятором[1].

Поэтому подавляющее большинство интерпретаторов действует так, что исполняет исходную программу последовательно, по мере ее поступления на вход интерпретатора. Тогда пользователю не надо ждать завершения компиляции всей исходной программы. Более того, он может последовательно вводить исходную программу и тут же наблюдать результат ее выполнения [15, 18, 35, 58, 63, 65].

При таком порядке работы интерпретатора проявляется существенная особенность, которая отличает его от компилятора — если интерпретатор исполняет команды по мере их поступления, то он не может выполнять оптимизациюисходной программы. Следовательно, фаза оптимизации в общей структуре интерпретатора будет отсутствовать. В остальном структура интерпретатора будет мало отличаться от структуры аналогичного компилятора. Следует только учесть, что на последнем этапе — генерации кода — машинные команды не записываются в объектный файл, а выполняются по мере их порождения.

Далеко не все языки программирования допускают построение интерпретаторов, которые могли бы выполнять исходную программу по мере поступления команд.

Отсутствие шага оптимизации ведет к тому, что выполнение программы с помощью интерпретатора является менее эффективным, чем с помощью аналогичного компилятора. Кроме того, при интерпретации исходная программа должна заново разбираться всякий раз при ее выполнении, в то время как при компиляции она разбирается только один раз, а после этого всегда используется объектный файл. Также очевидно, что объектный код будет исполняться всегда быстрее, чем происходит интерпретация аналогичной исходной программы. Таким образом, интерпретаторы всегда проигрывают компиляторам в производительности.

Преимуществом интерпретатора является независимость выполнения программы от архитектуры целевой вычислительной системы. В резуль тате компиляции получается объектный код, который всегда ориентирован на определенную целевую вычислительную систему. Для перехода на другую целевую вычислительную систему исходную программу требуется откомпилировать заново. А для интерпретации программы необходимо иметь только ее исходный текст и интерпретатор с соответствующего языка.

---

[1] Иногда такая схема используется в современных системах программирования при отладке программ — более подробно см. главу «Современные системы программирования».

Интерпретаторы долгое время значительно уступали в распространенности компиляторам. Как правило, интерпретаторы существовали для ограниченного круга относительно простых языков программирования (таких, например, как Basic). Высокопроизводительные профессиональные средства разработки программного обеспечения строились на основе компиляторов.

Новый импульс развитию интерпретаторов придало распространение глобальных вычислительных сетей. Такие сети могут включать в свой состав компьютеры различной архитектуры, и тогда требование единообразного выполнения на каждом из них текста исходной программы становится определяющим. Поэтому с развитием глобальных сетей и распространением Всемирной сети Интернет появилось много новых систем, интерпретирующих текст исходной программы. Многие языки программирования, применяемые во Всемирной сети, предполагают именно интерпретацию текста исходной программы без порождения объектного кода.

Некоторые современные языки программирования предполагают две стадии разработки: сначала исходная программа компилируется в промежуточный код (некоторый язык низкого уровня), а затем этот результат компиляции выполняется с помощью интерпретатора данного промежуточного языка. Более подробно варианты таких систем рассмотрены в главе «Современные системы программирования».

Широко распространенным примером интерпретируемого языка может служить HTML (Hypertext Markup Language) — язык описания гипертекста. На его основе в настоящее время функционирует практически вся структура сети Интернет . Другой пример — языки Java и JavaScript сочетают в себе функции компиляции и интерпретации (текст исходной программы компилируется в промежуточный код, не зависящий от архитектуры целевой вычислительной системы, этот код распространяется по сети и интерпретируется на принимающей стороне).

Трансляторы с языка ассемблера («ассемблеры»)

Определение языка ассемблера

Язык ассемблера — это язык низкого уровня. Структура цепочек этого языка близка к машинным командам целевой вычислительной системы, где должна выполняться результирующая программа. Применение языка ассемблера позволяет разработчику управлять ресурсами (процессором, оперативной памятью, внешними устройствами и т. п.) целевой вычислительной системы на уровне машинных команд. Каждая команда исходной программы на языке ассемблера в результате компиляции преобразуется в одну машинную команду.

Компилятор с языка ассемблера зачастую просто называют «ассемблер».

Язык ассемблера, как правило, содержит мнемонические коды машинных команд. Чаще всего используется англоязычная мнемоника команд. Поэтому язык ассемблера раньше носил название «язык мнемокодов» (сейчас это название уже практически не употребляется). Все команды в каждом языке ассемблера можно разбить на две группы. В первую группу входят команды языка, которые в процессе трансляции преобразуются в машинные команды; вторую группу составляют специальные команды языка, которые в машинные команды не преобразуются, но используются компилятором для выполнения задач компиляции (таких, например, как распределение памяти).

Синтаксис языка ассемблера чрезвычайно прост. Команды исходной программы записываются обычно таким образом, чтобы на одной строке программы располагалась одна команда. Каждая команда языка ассемблера, как правило, может быть разделена на три составляющие, расположенные последовательно одна за другой: поле метки, код операции и поле операндов. Компилятор с языка ассемблера предусматривает также наличие во входной программе комментариев [64, 69].

Поле метки содержит идентификатор, представляющий собой метку, либо является пустым. Каждый идентификатор метки может встречаться в программе на языке ассемблера только один раз. Метка считается описанной там, где она непосредственно встретилась в программе (предварительное описание меток не требуется). Метка может быть использована для передачи управления на помеченную ею команду . Нередко метка отделяется от остальной части команды специальным разделителем (чаще всего — двоеточием «:»).

Код операции всегда представляет собой строго определенную мнемонику одной из возможных команд процессора или также строго определенную команду самого компилятора. Код операции записывается алфавитными символами входного языка. Чаще всего его длина составляет 3—4, реже — 5 или 6 символов.

Поле операндов либо является пустым, либо представляет собой список из одного, двух, реже — трех операндов. Количество операндов строго определено и зависит от кода операции — каждая операция языка ассемблера предусматривает строго заданное число своих операндов. Каждому из этих вариантов соответствуют безадресные, одноадресные, двухадресные или трехадресные команды (большее число операндов практически не используется, поскольку в современных компьютерах даже трехадресные команды встречаются редко). В качестве операндов могут выступать идентификаторы или константы.

Среди всех возможных идентификаторов, используемых для обозначения операндов команд на языке ассемблера, выделяется набор заданных предопределенных идентификаторов. Эти предопределенные идентификаторы, в зависимости от архитектуры целевой вычислительной системы, обозначают регистры, порты, ячейки памяти и т. п. Чаще всего предопределенные идентификаторы используются для обозначения регистров процессора, доступных программисту.

Реализация компиляторов с языка ассемблера

Компилятор с языка ассемблера по своей структуре и назначению ничем не отличается от всех прочих компиляторов. Однако язык ассемблера имеет ряд особенностей, которые упрощают реализацию компилятора для этого языка по сравнению с любым компилятором для языка высокого уровня.

Первой особенностью языка ассемблера является то, что ряд идентификаторов в нем выделяется специально для обозначения регистров процессора . Такие идентификаторы, с одной стороны, не требуют предварительного описания, но, с другой, они не могут быть использованы пользователем для иных целей. Набор этих идентификаторов предопределен для каждого языка ассемблера.

Иногда язык ассемблера допускает использование в качестве операндов определенных ограниченных сочетаний обозначений регистров, переменных и констант , которые объединены некоторыми знаками операций. Такие сочетания чаще всего

используются для обозначения типов адресации, допустимых в машинных командах целевой вычислительной системы.

Например, следующая последовательность команд

```
datas   db    16  dup(0)
loops:  mov   datas[bx+4],cx
        dec    bx
        jnz    loops
```

представляет собой пример последовательности команд языка ассемблера процессоров семейства Intel 80x86. Здесь присутствуют команда описания набора данных (`db`), метка (`loops`), коды операций (`mov`, `dec` и `jnz`). Операндами являются идентификатор набора данных (`datas`), обозначения регистров процессора (`bx` и `cx`), метка (`loops`) и константа (`4`). Составной операнд `datas[bx+4]` отображает косвенную адресацию набора данных `datas` по базовому регистру `bx` со смещением `4`.

Подобный синтаксис языка может быть описан с помощью регулярной или простейшей КС-грамматики. Поэтому построение распознавателя для языка ассемблера не представляет труда. По этой же причине в компиляторах с языка ассемблера лексический и синтаксический разборы, как правило, совмещены в один распознаватель.

Семантика языка ассемблера целиком и полностью определяется целевой вычислительной системой, на которую ориентирован данный язык. Семантика языка ассемблера определяет, какая машинная команда соответствует каждой команде языка ассемблера, а также, какие операнды и в каком количестве допустимы для того или иного кода операции.

Поэтому семантический анализ в компиляторе с языка ассемблера так же прост, как и синтаксический. Основной его задачей является проверить допустимость операндов для каждого кода операции, а также проверить, что все идентификаторы и метки, встречающиеся во входной программе, описаны и обозначающие их идентификаторы не совпадают с предопределенными идентификаторами, используемыми для обозначения кодов операции и регистров процессора.

Именно простота построения компилятора определила тот факт, что компиляторы с языка ассемблера исторически явились первыми компиляторами, созданными для ЭВМ. Существует также ряд других особенностей, которые присущи именно языкам ассемблера и упрощают построение компиляторов для них.

Во-первых, в компиляторах с языка ассемблера не выполняется дополнительная идентификация переменных — все переменные языка сохраняют имена, присвоенные им пользователем. За уникальность имен в исходной программе отвечает ее разработчик, семантика языка никаких дополнительных требований на этот процесс не накладывает.

Во-вторых, в компиляторах с языка ассемблера предельно упрощено распределение памяти. Компилятор с языка ассемблера работает только со статической памятью. Если используется динамическая память, то для работы с нею нужно использовать соответствующую библиотеку или функции ОС, а за распределение памяти отвечает разработчик исходной программы. За передачу параметров и организацию дисплея памяти процедур и функций также отвечает разработчик исходной программы. Он же должен позаботиться и об отделении данных от кода программы — компиля-

тор с языка ассемблера, в отличие от компиляторов с языков высокого уровня, автоматически такого разделения не выполняет (более подробно вопросы распределения памяти компилятором рассмотрены в разделе «Распределение памяти» главы 5).

В-третьих, на этапе генерации кода в компиляторе с языка ассемблера не производится оптимизация, поскольку разработчик исходной программы сам отвечает за организацию вычислений, последовательность машинных команд и распределение регистров процессора.

Компиляторы с языка ассемблера реализуются чаще всего по двухпроходной схеме. На первом проходе компилятор выполняет разбор исходной программы, ее преобразование в машинные коды и одновременно заполняет таблицу идентификаторов. Но на первом проходе в машинных командах остаются незаполненными адреса тех операндов, которые размещаются в оперативной памяти. На втором проходе компилятор заполняет эти адреса и одновременно обнаруживает неописанные идентификаторы. Это связано с тем, что метка может быть указана в программе после того, как она первый раз была использована (переход вперед). Тогда ее адрес еще не известен на момент построения машинной команды, а поэтому требуется второй проход.

Для облегчения труда разработчика исходных программ на языке ассемблера используются макрокоманды. Практически все современные компиляторы с языков ассемблера предусматривают в своем составе развитый препроцессор, выполняющий макроподстановки по тексту исходной программы. Развитая система макрокоманд позволяет существенно расширить возможности компиляторов с языка ассемблера, но при этом следует всегда помнить, что макрокоманды представляют собой обработку текста исходной программы, а не функции, помещаемые в результирующую программу.

Макроязыки и макрогенерация

Определения макрокоманд и макрогенерации

Разработка программ на языке ассемблера — достаточно трудоемкий процесс, требующий зачастую простого повторения одних и тех же многократно встречающихся операций. Для облегчения труда разработчика были созданы так называемые макрокоманды.

Макрокоманда представляет собой текстовую подстановку, в ходе выполнения которой каждый идентификатор определенного вида заменяется на цепочку символов из некоторого хранилища данных. Процесс выполнения макрокоманды называется *макрогенерацией*, а цепочка символов, получаемая в результате выполнения макрокоманды, — *макрорасширением*.

Процесс выполнения макрокоманд заключается в последовательном просмотре текста исходной программы, обнаружении в нем определенных идентификаторов и замене их на соответствующие строки символов. Причем выполняется именно текстовая замена одной цепочки символов (идентификатора) на другую цепочку символов (строку). Такая замена называется *макроподстановкой* [50, 64, 69].

Для того чтобы указать, какие идентификаторы на какие строки необходимо заменять, служат *макроопределения*. Макроопределения присутствуют непосредственно в тексте исходной программы. Они выделяются специальными ключевыми словами

либо разделителями, которые не могут встречаться нигде больше в тексте программы. В процессе обработки все макроопределения полностью исключаются из текста входной программы, а содержащаяся в них информация запоминается в хранилище данных для обработки в процессе выполнения макрокоманд.

Макроопределение может содержать параметры. Тогда каждая соответствующая ему макрокоманда должна при вызове содержать строку символов вместо каждого параметра. Эта строка подставляется при выполнении макрокоманды везде, где в макроопределении встречается соответствующий параметр. В качестве параметра макрокоманды может оказаться другая макрокоманда, тогда она будет рекурсивно вызвана всякий раз, когда необходимо выполнить подстановку параметра. В принципе макрокоманды могут образовывать последовательность рекурсивных вызовов, аналогичную последовательности рекурсивных вызовов процедур и функций, но только вместо вычислений и передачи параметров они выполняют текстовые подстановки[1].

Макрокоманды и макроопределения обрабатываются специальным модулем, называемым *макропроцессором* ,или *макрогенератором*. Макрогенератор получает на вход текст исходной программы, содержащий макроопределения и макрокоманды, а на выходе его появляется текст макрорасширения исходной программы, не содержащий макроопределений и макрокоманд. Оба текста являются текстами исходной программы на входном языке, никакая другая обработка не выполняется.

Примеры макрокоманд

Синтаксис макрокоманд и макроопределений зависит от реализации макропроцессора. Но сам принцип выполнения макроподстановок в тексте программы неизменен и не зависит от их синтаксиса.

Например, следующий текст макроопределения определяет макрокоманду `push_0` в языке ассемблера процессора типа Intel 8086:

```
push_0  macro
        xor    ax,ax
        push   ax
        endm
```

Семантика этой макрокоманды заключается в записи числа «0» в стек через регистр процессора `ax`. Тогда везде в тексте программы, где встретится макрокоманда

```
push_0
```

она будет заменена в результате макроподстановки на последовательность команд

```
xor    ax,ax
push   ax
```

Это самый простой вариант макроопределения. Существует возможность создавать более сложные макроопределения с параметрами. Одно из таких макроопределений описано ниже:

[1] Глубина такой рекурсии, как правило, сильно ограничена. На последовательность рекурсивных вызовов макрокоманд налагаются обычно существенно более жесткие ограничения, чем на последовательность рекурсивных вызовов процедур и функций, которая при стековой организации дисплея памяти ограничена только размером стека передачи параметров.

```
add_abx macro  x1,x2
        add    ax,x1
        add    bx,x1
        add    cx,x2
        push   ax
        endm
```

Тогда в тексте программы макрокоманда также должна быть указана с соответствующим числом параметров. В данном примере макрокоманда

```
add_abx   4,8
```

будет в результате макроподстановки заменена на последовательность команд

```
add    ax,4
add    bx,4
add    cx,8
push   ax
```

Во многих макропроцессорах возможны более сложные конструкции, которые могут содержать локальные переменные и метки. Примером такой конструкции может служить макроопределение

```
loop_ax macro   x1,x2,y1
        local   loopax
        mov     ax,x1
        xor     bx,bx
loopax: add     bx,y1
        sub     ax,x2
        jge     loopax
        endm
```

Здесь метка `loopax` является локальной, определенной только внутри данного макроопределения. В этом случае уже не может быть выполнена простая текстовая подстановка макрокоманды в текст программы, поскольку если данную макрокоманду выполнить дважды, то это приведет к появлению в тексте программы двух одинаковых меток `loopax`. В таком варианте макрогенератор должен использовать более сложные методы текстовых подстановок, аналогичные тем, что используются в компиляторах при идентификации лексических элементов входной программы, чтобы дать всем возможным локальным переменным и меткам макрокоманд уникальные имена в пределах всей программы.

Макроязыки и препроцессоры

Макроопределения и макрокоманды нашли применение не только в языках ассемблера, но и во многих языках высокого уровня. Т ам их обрабатывает специальный модуль, называемый препроцессором языка (например, широко известен препроцессор языка C). Принцип обработки остается тем же самым, что и для программ на языке ассемблера, — препроцессор выполняет текстовые подстановки непосредственно над строками самой исходной программы.

В языках высокого уровня макроопределения должны быть отделены от текста самой исходной программы, чтобы препроцессор не мог спутать их с синтаксическими конструкциями входного языка. Для этого используются либо специальные символы и команды (команды препроцессора), которые никогда не могут встречаться в тексте исходной программы, либо макроопределения встречаются внутри незначащей части исходной программы — входят в состав комментариев (такая реализация существует, например, в компиляторе с языка P ascal, созданном фирмой Borland). Макрокоманды, напротив, могут встречаться в произвольном месте исходного текста программы, и синтаксически их вызов может не отличаться от вызова функций во входном языке.

В совокупности все макроопределения и макрокоманды в тексте исходной программы являются предложениями некоторого входного языка, который называется *макроязыком*. Макроязык является чисто формальным языком, поскольку он лишен какого-либо смысла. Семантика макроязыка полностью определяется макрорасширением текста исходной программы, которое получается после обработки всех предложений макроязыка, а сами по себе предложения макроязыка не несут никакой семантики.

Препроцессор макроязыка (макрогенератор) в общем случае представляет собой транслятор, на вход которого поступает исходная программа, содержащая макроопределения и макрокоманды, а резуль тирующей программой является макрорасширение программы на исходном языке. Общая схема работы макропроцессора соответствует схеме работы транслятора — он содержит лексический и синтаксический анализаторы для выделения предложений макроязыка в тексте исходной программы, таблицы идентификаторов для хранения макроопределений, генератор предложений исходного языка для выполнения макроподстановок. Отличием является отсутствие семантической обработки входного макроязыка и фазы подготовки к генерации кода.

В простейшем варианте реализации макрокоманд, когда выполняются только макроподстановки без параметров и внутренних переменных, препроцессор макроязыка может быть построен на основе регулярной грамматики и реализован в виде конечного автомата. Более сложные препроцессоры предусматривают синтаксический анализ входных конструкций макроязыка, сопоставимый по сложности с анализом предложений входного языка. Их реализация выполняется на тех же принципах, что и реализация синтаксических анализаторов в компиляторах для языков программирования.

Макрогенератор, или препроцессор, чаще всего не существует в виде отдельного программного модуля, а входит в состав компилятора. Именно макрорасширение исходной программы поступает на вход компилятора и является для него исходной программой. Макрорасширение исходной программы обычно недоступно ее разработчику. Более того, макроподстановки могут выполняться последовательно при разборе исходного текста на первом проходе компилятора вместе с разбором всего текста программы, и тогда макрорасширение исходной программы в целом может и вовсе не существовать как таковое.

Следует помнить, что, несмотря на схожесть синтаксиса вызова, макрокоманды принципиально отличаются от процедур и функций, поскольку не порождают результирующего кода, а представляют собой текстовую подстановку , выполняемую прямо в тексте исходной программы. Резуль тат вызова функции и макрокоманды может из-за этого серьезно различаться.

Рассмотрим пример на языке С.

Если описана функция

```
int f1(int a) { return a + a; }
```

и аналогичная ей макрокоманда

```
#define f2(a) ((a) + (a))
```

то результат их вызова не всегда будет одинаков.

Действительно, вызовы `j=f1(i)` и `j=f2(i)` (где `i` и `j` — некоторые целочисленные переменные) приведут к одному и тому же резуль тату. Но вызовы `j=f1(++i)` и `j=f2(++i)` дадут разные значения переменной `j`. Дело в том, что поскольку `f2` — это макроопределение, то во втором случае будет выполнена текстовая подстановка, которая приведет к последовательности операторов `j=((++i) + (++i))`. Видно, что в этой последовательности операция `++i` будет выполнена дважды, в отличие от вызова функции `f1(++i)`, где она выполняется только один раз.

Таблицы идентификаторов. Организация таблиц идентификаторов

Назначение и особенности построения таблиц идентификаторов

Проверка правильности семантики и генерация кода требуют знания характеристик переменных, констант, функций и других элементов, встречающихся в программе на исходном языке. Все эти элементы в исходной программе, как правило, обозначаются идентификаторами. Выделение идентификаторов и других элементов исходной программы происходит в фазе лексического анализа . Их характеристики определяются в фазах синтаксического разбора, семантического анализа и подготовки к генерации кода. Состав возможных характеристик и методы их определения зависят от семантики входного языка.

В любом случае компилятор должен иметь возможность хранить все найденные идентификаторы и связанные с ними характеристики в течение всего процесса компиляции, чтобы иметь возможность использовать их на различных фазах компиляции. Для этой цели, как было сказано выше, в компиляторах используются специальные хранилища данных, называемые *таблицами символов*, или *таблицами идентификаторов*.

Любая таблица идентификаторов состоит из набора полей, количество которых равно количесиву различных идентификаторов, найденных в исходной программе. Каждое поле содержит в себе полную информацию о данном элементе таблицы. Компилятор может работать с одной или несколькими таблицами идентификаторов — их количество зависит от реализации компилятора. Например, можно организовывать различные таблицы идентификаторов для различных модулей исходной программы или для различных типов элементов входного языка.

Состав информации, хранимой в таблице идентификаторов для каждого элемента исходной программы, зависит от семантики входного языка и типа элемента. Примеры такой информации указаны в [5, 18, 34, 42, 58, 59]. Конкретное наполнение

таблиц идентификаторов зависит от реализации компилятора. Не вся информация, хранимая в таблице идентификаторов, заполняется компилятором сразу — он может несколько раз выполнять обращение к данным в таблице идентификаторов на различных фазах компиляции. Например, имена переменных могут быть выделены на фазе лексического анализа, типы данных для переменных — на фазе синтаксического разбора, а область памяти связывается с переменной только на фазе подготовки к генерации кода. На различных фазах компиляции компилятор вынужден многократно обращаться к таблице для поиска информации и записи новых данных.

ВНИМАНИЕ

Компилятору приходится выполнять поиск необходимого элемента в таблице идентификаторов по имени чаще, чем помещать новый элемент в таблицу потому что каждый идентификатор может быть описан только один раз, а использован — несколько раз.

Отсюда можно сделать вывод, что таблицы идентификаторов должны быть организованы таким образом, чтобы компилятор имел возможность максимально быстрого поиска нужного ему элемента [5, 42].

Простейшие методы построения таблиц идентификаторов

Простейший способ организации таблицы состоит в том, чтобы добавлять элементы в порядке их поступления. Тогда таблица идентификаторов будет представлять собой неупорядоченный массив информации, каждая ячейка которого будет содержать данные о соответствующем элементе таблицы.

Поиск нужного элемента в таблице будет в этом случае заключаться в последовательном сравнении искомого элемента с каждым элементом таблицы, пока не будет найден подходящий. Тогда, если за единицу времени принять время, затрачиваемое компилятором на сравнение двух элементов (как правило, это сравнение двух строк), для таблицы, содержащей N элементов, в среднем будет выполнено N/2 сравнений [11].

Заполнение такой таблицы будет происходить элементарно просто — добавлением нового элемента в ее конец, и время, требуемое на добавление элемента ($T_з$), не будет зависеть от числа элементов в таблице N. Но если N велико, то поиск потребует значительных затрат времени. Время поиска ($T_п$) в такой таблице можно оценить как $T_п = O(N)$. Поскольку поиск в таблице идентификаторов является наиболее часто выполняемой компилятором операцией, а количество различных идентификаторов в реальной исходной программе достаточно велико (от нескольких сотен до нескольких тысяч элементов), то такой способ организации таблиц идентификаторов является неэффективным.

Поиск может быть выполнен более эффективно, если элементы таблицы упорядочены (отсортированы) согласно некоторому естественному порядку. Поскольку поиск осуществляется по имени идентификатора, наиболее естественным решением будет расположить элементы таблицы в прямом или обратном алфавитном порядке. Эффективным методом поиска в упорядоченном списке из N элементов является *бинарный*, или *логарифмический, поиск*.

Алгоритм логарифмического поиска заключается в следующем: искомый символ сравнивается с элементом (N + 1)/2 в середине таблицы. Если этот элемент не является искомым, то мы должны просмотреть только блок элементов, пронумерованных от 1 до (N + 1)/2—1, или блок элементов от (N + 1)/2 + 1 до N в зависимости от того, меньше или больше искомый элемент того, с которым его сравнили. Затем процесс повторяется над нужным блоком в два раза меньшего размера. Так продолжается до тех пор, пока либо искомый элемент будет найден, либо алгоритм дойдет до очередного блока, содержащего один или два элемента.

Так как на каждом шаге число элементов, которые могут содержать искомый элемент сокращается в 2 раза, максимальное число сравнений равно $1 + \log_2(N)$. Тогда время поиска элемента в таблице идентификаторов можно оценить как $T_п = O(\log_2 N)$. Для сравнения: при N = 128 бинарный поиск потребует максимум 8 сравнений, а поиск в неупорядоченной таблице — в среднем 64 сравнения.

Этот метод называют бинарным поиском, поскольку на каждом шаге объем рассматриваемой информации сокращается в два раза, а логарифмическим — поскольку время, затрачиваемое на поиск нужного элемента в массиве, имеет логарифмическую зависимость от общего количества элементов в нем.

Недостатком логарифмического поиска является требование упорядочивания таблицы идентификаторов. Так как массив информации, в котором выполняется поиск, должен быть упорядочен, то время его заполнения уже будет зависеть от числа элементов в массиве. Т аблица идентификаторов зачастую просматривается компилятором еще до того, как она заполнена полностью, поэтому требуется, чтобы условие упорядоченности выполнялось на всех этапах обращения к ней. Следовательно, для построения такой таблицы можно пользоваться только алгоритмом прямого упорядоченного включения элементов.

При добавлении каждого нового элемента в таблицу сначала надо определить место, куда поместить новый элемент , а потом выполнить перенос части информации в таблице, если элемент добавляется не в ее конец. Если пользоваться стандартными алгоритмами, применяемыми для организации упорядоченных массивов данных [11], то среднее время, необходимое на помещение всех элементов в таблицу, можно оценить следующим образом:

$$T_з = O(N*\log_2 N) + k*O(N^2).$$

Здесь k — некоторый коэффициент, отражающий соотношение между временем, затрачиваемым на выполнение операции сравнения, и временем на операцию переноса данных.

В итоге при организации логарифмического поиска в таблице идентификаторов мы добиваемся существенного сокращения времени поиска нужного элемента за счет увеличения времени на помещение нового элемента в таблицу . Поскольку добавление новых элементов в таблицу идентификаторов происходит существенно реже[1], чем обращение к ним, этот метод следует признать более эффективным, чем метод организации неупорядоченной таблицы.

[1] Как минимум при добавлении нового идентификатора в таблицу компилятор должен проверить, существует или нет там такой идентификатор, так как в большинстве языков программирования ни один идентификатор не может быть описан более одного раза. Следовательно, каждая операция добавления нового элемента влечет, как правило, не менее одной операции поиска.

Построение таблиц идентификаторов по методу бинарного дерева

Можно сократить время поиска искомого элемента в таблице идентификаторов, не увеличивая значительно время, необходимое на ее заполнение. Для этого надо отказаться от организации таблицы в виде непрерывного массива данных.

Существует метод построения таблиц, при котором таблица имеет форму бинарного дерева. Каждый узел дерева представляет собой элемент таблицы, причем корневой узел является первым элементом, встреченным при заполнении таблицы. Дерево называется бинарным, так как каждая вершина в нем может иметь не более двух ветвей. Для определенности будем называть две ветви «правая» и «левая».

Рассмотрим алгоритм заполнения бинарного дерева. Будем считать, что алгоритм работает с потоком входных данных, содержащим идентификаторы (в компиляторе этот поток данных порождается в процессе разбора текста исходной программы). Первый идентификатор, как уже было сказано, помещается в вершину дерева. Все дальнейшие идентификаторы попадают в дерево по следующему алгоритму.

Шаг 1. Выбрать очередной идентификатор из входного потока данных. Если очередного идентификатора нет, то построение дерева закончено.

Шаг 2. Сделать текущим узлом дерева корневую вершину.

Шаг 3 . Сравнить очередной идентификатор с идентификатором, содержащимся в текущем узле дерева.

Шаг 4. Если очередной идентификатор меньше, то перейти к шагу 5, если равен — сообщить об ошибке и прекратить выполнение алгоритма (двух одинаковых идентификаторов быть не должно!), иначе — перейти к шагу 7.

Шаг 5. Если у текущего узла существует левая вершина, то сделать ее текущим узлом и вернуться к шагу 3, иначе перейти к шагу 6.

Шаг 6. Создать новую вершину, поместить в нее очередной идентификатор, сделать эту новую вершину левой вершиной текущего узла и вернуться к шагу 1.

Шаг 7. Если у текущего узла существует правая вершина, то сделать ее текущим узлом и вернуться к шагу 3, иначе перейти к шагу 8.

Шаг 8. Создать новую вершину, поместить в нее очередной идентификатор, сделать эту новую вершину правой вершиной текущего узла и вернуться к шагу 1.

Рассмотрим в качестве примера последовательность идентификаторов `GA`, `D1`, `M22`, `E`, `A12`, `BC`, `F`. На рис. 2.2 проиллюстрирован весь процесс построения бинарного дерева для этой последовательности идентификаторов.

Поиск нужного элемента в дереве выполняется по алгоритму, схожему с алгоритмом заполнения дерева.

Шаг 1. Сделать текущим узлом дерева корневую вершину.

Шаг 2. Сравнить искомый идентификатор с идентификатором, содержащимся в текущем узле дерева.

Шаг 3. Если идентификаторы совпадают, то искомый идентификатор найден, алгоритм завершается, иначе надо перейти к шагу 4.

Шаг 4. Если искомый идентификатор меньше, то перейти к шагу 5, иначе — перейти к шагу 6.

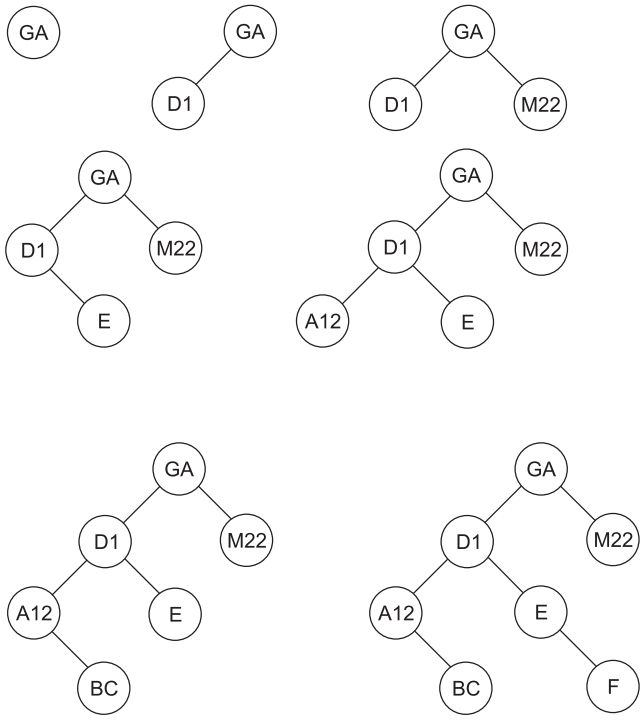

Рис. 2.2. Заполнение бинарного дерева для последовательности идентификаторов GA, D1, M22, E, A12, BC, F

Шаг 5 . Если у текущего узла существует левая вершина, то сделать ее текущим узлом и вернуться к шагу 2, иначе искомый идентификатор не найден, алгоритм завершается.

Шаг 6 . Если у текущего узла существует правая вершина, то сделать ее текущим узлом и вернуться к шагу 2, иначе искомый идентификатор не найден, алгоритм завершается.

Например, произведем поиск в дереве, изображенном на рис. 2.2, идентификатора `A12`. Берем корневую вершину (она становится текущим узлом), сравниваем идентификаторы `GA` и `A12`. Искомый идентификатор меньше — текущим узлом становится левая вершина `D1`. Опять сравниваем идентификаторы. Искомый идентификатор меньше — текущим узлом становится левая вершина `A12`. При следующем сравнении искомый идентификатор найден.

Если искать отсутствующий идентификатор — например `A11`, — то поиск опять пойдет от корневой вершины. Сравниваем идентификаторы `GA` и `A11`. Искомый идентификатор меньше — текущим узлом становится левая вершина `D1`. Опять сравниваем идентификаторы. Искомый идентификатор меньше — текущим узлом становится левая вершина `A12`. Искомый идентификатор меньше, но левая вершина у узла `A12` отсутствует, поэтому в данном случае искомый идентификатор не найден.

Для данного метода число требуемых сравнений и форма получившегося дерева зависят от того порядка, в котором поступают идентификаторы. Например, если в рас-

смотренном выше примере вместо последовательности идентификаторов GA, D1, M22, E, A12, BC, F взять последовательность A12, GA, D1, M22, E, BC, F, то полученное дерево будет иметь иной вид. А если в качестве примера взять последовательность идентификаторов A12, BC, D1, E, F, GA, M22, то дерево выродится в упорядоченный однонаправленный связный список. Эта особенность является недостатком данного метода организации таблиц идентификаторов. Другим недостатком является необходимость работы с динамическим выделением памяти при построении дерева.

Если предположить, что последовательность идентификаторов в исходной программе является статистически неупорядоченной (что в целом соответствует действительности), то можно считать, что построенное бинарное дерево будет невырожденным. Тогда среднее время на заполнение дерева ($T_з$) и на поиск элемента в нем ($T_п$) можно оценить следующим образом [4 т.2]:

$$T_з = O(N*\log_2 N),$$
$$T_п = O(\log_2 N).$$

В целом метод бинарного дерева является довольно удачным механизмом для организации таблиц идентификаторов. Он нашел свое применение в ряде компиляторов. Иногда компиляторы строят несколько различных деревьев для идентификаторов разных типов и разной длины [18, 58, 59].

Хеш-функции и хеш-адресация

Принципы работы хеш-функций

Логарифмическая зависимость времени поиска и времени заполнения таблицы идентификаторов — это самый хороший резуль тат, которого можно достичь за счет применения различных методов организации таблиц. Однако в реальных исходных программах количество идентификаторов столь велико, что даже логарифмическую зависимость времени поиска от их числа нельзя признать удовлетворительной. Необходимы более эффективные методы поиска информации в таблице идентификаторов.

Лучших результатов можно достичь, если применить методы, связанные с использованием хеш-функций и хеш-адресации.

Хеш-функцией F называется некоторое отображение множества входных элементов **R** на множество целых неотрицательных чисел **Z**: $F(r) = n$, $r \in \mathbf{R}$, $n \in \mathbf{Z}$. Множество допустимых входных элементов **R** называется областью определения хеш-функции. Множеством значений хеш-функции F называется подмножество **M** из множества целых неотрицательных чисел **Z**: $\mathbf{M} \subseteq \mathbf{Z}$, содержащее все возможные значения, возвращаемые функцией F: $\forall r \in \mathbf{R}$: $F(r) \in \mathbf{M}$ и $\forall m \in \mathbf{M}$: $\exists r \in \mathbf{R}$: $F(r)=m$. Процесс отображения области определения хеш-функции на множество значений называется «хешированием».

Сам термин «хеш-функция» происходит от английского термина «hash function» (hash — «мешать», «смешивать», «путать»). Вместо термина «хеширование» иногда используются термины «рандомизация», «переупорядочивание».

При работе с таблицей идентификаторов хеш-функция должна выполнять отображение имен идентификаторов на множество целых неотрицательных чисел. Областью определения хеш-функции будет множество всех возможных имен идентификаторов.

Хеш-адресация заключается в использовании значения, возвращаемого хеш-функцией, в качестве адреса ячейки из некоторого массива данных. Т огда размер массива данных должен соответствовать области значений используемой хеш-функции. Следовательно, в реальном компиляторе область значений хеш-функции никак не должна превышать размер доступного адресного пространства компьютера.

Метод организации таблиц идентификаторов , основанный на использовании хеш-адресации, заключается в размещении каждого элемента таблицы в ячейке, адрес которой возвращает хеш-функция, вычисленная для этого элемента. Тогда в идеальном случае для размещения любого элемента в таблице идентификаторов достаточно только вычислить его хеш-функцию и обратиться к нужной ячейке массива данных. Для поиска элемента в таблице необходимо вычислить хеш-функцию для искомого элемента и проверить, не является ли заданная ею ячейка массива пустой: если она не пуста — элемент найден, если пуста — не найден. Первоначально таблица идентификаторов должна быть заполнена информацией, которая позволила бы говорить о том, что все ее ячейки являются пустыми.

На рис. 2.3 проиллюстрирован метод организации таблиц идентификаторов с использованием хеш-адресации. Трем различным идентификаторам, A_1, A_2, A_3, соответствуют на рисунке три значения хеш-функции, n_1, n_2, n_3. В ячейки, адресуемые n_1, n_2, n_3, помещается информация об идентификаторах A_1, A_2, A_3. При поиске идентификатора A_3 вычисляется значение адреса n_3 и выбираются данные из соответствующей ячейки таблицы.

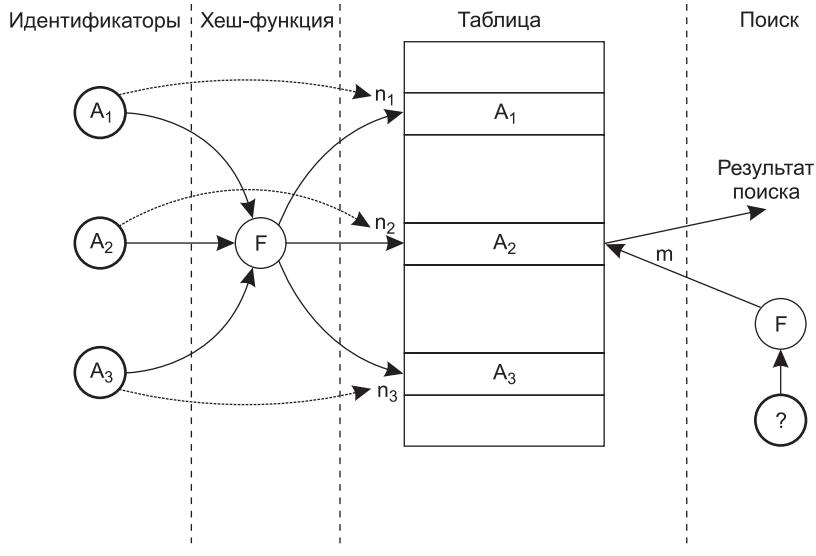

Рис. 2.3. Организация таблицы идентификаторов с использованием хеш-адресации

Этот метод весьма эффективен, поскольку как время размещения элемента в таблице, так и время его поиска определяются только временем, затрачиваемым на вычисление хеш-функции, которое в общем случае несопоставимо меньше времени, необходимого для выполнения многократных сравнений элементов таблицы.

Метод имеет два очевидных недостатка. Первый из них — неэффективное использование объема памяти под таблицу идентификаторов: размер массива для ее хранения должен соответствовать области значений хеш-функции, в то время как реально хранимых в таблице идентификаторов может быть существенно меньше. Второй недостаток — необходимость соответствующего разумного выбора хеш-функции. Этому существенному вопросу посвящены следующие два подраздела.

Построение таблиц идентификаторов на основе хеш-функций

Существуют различные варианты хеш-функций . Получение резуль тата хеш-функции — «хеширование» — обычно достигается за счет выполнения над цепочкой символов некоторых простых арифметических и логических операций. Самой простой хеш-функцией для символа является код внутреннего представления в компьютере литеры символа. Эту хеш-функцию можно использовать и для цепочки символов, выбирая первый символ в цепочке. Так, если двоичное ASCII представление символа `A` есть двоичный код 00100001_2, то результатом хеширования идентификатора `ATable` будет код 00100001_2.

Хеш-функция, предложенная выше, очевидно не удовлетворительна: при использовании такой хеш-функции возникнет проблема — двум различным идентификаторам, начинающимся с одной и той же буквы, будет соответствовать одно и то же значение хеш-функции. Тогда при хеш-адресации в одну ячейку таблицы идентификаторов по одному и тому же адресу должны быть помещены два различных идентификатора, что явно невозможно. Такая ситуация, когда двум или более идентификаторам соответствует одно и то же значение хеш-функции, называется *коллизией*.

Естественно, что хеш-функция, допускающая коллизии, не может быть напрямую использована для хеш-адресации в таблице идентификаторов. Причем достаточно получить хотя бы один случай коллизии на всем множестве идентификаторов, чтобы такой хеш-функцией нельзя было пользоваться непосредственно. Но в примере взята самая элементарная хеш-функция. А возможно ли построить хеш-функцию, которая полностью исключала бы возникновение коллизий?

Очевидно, что для полного исключения коллизий хеш-функция должна быть взаимно однозначной: каждому элементу из области определения хеш-функции должно соответствовать одно значение из ее множества значений и каждому значению из множества значений этой функции должен соответствовать только один элемент из области ее определения. Т огда любым двум произвольным элементам из области определения хеш-функции будут всегда соответствовать два различных ее значения. Теоретически для идентификаторов такую хеш-функцию построить можно, так как и область определения хеш-функции (все возможные имена идентификаторов), и область ее значений (целые неотрицательные числа) являются бесконечными счетными множествами. Теоретически можно организовать взаимно однозначное отображение одного счетного множества на другое[1].

Практически существует ограничение, делающее создание взаимно однозначной хеш-функции для идентификаторов невозможным. Дело в том, что в реальности

[1] Элементарным примером такого отображения для строки символов является сопоставление ей двоичного числа, полученного путем конкатенации кодов символов, входящих в строку. Фактически сама строка тогда будет выступать адресом при хеш-адресации. Практическая ценность такого отображения весьма сомнительна.

область значений любой хеш-функции ограничена размером доступного адресного пространства компьютера. При организации хеш-адресации значение, используемое в качестве адреса таблицы идентификаторов, не может выходить за пределы, заданные разрядностью адреса компьютера[1]. Множество адресов любого компьютера с традиционной архитектурой может быть велико, но всегда конечно, то есть ограничено. Организовать взаимно однозначное отображение бесконечного множества на конечное даже теоретически невозможно. Можно учесть, что длина принимаемой во внимание части имени идентификатора в реальных компиляторах также практически ограничена — обычно она лежит в пределах от 32 до 128 символов (то есть и область определения хеш-функции конечна). Но и тогда количество элементов в конечном множестве, составляющем область определения функции, будет превышать их количество в конечном множестве области значений функции (количество всех возможных идентификаторов все равно больше количества допустимых адресов в современных компьютерах). Таким образом, создать взаимно однозначную хеш-функцию практически ни в каком варианте невозможно. Следовательно, невозможно избежать возникновения коллизий.

Для решения проблемы коллизии можно использовать много способов. Одним из них является метод *рехеширования* (или расстановки). Согласно этому методу, если для элемента A адрес h(A), вычисленный с помощью хеш-функции h, указывает на уже занятую ячейку , необходимо вычислить значение функции $n_1 = h_1(A)$ и проверить занятость ячейки по адресу n_1. Если и она занята, то вычисляется значение $h_2(A)$, и так до тех пор, пока либо не будет найдена свободная ячейка, либо очередное значение $h_i(A)$ совпадет с h(A). В последнем случае считается, что таблица идентификаторов заполнена и места в ней больше нет — выдается информация об ошибке размещения идентификатора в таблице.

Такую таблицу идентификаторов можно организовать по следующему алгоритму размещения элемента .

Шаг 1. Вычислить значение хеш-функции n = h(A) для нового элемента A.

Шаг 2. Если ячейка по адресу n пустая, то поместить в нее элемент A и завершить алгоритм, иначе i := 1 и перейти к шагу 3.

Шаг 3. Вычислить $n_i = h_i(A)$. Если ячейка по адресу n_i пустая, то поместить в нее элемент A и завершить алгоритм, иначе перейти к шагу 4.

Шаг 4. Если $n = n_i$, то сообщить об ошибке и завершить алгоритм, иначе i : = i + 1 и вернуться к шагу 3.

Тогда поиск элемента A в таблице идентификаторов, организованной таким образом, будет выполняться по следующему алгоритму.

Шаг 1. Вычислить значение хеш-функции n = h(A) для искомого элемента A.

Шаг 2. Если ячейка по адресу n пустая, то элемент не найден, алгоритм завершен, иначе сравнить имя элемента в ячейке n с именем искомого элемента A. Если они совпадают, то элемент найден и алгоритм завершен, иначе i : = 1 и перейти к шагу 3.

[1] Можно, конечно, организовать адресацию с использованием внешних накопителей для организации виртуальной памяти, но накладные затраты для такой адресации будут весьма велики. И даже в таком варианте адресное пространство никогда не будет бесконечным.

Шаг 3. Вычислить $n_i = h_i(A)$. Если ячейка по адресу n_i пустая или $n = n_i$, то элемент не найден и алгоритм завершен, иначе сравнить имя элемента в ячейке n_i с именем искомого элемента A. Если они совпадают, то элемент найден и алгоритм завершен, иначе i := i + 1 и повторить шаг 3.

Алгоритмы размещения и поиска элемента схожи по выполняемым операциям. Поэтому они будут иметь одинаковые оценки времени, необходимого для их выполнения.

При такой организации таблиц идентификаторов в случае возникновения коллизии алгоритм размещает элементы в пустых ячейках таблицы, выбирая их определенным образом. При этом элементы могут попадать в ячейки с адресами, которые потом будут совпадать со значениями хеш-функции, что приведет к возникновению новых, дополнительных коллизий. Таким образом, количество операций, необходимых для поиска или размещения в таблице элемента, зависит от заполненности таблицы.

Для организации таблицы идентификаторов по методу рехеширования необходимо определить все хеш-функции h_i для всех i. Чаще всего функции h_i определяют как некоторые модификации хеш-функции h. Например, самым простым методом вычисления функции $h_i(A)$ является ее организация в виде $h_i(A) = (h(A) + p_i) \bmod N_m$, где p_i — некоторое вычисляемое целое число, а N_m — максимальное значение из области значений хеш-функции h. В свою очередь, самым простым подходом здесь будет положить $p_i = i$. Тогда получаем формулу $h_i(A) = (h(A) + i) \bmod N_m$. В этом случае при совпадении значений хеш-функции для каких-либо элементов поиск свободной ячейки в таблице начинается последовательно от текущей позиции, заданной хеш-функцией h(A).

Этот способ нельзя признать особенно удачным — при совпадении хеш-адресов элементы в таблице начинают группироваться вокруг них, что увеличивает число необходимых сравнений при поиске и размещении. Среднее время поиска элемента в такой таблице в зависимости от числа операций сравнения можно оценить следующим образом [18]:

$$T_п = O((1 - Lf/2)/(1 - Lf)).$$

Здесь Lf — (load factor) степень заполненности таблицы идентификаторов — отношение числа занятых ячеек N таблицы к максимально допустимому числу элементов в ней: $Lf = N/N_m$.

Рассмотрим в качестве примера ряд последовательных ячеек таблицы n_1, n_2, n_3, n_4, n_5 и ряд идентификаторов, которые надо разместить в ней, A_1, A_2, A_3, A_4, A_5 при условии, что $h(A_1) = h(A_2) = h(A_5) = n_1$; $h(A_3) = n_2$; $h(A_4) = n_4$. Последовательность размещения идентификаторов в таблице при использовании простейшего метода рехеширования показана на рис. 2.4. В итоге после размещения в таблице для поиска идентификатора A_1 потребуется 1 сравнение, для A_2 — 2 сравнения, для A_3 — 2 сравнения, для A_4 — 1 сравнение и для A_5 — 5 сравнений.

Даже такой примитивный метод рехеширования является достаточно эффективным средством организации таблиц идентификаторов при неполном заполнении таблицы. Имея, например, заполненную на 90 % таблицу для 1024 идентификаторов, в среднем необходимо выполнить 5,5 сравнения для поиска одного идентификатора, в то время как даже логарифмический поиск дает в среднем от 9 до 10 сравнений. Сравнительная эффективность метода будет еще выше при росте числа идентификаторов и снижении заполненности таблицы.

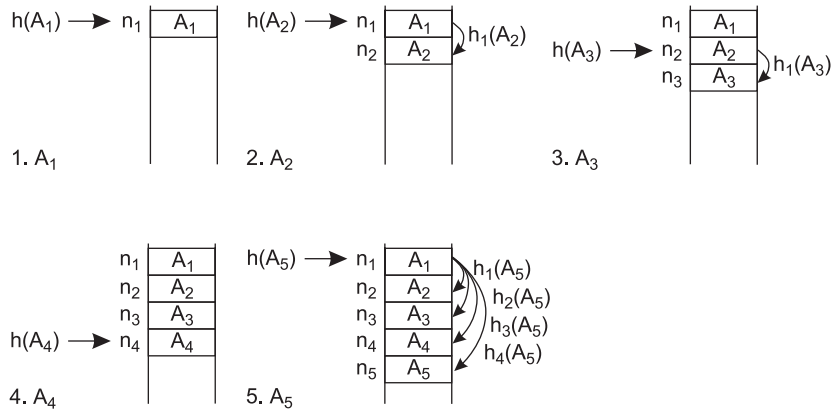

Рис. 2.4. Заполнение таблицы идентификаторов при использовании простейшего рехеширования

Среднее время на помещение одного элемента в таблицу и на поиск элемента в таблице можно снизить, если применить более совершенный метод рехеширования. Одним из таких методов является использование в качестве p_i для функции $h_i(A) = (h(A) + p_i) \bmod N_m$ последовательности псевдослучайных целых чисел $p_1, p_2, ..., p_k$. При хорошем выборе генератора псевдослучайных чисел длина последовательности k будет $k = N_m$. Тогда среднее время поиска одного элемента в таблице можно оценить следующим образом [18]:

$$E_п = O((1/Lf)*\log_2(1 - Lf)).$$

Существуют и другие методы организации функций рехеширования $h_i(A)$, основанные на квадратичных вычислениях или, например, на вычислении по формуле $h_i(A) = (h(A)*i) \bmod N_m$, если N_m — простое число. В целом рехеширование позволяет добиться неплохих результатов для эффективного поиска элемента в таблице (лучших, чем бинарный поиск и бинарное дерево), но эффективность метода сильно зависит от заполненности таблицы идентификаторов и качества используемой хеш-функции — чем реже возникают коллизии, тем выше эффективность метода. Требование неполного заполнения таблицы ведет к неэффективному использованию объема доступной памяти.

Построение таблиц идентификаторов по методу цепочек

Неполное заполнение таблицы идентификаторов при применении хеш-функций ведет к неэффективному использованию объема памяти, доступного компилятору. Причем объем неиспользуемой памяти будет тем выше, чем больше информации хранится для каждого идентификатора. Этого недостатка можно избежать, если дополнить таблицу идентификаторов некоторой промежуточной хеш-таблицей.

В ячейках хеш-таблицы может храниться либо пустое значение, либо указатель на некоторую область памяти из таблицы идентификаторов. Тогда хеш-функция вычисляет адрес, по которому происходит обращение сначала к хеш-таблице, а потом уже через нее — к самой таблице идентификаторов. Если соответствующая ячейка таблицы идентификаторов пуста, то ячейка хеш-таблицы будет содержать пустое

значение. Тогда вовсе не обязательно иметь в самой таблице идентификаторов ячейку для каждого возможного значения хеш-функции — таблицу можно сделать динамической так, чтобы ее объем рос по мере заполнения.

Такой подход позволяет добиться двух положительных результатов: во-первых, нет необходимости заполнять пустыми значениями таблицу идентификаторов — это можно сделать только для хеш-таблицы; во-вторых, каждому идентификатору будет соответствовать строго одна ячейка в таблице идентификаторов. Пустые ячейки в таком случае будут только в хеш-таблице, и объем неиспользуемой памяти не будет зависеть от объема информации, хранимой для каждого идентификатора, — для каждого значения хеш-функции будет расходоваться только память, необходимая для хранения одного указателя на основную таблицу идентификаторов.

На основе этой схемы можно реализовать еще один способ организации таблиц идентификаторов с помощью хеш-функций, называемый «*метод цепочек*». Для метода цепочек в таблицу идентификаторов для каждого элемента добавляется еще одно поле, в котором может содержаться ссылка на любой элемент таблицы. Первоначально это поле всегда пустое (никуда не указывает). Также для этого метода необходимо иметь одну специальную переменную, которая всегда указывает на первую свободную ячейку основной таблицы идентификаторов (первоначально — указывает на начало таблицы).

Метод цепочек работает по следующему алгоритму.

Шаг 1. Во все ячейки хеш-таблицы поместить пустое значение, таблица идентификаторов пуста, переменная `FreePtr` (указатель первой свободной ячейки) указывает на начало таблицы идентификаторов; i := 1.

Шаг 2. Вычислить значение хеш-функции n_i для нового элемента A_i. Если ячейка хеш-таблицы по адресу n_i пустая, то поместить в нее значение переменной `FreePtr` и перейти к шагу 5; иначе перейти к шагу 3.

Шаг 3. Положить j := 1, выбрать из хеш-таблицы адрес ячейки таблицы идентификаторов m_j и перейти к шагу 4.

Шаг 4. Для ячейки таблицы идентификаторов по адресу m_j проверить значение поля ссылки. Если оно пустое, то записать в него адрес из переменной `FreePtr` и перейти к шагу 5; иначе j := j + 1, выбрать из поля ссылки адрес m_j и повторить шаг 4.

Шаг 5. Добавить в таблицу идентификаторов новую ячейку, записать в нее информацию для элемента A_i (поле ссылки должно быть пустым), в переменную `FreePtr` поместить адрес за концом добавленной ячейки. Если больше нет идентификаторов, которые надо разместить в таблице, то выполнение алгоритма закончено, иначе i := i + 1 и перейти к шагу 2.

Поиск элемента в таблице идентификаторов, организованной таким образом, будет выполняться по следующему алгоритму.

Шаг 1. Вычислить значение хеш-функции n для искомого элемента A. Если ячейка хеш-таблицы по адресу n пустая, то элемент не найден и алгоритм завершен, иначе положить j := 1, выбрать из хеш-таблицы адрес ячейки таблицы идентификаторов m_j=n.

Шаг 2. Сравнить имя элемента в ячейке таблицы идентификаторов по адресу m_j с именем искомого элемента A. Если они совпадают, то искомый элемент найден и алгоритм завершен, иначе перейти к шагу 3.

Шаг 3. Проверить значение поля ссылки в ячейке таблицы идентификаторов по адресу m_j. Если оно пустое, то искомый элемент не найден и алгоритм завершен; иначе $j := j + 1$, выбрать из поля ссылки адрес m_j и перейти к шагу 2.

При такой организации таблиц идентификаторов в случае возникновения коллизии алгоритм размещает элементы в ячейках таблицы, связывая их друг с другом последовательно через поле ссылки. При этом элементы не могут попадать в ячейки с адресами, которые потом будут совпадать со значениями хеш-функции. Таким образом, дополнительные коллизии не возникают. В итоге в таблице возникают своеобразные цепочки связанных элементов, откуда происходит и название данного метода — «метод цепочек».

На рис. 2.5 проиллюстрировано заполнение хеш-таблицы и таблицы идентификаторов для примера, который ранее был рассмотрен на рис. 2.4 для метода простейшего рехеширования. После размещения в таблице для поиска идентификатора A_1 потребуется 1 сравнение, для A_2 — 2 сравнения, для A_3 — 1 сравнение, для A_4 — 1 сравнение и для A_5 — 3 сравнения (сравните с результатами простого рехеширования).

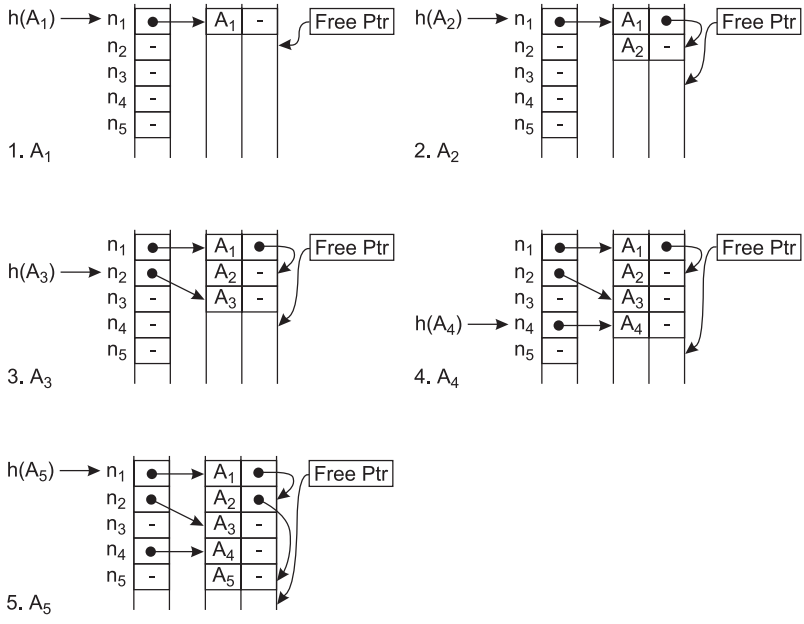

Рис. 2.5. Заполнение хеш-таблицы и таблицы идентификаторов при использовании метода цепочек

Метод цепочек является очень эффективным средством организации таблиц идентификаторов. Среднее время на размещение одного элемента и на поиск элемента в таблице для него зависит только от среднего числа коллизий, возникающих при вычислении хеш-функции. Накладные расходы памяти, связанные с необходимостью иметь одно дополнительное поле указателя в таблице идентификаторов на каждый ее элемент, можно признать вполне оправданными. Этот метод позволяет более экономно использовать память, но требует организации работы с динамическими массивами данных.

Комбинированные способы построения таблиц идентификаторов

Выше в примере была рассмотрена весьма примитивная хеш-функция, которую никак нельзя назвать удовлетворительной. Хорошая хеш-функция распределяет поступающие на ее вход идентификаторы равномерно на все имеющиеся в распоряжении адреса. Существует большое множество хеш-функций. Каждая из них стремится распределить адреса под идентификаторы по своему алгоритму, но, как было показано выше, идеального хеширования достичь невозможно.

В реальных компиляторах так или иначе используется хеш-адресация. Алгоритм хеш-функции составляет «ноу-хау» разработчиков компилятора. Обычно при разработке хеш-функции создатели компилятора стремятся свести к минимуму количество коллизий не на всем множестве возможных идентификаторов, а на тех их вариантах, которые наиболее часто встречаются во входных программах. Конечно, принять во внимание все допустимые исходные программы невозможно. Чаще всего выполняется статистическая обработка встречающихся имен идентификаторов на некотором множестве типичных исходных программ, а также принимаются во внимание соглашения о выборе имен идентификаторов, общепринятые для входного языка. Хорошая хеш-функция — это шаг к значительному ускорению работы компилятора, поскольку обращение к таблицам идентификаторов выполняется многократно на различных фазах компиляции.

То, какой конкретно метод применяется в компиляторе для организации таблиц идентификаторов, зависит от реализации компилятора. Компилятор может иметь несколько таблиц идентификаторов, организованных на основе различных методов.

Как правило, применяются комбинированные методы. В этом случае, как и для метода цепочек, в таблице идентификаторов организуется специальное дополнительное поле ссылки. Но, в отличие от метода цепочек, оно имеет несколько иное значение. При отсутствии коллизий для выборки информации из таблицы используется хеш-функция, поле ссылки остается пустым. Если же возникает коллизия, то с помощью поля ссылки организуется поиск идентификаторов, для которых значения хеш-функции совпадают по одному из рассмотренных выше методов. При хорошо построенной хеш-функции коллизии будут возникать редко, поэтому количество идентификаторов, для которых значения хеш-функции совпали, будет не столь велико. Тогда и время поиска одного среди них будет незначительным (в принципе при высоком качестве хеш-функции подойдет даже перебор по неупорядоченному списку).

Такой подход имеет преимущество по сравнению с методом цепочек, поскольку не требует использования промежуточной хеш-таблицы. Недостатком метода является необходимость работы с динамически распределяемыми областями памяти. Эффективность такого метода, очевидно, в первую очередь зависит от качества применяемой хеш-функции, а во вторую — от метода организации дополнительных хранилищ данных.

Хеш-адресация — это метод, который применяется не только для организации таблиц идентификаторов в компиляторах. Данный метод нашел свое применение и в операционных системах, и в системах управления базами данных. Интересующиеся читатели могут обратиться к соответствующей литературе [4 т.2, 5, 18, 59].

Контрольные вопросы и задачи

Вопросы

1. Перечислите основные этапы и фазы компиляции.
2. Верно ли, что любой компилятор является транслятором? Может ли существовать транслятор, который является компилятором? Может ли существовать компилятор, который является интерпретатором?
3. Можно ли построить компилятор, который не содержит лексический анализатор? Как могут быть связаны между собой лексический и синтаксический анализ?
4. Какие фазы работы компилятора, приведенного в этом пособии, будут отсутствовать у интерпретатора?
5. Какие особенности присущи компилятору с языка ассемблера? Какие фазы компиляции в нем могут отсутствовать?
6. От чего зависит количество проходов, необходимых компилятору для построения результирующей объектной программы на основе исходной программы? Как влияют на необходимое количество проходов синтаксис исходного языка программирования, семантика этого языка, архитектура целевой вычислительной системы?
7. Почему в Глобальной сети Интернет используются в основном языки программирования, предусматривающие интерпретацию исходного кода?
8. Почему для языка ассемблера, который имеет простую (часто регулярную) грамматику, чаще всего используется двухпроходный компилятор? Что мешает свести компиляцию исходного кода на языке ассемблера в один проход?
9. Зачем в макроопределении `#define f2(a) ((a)*(a))` параметр `a` взят в скобки, хотя операция этого не требует?
10. Какая информация может храниться в таблице идентификаторов?
11. Исходя из каких характеристик оценивается эффективность того или иного метода организации таблицы?
12. Какие существуют способы организации таблиц идентификаторов?
13. Что такое коллизия? Почему она происходит при использовании хеш-функций для организации таблиц идентификаторов?
14. В чем заключаются преимущества и недостатки метода цепочек по сравнению с методом рехеширования?
15. Метод логарифмического поиска позволяет значительно сократить время поиска идентификатора в таблице. Однако он же значительно увеличивает время на добавление нового идентификатора в таблицу . Почему тем не менее можно говорить о преимуществах этого метода по сравнению с поиском методом прямого перебора?
16. Проблемы создания хорошей хеш-функции связаны с ограниченной разрядной сеткой ЭВМ и, следовательно, ограниченным размером доступного адресного пространства. Но, как сказано в литературе [14, 62], размер адресного пространства можно увеличить, используя внешние накопители данных (прежде всего

жесткие диски) и механизм виртуальной памяти. Почему эти возможности не используются компиляторами при организации хеш-функций?

Задачи

1. В программе на языке C макрокоманда Inc1(a,b) и функция Inc2(a,b), выполняющие действие a + 2*b, описаны следующим образом:

```
#define Inc1(a,b) ((a)+(b)+(b))
int Inc2(unt a, int b) {return a+b+b;}
```

   К каким результатам приведут следующие вызовы для целочисленных переменных i, j и k, сравните:

```
k=Inc1(i,1); и k=Inc2(i,1);
k= Inc1(i,j); и k= Inc2(i,j);
k= Inc1(i,j+1); и k= Inc2(i,j+1);
k= Inc1(i,j++); и k= Inc2(i,j++);
k= Inc1(i++,Inc2(j,1)); и k= Inc2(i++,Inc1(j,1)).
```

   В каких из перечисленных случаев будет получен логически неверный результат?

2. Изменится ли результат, полученный при выполнении макрокоманды Inc1(a,b) из предыдущей задачи, если ее определение будет дано следующим образом:

```
#define Inc1(a,b) ((a)+(2*(b)))
```

   Приведет ли это к логически правильным резуль татам во всех вышеприведенных случаях?

3. Будет ли верным следующее определение макрокоманды:

```
#define Inc1(a,b) ((a)+(2*b))
```

   Если оно содержит ошибку, укажите, в каком случае выполнение данной макрокоманды даст неверный результат.

Упражнения

1. Напишите программу, реализующую метод логарифмического поиска в упорядоченном массиве строк. В качестве исходных данных для заполнения массива возьмите любой текстовый файл, считая, что все слова в нем являются идентификаторами.
2. Напишите программу, реализующую метод построения бинарного дерева. Используйте динамические структуры данных для организации дерева. В качестве исходных данных для заполнения дерева возьмите любой текстовый файл, считая, что все слова в нем являются идентификаторами. Проверьте эффективность метода на случайных текстовых файлах, а также попробуйте в качестве источника данных файл с упорядоченным текстом.
3. Напишите программу, создающую таблицу идентификаторов с помощью хеш-функций на основе метода простого рехэширования. В качестве исходных данных для заполнения дерева возьмите любой текстовый файл, считая, что все слова в нем являются идентификаторами. Организуйте программу таким образом, чтобы в ней можно было бы легко подменять используемую хеш-функцию.

Подсчитывая число коллизий и среднее количество сравнений для поиска идентификатора, сравните результаты для различных хеш-функций. В качестве исходных данных для хеш-функции можно предложить:

- коды первых двух букв идентификатора;
- коды последних двух букв идентификатора;
- код первой и код последней букв идентификатора;
- коды первой, последней и средней букв идентификатора.

4. Выполните действия, описанные в упражнении № 3, для таблицы идентификаторов, организованной на основе метода цепочек. Сравните результаты.
5. Напишите программу, строящую таблицу идентификаторов по комбинированному способу: при возникновении коллизии новый идентификатор помещается в динамический неупорядоченный список через ссылку в поле основной таблицы идентификаторов. При поиске внутри списка используется простой перебор. Подсчитывая число коллизий и среднее количество сравнений для поиска идентификатора, сравните результаты для хеш-функций, предложенных в упражнении № 3. Попробуйте предложить свой вариант хеш-функции и сравните полученные результаты.

ГЛАВА 3 Лексические анализаторы

Лексические анализаторы (сканеры). Принципы построения сканеров

Назначение лексического анализатора

Прежде чем перейти к рассмотрению лексических анализаторов, необходимо дать четкое определение того, что же такое лексема.

Лексема (лексическая единица языка) — это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своем составе других структурных единиц языка.

Лексемами языков естественного общения являются слова[1]. Лексемами языков программирования являются идентификаторы, константы, ключевые слова языка, знаки операций и разделители. Состав возможных лексем каждого конкретного языка программирования определяется синтаксисом этого языка.

Лексический анализатор (или сканер) — это часть компилятора, которая читает исходную программу и выделяет в ее тексте лексемы входного языка. На вход лексического анализатора поступает текст исходной программы.

Результатом работы лексического анализатора является перечень всех найденных в тексте исходной программы лексем. Этот перечень лексем можно представить в виде таблицы, называемой *таблицей лексем*. Каждой лексеме в таблице лексем соответствует некий уникальный условный код, зависящий от типа лексемы, и дополнительная служебная информация. Кроме того, информация о некоторых типах лексем, найденных в исходной программе, должна помещаться в таблицу идентификаторов (или в одну из таблиц идентификаторов, если компилятор предусматривает различные таблицы идентификаторов для различных типов лексем).

ВНИМАНИЕ

Не следует путать таблицу лексем и таблицу идентификаторов — это две принципиально разные таблицы, обрабатываемые лексическим анализатором.

[1] В языках естественного общения *лексикой* называется словарный запас языка. Лексический состав языка изучается лексикологией и фразеологией, а значение лексем (слов языка) — семасиологией. В языках программирования словарный запас, конечно, не столь интересен и специальной наукой не изучается.

Таблица лексем фактически содержит весь текст исходной программы, обработанный лексическим анализатором. В нее входят все возможные типы лексем, кроме того, любая лексема может встречаться в ней любое количество раз. Таблица идентификаторов содержит только определенные типы лексем — идентификаторы и константы. В нее не попадают такие лексемы, как ключевые (служебные) слова входного языка, знаки операций и разделители. Кроме того, каждая лексема (идентификатор или константа) может встречаться в таблице идентификаторов только один раз. Т акже можно отметить, что лексемы в таблице лексем обязательно располагаются в том же порядке, что и в исходной программе (порядок лексем в ней не меняется), а в таблице идентификаторов лексемы располагаются в любом порядке так, чтобы обеспечить удобство поиска (методы организации таблиц идентификаторов рассмотрены в предыдущей главе).

В качестве примера можно рассмотреть некоторый фрагмент исходного кода на языке Pascal и соответствующую ему таблицу лексем, представленную в табл. 3.1:

```
...
begin
    for I := 1 to N do fg := fg * 0.5
...
```

Таблица 3.1. Лексемы программы

Лексема	Тип лексемы	Значение
begin	Ключевое слово	X1
for	Ключевое слово	X2
I	Идентификатор	I : 1
:=	Знак присваивания	S1
1	Целочисленная константа	1
to	Ключевое слово	X3
N	Идентификатор	N : 2
do	Ключевое слово	X4
fg	Идентификатор	fg : 3
:=	Знак присваивания	S1
fg	Идентификатор	fg : 3
*	Знак арифметической операции	A1
0.5	Вещественная константа	0.5

Поле «Значение» в табл. 3.1 подразумевает некое кодовое значение, которое будет помещено в итоговую таблицу лексем в результате работы лексического анализатора. Конечно, значения, которые записаны в примере, являются условными. Конкретные коды выбираются разработчиками при реализации компилятора. Важно отметить также, что устанавливается связка таблицы лексем с таблицей идентификаторов (в примере это отражено некоторым индексом, следующим после идентификатора за знаком «:», а в реальном компиляторе определяется его реализацией).

Принципы построения лексических анализаторов

С теоретической точки зрения лексический анализатор не является обязательной частью компилятора. Все его функции могут выполняться на этапе синтаксического

разбора, поскольку полностью регламентированы синтаксисом входного языка. Однако существует несколько причин, по которым в состав практически всех компиляторов включают лексический анализ:

- применение лексического анализатора сокращает объем информации, обрабатываемой на этапе синтаксического разбора;
- некоторые задачи, требующие использования сложных вычислительных методов на этапе синтаксического анализа, могут быть решены более простыми методами на этапе лексического анализа (например, задача различения унарного минуса и бинарной операции вычитания, обозначаемых одним и тем же знаком «–»);
- лексический анализатор отделяет сложный по конструкции синтаксический анализатор от работы непосредственно с текстом исходной программы, структура которого может варьироваться в зависимости от архитектуры вычислительной системы, где выполняется компиляция, — при такой конструкции компилятора для перехода на другую вычислительную систему достаточно только перестроить относительно простой лексический анализатор;
- в современных системах программирования лексический анализатор может выполнять обработку текста исходной программы параллельно с его подготовкой пользователем — это дает системе программирования принципиально новые возможности, которые позволяют снизить трудоемкость разработки программ (более подробно об этом сказано в главе «Современные системы программирования»).

Функции, выполняемые лексическим анализатором, и состав типов лексем, которые он выделяет в тексте исходной программы, могут меняться в зависимости от реализации компилятора. То, какие функции должен выполнять лексический анализатор, а какие оставлять для этапа синтаксического разбора, решают разработчики компилятора. В основном лексические анализаторы выполняют исключение из текста исходной программы комментариев и незначащих символов (пробелов, символов табуляции и перевода строки), а также выделение лексем следующих типов: идентификаторов, строковых, символьных и числовых констант, ключевых (служебных) слов входного языка, знаков операций и разделителей.

Лексический анализатор имеет дело с такими объектами, как различного рода константы и идентификаторы (к последним относятся и ключевые слова). Язык констант и идентификаторов является регулярным — то есть может быть описан с помощью регулярных грамматик. Распознавателями для регулярных языков являются конечные автоматы. Следовательно, основой для реализации лексических анализаторов служат регулярные грамматики и конечные автоматы.

Конечный автомат для каждой входной цепочки языка дает ответ на вопрос о том, принадлежит или нет цепочка языку, заданному автоматом. Однако в общем случае задача лексического анализатора несколько шире, чем просто проверка цепочки символов лексемы на соответствие входному языку. Кроме этого, он должен выполнить следующие действия:

- определить границы лексем, которые в тексте исходной программы явно не указаны;
- выполнить действия для сохранения информации об обнаруженной лексеме (или выдать сообщение об ошибке, если лексема неверна).

Эти действия связаны с определенными проблемами. Далее рассмотрено, как эти проблемы решаются в лексических анализаторах.

Проблемы построения лексических анализаторов

Определение границ лексем

Выделение границ лексем является нетривиальной задачей. Ведь в тексте исходной программы лексемы никак не ограничены. Если говорить в терминах лексического анализатора, то определение границ лексем — это выделение тех строк в общем потоке входных символов, для которых надо выполнять распознавание.

Иллюстрацией случая, когда определение границ лексемы вызывает определенные сложности, может служить пример оператора программы на языке FORTRAN : по фрагменту исходного кода `DO 10 I=1` невозможно определить тип оператора языка (а соответственно, и границы лексем). В случае `DO 10 I=1.15` это присвоение вещественной переменной `DO10I` значения константы `1.15` (пробелы в языке FORTRAN игнорируются), а в случае `DO 10 I=1,15` — это цикл с перечислением от `1` до `15` по целочисленной переменной `I` до метки `10`.

Другой иллюстрацией может служить оператор языка C, имеющий вид `k=i+++++j;`. Существует только одна единственно верная трактовка этого оператора: `k = i++ + ++j;` (если явно пояснить ее с помощью скобок, то данная конструкция имеет вид `k = (i++) + (++j);`). Однако найти ее лексический анализатор может, лишь просмотрев весь оператор до конца и перебрав все варианты, причем неверные варианты могут быть обнаружены только на этапе семантического анализа (например, вариант `k = (i++)++ + j;` является синтаксически правильным, но семантикой языка C не допускается). Конечно, чтобы эта конструкция была в принципе допустима, входящие в нее операнды `k`, `i` и `j` должны быть описаны и должны допускать выполнение операций языка `++` и `+`.

Поэтому в большинстве компиляторов лексический и синтаксический анализаторы — это взаимосвязанные части. Возможны два принципиально различных метода взаимосвязи лексического и синтаксического анализа:

- последовательный ;
- параллельный[1].

При последовательном варианте лексический анализатор просматривает весь текст исходной программы от начала до конца и преобразует его в таблицу лексем. Таблица лексем заполняется сразу полностью, компилятор использует ее для последующих фаз компиляции. Дальнейшую обработку таблицы лексем выполняют следующие фазы компиляции. Если лексический анализатор не смог правильно определить тип лексемы, то считается, что исходная программа содержит ошибку.

Работа синтаксического и лексического анализаторов в варианте их последовательного взаимодействия изображена в виде схемы на рис. 3.1.

[1] Параллельный метод работы лексического анализатора и синтаксического разбора вовсе не означает, что они должны будут выполняться как параллельные взаимодействующие процессы. Такой вариант возможен, но необязателен.

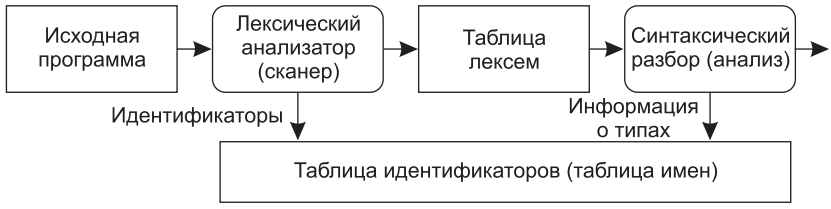

Рис. 3.1. Последовательное взаимодействие лексического и синтаксического анализаторов

При параллельном варианте лексический анализ текста исходной программы выполняется поэтапно по шагам. Лексический анализаторвыделяет очередную лексему в исходном коде и передает ее синтаксическому анализатору. Синтаксический анализатор может подтвердить правильность найденной лексемы и обратиться к лексическому анализатору за следующей лексемой либо же отвергнуть найденную лексему. Во втором случае он может проинформировать лексический анализатор о том, что надо вернуться назад к уже просмотренному ранее фрагменту исходного кода и сообщить ему дополнительную информацию о том, лексему какого типа следует ожидать. Взаимодействуя между собой таким образом, лексический и синтаксический анализаторы могут перебрать несколько возможных вариантов лексем, и, если ни один из них не подойдет, будет считаться, что исходная программа содержит ошибку. Только после того, как синтаксический анализатор успешно выполнит разбор очередной конструкции исходного языка, лексический анализатор помещает найденные лексемы в таблицу лексем и в таблицу идентификаторов и продолжает разбор дальше в том же порядке.

Работа синтаксического и лексического анализаторов в варианте их параллельного взаимодействия изображена в виде схемы на рис. 3.2.

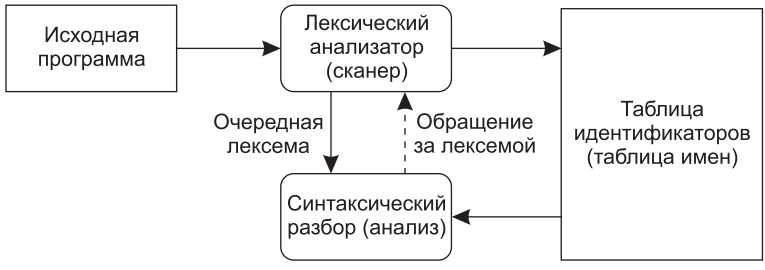

Рис. 3.2. Параллельное взаимодействие лексического и синтаксического анализаторов

Последовательная работа лексического и синтаксического анализаторов, представленная на рис. 3.1, является более простым вариантом их взаимодействия. Она проще в реализации и обеспечивает более высокую скорость работы компилятора, чем их параллельное взаимодействие, показанное на рис. 3.2. Поэтому разработчики компиляторов стремятся организовать взаимодействие лексического и синтаксического анализаторов именно таким образом.

Для большинства языков программирования границы лексем распознаются по заданным терминальным символам. Эти символы — пробелы, знаки операций, символы комментариев, а также разделители (запятые, точки с запятой и т . п.). Набор

таких терминальных символов зависит от синтаксиса входного языка. Важно отметить, что знаки операций и разделители сами также являются лексемами.

Но для многих языков программирования на этапе лексического анализа может быть недостаточно информации для однозначного определения типа и границ очередной лексемы. Однако даже и тогда разработчики компиляторов стремятся избежать параллельной работы лексического и синтаксического анализаторов. В ряде случаев помогает принцип выбора из всех возможных лексем лексемы наибольшей длины: очередной символ из входного потока добавляется в лексему всегда, когда он может быть туда добавлен. Когда символ не может быть добавлен в лексему , то считается, что он является границей лексемы и началом следующей лексемы.

Такой принцип не всегда позволяет правильно определить границы лексем в том случае, когда они не разделены пустыми символами. Например, приведенная выше строка языка C `k = i+++++j;` будет разбита на лексемы следующим образом: `k = i++ ++ + j;` — и это разбиение неверное. Лексический анализатор, разбирая строку из 5 знаков `+` дважды выбрал лексему наибольшей возможной длины — знак операции инкремента (увеличения значения переменной на 1) `++`, хотя это неправильно. Компилятор должен будет выдать пользователю сообщение об ошибке, при том что правильный вариант распознавания этой строки существует.

Разработчики компиляторов сознательно идут на то, что отсекают некоторые правильные, но не вполне читаемые варианты исходных программ. Попытки усложнить лексический распознаватель неизбежно приведут к необходимости организации его параллельной работы с синтаксическим разбором. Это снизит эффективность работы всего компилятора. Возникшие накладные расходы никак не оправдываются достигаемым эффектом — распознаванием строк с сомнительными лексемами. Достаточно обязать пользователя явно указать с помощью пробелов (или других незначащих символов) границы лексем, что значительно проще[1].

Не для всех входных языков такой подход возможен. Например, для рассмотренного выше примера с языка FORTRAN невозможно применить указанный метод — разница между оператором цикла и оператором присваивания слишком существенна. В таком случае приходится организовывать параллельную работу лексического и синтаксического анализаторов[2].

Большинство современных широко распространенных языков программирования, таких как C, C++, Java и Object P ascal, тем не менее позволяют построить лексический анализ по более простому, последовательному, методу.

Выполнение действий, связанных с лексемами

Выполнение действий в процессе распознавания лексем представляет для лексического анализатора гораздо меньшую проблему , чем определение границ лексем. Фактически конечный автомат, который лежит в основе лексического анализатора,

[1] Желающие могут воспользоваться любым доступным компилятором C и проверить, насколько он способен разобрать приведенный здесь пример.

[2] Приведенные здесь два оператора на языке FORTRAN, различающиеся только на один символ — «.» (точка) или «,» (запятая), — представляют собой проблему не столько для компилятора, сколько для программиста, поскольку позволяют допустить очень неприятную и трудно обнаруживаемую ошибку [6].

должен иметь не только входной язык, но и выходной. Он должен не только уметь распознать правильную лексему на входе, но и породить связанную с ней последовательность символов на выходе. В такой конфигурации конечный автомат преобразуется в конечный преобразователь [3, 4 т.1, 5, 29, 34].

Для лексического анализатора действия по обнаружению лексемы могут трактоваться несколько шире, чем только порождение цепочки символов выходного языка. Он должен уметь выполнять такие действия, как запись найденной лексемы в таблицу лексем, поиск и запись лексемы в таблице идентификаторов. Набор выполняемых действий определяется реализацией компилятора. Обычно эти действия выполняются сразу же при обнаружении конца распознаваемой лексемы.

В конечном автомате, лежащем в основе лексического анализатора, эти действия можно отобразить довольно просто — достаточно иметь возможность с каждым переходом на графе автомата (или в функции переходов автомата) связать выполнение некоторой произвольной функции f(q,a), где q — текущее состояние автомата, a — текущий входной символ. Функция f(q,a) может выполнять любые действия, доступные лексическому анализатору:

- помещать новую лексему в таблицу лексем;
- проверять наличие найденной лексемы в таблице идентификаторов;
- добавлять новую лексему в таблицу идентификаторов;
- выдавать сообщения пользователю о найденных ошибках и предупреждения об обнаруженных неточностях в программе;
- прерывать процесс компиляции.

Возможны и другие действия, предусмотренные реализацией компилятора. Такую функцию f(q,a), если она есть, обычно записывают на графе переходов конечного автомата под дугами, соединяющими состояния автомата. Функция f(q,a) может быть пустой (не выполнять никаких действий), тогда соответствующая запись отсутствует.

Регулярные языки и грамматики

Чтобы перейти к примерам реализации лексических анализаторов, необходимо более подробно рассмотреть регулярные языки и грамматики, лежащие в их основе.

Регулярные и автоматные грамматики

Регулярные грамматики

К регулярным относятся два типа грамматик: леволинейные и праволинейные.

Леволинейные грамматики $G(\mathbf{VT},\mathbf{VN},\mathbf{P},S)$, $\mathbf{V} = \mathbf{VN} \cup \mathbf{VT}$ могут иметь правила двух видов: $A \to B\gamma$ или $A \to \gamma$, где $A,B \in \mathbf{VN}$, $\gamma \in \mathbf{VT}^*$.

В свою очередь, праволинейные грамматики $G(\mathbf{VT},\mathbf{VN},\mathbf{P},S)$, $\mathbf{V} = \mathbf{VN} \cup \mathbf{VT}$ могут иметь правила также двух видов: $A \to \gamma B$ или $A \to \gamma$, где $A,B \in \mathbf{VN}$, $\gamma \in \mathbf{VT}^*$.

Доказано, что эти два класса грамматик эквивалентны. Разница между леволинейными и праволинейными грамматиками заключается в основном в том, в каком по-

рядке строятся предложения языка: слева направо для леволинейных либо справа налево для праволинейных. Поскольку предложения языков программирования строятся в порядке слева направо, то в дальнейшем в разделе регулярных грамматик будет идти речь о леволинейных грамматиках.

Автоматные грамматики

Среди всех регулярных грамматик можно выделить отдельный класс — автоматные грамматики. Они также могут быть леволинейными и праволинейными.

Леволинейные автоматные грамматики $G(\mathbf{VT},\mathbf{VN},\mathbf{P},S)$, $\mathbf{V} = \mathbf{VN} \cup \mathbf{VT}$ могут иметь правила двух видов: $A \to Bt$ или $A \to t$, где $A,B \in \mathbf{VN}$, $t \in \mathbf{VT}$.

Праволинейные автоматные грамматики $G(\mathbf{VT},\mathbf{VN},\mathbf{P},S)$, $\mathbf{V} = \mathbf{VN} \cup \mathbf{VT}$ могут иметь правила двух видов: $A \to tB$ или $A \to t$, где $A,B \in \mathbf{VN}$, $t \in \mathbf{VT}$.

Разница между автоматными грамматиками и обычными регулярными грамматиками заключается в следующем: там, где в правилах обычных регулярных грамматик может присутствовать цепочка терминальных символов, в автоматных грамматиках может присутствовать только один терминальный символ. Любая автоматная грамматика является регулярной, но не наоборот — не всякая регулярная грамматика является автоматной.

Доказано, что классы обычных регулярных грамматик и автоматных грамматик почти эквивалентны. Это значит, что для любого языка, который задан регулярной грамматикой, можно построить автоматную грамматику , определяющую почти эквивалентный язык (обратное утверждение очевидно).

Чтобы классы автоматных и регулярных грамматик были полностью эквивалентны, в автоматных грамматиках разрешается дополнительное правило вида $S \to \lambda$, где S — целевой символ грамматики. При этом символ S не должен встречаться в правых частях других правил грамматики. Т огда язык, заданный автоматной грамматикой **G**, может включать в себя пустую цепочку: $\lambda \in L(\mathbf{G})$.

Автоматные грамматики, так же как обычные леволинейные и праволинейные грамматики, задают регулярные языки. Поскольку реально используемые языки, как правило, не содержат пустую цепочку символов, разница на пустую цепочку между этими двумя типами грамматик значения не имеет и правила вида $S \to \lambda$ далее рассматриваться не будут.

Существует алгоритм, который позволяет преобразовать произвольную регулярную грамматику к автоматному виду — то есть построить эквивалентную ей автоматную грамматику. Этот алгоритм рассмотрен ниже. Он является исключительно полезным, поскольку позволяет существенно облегчить построение распознавателей для регулярных грамматик.

Преобразование регулярной грамматики к автоматному виду

Имеется регулярная грамматика $G(\mathbf{VT},\mathbf{VN},\mathbf{P},S)$, необходимо преобразовать ее в почти эквивалентную автоматную грамматику $G'(\mathbf{VT},\mathbf{VN}',\mathbf{P}',S')$. Как уже было сказано выше, будем рассматривать леволинейные грамматики (для праволинейных грамматик можно легко построить аналогичный алгоритм).

Алгоритм преобразования прост, и заключается он в следующей последовательности действий.

Шаг 1. Все нетерминальные символы из множества **VN** грамматики **G** переносятся во множество **VN'** грамматики **G'**.

Шаг 2. Необходимо просматривать все множество правил **P** грамматики **G**.

Если встречаются правила вида $A \to Ba_1$, $A,B \in \mathbf{VN}$, $a_1 \in \mathbf{VT}$, или вида $A \to a_1$, $A \in \mathbf{VN}$, $a_1 \in \mathbf{VT}$, то они переносятся во множество **P'** правил грамматики **G'** без изменений.

Если встречаются правила вида $A \to Ba_1a_2...a_n$, $n > 1$, $A,B \in \mathbf{VN}$, $\forall n > i > 0$: $a_i \in \mathbf{VT}$, то во множество нетерминальных символов **VN'** грамматики **G'** добавляются символы $A_1,A_2,...,A_{n-1}$, а во множество правил **P'** грамматики **G'** добавляются правила:

$A \to A_{n-1}a_n$;
$A_{n-1} \to A_{n-2}a_{n-1}$;
...
$A_2 \to A_1a_2$;
$A_1 \to Ba_1$.

Если встречаются правила вида $A \to a_1a_2...a_n$, $n > 1$, $A \in \mathbf{VN}$, $\forall n > i > 0$: $a_i \in \mathbf{VT}$, то во множество нетерминальных символов **VN'** грамматики **G'** добавляются символы $A_1,A_2,...,A_{n-1}$, а во множество правил **P'** грамматики **G'** добавляются правила:

$A \to A_{n-1}a_n$;
$A_{n-1} \to A_{n-2}a_{n-1}$;
...
$A_2 \to A_1a_2$;
$A_1 \to a_1$.

Если встречаются правила вида $A \to B$ или вида $A \to \lambda$, то они переносятся во множество правил **P'** грамматики **G'** без изменений.

Шаг 3. Просматривается множество правил **P'** грамматики **G'**. В нем ищутся правила вида $A \to B$ или вида $A \to \lambda$.

Если находится правило вида $A \to B$, то просматривается множество правил **P'** грамматики **G'**. Если в нем присутствуют правила вида $B \to C$, $B \to Ca$, $B \to a$ или $B \to \lambda$, то в него добавляются правила вида $A \to C$, $A \to Ca$, $A \to a$ и $A \to \lambda$, соответственно, $\forall A,B,C \in \mathbf{VN}'$, $\forall a \in \mathbf{VT}'$. При этом следует учитывать, что в грамматике не должно быть совпадающих правил, и если какое-то правило уже присутствует в грамматике **G'**, то повторно его туда добавлять не следует. Правило $A \to B$ удаляется из множества правил **P'**.

Если находится правило вида $A \to \lambda$ (и символ A не является целевым символом S), то просматривается множество правил **P'** грамматики **G'**. Если в нем присутствует правило вида $B \to A$ или $B \to Aa$, то в него добавляются правила вида $B \to \lambda$ и $B \to a$, соответственно, $\forall A,B \in \mathbf{VN}'$, $\forall a \in \mathbf{VT}'$ (при этом также следует учитывать, что в грамматике не должно быть совпадающих правил). Правило $A \to \lambda$ удаляется из множества правил **P'**.

Шаг 4. Если на шаге 3 было найдено хотя бы одно правило вида $A \to B$ или $A \to \lambda$ во множестве правил **P'** грамматики **G'**, то надо повторить шаг 3, иначе перейти к шагу 5.

Шаг 5. Целевым символом S' грамматики **G'** становится символ S.

Шаги 3 и 4 алгоритма в принципе можно не выполнять, если грамматика не содержит правил вида A → B (такие правила называются *цепными*) или вида A → λ (такие правила называются λ-*правилами*). Реальные регулярные грамматики обычно не содержат правил такого вида. Кроме того, эти правила можно было бы предварительно устранить с помощью специальных алгоритмов преобразования грамматик (эти алгоритмы рассмотрены в главе «Синтаксические анализаторы»).

Пример преобразования регулярной грамматики к автоматному виду

Рассмотрим в качестве примера следующую простейшую регулярную грамматику : `G({"a","(","*",")","{","}"},{S,C,K},P,S)` (символы `a, (, *, ), {, }` из множества терминальных символов грамматики взяты в кавычки, чтобы выделить их среди фигурных скобок, обозначающих само множество):

P:

```
S → C*) | K}
C → (* | Ca | C{ | C} | C( | C* | C)
K → { | Ka | K( | K* | K) | K{
```

Если предположить, что `a` здесь — это любой алфавитно-цифровой символ, кроме символов `(, *, ), {, }`, то эта грамматика описывает два типа комментариев, допустимых в языке программирования Borland Pascal. Преобразуем ее в автоматный вид.

Шаг 1. Построим множество **VN'** = {S, C, K}.

Шаг 2. Начинаем просматривать множество правил **P** грамматики **G**.

Для правила `S → C*)` во множество **VN'** включаем символ S_1, а само правило разбиваем на два: `S →` S_1`)` и S_1 `→ C*`; включаем эти правила во множество правил **P'**.

Правило `S → K}` переносим во множество правил **P'** без изменений.

Для правила `C → (*` во множество **VN'** включаем символ C_1, а само правило разбиваем на два: `C →` C_1`*` и C_1 `→ (`; включаем эти два правила во множество правил **P'**.

Правила `C → Ca | C{ | C} | C( | C* | C)` переносим во множество правил **P'** без изменений.

Правила `K → { | Ka | K( | K* | K) | K{` переносим во множество правил **P'** без изменений.

Шаг 3. Правил вида A→B или A→λ во множестве правил **P'** не содержится.

Шаг 4. Переходим к шагу 5.

Шаг 5. Целевым символом грамматики **G'** становится символ `S`.

В итоге получаем автоматную грамматику:

G'({"a","(","*",")","{","}"}, {S,S_1,C,C_1,K},**P'**,S):

P':

```
S → S1) | K}
S1 → C*
C → C1* | Ca | C{ | C} | C( | C* | C)
C1 → (
K → { | Ka | K( | K* | K) | K{
```

Эта грамматика, так же как и рассмотренная выше, описывает два типа комментариев, допустимых в языке программирования Borland Pascal.

Конечные автоматы

Определение конечного автомата

Конечным автоматом (КА) называют пятерку следующего вида:

$M(\mathbf{Q}, \mathbf{V}, \delta, q_0, \mathbf{F})$,

где

$\mathbf{Q}$ — конечное множество состояний автомата;

$\mathbf{V}$ — конечное множество допустимых входных символов (алфавит автомата);

δ — функция переходов, отображающая $\mathbf{V}^*\mathbf{Q}$ (декартово произведение множеств) во множество подмножеств $\mathbf{Q}$: $R(\mathbf{Q})$, то есть $\delta(a,q)=\mathbf{R}$, $a\in\mathbf{V}$, $q\in\mathbf{Q}$, $\mathbf{R}\subseteq\mathbf{Q}$;

q_0 — начальное состояние автомата $\mathbf{Q}$, $q_0\in\mathbf{Q}$;

$\mathbf{F}$ — непустое множество конечных состояний автомата, $\mathbf{F}\subseteq\mathbf{Q}$, $\mathbf{F}\neq\varnothing$.

КА называют *полностью определенным*, если в каждом его состоянии существует функция перехода для всех возможных входных символов, то есть $\forall a\in\mathbf{V}$, $\forall q\in\mathbf{Q}$ $\exists\delta(a,q)=\mathbf{R}$, $\mathbf{R}\subseteq\mathbf{Q}$.

Работа конечного автомата представляет собой последовательность шагов (или тактов). На каждом шаге работы автомат находится в одном из своих состояний $q\in\mathbf{Q}$ (в текущем состоянии), на следующем шаге он может перейти в другое состояние или остаться в текущем состоянии. То, в какое состояние автомат перейдет на следующем шаге работы, определяет функция переходов δ. Она зависит не только от текущего состояния q, но и от символа a из алфавита $\mathbf{V}$, поданного на вход автомата. Когда функция перехода допускает несколько следующих состояний автомата, то КА может перейти в любое из этих состояний. В начале работы автомат всегда находится в начальном состоянии q_0. Работа КА продолжается до тех пор, пока на его вход поступают символы из входной цепочки $\omega\in\mathbf{V}^+$.

Видно, что конфигурацию КА на каждом шаге работы можно определить в виде (q,ω,n), где q — текущее состояние автомата, $q\in\mathbf{Q}$; ω — цепочка входных символов, $\omega\in\mathbf{V}^+$; n — положение указателя в цепочке символов, $n\in\mathbf{N}\cup\{0\}$, $n\le|\omega|$ ($\mathbf{N}$ — множество натуральных чисел). Конфигурация автомата на следующем шаге — это $(q',\omega,n+1)$, если $q'\in\delta(a,q)$ и символ $a\in\mathbf{V}$ находится в позиции n+1 цепочки ω. Начальная конфигурация автомата: $(q_0,\omega,0)$; заключительная конфигурация автомата: (f,ω,n), $f\in\mathbf{Q}$, $n=|\omega|$, она является конечной конфигурацией, если $f\in\mathbf{F}$.

Язык, заданный конечным автоматом

КА $M(\mathbf{Q},\mathbf{V},\delta,q_0,\mathbf{F})$ *принимает цепочку символов* $\omega\in\mathbf{V}^+$, если, получив на вход эту цепочку, он из начального состояния q_0 может перейти в одно из конечных состояний $f\in\mathbf{F}$. В противном случае КА не принимает цепочку символов.

Язык $L(\mathbf{M})$, заданный КА $M(\mathbf{Q},\mathbf{V},\delta,q_0,\mathbf{F})$, — это множество всех цепочек символов, которые принимаются этим автоматом.

Два КА M и **M'** *эквивалентны*, если они задают один и тот же язык:
M≡**M'** ⇔ L(**M**)=L(**M'**).

Все КА являются распознавателями для регулярных языков [4 т.1, 5, 15, 29].

Граф переходов конечного автомата

КА часто представляют в виде диаграммы или графа переходов автомата.

Граф переходов КА — это направленный граф, вершины которого помечены символами состояний КА и в котором есть дуга (p,q) p,q∈**Q**, помеченная символом a∈**V**, если в КА определена δ(a,p) и q∈δ(a,p). Начальное и конечные состояния автомата на графе состояний помечаются специальным образом (в данном пособии начальное состояние — дополнительной пунктирной линией, конечное состояние — дополнительной сплошной линией).

Рассмотрим конечный автомат: `M({H,A,B,S},{a,b},δ,H,{S});` δ: `δ(H,b)=B, δ(B,a)=A, δ(A,b)={B,S}`. Ниже на рис. 3.3 приведен пример графа состояний для этого КА.

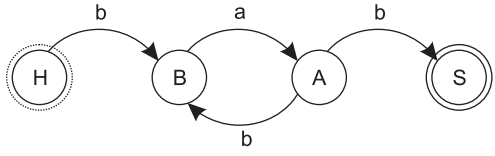

Рис. 3.3. Граф переходов недетерминированного конечного автомата

Для моделирования работы КА его удобно привести к полностью определенному виду, чтобы исключить ситуации, из которых нет переходов по входным символам. Для этого в КА добавляют еще одно состояние, которое можно условно назвать «ошибка». На это состояние замыкают все неопределенные переходы, а все переходы из самого состояния «ошибка» замыкают на него же.

Если преобразовать подобным образом рассмотренный выше автомат**M**, то получим полностью определенный автомат:`M({H,A,B,E,S},{a,b},δ,H,{S});` δ: `δ(H,a)=E, δ(H,b)=B, δ(B,a)=A, δ(B,b)=E, δ(A,a)={E}, δ(A,b)={B,S}, δ(E,a)={E}, δ(E,b)={E}, δ(S,a)={E}, δ(S,b)={E}`. Состояние E как раз соответствует состоянию «ошибка». Граф переходов этого КА представлен на рис. 3.4.

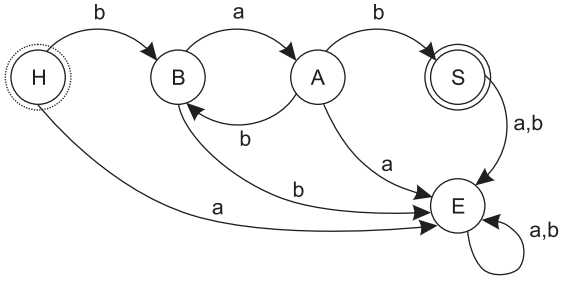

Рис. 3.4. Граф переходов полностью определенного недетерминированного конечного автомата

Детерминированные и недетерминированные конечные автоматы

Определение детерминированного конечного автомата

Конечный автомат M(**Q**,**V**,δ,q_0,**F**) называют *детерминированным конечным автоматом* (ДКА), если в каждом из его состояний для любого входного символа функция перехода содержит не более одного состояния: $\forall a \in \mathbf{V}$, $\forall q \in \mathbf{Q}$: либо $\delta(a,q)=\{r\}$, $r \in \mathbf{Q}$, либо $\delta(a,q)=\varnothing$.

В противном случае конечный автомат называют *недетерминированным*.

Из этого определения видно, что автоматы, представленные ранее на рис. 3.3 и 3.4, являются недетерминированными КА.

ДКА может быть задан в виде пятерки:

M(**Q**, **V**, δ, q_0, **F**),

где

Q — конечное множество состояний автомата;

V — конечное множество допустимых входных символов;

δ — функция переходов, отображающая **V*****Q** в множество **Q**: $\delta(a,q)=r$, $a \in \mathbf{V}$, $q,r \in \mathbf{Q}$;

q_0 — начальное состояние автомата **Q**, $q_0 \in \mathbf{Q}$;

F — непустое множество конечных состояний автомата, $\mathbf{F} \subseteq \mathbf{Q}$, $\mathbf{F} \neq \varnothing$.

Если функция переходов ДКА определена для каждого допустимого входного символа и для каждого состояния автомата, то автомат называется *полностью определенным* ДКА :$\forall a \in \mathbf{V}$, $\forall q \in \mathbf{Q}$: $\exists \delta(a,q)=r$, $r \in \mathbf{Q}$.

Доказано, что для любого КА можно построить эквивалентный ему ДКА. Моделировать работу ДКА существенно проще, чем работу произвольного КА, поэтому произвольный КА стремятся преобразовать в ДКА. При построении компиляторов чаще всего используют полностью определенный ДКА.

Преобразование конечного автомата к детерминированному виду

Алгоритм преобразования произвольного КА M(**Q**,**V**,δ,q_0,**F**) в эквивалентный ему ДКА M'(**Q**',**V**,δ',q'_0,**F**') заключается в следующем.

1. Множество состояний **Q**' автомата **M**' строится из комбинаций всех состояний множества **Q** автомата **M**. Если $q_1, q_2, ..., q_n$, $n>0$ — состояния автомата **M**, $\forall 0<i\leqslant n$ $q_i \in \mathbf{Q}$, то всего будет 2^n-1 состояний автомата **M**'. Обозначим их так: $[q_1 q_2 ... q_m]$, $0<m\leq n$.
2. Функция переходов δ' автомата **M**' строится так: $\delta'(a,[q_1 q_2 ... q_m])=[r_1 r_2 ... r_k]$, где $\forall\ 0< i\leq m\ \exists\ 0<j\leq k$ так, что $r_j \in \delta(a,q_i)$.
3. Обозначим $q'_0 = [q_0]$.
4. Пусть $f_1, f_2, ..., f_l$, $l>0$ — конечные состояния автомата **M**, $\forall 0<i\leqslant l$ $f_i \in \mathbf{F}$, тогда множество конечных состояний **F**' автомата **M**' строится из всех состояний, имеющих вид $[...f_i...]$, $f_i \in \mathbf{F}$.

Доказано, что описанный выше алгоритм строит ДКА, эквивалентный заданному произвольному КА [4 т.1, 29].

После построения из нового ДКА необходимо удалить все недостижимые состояния.

Состояние $q \in \mathbf{Q}$ в КА $M(\mathbf{Q},\mathbf{V},\delta,q_0,\mathbf{F})$ называется недостижимым, если ни при какой входной цепочке $\omega \in \mathbf{V}^+$ невозможен переход автомата из начального состояния q_0 в состояние q. Иначе состояние называется достижимым.

Для работы алгоритма удаления недостижимых состояний используются два множества: множество достижимых состояний $\mathbf{R}$ и множество текущих активных состояний $\mathbf{P}_i$. Результатом работы алгоритма является полное множество достижимых состояний $\mathbf{R}$. Рассмотрим работу алгоритма по шагам.

Шаг 1. $\mathbf{R}:=\{q_0\}$; i:=0; $\mathbf{P_0}:=\{q_0\}$.

Шаг 2. $\mathbf{P}_{i+1}:=\varnothing$.

Шаг 3. $\forall a \in \mathbf{V}$, $\forall q \in \mathbf{P}_i$: $\mathbf{P}_{i+1}:=\mathbf{P}_{i+1} \cup \delta(a,q)$.

Шаг 4. Если $\mathbf{P}_{i+1} - \mathbf{R} = \varnothing$, то выполнение алгоритма закончено, иначе $\mathbf{R}:=\mathbf{R} \cup \mathbf{P}_{i+1}$, i:=i+1 и перейти к шагу 3.

После выполнения данного алгоритма из КА можно исключить все состояния, не входящие в построенное множество $\mathbf{R}$.

Рассмотрим работу алгоритма преобразования произвольного КА в ДКА на примере автомата `M({H,A,B,S},{a,b},δ,H,{S})`; `δ: δ(H,b)=B, δ(B,a)=A, δ(A,b)={B,S}`. Видно, что это недетерминированный КА (из состояния `A` возможны два различных перехода по символу `b`). Граф переходов для этого автомата был изображен выше на рис. 3.3.

Построим множество состояний эквивалентного ДКА:

```
Q'={[H],[A],[B],[S],[HA],[HB],[HS],[AB],[AS],[BS],[HAB],[HAS],[HBS],[ABS],[HABS]}.
```

Построим функцию переходов эквивалентного ДКА:

```
δ'([H],b)=[B]      δ'([A],b)=[BS]     δ'([B],a)=[A]
δ'([HA],b)=[BS]    δ'([HB],a)=[A]     δ'([HB],b)=[B]
δ'([HS],b)=[B]     δ'([AB],a)=[A]     δ'([AB],b)=[BS]
δ'([AS],b)=[BS]    δ'([BS],a)=[A]     δ'([HAB],a)=[A]
δ'([HAB],b)=[BS]   δ'([HAS],b)=[BS]   δ'([HBS],b)=[B]
δ'([HBS],a)=[A]    δ'([ABS],b)=[BS]   δ'([ABS],a)=[A]
δ'([HABS],a)=[A]   δ'([HABS],b)=[BS].
```

Начальное состояние эквивалентного ДКА

```
q0' = [H].
```

Множество конечных состояний эквивалентного ДКА

```
F' = {[S],[HS],[AS],[BS],[HAS],[HBS],[ABS],[HABS]}.
```

После построения ДКА исключим недостижимые состояния. Множество достижимых состояний ДКА будет следующим: `R={[H],[B],[A],[BS]}`. В итоге, исключив все недостижимые состояния, получим ДКА:

```
M'({[H],[B],[A],[BS]},{a,b},[H],{[BS]}),
δ([H],b)=[B], δ([B],a)=[A], δ([A],b)=[BS], δ([BS],a)=[A].
```

Ничего не изменяя, переобозначим состояния ДКА. Получим

M'({H,B,A,S},{a,b},H,{S}),

δ(H,b)=B, δ(B,a)=A, δ(A,b)=S, δ(S,a)=A.

Граф переходов полученного ДКА изображен на рис. 3.5.

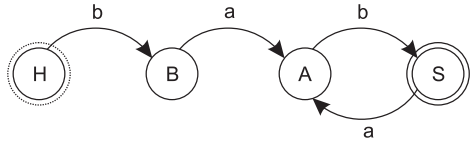

Рис. 3.5. Граф переходов детерминированного конечного автомата

Этот автомат можно преобразовать к полностью определенному виду Получим граф состояний, изображенный на рис. 3.6 (состояние E — это состояние «ошибка»).

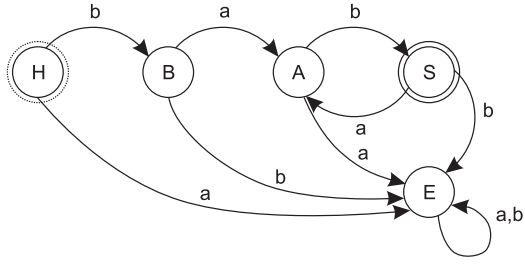

Рис. 3.6. Граф переходов полностью определенного детерминированного конечного автомата

ВНИМАНИЕ

При построении распознавателей к вопросу о необходимости преобразования КА в ДКА надо подходить, основываясь на принципе разумной достаточности. Моделировать работу ДКА существенно проще, чем произвольного КА, но при выполнении преобразования число состояний автомата может возрасти и в худшем случае составит 2^n-1, где n — количество состояний исходного КА. Поэтому не всегда выполнение преобразования КА в ДКА является обязательным.

Минимизация конечных автоматов

Многие КА можно минимизировать. *Минимизация* КА заключается в построении эквивалентного КА с меньшим числом состояний. В процессе минимизации необходимо построить автомат с минимально возможным числом состояний, эквивалентный данному КА.

Для минимизации автомата используется алгоритм построения эквивалентных состояний КА. Два различных состояния в конечном автомате $M(\mathbf{Q},\mathbf{V},\delta,q_0,\mathbf{F})$ $q\in\mathbf{Q}$ и $q'\in\mathbf{Q}$ называются n-эквивалентными (n-неразличимыми), $n\geq 0$, $n\in\mathbf{N}\cup\{0\}$, если, находясь в одном из этих состояний и получив на вход любую цепочку символов ω, $\omega\in\mathbf{V}^*$ $|\omega|\leq n$, автомат может перейти в одно и то же множество конечных состояний. Очевидно, что 0-эквивалентными состояниями автомата $M(\mathbf{Q},\mathbf{V},\delta,q_0,\mathbf{F})$ являются два множества его состояний: **F** и **Q**–**F**. Множества эквивалентных состояний автомата

называют классами эквивалентности, а всю их совокупность — множеством классов эквивалентности $\mathbf{R}(n)$, причем $R(0) = \{\mathbf{F}, \mathbf{Q}–\mathbf{F}\}$.

Рассмотрим работу алгоритма построения эквивалентных состояний по шагам.

Шаг 1. n:=0, строим R(0).

Шаг 2. n:=n+1, строим R(n) на основе R(n−1): $R(n)=\{r_i(n):\{q_{ij}\in \mathbf{Q}: \forall a\in \mathbf{V}\,\delta(a,q_{ij})\subseteq r_j(n-1)\}\ \forall i,j\in \mathbf{N}\}$. То есть в классы эквивалентности на шаге n входят те состояния, которые по одинаковым символам переходят в n–1 эквивалентные состояния.

Шаг 3. Если R(n)=R(n−1), то работа алгоритма закончена, иначе необходимо вернуться к шагу 2.

Доказано, что алгоритм построения множества классов эквивалентности завершится максимум для n=m−2, где m — общее количество состояний автомата.

Алгоритм минимизации КА заключается в следующем.

1. Из автомата исключаются все недостижимые состояния.
2. Строятся классы эквивалентности автомата.
3. Классы эквивалентности состояний исходного КА становятся состояниями результирующего минимизированного КА.
4. Функция переходов результирующего КА очевидным образом строится на основе функции переходов исходного КА.

Для этого алгоритма доказано: во-первых, что он строит минимизированный КА, эквивалентный заданному; во-вторых, что он строит КА с минимально возможным числом состояний (минимальный КА).

Рассмотрим пример: задан автомат `M({A,B,C,D,E,F,G},{0,1},δ,A,{D,E})`, `δ: δ(A,0)={B}, δ(A,1)={C}, δ(B,1)={D}, δ(C,1)={E}, δ(D,0)={C}, δ(D,1)={E}, δ(E,0)={B}, δ(E,1)={D}, δ(F,0)={D}, δ(F,1)={G}, δ(G,0)={F}, δ(G,1)={F}`; необходимо построить эквивалентный ему минимальный КА.

Граф переходов этого автомата приведен на рис. 3.7.

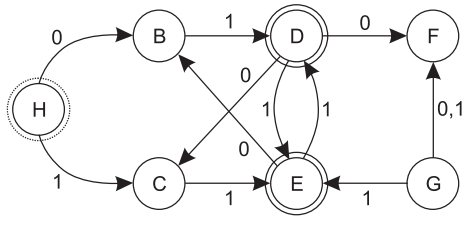

Рис. 3.7. Граф переходов конечного автомата до его минимизации

Состояния `F` и `G` являются недостижимыми, они будут исключены на первом шаге алгоритма. Построим классы эквивалентности автомата:

`R(0)={{A,B,C},{D,E}}, n=0;`

`R(1)={{A},{B,C},{D,E}}, n=1;`

`R(2)={{A},{B,C},{D,E}}, n=2.`

Обозначим соответствующим образом состояния полученного минимального КА и построим автомат: M({A,BC,DE},{0,1},δ',A,{DE}), δ': δ'(A,0)={BC}, δ'(A,1)={BC}, δ'(BC,1)={DE}, δ'(DE,0)={BC}, δ'(DE,1)={DE}.

Граф переходов минимального КА приведен на рис. 3.8.

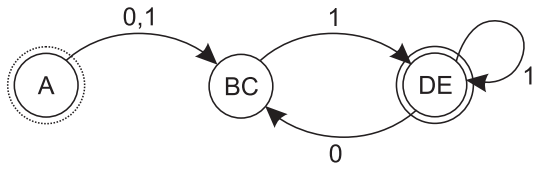

Рис. 3.8. Граф переходов конечного автомата после его минимизации

Минимизация конечных автоматов позволяет при построении распознавателей получить автомат с минимально возможным числом состояний и тем самым в дальнейшем упростить функционирование распознавателя.

Регулярные множества и регулярные выражения

Определение регулярного множества

Определим над множествами цепочек символов из алфавита **V** операции конкатенации и итерации следующим образом:

PQ — конкатенация **P**∈ **V*** и **Q**∈ **V***: **PQ** = {pq ∀p∈ **P**, ∀q∈ **Q**};

P* — итерация **P**∈ **V***: **P*** = {p^n ∀p∈ **P**, ∀n⩾0}.

Тогда для алфавита **V** регулярные множества определяются рекурсивно:

1) ∅ — регулярное множество;
2) {λ} — регулярное множество;
3) {a} — регулярное множество ∀a∈ **V**;
4) если **P** и **Q** — произвольные регулярные множества, то множества **P**∪**Q**, **PQ** и **P*** также являются регулярными множествами;
5) ничто другое не является регулярным множеством.

Фактически регулярные множества — это множества цепочек символов над заданным алфавитом, построенные определенным образом (с использованием операций объединения, конкатенации и итерации).

Все регулярные языки представляют собой регулярные множества [4 т.1, 29, 58].

Регулярные выражения. Свойства регулярных выражений

Регулярные множества можно обозначать с помощью регулярных выражений.

Эти обозначения вводятся следующим образом:

1) 0 — регулярное выражение, обозначающее ∅;
2) λ — регулярное выражение, обозначающее {λ};

3) a — регулярное выражение, обозначающее {a} $\forall a \in \mathbf{V}$;
4) если p и q — регулярные выражения, обозначающие регулярные множества **P** и **Q**, то p+q, pq, p^* — регулярные выражения, обозначающие регулярные множества $\mathbf{P} \cup \mathbf{Q}$, **PQ** и $\mathbf{P}^*$ соответственно.

ПРИМЕЧАНИЕ

Иногда для удобства обозначений вводят также операцию непустой итерации, которая обозначается p^+ и для любого регулярного выражения p справедливо: $p^+=pp^*=p^*p$.

Два регулярных выражения, α и β, равны, $\alpha=\beta$, если они обозначают одно и то же множество. Каждое регулярное выражение обозначает одно и только одно регулярное множество, но для одного регулярного множества может существовать сколь угодно много регулярных выражений, обозначающих это множество.

При записи регулярных выражений будут использоваться круглые скобки, как и для обычных арифметических выражений. При отсутствии скобок операции выполняются слева направо с учетом приоритета. Приоритет для операций принят следующий: первой выполняется итерация (высший приоритет), затем конкатенация, потом — объединение множеств (низший приоритет).

Если α, β и γ — регулярные выражения, то свойства регулярных выражений можно записать в виде следующих формул.

1. $\lambda + \alpha\alpha^* = \lambda + \alpha^*\alpha = \alpha^*$ (или, что то же самое, $\lambda + \alpha^+ = \alpha^*$).
2. $\alpha + \beta = \beta + \alpha$.
3. $\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$.
4. $(\alpha\ \beta + \gamma) = \alpha\beta + \alpha\gamma$.
5. $(\beta + \gamma)\alpha = \beta\alpha + \gamma\alpha$.
6. $(\alpha\ \beta\gamma) = (\alpha\beta)\gamma$.
7. $\alpha + \alpha = \alpha$.
8. $\alpha + \alpha^* = \alpha^*$.
9. $\lambda + \alpha^* = \alpha^* + \lambda = \alpha^*$.
10. $0^* = \lambda$.
11. $0\alpha = \alpha 0 = 0$.
12. $0 + \alpha = \alpha + 0 = \alpha$.
13. $\lambda\alpha = \alpha\lambda = \alpha$.
14. $(\alpha^*)^* = \alpha^*$.

Все эти свойства можно легко доказать, основываясь на теории множеств, так как регулярные выражения — это только обозначения для соответствующих множеств.

Следует также обратить внимание на то, что среди прочих свойств отсутствует равенство $\alpha\beta = \beta\alpha$, то есть операция конкатенации не обладает свойством коммутативности. Это и неудивительно, поскольку для этой операции важен порядок следования символов.

Уравнения с регулярными коэффициентами

На основе регулярных выражений можно построить уравнения с регулярными коэффициентами [5, 15, 24]. Простейшие уравнения с регулярными коэффициентами будут выглядеть следующим образом:

$$X = \alpha X + \beta;$$
$$X = X\alpha + \beta,$$

где $\alpha,\beta \in \mathbf{V}^*$ — регулярные выражения над алфавитом **V**, а переменная $X \notin \mathbf{V}$.

Решениями таких уравнений будут регулярные множества . Это значит , что если взять регулярное множество, являющееся решением уравнения, обозначить его в виде соответствующего регулярного выражения и подставить в уравнение, то получим тождественное равенство. Два вида записи уравнений связаны с тем, что для регулярных выражений операция конкатенации не обладает свойством коммутативности, поэтому коэффициент можно записать как справа, так и слева от переменной и при этом получатся различные уравнения.

Решением первого уравнения является множество, обозначенное регулярным выражением $\alpha^*\beta$. Проверим это решение, подставив его в уравнение вместо переменной X:

$$\alpha X + \beta = \alpha(\alpha^*\beta) + \beta =^{6} (\alpha\alpha^*)\beta + \beta =^{13} (\alpha\alpha^*)\beta + \lambda\beta =^{5} (\alpha\alpha^* + \lambda)\beta =^{1} \alpha^*\beta = X.$$

Над знаками равенства указаны номера свойств регулярных выражений, которые были использованы для выполнения преобразований.

Решением второго уравнения является множество, обозначенное регулярным выражением $\beta\alpha^*$:

$$X\alpha + \beta = (\beta\alpha^*)\alpha + \beta =^{6} \beta(\alpha^*\alpha) + \beta =^{13} \beta(\alpha^*\alpha) + \beta\lambda =^{4} \beta(\alpha^*\alpha + \lambda) =^{1} \beta\alpha^* = X.$$

ПРИМЕЧАНИЕ

Указанные решения уравнений не всегда являются единственными. Однако доказано, что $X = \alpha^*\beta$ и $X = \beta\alpha^*$ — это решения, наименьшие из возможных решений для данных двух уравнений [4 т.1, 15].

Из уравнений с регулярными коэффициентами можно формировать систему уравнений с регулярными коэффициентами. Система уравнений с регулярными коэффициентами имеет вид (правосторонняя запись)

$$X_1 = \alpha_{10} + \alpha_{11}X_1 + \alpha_{12}X_2 + ... + \alpha_{1n}X_n;$$
$$X_2 = \alpha_{20} + \alpha_{21}X_1 + \alpha_{22}X_2 + ... + \alpha_{2n}X_n;$$
$$...$$
$$X_i = \alpha_{i0} + \alpha_{i1}X_1 + \alpha_{i2}X_2 + ... + \alpha_{in}X_n;$$
$$...$$
$$X_n = \alpha_{n0} + \alpha_{n1}X_1 + \alpha_{12}X_2 + ... + \alpha_{nn}X_n;$$

или (левосторонняя запись):

$$X_1 = \alpha_{10} + X_1\alpha_{11} + X_2\alpha_{12} + ... + X_n\alpha_{1n};$$
$$X_2 = \alpha_{20} + X_1\alpha_{21} + X_2\alpha_{22} + ... + X_n\alpha_{2n};$$
$$...$$

$$X_i = \alpha_{i0} + X_1\alpha_{i1} + X_2\alpha_{i2} + ... + X_n\alpha_{in};$$

...

$$X_n = \alpha_{n0} + X_1\alpha_{n1} + X_2\alpha_{12} + ... + X_n\alpha_{nn};$$

В системе уравнений с регулярными коэффициентами все коэффициенты α_{ij} являются регулярными выражениями над алфавитом **V**, а переменные не входят в алфавит **V**: $\forall i\ X_i \notin \mathbf{V}$. Оба варианта записи равноправны, но в общем случае могут иметь различные решения при одинаковых коэффициентах при переменных. Чтобы решить систему уравнений с регулярными коэффициентами, надо найти такие регулярные множества $\mathbf{X_i}$, при подстановке которых в систему все уравнения превращаются в тождества. Иными словами, решением системы является некоторое отображение $f(\mathbf{X})$ множества переменных уравнения $\Delta = \{\mathbf{X_i}: n \geqslant i > 0\}$ на множество языков над алфавитом $\mathbf{V}^*$.

Системы уравнений с регулярными коэффициентами решаются методом последовательных подстановок. Рассмотрим метод решения для правосторонней записи. Алгоритм решения работает с переменной номера шага i и состоит из следующих шагов.

Шаг 1. Положить i:=1.

Шаг 2. Если i=n, то перейти к шагу 4, иначе записать i-е уравнение в виде $X_i=\alpha_iX_i+\beta_i$, где $\alpha_i=\alpha_{ii}$, $\beta_i=\beta_{i0}+\beta_{ii+1}X_{i+1}+...+\beta_{in}X_n$. Решить уравнение и получить $X_i=\alpha_i^*\beta_i$. Затем для всех уравнений с переменными X_{i+1}, ..., X_n подставить в них найденное решение вместо X_i.

Шаг 3. Увеличить i на 1 (i:=i+1) и вернуться к шагу 2.

Шаг 4. После всех подстановок уравнение для X_n будет иметь вид $X_n=\alpha_nX_n+\beta$, где $\alpha_n=\alpha_{nn}$. Причем β будет регулярным выражением над алфавитом $\mathbf{V}^*$ (не содержит в своем составе переменных системы уравнений X_i). Тогда можно найти окончательное решение для X_n: $X_n=\alpha_n^*\beta$. Перейти к шагу 5.

Шаг 5. Уменьшить i на 1 (i: =i—1). Если i =0, то алгоритм завершен, иначе перейти к шагу 6.

Шаг 6. Берем найденное решение для $X_i=\alpha_iX_i+\beta_i$, где $\alpha_i=\alpha_{ii}$, $\beta_i=\beta_{i0}+\beta_{ii+1}X_{i+1}+...+\beta_{in}X_n$ и подставляем в него окончательные решения для переменных X_{i+1}, ..., X_n. Получаем окончательное решение для X_i. Перейти к шагу 5.

Для левосторонней записи системы уравнений алгоритм решения будет аналогичным, с разницей только в порядке записи коэффициентов.

Система уравнений с регулярными коэффициентами всегда имеет решение, но это решение не всегда единственное [4 т.1, 15].

В качестве примера рассмотрим систему уравнений с регулярными коэффициентами над алфавитом `V={"—","+",".",0,1}` (для ясности записи символы «—» и «+», а также . взяты в кавычки, чтобы не путать их со знаками операций):

$X_1 = (\text{«—»} + \text{«+»} + \lambda);$

$X_2 = X_1\text{».»}(0 + 1) + X_3\text{».»} + X_2(0 + 1);$

$X_3 = X_1(0 + 1) + X_3(0 + 1);$

$X_4 = X_2 + X_3.$

Решим эту систему уравнений.

Шаг 1.

i := 1.

Шаг 2.

Имеем i=1 < 4.

Берем уравнение для i=1. Имеем $X_1 = («-» + «+» + \lambda)$. Это уже и есть решение для X_1.

Подставляем его в другие уравнения. Получаем

$X_2 = («-» + «+» + \lambda)«.»(0 + 1) + X_3«.» + X_2(0 + 1);$

$X_3 = («-» + «+» + \lambda)(0 + 1) + X_3(0 + 1);$

$X_4 = X_2 + X_3.$

Шаг 3.

i:=i+1 = 2.

Возвращаемся к шагу 2.

Шаг 2.

Имеем i=2 < 4.

Берем уравнение для i=2. Имеем $X_2 = («-» + «+» + \lambda)«.»(0 + 1) + X_3«.» + X_2(0 + 1)$. Преобразуем уравнение к виду $X_2 = X_2(0 + 1) + ((«-» + «+» + \lambda)«.»(0 + 1) + X_3«.»)$. Тогда $\alpha_2 = (0 + 1)$, $\beta_2 = («-» + «+» + \lambda)«.»(0 + 1) + X_3«.»$, а решением для X_2 будет $X_2 = \beta_2\alpha_2^* = ((«-» + «+» + \lambda)«.»(0 + 1) + X_3«.»)(0 + 1)^* = («-» + «+» + \lambda)«.»(0 + 1)(0 + 1)^* + X_3«.»(0 + 1)^*$.

Подставим его в другие уравнения. Получаем

$X_3 = («-» + «+» + \lambda)(0 + 1) + X_3(0 + 1);$

$X_4 = («-» + «+» + \lambda)«.»(0 + 1)(0 + 1)^* + X_3«.»(0 + 1)^* + X_3.$

Шаг 3.

i:=i+1 = 3.

Возвращаемся к шагу 2.

Шаг 2.

Имеем i=3 < 4.

Берем уравнение для i=3. Имеем $X_3 = («-» + «+» + \lambda)(0 + 1) + X_3(0 + 1)$. Преобразуем уравнение к виду $X_3 = X_3(0 + 1) + («-» + «+» + \lambda)(0 + 1)$.

Тогда $\alpha_3 = (0 + 1)$, $\beta_3 = («-» + «+» + \lambda)(0 + 1)$. Решением для X_3 будет

$X_3 = \beta_3\alpha_3^* = («-» + «+» + \lambda)(0 + 1)(0 + 1)^*.$

Подставим его в другие уравнения. Получаем

$X_4 = («-» + «+» + \lambda)«.»(0 + 1)(0 + 1)^* + («-» + «+» + \lambda)(0 + 1)(0 + 1)^*«.»(0 + 1)^* + + («-» + «+» + \lambda)(0 + 1)(0 + 1)^*.$

Шаг 3.

i:=i+1 = 4.

Возвращаемся к шагу 2.

Шаг 2.

Имеем i=4 = 4. Переходим к шагу 4.

Шаг 4.

Уравнение для X_4 теперь имеет вид

$X_4 = («—» + «+» + \lambda)«.»(0 + 1)(0 + 1)^* + («—» + «+» + \lambda)(0 + 1)(0 + 1)^*«.»(0 + 1)^* + + («—» + «+» + \lambda)(0 + 1)(0 + 1)^*.$

Оно не нуждается в преобразованиях и содержит окончательное решение для X_4. Переходим к шагу 5.

Шаг 5.

i:=i—1 = 3 > 0.

Переходим к шагу 6.

Шаг 6.

Уравнение для X_3 имеет вид $X_3 = («—» + «+» + \lambda)(0 + 1)(0 + 1)^*$. Оно уже содержит окончательное решение для X_3. Переходим к шагу 5.

Шаг 5.

i:=i—1 = 2 > 0.

Переходим к шагу 6.

Шаг 6.

Уравнение для X_2 имеет вид $X_2 = («—» + «+» + \lambda)«.»(0 + 1)(0 + 1)^* + X_3«.»(0 + 1)^*$. Подставим в него окончательное решение для X_3. Получим окончательное решение для X_2: $X_2 = («—» + «+» + \lambda)«.»(0 + 1)(0 + 1)^* + («—» + «+» + \lambda)(0 + 1)(0 + 1)^*«.»(0 + 1)^*$. Переходим к шагу 5.

Шаг 5.

i:=i—1 = 1 > 0.

Переходим к шагу 6.

Шаг 6.

Уравнение для X_1 имеет вид $X_1 = («—» + «+» + \lambda)$. Оно уже содержит окончательное решение для X_1. Переходим к шагу 5.

Шаг 5.

i:=i—1 = 0 = 0.

Алгоритм завершен.

В итоге получили решение:

$X_1 = («—» + «+» + \lambda);$

$X_2 = («—» + «+» + \lambda)«.»(0 + 1)(0 + 1)^* + («—» + «+» + \lambda)(0 + 1)(0 + 1)^*«.»(0 + 1)^*;$

$X_3 = («—» + «+» + \lambda)(0 + 1)(0 + 1)^*;$

$X_4 = («—» + «+» + \lambda)«.»(0 + 1)(0 + 1)^* + («—» + «+» + \lambda)(0 + 1)(0 + 1)^*«.»(0 + 1)^* + + («—» + «+» + \lambda)(0 + 1)(0 + 1)^*.$

Выполнив несложные преобразования, это же решение можно представить в более простом виде

$X_1 = («—» + «+» + \lambda);$

$X_2 = («-» + «+» + \lambda)(«.»(0 + 1) + (0 + 1)^+«.»)(0 + 1)^*$;

$X_3 = («-» + «+» + \lambda)\ (0 + 1)^+$;

$X_4 = («-» + «+» + \lambda)(«.»(0 + 1) + (0 + 1)^+«.» + (0 + 1))(0 + 1)^*$.

Можно заметить, что регулярное выражение для X_4 описывает язык двоичных чисел с плавающей точкой.

Свойства регулярных языков

Основные свойства регулярных языков

Множество называется замкнутым относительно некоторой операции, если в результате выполнения этой операции над любыми элементами, принадлежащими данному множеству, получается новый элемент, принадлежащий тому же множеству.

Например, множество целых чисел замкнуто относительно операций сложения, умножения и вычитания, но оно не замкнуто относительно операции деления — при делении двух целых чисел не всегда получается целое число.

Регулярные множества (и однозначно связанные с ними регулярные языки) замкнуты относительно многих операций, которые применимы к цепочкам символов.

Например, регулярные языки замкнуты относительно следующих операций:

- пересечения;
- объединения;
- дополнения;
- итерации;
- конкатенации;
- гомоморфизма (изменения имен символов и подстановки цепочек вместо символов).

Поскольку регулярные множества замкнуты относительно операций пересечения, объединения и дополнения, они представляют булеву алгебру множеств. Существуют и другие операции, относительно которых замкнуты регулярные множества. Вообще говоря, таких операций достаточно много.

Проблемы, разрешимые для регулярных языков

Регулярные языки представляют собой очень удобный тип языков. Для них разрешимы многие проблемы, неразрешимые для других типов языков.

Например, доказано, что разрешимыми являются следующие проблемы.

- *Проблема эквивалентности.* Даны два регулярных языка, $L_1(\mathbf{V})$ и $L_2(\mathbf{V})$. Необходимо проверить, являются ли эти два языка эквивалентными.
- *Проблема принадлежности цепочки языку* . Дан регулярный язык $L(\mathbf{V})$ и цепочка символов $\alpha \in \mathbf{V}^*$. Необходимо проверить, принадлежит ли цепочка данному языку.

- *Проблема пустоты языка*. Дан регулярный язык L(**V**). Необходимо проверить, является ли этот язык пустым, то есть найти хотя бы одну цепочку $\alpha \neq \lambda$, такую что $\alpha \in L(\mathbf{V})$.

Эти проблемы разрешимы вне зависимости от того, каким из трех способов задан регулярный язык. Следовательно, эти проблемы разрешимы для всех способов представления регулярных языков: регулярных множеств, регулярных грамматик и конечных автоматов. На самом деле достаточно доказать разрешимость любой из этих проблем хотя бы для одного из способов представления языка, тогда для остальных способов можно воспользоваться алгоритмами преобразования, рассмотренными выше[1].

Для регулярных грамматик также разрешима проблема однозначности — доказано, что для любой регулярной грамматики можно построить эквивалентную ей однозначную регулярную грамматику.

Иногда бывает необходимо доказать, является или нет некоторый язык регулярным. Существует способ проверки, является или нет заданный язык регулярным. Этот метод основан на проверке так называемой леммы о разрастании языка. Доказано, что если для некоторого заданного языка выполняется лемма о разрастании регулярного языка, то этот язык является регулярным; если же лемма не выполняется, то и язык регулярным не является [4 т.1].

Построение лексических анализаторов

Три способа задания регулярных языков

Регулярные (праволинейные и леволинейные) грамматики, конечные автоматы (КА) и регулярные множества (равно как и обозначающие их регулярные выражения) — это три различных способа, с помощью которых можно задавать регулярные языки. Регулярные языки в принципе можно определять и другими способами, но именно три указанных способа представляют наибольший интерес.

Для этих трех способов определения регулярных языков можно записать следующие строгие утверждения.

Утверждение 1. Язык является регулярным множеством тогда и только тогда, когда он задан леволинейной (праволинейной) грамматикой.

Утверждение 2. Язык может быть задан леволинейной (праволинейной) грамматикой тогда и только тогда, когда он является регулярным множеством.

Утверждение 3. Язык является регулярным множеством тогда и только тогда, когда он задан с помощью конечного автомата.

Утверждение 4. Язык распознается с помощью конечного автомата тогда и только тогда, когда он является регулярным множеством.

Все три способа определения регулярных языков равноправны. Существуют алгоритмы, которые позволяют для регулярного языка, заданного одним из указанных способов, построить другой способ, определяющий тот же самый язык. Это не всегда

[1] Возможны и другие способы представления регулярных множеств, а для них разрешимость указанных проблем будет уже не очевидна.

справедливо для других способов, которыми можно определить регулярные языки [4 т.1, 15, 21, 29].

Из всех возможных преобразований практический интерес для компиляторов представляют построение регулярного выражения и построение КА на основе регулярной грамматики. Ниже рассмотрены алгоритмы, позволяющие выполнять эти преобразования.

Построение регулярного выражения для языка, заданного леволинейной грамматикой

Постановка задачи

Для любого регулярного языка, заданного регулярной грамматикой, можно получить регулярное выражение, определяющее тот же язык.

Задача формулируется следующим образом: имеется леволинейная грамматика G(**VT**,**VN**,**P**,S), необходимо найти регулярное выражение над алфавитом **VT**, определяющее язык L(**G**), заданный этой грамматикой.

Задача решается в два этапа.

1. На основе грамматики **G,** задающей язык L(**G**), строим систему уравнений с регулярными коэффициентами.
2. Решаем полученную систему уравнений. Решение, полученное для целевого символа грамматики S, будет представлять собой искомое регулярное выражение, определяющее язык L(**G**).

Поскольку алгоритм решения системы уравнений с регулярными коэффициентами известен, далее будет рассмотрен только алгоритм, позволяющий на основе грамматики G(**VT**,**VN**,**P**,S) построить систему уравнений с регулярными коэффициентами.

Построение системы уравнений с регулярными коэффициентами на основе регулярной грамматики

В данном случае преобразование не столь элементарно. Выполняется оно следующим образом.

1. Обозначим символы алфавита нетерминальных символов **VN** следующим образом: **VN** = $\{X_1, X_2, ..., X_n\}$. Тогда все правила грамматики будут иметь вид $X_i \to X_j\gamma$ или $X_i \to \gamma X_j, X_j \in \mathbf{VN}, \gamma \in \mathbf{VT}^*$; целевому символу грамматики S будет соответствовать некоторое обозначение X_k.
2. Построим систему уравнений с регулярными коэффициентами на основе переменных $X_1, X_2, ..., X_n$:

$$X_1 = \alpha_{01} + X_1\alpha_{11} + X_2\alpha_{21} + ... + X_n\alpha_{n1};$$
$$X_2 = \alpha_{02} + X_1\alpha_{12} + X_2\alpha_{22} + ... + X_n\alpha_{n2};$$
$$...$$
$$X_n = \alpha_{0n} + X_1\alpha_{1n} + X_2\alpha_{2n} + ... + X_n\alpha_{nn};$$

коэффициенты $\alpha_{01}, \alpha_{02}, ..., \alpha_{0n}$ выбираются следующим образом:

$$\alpha_{0i} = (\gamma_1 + \gamma_2 + ... + \gamma_m),$$

если во множестве правил **P** грамматики **G** существуют правила $X_i \rightarrow \gamma_1|\gamma_2|...|\gamma_m$;

$$\alpha_{0i} = \varnothing,$$

если правил такого вида не существует;

коэффициенты α_{j1}, α_{j2}, ..., α_{jn} для некоторого j выбираются следующим образом:

$$\alpha_{ji} = (\gamma_1 + \gamma_2 + ... + \gamma_m),$$

если во множестве правил **P** грамматики **G** существуют правила $X_i \rightarrow X_j\gamma_1|X_j\gamma_2|...|X_j\gamma_m$;

$$\alpha_{ji} = \varnothing,$$

если правил такого вида не существует.

3. Находим решение построенной системы уравнений.

Доказано, что решение для X_k, которое обозначает целевой символ S грамматики **G**, будет представлять собой искомое регулярное выражение, обозначающее язык, заданный грамматикой **G**.

Остальные решения системы будут представлять собой регулярные выражения, обозначающие понятия грамматики, соответствующие ее нетерминальным символам.

СОВЕТ

В принципе, для поиска регулярного выражения, обозначающего язык, заданный грамматикой, не нужно искать все решения — достаточно найти решение для X_k.

Пример построения регулярного выражения для языка, заданного леволинейной грамматикой

Например, рассмотрим леволинейную регулярную грамматику определяющую язык двоичных чисел с плавающей точкой (`{.,–,+,0,1}, {<знак>,<дробное>,<целое>, <число>}, P, <число>`):

P:

`<знак>` → –|+|λ

`<дробное>` → `<знак>.0|<знак>.1|<целое>.|<дробное>0|<дробное>1`

`<целое>` → `<знак>0|<знак>1|<целое>0|<целое>1`

`<число>` → `<дробное>|<целое>`

Обозначим символы множества VN ={<знак>,<дробное>,<целое>,<число>} соответствующими переменными X_i, получим VN={X_1,X_2,X_3,X_4}.

Построим систему уравнений на основе правил грамматики **G**:

$X_1 = («–» + «+» + \lambda)$;

$X_2 = X_1».»(0+1) + X_3«.» + X_2(0+1)$;

$X_3 = X_1(0+1) + X_3(0+1)$;

$X_4 = X_2 + X_3$.

Эта система уравнений уже была решена выше. В данном случае нас интересует только решение для X_4, которое соответствует целевому символу грамматики **G** <число>.

Решение для X_4 может быть записано в виде

$X_4 = («-» + «+» + \lambda)(«.»(0+1) + (0+1)(0+1)*«.» + (0+1))(0+1)*,$

то есть

```
<число> = ("-" + "+" + λ)("."(0+1) + (0+1)(0+1)*"." + (0+1))(0+1)*
```

Это и есть регулярное выражение, определяющее язык двоичных чисел с плавающей точкой, заданный грамматикой **G**.

Построение конечного автомата на основе леволинейной грамматики

Постановка задачи

На основе имеющейся регулярной грамматики можно построить эквивалентный ей КА и, наоборот, для заданного КА можно построить эквивалентную ему регулярную грамматику.

Это очень важное утверждение, поскольку регулярные грамматики используются для определения лексических конструкций языков программирования. Создав КА на основе известной грамматики, мы получаем распознаватель для лексических конструкций данного языка. Таким образом, удается решить задачу разбора для лексических конструкций языка, заданных произвольной регулярной грамматикой. Обратное утверждение также полезно, поскольку позволяет узнать грамматику, цепочки языка которой допускает заданный автомат.

Все языки программирования определяют нотацию записи «слева направо». В той же нотации работают и компиляторы. Поэтому далее рассмотрены алгоритмы для леволинейных грамматик. Доказано, что аналогичные построения возможно выполнить и для праволинейных грамматик.

Задача формулируется следующим образом: имеется леволинейная грамматика G(**VT**,**VN**,**P**,S), задающая язык L(**G**), необходимо построить эквивалентный ей конечный автомат M(**Q**,**V**,δ,q_0,**F**), задающий тот же язык: L(**G**) = L(**M**).

Задача решается в два этапа.

1. Исходную леволинейную грамматику **G** необходимо привести к автоматному виду **G**'.
2. На основе полученной автоматной леволинейной грамматики G'(**VT**,**VN**',**P**',S) строится искомый автомат M(**Q**,**V**,δ,q_0,**F**).

Алгоритм преобразования к автоматному виду был рассмотрен выше, поэтому здесь рассмотрим только алгоритм построения КА на основе автоматной леволинейной грамматики.

Алгоритм построения конечного автомата на основе автоматной леволинейной грамматики

Построение КА M(**Q**,**V**,δ,q_0,**F**) на основе автоматной леволинейной грамматики G(**VT**,**VN**,**P**,S) выполняется по следующему алгоритму.

Шаг 1. Строим множество состояний автомата **Q**. Состояния автомата строятся таким образом, чтобы каждому нетерминальному символу из множества **VN** грамматики **G** соответствовало одно состояние из множества **Q** автомата **M**. Кроме того, во множество состояний автомата добавляется еще одно дополнительное состояние, которое будем обозначать H. Сохраняя обозначения нетерминальных символов грамматики **G**, можно записать **Q** = **VN**∪{H}.

Шаг 2. Входным алфавитом автомата **M** является множество терминальных символов грамматики **G**: **V** = **VT**.

Шаг 3. Просматриваем все множества правил исходной грамматики.

Если встречается правило вида A →t∈ **P**, где A∈ **VN**, t∈ **VT**, то в функцию переходов δ(H,t) автомата **M** добавляем состояние A: A∈ δ(H,t).

Если встречается правило вида A →Bt∈ **P**, где A,B ∈ **VN**, t∈ **VT**, то в функцию переходов δ(B,t) автомата **M** добавляем состояние A: A∈ δ(B,t).

Шаг 4. Начальным состоянием автомата **M** является состояние H: q_0 = H.

Шаг 5. Множество конечных состояний автомата **M** состоит из одного состояния. Этим состоянием является состояние, соответствующее целевому символу грамматики **G**: **F** = {S}.

На этом построение автомата заканчивается.

Пример построения конечного автомата на основе заданной леволинейной грамматики

Рассмотрим грамматику `G({"a","(","*",")","{","}"},{S,C,K},P,S)` (символы `a`, `(`, `*`, `)`, `{`, `}` из множества терминальных символов грамматики взяты в кавычки, чтобы выделить их среди фигурных скобок, обозначающих множество):

```
P:
S → C*) | K}
C → (* | Ca | C{ | C} | C( | C* | C)
K → { | Ka | K( | K* | K) | K{
```

Это леволинейная регулярная грамматика . Преобразование ее к автоматному виду уже было выполнено ранее (в разделе «Регулярные и автоматные грамматики»).

Получим леволинейную автоматную грамматику следующего вида:

`G'({"a","(","*",")","{","}"}, {S,`S_1`,C,`C_1`,K},P',S):`

```
P':
S → S1) | K}
S1 → C*
C → C1* | Ca | C{ | C} | C( | C* | C)
C1 → (
K → { | Ka | K( | K* | K) | K{
```

Для удобства переобозначим нетерминальные символы C_1 и S_1 символами D и E. Получим новую автоматную леволинейную грамматику G'({"a","(","*",")","{","}"},{S,E,C,D,K},**P'**,S):

P':

S → E) | K}

E → C*

C → D* | Ca | C{ | C} | C(| C* | C)

D → (

K → { | Ka | K(| K* | K) | K{

Построим конечный автомат **M(Q,V**,δ,q_0,**F)**, эквивалентный указанной грамматике.

Шаг 1. Строим множество состояний автомата. Получаем

Q=VN∪{H}={S,E,C,D,K,H}.

Шаг 2. В качестве алфавита входных символов автомата берем множество терминальных символов грамматики. Получаем V={"a","(","*",")","{","}"}.

Шаг 3. Рассматриваем множество правил грамматики.

Для правил S→E)|K} имеем δ(E,")")={S}; δ(K,"}")={S}.

Для правила E→C* имеем δ(C,"*")={E}.

Для правил C→D*|Ca|C{|C}|C(|C*|C) имеем δ(D,"*")={C}; δ(C,"a")={C}; δ(C,"{")={C}; δ(C,"}")={C}; δ(C,"(")={C}; δ(C,"*")={E,C}; δ(C,")")={C}.

Для правила D→(имеем δ(H,"(")={D}.

Для правил K→{|Ka|K(|K*|K)|K{ имеем δ(H,"{")={K}; δ(K,"a")={K}; δ(K,"(")={K}; δ(K,"*")={K}; δ(K,")")={K}; δ(K,"{")={K}.

Шаг 4. Начальным состоянием автомата является состояние q_0 = H.

Шаг 5. Множеством конечных состояний автомата является множество F = {S}.

Выполнение алгоритма закончено.

В итоге получаем автомат M({S,E,C,D,K,H}, {"a","(","*",")","{","}"},δ,H,{S}) с функцией переходов:

δ(H,"{") = {K} δ(H,"(") = {D} δ(K,"a") = {K}

δ(K,"(") = {K} δ(K,"*") = {K} δ(K,")") = {K}

δ(K,"{") = {K} δ(K,"}") = {S} δ(D,"*") = {C}

δ(C,"a") = {C} δ(C,"{") = {C} δ(C,"}") = {C}

δ(C,"(") = {C} δ(C,"*") = {E,C} δ(C,")") = {C}

δ(E,")") = {S}

Граф переходов этого автомата изображен на рис. 3.9.

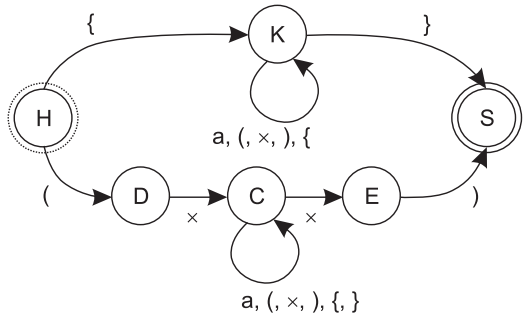

Рис. 3.9. Недетерминированный КА для языка комментариев в Borland Pascal

Это недетерминированный КА, поскольку существует состояние, в котором множество, получаемое с помощью функции переходов по одному и тому же символу, имеет более одного следующего состояния. Это состояние C и функция δ(C,"*")={E,C}.

Моделировать поведение недетерминированного КА — непростая задача, поэтому можно построить эквивалентный ему детерминированный КА. Полученный таким путем КА можно затем минимизировать.

В результате всех преобразований получаем детерминированный конечный автомат M'({S,E,C,D,K,H},{"a","(","*",")","{","}"},δ',H,{S}) с функцией переходов:

```
δ'(H,"{") = {K}   δ'(H,"(") = {D}   δ'(K,"a") = {K}
δ'(K,"(") = {K}   δ'(K,"*") = {K}   δ'(K,")") = {K}
δ'(K,"{") = {K}   δ'(K,"}") = {S}   δ'(D,"*") = {C}
δ'(C,"a") = {C}   δ'(C,"{") = {C}   δ'(C,"}") = {C}
δ'(C,"(") = {C}   δ'(C,")") = {C}   δ'(C,"*") = {E}
δ'(E,"a") = {C}   δ'(E,"{") = {C}   δ'(E,"}") = {C}
δ'(E,"(") = {C}   δ'(E,"*") = {E}   δ'(E,")") = {S}
```

Граф переходов этого автомата изображен на рис. 3.10.

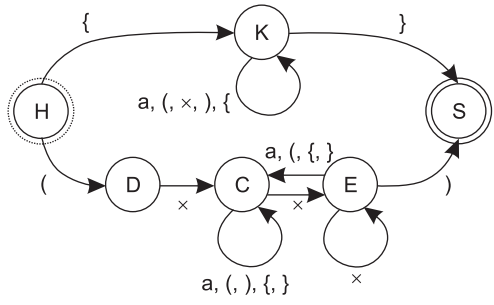

Рис. 3.10. Детерминированный КА для языка комментариев в Borland Pascal

На основании этого автомата можно легко построить распознаватель. В данном случае мы можем получить распознаватель для двух типов комментариев языка программирования Borland Pascal, если учесть, что a может означать любой алфавитно-цифровой символ, кроме символов (, *,), {, }.

Другие примеры построения лексических анализаторов можно найти в книгах [42, 58].

Автоматизация построения лексических анализаторов (программа LEX)

Лексические распознаватели (сканеры) — это не только важная часть компиляторов. Как было показано выше на примерах, лексический анализ применяется во многих других областях, связанных с обработкой текстовой информации на компьютере. Прежде всего, лексического анализа требуют все возможные командные процессоры и утилиты, предусматривающие ввод командных строк и параметров пользователем. Кроме того, хотя бы самый примитивный лексический анализ вводимого текста применяют практически все современные текстовые редакторы и текстовые процессоры. Практически любой более-менее серьезный разработчик программного обеспечения рано или поздно сталкивается с необходимостью решать задачу лексического анализа (разбора)[1].

С одной стороны, задачи лексического анализа имеют широкое распространение, а с другой — допускают четко формализованное решение на основе техники моделирования работы КА. Все это вместе предопределило возможность автоматизации решения данной задачи. И программы, ориентированные на ее решение, не замедлили появиться. Причем, поскольку математический аппарат КА известен уже длительное время, да и с задачами лексического анализа программисты столкнулись довольно давно, и программам, их решающим, насчитывается не один десяток лет.

Для решения задач построения лексических анализаторов существуют различные программы. Наиболее известной среди них является программа LEX.

LEX — программа для генерации сканеров (лексических анализаторов). Входной язык содержит описания лексем в терминах регулярных выражений. Резуль татом работы LEX является программа на некотором языке программирования, которая читает входной файл (обычно это стандартный ввод) и выделяет из него последовательности символов (лексемы), соответствующие заданным регулярным выражениям.

История программы LEX тесно связана с историей операционных систем типа UNIX. Эта программа появилась в составе утилит ОС UNIX и в настоящее время входит в поставку практически каждой ОС этого типа. Однако сейчас существуют версии программы LEX практически для любой ОС, в том числе для широко распространенных MS-DOS и MS Windows всех версий.

[1] С другой стороны, далеко не каждый разработчик, столкнувшись с этой задачей, понимает , что он имеет дело именно с лексическим анализом текста. Поэтому не всегда такие задачи решаются должным образом, в чем автор мог убедиться на собственном опыте.

Поскольку LEX появилась в среде ОС UNIX, то первоначально языком программирования, на котором строились порождаемые ею лексические анализаторы, был язык C. Но со временем появились версии LEX, порождающие сканеры и на основе других языков программирования (например, известны версии для языка Pascal).

СОВЕТ

Для поиска нужной версии программы лексического анализа LEX автор советует заинтересованным лицам прежде всего обратиться во Всемирную сеть Интернет . В настоящее время существует огромное множество версий этой и подобных ей программ автоматизации построения лексических анализаторов.

Принцип работы LEX достаточно прост: на вход ей подается текстовый файл, содержащий описания нужных лексем в терминах регулярных выражений, а на выходе получается файл с текстом исходной программы сканера на заданном языке программирования (обычно — на C). Т екст исходной программы сканера может быть дополнен вызовами любых функций из любых библиотек, поддерживаемых данным языком и системой программирования. Таким образом, LEX позволяет значительно упростить разработку лексических анализаторов, практически сводя эту работу к описанию требуемых лексем в терминах регулярных выражений.

Более подробную информацию о работе с программой LEX можно получить в [3, 9, 22, 31, 53, 55, 58, 62].

Контрольные вопросы и задачи

Вопросы

1. Какие грамматики относятся к регулярным грамматикам? Назовите два класса регулярных грамматик. Как они соотносятся между собой?
2. В чем заключается отличие автоматных грамматик от других регулярных грамматик? Всякая ли регулярная грамматика является автоматной? Всякая ли регулярная грамматика может быть преобразована к автоматному виду?
3. Если язык, заданный регулярной грамматикой, содержит пустую цепочку, то может ли он быть задан автоматной грамматикой?
4. Можно ли граф переходов конечного автомата использовать для однозначного определения данного автомата (и если нет — то почему)?
5. Всегда ли недетерминированный КА может быть преобразован к детерминированному виду (если нет — то в каких случаях)?
6. Какое максимальное количество состояний может содержать ДКА после преобразования из недетерминированного КА? Всегда ли это количество состояний можно сократить?
7. Чем различаются таблица лексем и таблица идентификаторов? В какую из этих таблиц лексический анализатор не должен помещать ключевые слова, разделители и знаки операций?
8. Являются ли регулярными множествами следующие множества:
   - множество целых натуральных чисел;

- множество вещественных чисел;
- множество всех слов русского языка;
- множество всех возможных строковых констант в языке Pascal;
- множество иррациональных чисел;
- множество всех возможных вещественных констант языка C?

9. На основании свойств регулярных выражений докажите следующие тождества для произвольных регулярных выражений α, β, γ и δ:

1) $(\alpha + \beta)(\delta + \gamma) = \alpha\delta + \alpha\gamma + \beta\delta + \beta\gamma$;
2) $(\delta\alpha + \beta)\gamma = \delta\alpha\gamma + \delta\beta\gamma$;
3) $\beta + \beta\alpha + \beta\alpha^* = \beta\alpha^*$;
4) $\beta + \beta\alpha\alpha^* + \beta\alpha\alpha\alpha^* = \beta\alpha^*$;
5) $(\delta\alpha\gamma)^* + \delta\alpha^*\gamma + \delta\gamma = \delta\alpha^*\gamma$;
6) $(0^* + \alpha\alpha^* + \alpha\alpha\alpha^*)^* = \alpha^*$.

7. Используя свойства регулярных множеств, проверьте, являются ли истинными следующие тождества для произвольных регулярных выражений α и β:

1) $(\alpha + \beta)^* = (\alpha^*\beta^*)^*$;
2) $(\alpha\beta)^* = \alpha^*\beta^*$;
3) $(\alpha + \lambda)^* = \alpha^*$;
4) $\alpha^*\beta^* + \beta^*\alpha^* = \alpha^*\beta^*\alpha^*$;
5) $\alpha^*\alpha^* = \alpha^*$.

6. Почему возможны две формы записи уравнений с регулярными коэффициентами? Как они соотносятся с двумя классами регулярных грамматик?
7. Можно ли для языка, заданного леволинейной грамматикой, построить праволинейную грамматику, задающую эквивалентный язык?
8. Всякая ли регулярная грамматика является однозначной? Если нет, то приведите пример неоднозначной регулярной грамматики.
9. Какие из следующих утверждений являются истинными:
   - если язык задан регулярным множеством, то он может быть задан праволинейной грамматикой;
   - если язык задан КА, то он может быть задан КС-грамматикой;
   - если язык задан КС-грамматикой, то для него можно построить регулярное выражение;
   - если язык задан КА, то для него можно построить регулярное выражение.
10. Можно ли для двух леволинейных грамматик доказать, что они задают один и тот же язык? Можно ли выполнить то же самое доказательство для леволинейной и праволинейной грамматик?

Задачи

1. Определите, какие из перечисленных ниже грамматик являются регулярными, леволинейными, праволинейными, автоматными:

G_1({".",+,−,0,1}, {<число>,<часть>,<цифра>,<осн>}, $\mathbf{P_1}$, <число>);
$\mathbf{P_1}$: <число> → +<осн> | −<осн> | <осн>
<осн> → <часть>.<часть> | <часть>. | <часть>
<часть> → <цифра> | <часть><цифра>
<цифра> → 0 | 1
G_2({".",+,−,0,1}, {<число>,<часть>,<осн>}, $\mathbf{P_2}$, <число>);
$\mathbf{P_2}$: <число> → +<осн> | −<осн> | <осн>
<осн> → <часть>. | <часть> | <осн>0 | <осн.>1
<часть> → 0 | 1 | <часть>0 | <часть>1
G_3({".",+,−,0,1}, {<число>,<часть>,<осн>}, $\mathbf{P_3}$, <число>);
$\mathbf{P_3}$: <число> → +<осн> | −<осн> | <осн>
<осн> → 0 | 1 | 0.<часть> | 1.<часть> | 0<осн> | 1<осн>
<часть> → 0 | 1 | 0<часть> | 1<часть>
G_4({".",+,−,0,1}, {<знак>,<число>,<часть>}, $\mathbf{P_4}$, <число>);
$\mathbf{P_4}$: <число> → <знак>0 | <знак>1 | <часть>. | <число>0 | <число>1
<часть> → <знак>0 | <знак>1 | <часть>0 | <часть>1
<знак> → λ | + | −
G_5({".",+,−,0,1}, {<знак>,<число>,<часть>,<осн>}, $\mathbf{P_5}$, <число>);
$\mathbf{P_5}$: <число> → <часть>. | <осн>0 | <осн>1 | <часть>0 | <часть>1
<осн> → <часть>. | <осн>0 | <осн>1
<часть> → 0 | 1 | <знак>0 | <знак>1 | <часть>0 | <часть>1
<знак> → + | −

2. Преобразуйте к автоматному виду грамматики $\mathbf{G_3}$ и $\mathbf{G_4}$ из задачи 1.
3. Постройте КА на основании грамматик $\mathbf{G_4}$ или $\mathbf{G_5}$ из задачи 1. Преобразуйте построенный автомат к детерминированному виду.
4. Задан КА:M({H,I,S,E,F,G}, {b,d}, δ, H, {S}),δ(H,b)={I,S}, δ(H,d)={E}, δ(I,b)={I,S}, δ(I,d)={I,S}, δ(E,b)={E,F}, δ(E,d)={E,F}, δ(F,b)={E}, δ(F,d)={E}, δ(G,b)={F,S}, δ(G,d)={S}. Минимизируйте его и постройте эквивалентный ДКА.
5. Задан КА: M({S,R,Z}, {a,b}, δ, S, {Z}), δ(S,a)={S,R}, δ(R,b)={R}, δ(R,a)={Z}. Преобразуйте его к детерминированному виду и минимизируйте полученный КА.
6. Постройте и решите систему уравнений с регулярными коэффициентами для грамматики $\mathbf{G_4}$ из задачи 1. Какой язык задает эта грамматика?
7. Докажите, что уравнение с регулярными коэффициентами $X = \alpha X + \beta$ имеет решение $X = \alpha^*(\beta+\gamma)$, где γ обозначает произвольное множество, в том случае, когда множество, обозначенное выражением α, содержит пустую цепочку. (Указание: поскольку множество, обозначенное выражением α, содержит пустую цепочку, то выражение α можно представить в виде $\alpha=\delta+\lambda$, при этом $\alpha^*=\delta^*$.)
8. Решите систему уравнений с регулярными коэффициентами над алфавитом $\mathbf{V} = \{0,1\}$:

$X_1 = 0X_2 + 1X_1 + \lambda;$

$X_2 = 0X_3 + 1X_2$;
$X_3 = 0X_1 + 1X_3$.

9. В языке C++ возможны два типа комментариев: обычный комментарий, ограниченный символами `/*` и `*/`, и строчный комментарий, начинающийся с символов `//` и заканчивающийся концом строки. Постройте грамматику для этого языка, преобразуйте ее к автоматному виду.

Упражнения

1. Докажите (любым способом), что грамматики $\mathbf{G_3}$, $\mathbf{G_4}$ и $\mathbf{G_5}$ из задачи 1 задают один и тот же язык.
2. Постройте регулярную грамматику, которая бы описывала язык строк, целочисленных констант и строковых констант языка Pascal. Постройте и решите систему уравнений с регулярными коэффициентами на основе этой грамматики.
3. В языке C++ возможны два типа комментариев: обычный комментарий, ограниченный символами `/*` и `*/`, и строчный комментарий, начинающийся с символов `//` и заканчивающийся концом строки. Постройте сканер, который бы находил и исключал из входного текста все комментарии языка C++.
4. С помощью леммы о разрастании для регулярных языков [4 т .1, 15] докажите, что язык $L_1 = \{a^n,b^n \mid n > 2\}$ не является регулярным, а язык $L_2 = \{a^n b^m \mid n > 1, m > 2\}$ является регулярным.
5. Постройте сканер, который выделял бы из текста входной программы на некотором языке программирования все содержащиеся в ней строковые константы и записывал их в отдельный файл.

ГЛАВА 4 Синтаксические анализаторы

Основные принципы работы синтаксических анализаторов

Назначение синтаксических анализаторов

Синтаксический анализатор (синтаксический разбор) — это часть компилятора , которая отвечает за выявление и проверку синтаксических конструкций входного языка. В задачу синтаксического анализа входят следующие функции:

- найти и выделить синтаксические конструкции в тексте исходной программы;
- установить тип и проверить правильность каждой синтаксической конструкции;
- представить синтаксические конструкции в виде, удобном для дальнейшей генерации текста результирующей программы.

Синтаксический анализатор — это основная часть компилятора на этапе анализа. Без выполнения синтаксического разбора работа компилятора бессмысленна, в то время как лексический разбор, в принципе, является необязательной фазой. Все задачи по проверке синтаксиса входного языка могут быть решены на этапе синтаксического разбора. Лексический анализатор только позволяет избавить сложный по структуре синтаксический анализатор от решения примитивных задач по выявлению и запоминанию лексем исходной программы.

Синтаксический анализатор воспринимает выход лексического анализатора и разбирает его в соответствии с грамматикой входного языка. Однако в грамматике входного языка программирования обычно не уточняется, какие конструкции следует считать лексемами . Примерами конструкций, которые обычно распознаются во время лексического анализа, служат ключевые слова, константы и идентификаторы. Но эти же конструкции могут распознаваться и синтаксическим анализатором. На практике не существует жесткого правила, определяющего, какие конструкции должны распознаваться на лексическом уровне, а какие надо оставлять синтаксическому анализатору . Обычно это определяет разработчик компилятора, исходя из технологических аспектов программирования, а также синтаксиса и семантики входного языка. Принципы взаимодействия лексического и синтаксического анализаторов были рассмотрены ранее в главе «Лексические анализаторы».

В основе синтаксического анализатора лежит распознаватель текста исходной программы, построенный на основе грамматики входного языка. Как правило, синтаксические конструкции языков программирования могут быть описаны с помощью КС-грамматик, реже встречаются языки, которые могут быть описаны с помощью регулярных грамматик .

ПРИМЕЧАНИЕ

Чаще всего регулярные грамматики применимы к языкам ассемблера, а синтаксис языков высокого уровня основан на грамматике КС-языков.

Главную роль в том, как функционирует синтаксический анализатор и какой алгоритм лежит в его основе, играют принципы построения распознавателей для КС-языков. Без применения этих принципов невозможно выполнить эффективный синтаксический разбор предложений входного языка.

Распознавателями для КС-языков являются автоматы с магазинной памятью — МП-автоматы — односторонние недетерминированные распознаватели с линейно-ограниченной магазинной памятью (классификация распознавателей приведена в соответствующем разделе главы «Формальные языки и грамматики»). Поэтому важно рассмотреть, как функционирует МП-автомат и как для КС-языков решается задача разбора — построение распознавателя языка на основе заданной грамматики. Ниже в этой главе рассмотрены технические аспекты, связанные с реализацией синтаксических анализаторов.

Автоматы с магазинной памятью (МП-автоматы)

Определение МП-автомата

Контекстно-свободными (КС) называются языки, определяемые грамматиками типа G(**VT**,**VN**,**P**,S), в которых правила **P** имеют вид $A \to \beta$, где $A \in \mathbf{VN}$ и $\beta \in \mathbf{V}^*$, $\mathbf{V}=\mathbf{VT} \cup \mathbf{VN}$. Распознавателями КС-языков служат автоматы с магазинной памятью (*МП-автоматы*).

В общем виде МП-автомат можно определить следующим образом:

$R(\mathbf{Q},\mathbf{V},\mathbf{Z},\delta,q_0,z_0,\mathbf{F})$,

где

Q — множество состояний автомата;

V — алфавит входных символов автомата;

Z — специальный конечный алфавит магазинных символов автомата, $\mathbf{V} \subseteq \mathbf{Z}$;

δ — функция переходов автомата, которая отображает множество $\mathbf{Q} \times (\mathbf{V} \cup \{\lambda\}) \times \mathbf{Z}$ на конечное множество подмножеств $P(\mathbf{Q} \times \mathbf{Z}^*)$;

$q_0 \in \mathbf{Q}$ — начальное состояние автомата;

$z_0 \in \mathbf{Z}$ — начальный символ магазина;

$\mathbf{F} \subseteq \mathbf{Q}$ — множество конечных состояний.

МП-автомат в отличие от обычного КА имеет стек (магазин), в который можно помещать специальные «магазинные» символы (обычно это терминальные и нетерминальные символы грамматики языка). Переход из одного состояния в другое зависит не

только от входного символа, но и от символа на верхушке стека. Таким образом, конфигурация автомата определяется тремя параметрами: состоянием автомата, текущим символом входной цепочки (положением указателя в цепочке) и содержимым стека.

МП-автомат условно можно представить в виде схемы на рис. 4.1.

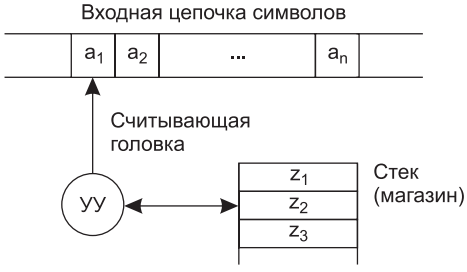

Рис. 4.1. Общая условная схема автомата с магазинной памятью (МП-автомата)

Конфигурация МП-автомата описывается в виде тройки $(q,\alpha,\omega)\in \mathbf{Q}\times\mathbf{V}^*\times\mathbf{Z}^*$, которая определяет текущее состояние автомата q, цепочку еще непрочитанных символов α на входе автомата и содержимое магазина (стека) ω. Вместо α в конфигурации можно указать пару (β,n), где $\beta\in \mathbf{V}^*$ — вся цепочка входных символов, а $n\in \mathbf{N}\cup\{0\}$, $n\geq 0$ — положение считывающего указателя в цепочке.

Тогда один такт работы автомата можно описать в виде $(q,a\ \alpha,z\omega) \div (q',\alpha,\gamma\omega)$, если $(q',\gamma)\in \delta(q,a,z)$, где $q,q'\in \mathbf{Q}$, $a\in \mathbf{V}\cup\{\lambda\}$, $\alpha\in \mathbf{V}^*$, $z\in \mathbf{Z}$, $\gamma,\omega\in \mathbf{Z}^*$. При выполнении такта (перехода) из стека удаляется верхний символ, соответствующий условию перехода, и добавляется цепочка, соответствующая правилу перехода. Первый символ цепочки становится верхушкой стека. Допускаются переходы, при которых входной символ игнорируется (и тем самым он будет входным символом при следующем переходе). Эти переходы (такты) называются λ-*переходами* (λ-тактами).

Начальная конфигурация МП-автомата, очевидно, определяется как (q_0,α,z_0), $\alpha\in \mathbf{V}^*$. Множество конечных конфигураций автомата — (q,λ,ω), $q\in \mathbf{F}$, $\omega\in \mathbf{Z}^*$.

МП-автомат допускает (принимает) цепочку символов, если, получив эту цепочку на вход, он может перейти в одну из конечных конфигураций — когда при окончании цепочки автомат находится в одном из конечных состояний, а стек содержит некоторую заданную цепочку символов (возможно, пустую). Иначе цепочка символов не принимается.

Язык, определяемый МП-автоматом, — это множество всех цепочек символов, которые допускает данный автомат. Язык, определяемый МП-автоматом $\mathbf{R}$, обозначается как $L(\mathbf{R})$. Два МП-автомата называются эквивалентными, если они определяют один и тот же язык: $\mathbf{R}_1\equiv\mathbf{R}_2 \Leftrightarrow L(\mathbf{R}_1)=L(\mathbf{R}_2)$.

МП-автомат допускает цепочку символов с опустошением магазина, если при окончании разбора цепочки автомат находится в одном из конечных состояний, а стек пуст — конфигурация (q,λ,λ), $q\in \mathbf{F}$. Если язык задан МП-автоматом $\mathbf{R}$, который допускает цепочки с опустошением стека, это обозначается так: $L^{\lambda}(\mathbf{R})$. Доказано, что для любого МП-автомата всегда можно построить эквивалентный ему МП-автомат, допускающий цепочки заданного языка с опустошением стека [4 т .1, 5]. То есть $\forall$ МП-автомата $\mathbf{R}$: $\exists$ МП-автомат $\mathbf{R}'$, такой, что: $L(\mathbf{R})=L^{\lambda}(\mathbf{R}')$.

Расширенные МП-автоматы

Кроме обычного МП-автомата, существует также понятие расширенного МП-автомата .

Расширенный МП-автомат может заменять цепочку символов конечной длины в верхней части стека на другую цепочку символов конечной длины, в отличие от обычного МП-автомата. В отличие от обычного МП-автомата, который на каждом такте работы может изымать из стека только один символ, расширенный МП-автомат может изымать за один такт цепочку символов, находящуюся на вершине стека. Функция переходов δ для расширенного МП-автомата отображает множество $\mathbf{Q}\times(\mathbf{V}\cup\{\lambda\})\times\mathbf{Z}^*$ на конечное множество подмножеств $P(\mathbf{Q}\times\mathbf{Z}^*)$.

Доказано, что для любого расширенного МП-автомата всегда можно построить эквивалентный ему обычный МП-автомат (обратное утверждение очевидно, так как любой обычный МП-автомат является и расширенным МП-автоматом) [4 т1, 5]. Таким образом, классы МП-автоматов и расширенных МП-автоматов эквивалентны и задают один и тот же тип языков.

Доказано, что для произвольной КС-грамматики всегда можно построить МП-автомат, распознающий заданный этой грамматикой язык [4 т.1, 5, 15]. Поэтому можно говорить, что МП-автоматы распознают КС-языки. Существует также доказательство того, что для произвольного МП-автомата всегда можно построить КС-грамматику, которая задает язык, распознаваемый этим автоматом. Т о есть КС-грамматики и МП-автоматы задают один и тот же тип языков — КС-языки.

Поскольку класс расширенных МП-автоматов эквивалентен классу обычных МП-автоматов и задает тот же самый тип языков, то можно утверждать, что и расширенные МП-автоматы распознают языки из типа КС-языков. Следовательно, язык, который может распознавать расширенный МП-автомат , также может быть задан с помощью КС-грамматики.

Детерминированные МП-автоматы

МП-автомат называется *детерминированным*, если из каждой его конфигурации возможно не более одного перехода в следующую конфигурацию. В противном случае МП-автомат называется недетерминированным.

Формально для детерминированного МП-автомата (ДМП-автомата) $R(\mathbf{Q},\mathbf{V},\mathbf{Z},\delta,q_0,z_0,\mathbf{F})$ функция переходов δ может $\forall q\in\mathbf{Q}$, $\forall a\in\mathbf{V}$, $\forall z\in\mathbf{Z}$ иметь один из следующих трех видов:

1) $\delta(q,a,z)$ содержит один элемент: $\delta(q,a,z)=\{(q',\gamma)\}$, $\gamma\in\mathbf{Z}^*$ и $\delta(q,\lambda,z)=\varnothing$;
2) $\delta(q,a,z)=\varnothing$ и $\delta(q,\lambda,z)$ содержит один элемент: $\delta(q,\lambda,z)=\{(q',\gamma)\}$, $\gamma\in\mathbf{Z}^*$;
3) $\delta(q,a,z)=\varnothing$ и $\delta(q,\lambda,z)=\varnothing$.

Класс ДМП-автоматов и соответствующих им языков значительно уже, чем весь класс МП-автоматов и КС-языков.

ДМП-автоматы определяют очень важный класс среди всех КС-языков, называемый детерминированными КС-языками (ДКС-языками) . Доказано, что все языки, принадлежащие к классу ДКС-языков, могут быть построены с помощью однозначных КС-грамматик. Поскольку однозначность — это важное и обязательное требование

к грамматике любого языка программирования, ДМП-автоматы представляют особый интерес для создания компиляторов.

ВНИМАНИЕ

В отличие от КА, когда для любого недетерминированного КА можно построить эквивалентный ему детерминированный КА, не для каждого МП-автомата можно построить эквивалентный ему ДМП-автомат. Иными словами, в общем случае невозможно преобразовать недетерминированный МП-автомат в детерминированный.

Все без исключения синтаксические конструкции языков программирования задаются с помощью однозначных КС-грамматик. Следовательно, синтаксические структуры этих языков относятся к классу ДКС-языков и могут распознаваться с помощью ДМП-автоматов. Поэтому большинство распознавателей, которые будут рассмотрены далее, относятся к классу ДКС-языков.

ПРИМЕЧАНИЕ

Любой ДКС-язык может быть задан однозначной КС-грамматикой, но обратное неверно: не всякий язык, заданный однозначной КС-грамматикой, является детерминированным. Например, однозначная грамматика G({a,b}, {S,A,B}, {S→A|B, A→aAb|ab, B→aBbb|abb}, S) задает КС-язык, который не является детерминированным.

Построение синтаксических анализаторов

Проблемы построения синтаксических анализаторов

Синтаксический анализатор должен распознавать весь текст исходной программы. Поэтому, в отличие от лексического анализатора, ему нет необходимости искать границы распознаваемой строки символов. Он должен воспринимать всю информацию, поступающую ему на вход, и либо подтвердить ее принадлежность входному языку, либо сообщить об ошибке в исходной программе.

Но, как и в случае лексического анализа, задача синтаксического анализа не ограничивается только проверкой принадлежности цепочки заданному языку. Необходимо оформить найденные синтаксические конструкции для дальнейшей генерации текста результирующей программы. Синтаксический анализатор должен иметь некий выходной язык, с помощью которого он передает следующим фазам компиляции информацию о найденных и разобранных синтаксических структурах. В таком случае он уже является не разновидностью МП-автомата, а преобразователем с магазинной памятью — МП-преобразователем [3, 4 т.1, 5, 15].

ПРИМЕЧАНИЕ

Во всех рассматриваемых далее примерах результатом работы МП-автомата является не только ответ на вопрос о принадлежности входной цепочки заданному языку («да» или «нет»), но и последовательность номеров правил грамматики, использованных для построения входной цепочки. Затем на основе этих правил строятся цепочка вывода и дерево вывода. Поэтому, строго говоря, все рассмотренные примеры МП-автоматов являются не только МП-автоматами, но и МП-преобразователями.

Вопросы, связанные с представлением информации, являющейся результатом работы синтаксического анализатора, и с порождением на основе этой информации

текста результирующей программы, рассмотрены в следующей главе, поэтому здесь на них останавливаться не будем. Однако проблемы, связанные с созданием синтаксического анализатора, не ограничиваются только двумя перечисленными, как это было для лексического анализа. Дело в том, что процесс построения синтаксического анализатора гораздо сложнее аналогичного процесса для лексического анализатора. Так происходит, поскольку КС-грамматики и МП-автоматы, лежащие в основе синтаксического анализа, сложнее, чем регулярные грамматики и КА, лежащие в основе лексического анализа.

Алгоритм создания МП-автомата на основе произвольной КС-грамматики прост [4 т.1, 5, 15], но если напрямую воспользоваться этим алгоритмом, то может оказаться, что работу полученного МП-автомата невозможно будет промоделировать на компьютере. Следовательно, такой автомат практического применения в компиляторе иметь не может . Но, даже если реализовать распознаватель на основе МП-автомата удастся, может оказаться, что время его работы непомерно велико и в худшем случае экспоненциально зависит от длины входной цепочки.

С этими проблемами столкнулись в свое время разработчики первых компиляторов с языков высокого уровня. Для их решения были предприняты определенные исследования в теории формальных языков и, в частности, в области КС-языков.

Во-первых, очевидно, что, поскольку любой язык программирования должен быть задан однозначной КС-грамматикой, а распознавателями для КС-языков, заданных однозначными КС-грамматиками, являются ДМП-автоматы, нужно стремиться построить синтаксический анализатор на основе ДМП-автомата. Моделировать работу ДМП-автомата существенно проще, чем работу произвольного МП-автомата.

Во-вторых, было установлено, что существуют преобразования правил КС-грамматик, выполнив которые, можно привести грамматику к такому виду , когда построение и моделирование работы МП-автомата, распознающего язык, заданный грамматикой, существенно упрощаются. Были найдены и описаны специальные формы правил КС-грамматик, называемые «нормальными формами» , для которых построены соответствующие МП-автоматы [4 т .1, 5, 15]. Преобразовав произвольную КС-грамматику к одной из известных нормальных форм, можно сразу же построить соответствующий МП-автомат (нормальные формы КС-грамматик в данном учебнике не рассматриваются).

Наконец, было найдено множество различных классов КС-грамматик, для которых возможно построить ДМП-автомат, имеющий линейную зависимость времени разбора входной цепочки от ее длины. Такие распознаватели для КС-языков называются *линейными распознавателями*. Часть таких классов КС-грамматик будет рассматриваться далее в этой главе.

В принципе было бы достаточно знать хотя бы один такой класс, если бы для КС-грамматик были разрешимы проблема преобразования и проблема эквивалентности. Но, поскольку в общем случае это не так, одним классом КС-грамматик, для которого существуют линейные распознаватели, ограничиться не удается. По этой причине для всех классов КС-грамматик существует принципиально важное ограничение: в общем случае невозможно преобразовать произвольную КС-грамматику к виду , требуемому данным классом КС-грамматик, либо же доказать, что такого преобразования не существует . То, что проблема неразрешима в общем случае, не говорит

о том, что она не решается в каждом конкретном частном случае, и зачастую удается найти такие преобразования. И чем шире набор классов КС-грамматик с линейными распознавателями, тем проще их искать.

Среди универсальных распознавателей лучшими по эффективности являются табличные, функционирование которых построено на иных принципах, нежели моделирование работы МП-автомата [4 т .1, 5, 15]. Табличные распознаватели обладают полиномиальными характеристиками требуемых вычислительных ресурсов в зависимости от длины входной цепочки: для КС-языков, заданных однозначной грамматикой, удается добиться квадратичной зависимости, для всех прочих КС-языков — кубической зависимости. Но в данном учебнике табличные распознаватели не рассматриваются, поскольку они достаточно сложны, требуют значительного объема памяти (объем необходимой памяти квадратично зависит от длины входной цепочки), но при этом не позволяют добиться практически значимых результатов.

Варианты синтаксических анализаторов

Построение синтаксического анализатора — это менее рутинный процесс, чем построение лексического анализатора. Этот процесс не всегда может быть полностью формализован.

Имея грамматику входного языка, разработчик синтаксического анализатора должен в первую очередь выполнить ряд формальных преобразований над этой грамматикой, облегчающих построение распознавателя. После этого он должен проверить, подпадает ли полученная грамматика под один из известных классов КС-языков, для которых существуют линейные распознаватели. Если такой класс найден, можно строить распознаватель (если найдено несколько классов — выбрать тотдля которого построение распознавателя проще, либо построенный распознаватель обладает лучшими характеристиками). Если же такой класс КС-языков найти не удалось, то разработчик должен попытаться выполнить над грамматикой некоторые преобразования, чтобы привести ее к одному из известных классов. Вот эти преобразования не могут быть описаны формально, и в каждом конкретном случае разработчик должен попытаться найти их сам. Иногда преобразования имеет смысл искать даже в том случае, когда грамматика подпадает под один из известных классов КС-языков, сцелью найти другой класс, для которого можно построить лучший по характеристикам распознаватель.

Только в том случае, когда в результате всех этих действий не удалось найти соответствующий класс КС-языков, разработчик вынужден строить универсальный распознаватель. Характеристики такого распознавателя будут существенно хуже, чем у линейного распознавателя. Такие случаи бывают редко, поэтому все современные компиляторы построены на основе линейных распознавателей (иначе время их работы было бы недопустимо велико).

Для каждого класса КС-языков существует свой класс распознавателей, но все они функционируют на основе общих принципов, на которых основано моделирование работы МП-автоматов. Все распознаватели для КС-языков можно разделить на две большие группы: нисходящие и восходящие.

Нисходящие распознаватели просматривают входную цепочку символов слева направо и порождают левосторонний вывод. При этом получается, что дерево вывода строится таким распознавателем от корня к листьям (сверху вниз), откуда и происходит название распознавателя.

Восходящие распознаватели также просматривают входную цепочку символов слева направо, но порождают при этом правосторонний вывод. Дерево вывода строится восходящим распознавателем от листьев к корню (снизу вверх), откуда и происходит название распознавателя.

Для моделирования работы этих двух групп распознавателей используются два алгоритма: алгоритм с подбором альтернатив — для нисходящих распознавателей и алгоритм «сдвиг—свертка» — для восходящих распознавателей. В общем случае эти два алгоритма универсальны. Они строятся на основе любой КС-грамматики после некоторых формальных преобразований и поэтому могут быть использованы для разбора цепочки любого КС-языка. В этом случае время разбора входной цепочки имеет экспоненциальную зависимость от длины цепочки.

Однако для линейных распознавателей эти алгоритмы могут быть модифицированы так, чтобы время разбора имело линейную зависимость от длины входной цепочки. Указанные модификации не влияют на принципы, на основе которых построена работа алгоритмов, но позволяют оптимизировать их выполнение при условии, что грамматика входного языка принадлежит к определенному классу КС-грамматик. Для каждого такого класса предусматривается своя модификация.

Далеко не все известные распознаватели с линейными характеристиками рассматриваются в данном пособии. Более полный набор распознавателей, а также описание связанных с ними классов КС-грамматик и КС-языков можно найти в [4 т.1, 5, 18, 58, 63].

Выбор распознавателя для синтаксического анализа

Часто одна и та же КС-грамматика может быть отнесена не к одному , а сразу к нескольким классам КС-грамматик, допускающих построение линейных распознавателей. В этом случае необходимо решить, какой из возможных распознавателей выбрать для практической реализации.

Ответить на этот вопрос не всегда легко, поскольку могут быть построены два принципиально разных распознавателя, алгоритмы работы которых несопоставимы. Речь идет о восходящих и нисходящих распознавателях: в основе первых лежит алгоритм «сдвиг—свертка», в основе вторых — алгоритм подбора альтернатив.

На вопрос о том, какой распознаватель — нисходящий или восходящий — выбрать для построения синтаксического анализатора, нет однозначного ответа. Эту проблему необходимо решать, опираясь на некую дополнительную информацию о том, как будут использованы результаты работы распознавателя.

СОВЕТ

Следует помнить, что синтаксический анализатор — это один из этапов компиляции. И с этой точки зрения результаты работы распознавателя служат исходными данными для следующих этапов компиляции. Поэтому выбор того или иного распознавателя во многом зависит от реализации компилятора, от того, какие принципы положены в его основу.

Восходящий синтаксический анализ , как правило, привлекательнее нисходящего, так как для языка программирования легче построить правосторонний (восходящий) распознаватель. Класс языков, заданных восходящими распознавателями, шире, чем класс языков, заданных нисходящими распознавателями.

С другой стороны, как будет показано далее, левосторонний (нисходящий) синтаксический анализ предпочтителен с точки зрения процесса трансляции, поскольку на его основе легче организовать процесс порождения цепочек результирующего языка. Ведь в задачу компилятора входит не только распознать входную программу на входном языке, но и построить результирующую программу. Более подробную информацию об этом можно получить в главе «Генерация и оптимизация кода» или в работах [4 т.1,2, 5]. Левосторонний анализ, основанный на нисходящем распознавателе, оказывается предпочтительным также при учете вопросов, связанных с обнаружением и локализацией ошибок [4 т.1,2, 5, 58, 65].

Желание использовать более простой класс грамматик для построения распознавателя может потребовать каких-то манипуляций с заданной грамматикой, необходимых для ее преобразования к требуемому классу. При этом нередко грамматика становится неестественной и малопонятной, что в дальнейшем затрудняет ее использование в схеме синтаксически управляемого перевода и трансляции на этапе генерации результирующего кода (см. главу «Генерация и оптимизация кода»). Поэтому бывает удобным использовать исходную грамматику такой, как она есть, не стремясь преобразовать ее к более простому классу.

Интерес представляют как левосторонний, так и правосторонний анализ. Конкретный выбор зависит от реализации конкретного компилятора, а также от сложности грамматики входного языка программирования.

Преобразование КС-грамматик. Приведенные грамматики

Цель преобразования КС-грамматик. Приведенные грамматики

Цель преобразования КС-грамматик

Как было сказано выше, для КС-грамматик невозможно в общем случае проверить их однозначность и эквивалентность. Но очень часто правила КС-грамматик можно и нужно преобразовать к некоторому заданному виду таким образом, чтобы получить новую грамматику, эквивалентную исходной. Заранее определенный вид правил грамматики облегчает создание распознавателей.

Можно выделить две основные цели преобразований КС-грамматик: упрощение правил грамматики и облегчение создания распознавателя языка. Не всегда эти две цели можно совместить. В случае с языками программирования, когда итогом работы с грамматикой является создание компилятора, именно вторая цель преобразования является основной. Поэтому упрощениями правил пренебрегают, если при этом удается упростить построение распознавателя языка [12, 24].

Все преобразования условно можно разбить на две группы:

- первая группа — это преобразования, связанные с исключением из грамматики избыточных правил и символов (именно эти преобразования позволяют выполнить упрощения грамматики);

- вторая группа — это преобразования, в результате которых изменяется вид и состав правил грамматики, при этом грамматика может дополняться новыми правилами, а ее словарь нетерминальных символов — новыми символами (то есть преобразования второй группы не связаны с упрощениями).

Следует еще раз подчеркнуть, что всегда в результате преобразований мы получаем новую КС-грамматику, эквивалентную исходной, то есть определяющую тот же самый язык.

Тогда формально преобразование можно определить следующим образом:

$$G(\mathbf{VT},\mathbf{VN},\mathbf{P},S)\rightarrow G'(\mathbf{VT}',\mathbf{VN}',\mathbf{P}',S'):\ L(\mathbf{G})=L(\mathbf{G}').$$

Приведенные грамматики

Приведенные грамматики — это КС-грамматики , которые не содержат недостижимых и бесплодных символов, циклов и λ-правил (правил с пустыми цепочками).

Для того чтобы преобразовать произвольную КС-грамматику к приведенному виду, необходимо выполнить следующие действия:

- удалить все бесплодные символы;
- удалить все недостижимые символы;
- удалить λ-правила ;
- удалить цепные правила .

Следует подчеркнуть, что шаги преобразования должны выполняться именно в указанном порядке.

Удаление бесплодных символов

В грамматике G(**VT**,**VN**,**P**,S) символ $A \in \mathbf{VN}$ называется *бесплодным*, если для него выполняется $\{\alpha \mid A \Rightarrow^* \alpha,\ \alpha \in \mathbf{VT}^*\} = \varnothing$. То есть нетерминальный символ является бесплодным тогда, когда из него нельзя вывести ни одной цепочки терминальных символов.

В простейшем случае символ является бесплодным, если во всех правилах, где этот символ стоит в левой части, он также встречается и в правой части. Более сложные варианты предполагают зависимости между цепочками бесплодных символов, когда они в любой последовательности вывода порождают друг друга.

Алгоритм удаления бесплодных символов работает со специальным множеством нетерминальных символов $\mathbf{Y}_i$. Первоначально в это множество попадают только те символы, из которых непосредственно можно вывести терминальные цепочки, затем оно пополняется на основе правил грамматики **G**.

Алгоритм удаления бесплодных символов по шагам

Шаг 1. $\mathbf{Y}_0 = \varnothing$, i:=1.

Шаг 2. $\mathbf{Y}_i = \{A \mid (A \rightarrow \alpha) \in \mathbf{P},\ \alpha \in (\mathbf{Y}_{i-1} \cup \mathbf{VT})^*\} \cup \mathbf{Y}_{i-1}$.

Шаг 3. Если $\mathbf{Y}_i \neq \mathbf{Y}_{i-1}$, то i:=i+1 и перейти к шагу 2, иначе перейти к шагу 4.

Шаг 4. $\mathbf{VN}'=\mathbf{Y}_i$, $\mathbf{VT}'=\mathbf{VT}$, в $\mathbf{P}'$ входят те правила из **P**, которые содержат только символы из множества $(\mathbf{VT} \cup \mathbf{Y}_i)$, S'=S.

Удаление недостижимых символов

Символ x∈(**VT**∪**VN**) называется *недостижимым*, если он не встречается ни в одной сентенциальной форме грамматики G(**VT**,**VN**,**P**,S). То есть символ недостижимый, если он не участвует ни в одной цепочке вывода из целевого символа грамматики. Очевидно, что такой символ в грамматике не нужен.

Алгоритм удаления недостижимых символов строит множество достижимых символов грамматики G(**VT**,**VN**,**P**,S) — $\mathbf{V}_i$. Первоначально в это множество входит только целевой символ S, затем оно пополняется на основе правил грамматики. Все символы, которые не войдут в данное множество, являются недостижимыми и могут быть исключены в новой грамматике **G'** из словаря и из правил.

Алгоритм удаления недостижимых символов по шагам

Шаг 1. $\mathbf{V}_0 = \{S\}$, i:=1.

Шаг 2. $\mathbf{V}_i = \{x \mid x \in (\mathbf{VT} \cup \mathbf{VN})$ и $(A \to \alpha x \beta) \in \mathbf{P}$, $A \in \mathbf{V}_{i-1}$, $\alpha,\beta \in (\mathbf{VT} \cup \mathbf{VN})^*\} \cup \mathbf{V}_{i-1}$.

Шаг 3. Если $\mathbf{V}_i \neq \mathbf{V}_{i-1}$, то i:=i+1 и перейти к шагу 2, иначе перейти к шагу 4.

Шаг 4. $\mathbf{VN}' = \mathbf{VN} \cap \mathbf{V}_i$, $\mathbf{VT}' = \mathbf{VT} \cap \mathbf{V}_i$, в **P'** входят те правила из**P**, которые содержат только символы из множества $\mathbf{V}_i$, S'=S.

Пример удаления недостижимых и бесплодных символов

Рассмотрим работу алгоритмов удаления недостижимых и бесплодных символов на примере грамматики:

```
G({a,b,c},{A,B,C,D,E,F,G,S},P,S)
P:    S → aAB | E
        A → aA | bB
        B → ACb | b
        C → A | bA | cC | aE
        E → cE | aE | Eb | ED | FG
        D → a | c | Fb
        F → BC | EC | AC
        G → Ga | Gb
```

ВНИМАНИЕ

Для правильного выполнения преобразований необходимо сначала удалить бесплодные символы, а потом — недостижимые символы, но не наоборот.

Удалим бесплодные символы.

1. $\mathbf{Y}_0 = \varnothing$, i:=1 (шаг 1).
2. $\mathbf{Y}_1 = \{B,D\}$, $\mathbf{Y}_1 \neq \mathbf{Y}_0$: i:=2 (шаги 2 и 3).
3. $\mathbf{Y}_2 = \{B,D,A\}$, $\mathbf{Y}_2 \neq \mathbf{Y}_1$: i:=3 (шаги 2 и 3).
4. $\mathbf{Y}_3 = \{B,D,A,S,C\}$, $\mathbf{Y}_3 \neq \mathbf{Y}_2$: i:=4 (шаги 2 и 3).
5. $\mathbf{Y}_4 = \{B,D,A,S,C,F\}$, $\mathbf{Y}_4 \neq \mathbf{Y}_3$: i:=5 (шаги 2 и 3).
6. $\mathbf{Y}_5 = \{B,D,A,S,C,F\}$, $\mathbf{Y}_5 = \mathbf{Y}_4$ (шаги 2 и 3).
7. Строим множества **VN'** = {A,B,C,D,F,S}, **VT'** = {a,b,c} и **P'** (шаг 4).

Получили грамматику

```
G'({a,b,c},{A,B,C,D,F,S},P',S)
P':    S → aAB
        A → aA | bB
        B → ACb | b
        C → A | bA | cC
        D → a | c | Fb
        F → BC | AC
```

Удалим недостижимые символы.

1. $\mathbf{V}_0 = \{S\}$, i:=1 (шаг 1).
2. $\mathbf{V}_1 = \{S,A,B\}$, $\mathbf{V}_1 \neq \mathbf{V}_0$: i:=2 (шаги 2 и 3).
3. $\mathbf{V}_2 = \{S,A,B,C\}$, $\mathbf{V}_2 \neq \mathbf{V}_1$: i:=3 (шаги 2 и 3).
4. $\mathbf{V}_3 = \{S,A,B,C\}$, $\mathbf{V}_3 = \mathbf{V}_2$ (шаги 2 и 3).
5. Строим множества $\mathbf{VN}» = \{A,B,C,S\}$, $\mathbf{VT}» = \{a,b,c\}$ и $\mathbf{P}»$ (шаг 4).

В итоге получили грамматику

```
G"({a,b,c},{A,B,C,S},P",S)
P":    S → aAB
        A → aA | bB
        B → ACb | b
        C → A | bA | cC
```

Алгоритмы удаления бесплодных и недостижимых символов относятся к первой группе преобразований КС-грамматик. Они всегда ведут к упрощению грамматики, сокращению количества символов алфавита и правил грамматики.

Устранение λ-правил

λ-правилами называются все правила грамматики вида $A \to \lambda$, где $A \in \mathbf{VN}$.

КС-грамматика $G(\mathbf{VT},\mathbf{VN},\mathbf{P},S)$ называется грамматикой без λ-правил, если в ней не существует правил $(A \to \lambda) \in \mathbf{P}$, $A \neq S$, и существует только одно правило $(S \to \lambda) \in \mathbf{P}$, в том случае когда $\lambda \in L(\mathbf{G})$, и при этом S не встречается в правой части ни одного правила.

Для того чтобы упростить построение распознавателей языка $L(\mathbf{G})$, грамматику **G** часто целесообразно преобразовать к виду без λ-правил. Существует алгоритм преобразования произвольной КС-грамматики к виду без λ-правил. Он работает с некоторым множеством нетерминальных символов $\mathbf{W}_i$.

Алгоритм устранения λ-правил по шагам

Шаг 1. $\mathbf{W}_0 = \{A: (A \to \lambda) \in \mathbf{P}\}$, i:=1.

Шаг 2. $\mathbf{W}_i = \mathbf{W}_{i-1} \cup \{A: (A \to \alpha) \in \mathbf{P}, \alpha \in \mathbf{W}_{i-1}^*\}$.

Шаг 3. Если $\mathbf{W}_i \neq \mathbf{W}_{i-1}$, то i:=i+1 и перейти к шагу 2, иначе перейти к шагу 4.

Шаг 4. $\mathbf{VN}' = \mathbf{VN}$, $\mathbf{VT}' = \mathbf{VT}$, в $\mathbf{P}'$ входят все правила из **P**, кроме правил вида $A \to \lambda$.

Шаг 5. Если $(A \to \alpha) \in \mathbf{P}$ и в цепочку α входят символы из множества $\mathbf{W}_i$, тогда на основе цепочки α строится множество цепочек $\{\alpha'_1,..., \alpha'_k\}$, $k>0$, путем исключения из α всех возможных комбинаций символов $\mathbf{W}_i$, все правила вида $A \to \alpha'_i$, $k \geqslant i > 0$, добавляются в $\mathbf{P}'$ (при этом надо отбрасывать дубликаты правил и бессмысленные правила вида $A \to A$).

Шаг 6. Если $S \in \mathbf{W}_i$, то значит $\lambda \in L(\mathbf{G})$, и тогда в $\mathbf{VN}'$ добавляется новый символ S', который становится целевым символом грамматики $\mathbf{G}'$, а в $\mathbf{P}'$ добавляются два новых правила: $S' \to \lambda|S$; иначе S'=S (целевой символ грамматики не меняется).

Данный алгоритм часто ведет к увеличению количества правил грамматики, но позволяет упростить построение распознавателя для заданного языка.

Пример устранения λ-правил

Рассмотрим грамматику

```
G({a,b,c},{A,B,C,S},P,S)
```

```
P:    S → AaB | aB | cC
        A → AB | a | b | B
        B → Ba | λ
        C → AB | c
```

Удалим λ-правила.

1. $\mathbf{W}_0 = \{B\}$, i:=1 (шаг 1).
2. $\mathbf{W}_1 = \{B,A\}$, $\mathbf{W}_1 \neq \mathbf{W}_0$, i:=2 (шаги 2 и 3).
3. $\mathbf{W}_2 = \{B,A,C\}$, $\mathbf{W}_2 \neq \mathbf{W}_1$, i:=3 (шаги 2 и 3).
4. $\mathbf{W}_3 = \{B,A,C\}$, $\mathbf{W}_3 = \mathbf{W}_2$ (шаги 2 и 3).
5. Построим $\mathbf{VN}' = \{A,B,C,S\}$, $\mathbf{VT}' = \{a,b,c\}$ и множество правил $\mathbf{P}'$ (шаг 4):

```
P':    S → AaB | aB | cC
       A → AB | a | b | B
       B → Ba
       C → AB | c
```

6. Рассмотрим все правила из множества P' (шаг 5):
   - из правил S→AaB|aB|cC исключим все комбинации $\mathbf{W}_3$={B,A,C} и получим новые правила S→Aa|aB|a|a|c, добавим их в P', исключая дубликаты, получим S→AaB|aB|cC|Aa|aB|a|c;
   - из правил A→AB|a|b|B исключим все комбинации $\mathbf{W}_3$={B,A,C} и получим новые правила A→A|B, в P' их добавлять не надо, поскольку правило A→B там уже есть, а правило A→A бессмысленно;
   - из правила B→Ba исключим все комбинации $\mathbf{W}_3$={B,A,C} и получим новое правило B→a, добавим его в P', получим B→Ba|a;
   - из правил C→AB|c исключим все комбинации $\mathbf{W}_3$={B,A,C} и получим новые правила C→A|B, добавим их в P', получим C→AB|A|B|c.
7. $S \notin \mathbf{W}_3$, поэтому в грамматику G' не надо добавлять новый целевой символ: S'=S (шаг 6).

Получим грамматику

```
G'({a,b,c},{A,B,C,S},P',S)
P':    S → AaB | aB | cC | Aa | a | c
       A → AB | a | b | B
       B → Ba | a
       C → AB | A | B | c
```

Устранение цепных правил

Циклом (циклическим выводом) в грамматике G(**VT**,**VN**,**P**,S) называется вывод вида A⇒*A, A∈**VN**. Очевидно, что такой вывод абсолютно бесполезен. В распознавателях КС-языков целесообразно избегать возможности появления циклов.

Циклы возможны только в том случае, если в КС-грамматике присутствуют *цепные правила* вида A→B, A,B∈**VN**. Чтобы исключить возможность появления циклов в цепочках вывода, достаточно устранить цепные правила из правил грамматики.

Чтобы устранить цепные правила в КС-грамматике G(**VT**,**VN**,**P**,S), для каждого нетерминального символа X∈**VN** строится специальное множество цепных символов $\mathbf{N}^X$, а затем на основании построенных множеств выполняются преобразования правил **P**. Поэтому алгоритм устранения цепных правил надо выполнить для всех нетерминальных символов грамматики из множества **VN**.

Алгоритм устранения цепных правил по шагам

Шаг 1. Для всех символов X из **VN** повторять шаги 2—4, затем перейти к шагу 5.

Шаг 2. $\mathbf{N}^X_0 = \{X\}$, i:=1.

Шаг 3. $\mathbf{N}^X_i = \mathbf{N}^X_{i-1} \cup \{B: (A \to B) \in \mathbf{P}, B \in \mathbf{N}^X_{i-1}\}$.

Шаг 4. Если $\mathbf{N}^X_i \neq \mathbf{N}^X_{i-1}$, то i:=i+1 и перейти к шагу 3, иначе $\mathbf{N}^X = \mathbf{N}^X_i - \{X\}$ и продолжить цикл по шагу 1.

Шаг 5. **VN'** = **VN**, **VT'** = **VT**, в **P'** входят все правила из **P**, кроме правил вида A → B, S' = S.

Шаг 6. Для всех правил (A →α)∈**P'**, если B∈$\mathbf{N}^A$, B≠A, то в **P'** добавляются правила вида B→α.

Данный алгоритм, так же как и алгоритм устранения λ-правил, ведет к увеличению числа правил грамматики, но упрощает построение распознавателей.

Пример устранения цепных правил

Рассмотрим в качестве примера грамматику арифметических выраженийнад символами a и b, которая уже рассматривалась ранее в этом учебном пособии:

```
G({+,—,/,*,a,b},{S,T,E},P,S):
P:    S → S+T | S—T | T
        T → T*E | T/E | E
        E → (S) | a | b
```

Устраним цепные правила.

1. $\mathbf{N}^S_0 = \{S\}$, i:=1.
2. $\mathbf{N}^S_1 = \{S,T\}$, $\mathbf{N}^S_1 \neq \mathbf{N}^S_0$, i:=2.
3. $\mathbf{N}^S_2 = \{S,T,E\}$, $\mathbf{N}^S_2 \neq \mathbf{N}^S_1$, i:=3.
4. $\mathbf{N}^S_3 = \{S,T,E\}$, $\mathbf{N}^S_3 = \mathbf{N}^S_2$, $\mathbf{N}^S = \{T,E\}$.
5. $\mathbf{N}^T_0 = \{T\}$, i:=1.
6. $\mathbf{N}^T_1 = \{T,E\}$, $\mathbf{N}^T_1 \neq \mathbf{N}^T_0$: i:=2.
7. $\mathbf{N}^T_2 = \{T,E\}$, $\mathbf{N}^T_2 = \mathbf{N}^T_1$, $\mathbf{N}^T = \{E\}$.
8. $\mathbf{N}^E_0 = \{E\}$, i:=1.
9. $\mathbf{N}^E_1 = \{E\}$, $\mathbf{N}^E_1 = \mathbf{N}^E_0$, $\mathbf{N}^E = \varnothing$.
10. Получили $\mathbf{N}^S = \{T,E\}$, $\mathbf{N}^T = \{E\}$, $\mathbf{N}^E = \varnothing$, S'=S, построим множества **VN'** = {S,T,E}, **VT'** = {+,-,/,*,a,b} и множество правил **P'**,
11. Рассмотрим все правила из множества **P'** — интерес представляют только правила для символов T и E, так как $\mathbf{N}^S = \{T,E\}$ и $\mathbf{N}^T = \{E\}$:
    - для правил `T→T*E|T/E` имеем новые правила `S→T*E|T/E`, поскольку $T \in \mathbf{N}^S$;
    - для правил `E→(S)|a|b` имеем новые правила `S→(S)|a|b` и `T→(S)|a|b`, поскольку $E \in \mathbf{N}^S$ и $E \in \mathbf{N}^T$.

Получим новую грамматику

```
G'({+,–,/,*,a,b},{S,T,E},P',S)
P': S   → S+T | S–T | T*E | T/E | (S) | a | b
       T → T*E | T/E | (S) | a  | b
       E → (S) |  a  | b
```

Эта грамматика далее будет использована для построения распознавателей КС-языков.

Устранение левой рекурсии

Определение левой рекурсии

Символ $A \in \mathbf{VN}$ в КС-грамматике G(**VT**,**VN**,P,S) называется рекурсивным, если для него существует цепочка вывода вида $A \Rightarrow^+ \alpha A \beta$, где $\alpha,\beta \in (\mathbf{VT} \cup \mathbf{VN})^*$.

Если $\alpha=\lambda$ и $\beta \neq \lambda$, то рекурсия называется левой, а грамматика **G** — *леворекурсивной*; если $\alpha \neq \lambda$ и $\beta=\lambda$, то рекурсия называется правой, а грамматика **G** — *праворекурсивной*. Если $\alpha=\lambda$ и $\beta=\lambda$, то рекурсия представляет собой цикл . Алгоритм исключения циклов был рассмотрен выше, поэтому далее циклы не рассматриваются.

Любая КС-грамматика может быть как леворекурсивной , так и праворекурсивной , а также леворекурсивной и праворекурсивной одновременно.

КС-грамматика называется *нелеворекурсивной*, если она не является леворекурсивной. Аналогично, КС-грамматика является *неправорекурсивной*, если не является праворекурсивной.

Некоторые алгоритмы левостороннего разбора для КС-языков не работают с леворекурсивными грамматиками, поэтому возникает необходимость исключить левую рекурсию из выводов грамматики. Далее будет рассмотрен алгоритм, который

позволяет преобразовать правила произвольной КС-грамматики таким образом, чтобы в выводах не встречалась левая рекурсия.

Следует отметить, что, поскольку рекурсия лежит в основе построения языков на основе правил грамматики в форме Бэкуса—Наура, полностью исключить рекурсию из выводов грамматики невозможно. Можно избавиться только от одного вида рекурсии — левого или правого, то есть преобразовать исходную грамматику **G** к одному из видов: нелеворекурсивному (избавиться от левой рекурсии) или неправорекурсивному (избавиться от правой рекурсии). Для левосторонних распознавателей интерес представляет избавление от левой рекурсии — то есть преобразование грамматики к нелеворекурсивному виду.

Причина устранения именно левой рекурсии кроется в том, что все существующие в реальных компиляторах распознаватели читают входной текст слева направо.

Доказано, что любую КС-грамматику можно преобразовать к нелеворекурсивному или неправорекурсивному виду [4 т.1, 5, 58].

Алгоритм устранения левой рекурсии

Условие: дана КС-грамматика G(**VT**,**VN**,**P**,S), необходимо построить эквивалентную ей нелеворекурсивную грамматику G'(**VN**',**VT**,**P**',S'): L(**G**)=L(**G**').

Алгоритм преобразования работает с множеством правил исходной грамматики **P**, множеством нетерминальных символов **VN** и двумя переменными счетчиками: i и j.

Шаг 1. Обозначим нетерминальные символы грамматики так: **VN**=$\{A_1,A_2,...,A_n\}$, i:=1.

Шаг 2. Рассмотрим правила для символа A_i. Если эти правила не содержат левой рекурсии, то перенесем их во множество правил **P**' без изменений, а символ A_i добавим во множество нетерминальных символов **VN**'.

Иначе запишем правила для A_i в виде $A_i \to A_i\alpha_1|A_i\alpha_2|...|A_i\alpha_m|\beta_1|\beta_2|...|\beta_p$, где $\forall p \geq j \geq 1$, ни одна из цепочек β_j не начинается с символов A_k, таких, что $k \leq i$.

Вместо этого правила во множество **P**' запишем два правила вида

$$A_i \to \beta_1|\beta_2|...|\beta_p|\beta_1A_i'|\beta_2A_i'|...|\beta_pA_i';$$
$$A_i' \to \alpha_1|\alpha_2|...|\alpha_m|\alpha_1A_i'|\alpha_2A_i'|...|\alpha_mA_i'.$$

Символы A_i и A_i' включаем во множество **VN**'.

Теперь все правила для A_i начинаются либо с терминального символа, либо с нетерминального символа A_k, такого, что k>i.

Шаг 3. Если i =n, то грамматика **G**' построена, перейти к шагу 6, иначе i: =i+1, j:=1 и перейти к шагу 4.

Шаг 4. Для символа A_j во множестве правил **P**' заменить все правила вида $A_i \to A_j\alpha$, где $\alpha \in (\mathbf{VT} \cup \mathbf{VN})^*$, на правила вида $A_i \to \beta_1\alpha|\beta_2\alpha|...|\beta_m\alpha$, причем $A_j \to \beta_1|\beta_2|...|\beta_m$ — все правила для символа A_j.

Так как правая часть правил $A_j \to \beta_1|\beta_2|...|\beta_m$ уже начинается с терминального символа или нетерминального символа A_k, k>j, то теперь и правая часть правил для символа A_i будет удовлетворять этому условию.

Шаг 5. Если j=i–1, то перейти к шагу 2, иначе j:=j+1 и перейти к шагу 4.

Шаг 6. Целевым символом грамматики **G'** становится символ A_k, соответствующий символу S исходной грамматики **G**.

Рассмотрим в качестве примера грамматику для арифметических выражений над символами a и b:

```
G({+,—,/,*,a,b},{S,T,E},P,S):
P:    S → S+T | S—T | T
        T → T*E | T/E | E
        E → (S) | a | b
```

Эта грамматика является леворекурсивной. Построим эквивалентную ей нелеворекурсивную грамматику **G'**.

Шаг 1. Обозначим **VN**={A_1,A_2,A_3}. i:=1.

Тогда правила грамматики **G** будут иметь вид

$A_1 \to A_1+A_2 \mid A_1—A_2 \mid A_2$

$A_2 \to A_2*A_3 \mid A_2/A_3 \mid A_3$

$A_3 \to (A_1) \mid a \mid b$

Шаг 2. Для A_1 имеем правила $A_1 \to A_1+A_2 | A_1—A_2 | A_2$. Их можно записать в виде $A_1 \to A_1\alpha_1 | A_1\alpha_2 | \beta_1$, где $\alpha_1=+A_2$, $\alpha_2=—A_2$, $\beta_1=A_2$.

Запишем новые правила для множества **P'**:

$A_1 \to A_2 \mid A_2A_1'$

$A_1' \to +A_2 \mid —A_2 \mid +A_2A_1' \mid —A_2A_1'$

Добавив эти правила в **P'**, а символы A_1 и A_1' во множество нетерминальных символов, получим: **VN'**={A_1,A_1'}.

Шаг 3. i=1 < 3. Построение не закончено: i:=i+1 = 2, j:=1.

Шаг 4. Для символа A_2 во множестве правил **P'** нет правила вида $A_2 \to A_1\alpha$, поэтому на этом шаге никаких действий не выполняем.

Шаг 5. j = 1 = i – 1, переходим опять к шагу 2.

Шаг 2. Для A_2 имеем правила $A_2 \to A_2*A_3 | A_2/A_3 | A_3$. Их можно записать в виде $A_2 \to A_2\alpha_1 | A_2\alpha_2 | \beta_1$, где $\alpha_1=*A_3$, $\alpha_2=/A_3$, $\beta_1=A_3$.

Запишем новые правила для множества **P'**:

$A_2 \to A_3 \mid A_3A_2'$

$A_2' \to *A_3 \mid /A_3 \mid *A_3A_2' \mid /A_3A_2'$

Добавим эти правила в **P'**, а символы A_2 и A_2' во множество нетерминальных символов, получим **VN'**={A_1,A_1',A_2,A_2'}.

Шаг 3. i=2 < 3. Построение не закончено: i:=i+1 = 3, j:=1.

Шаг 4. Для символа A_3 во множестве правил **P'** нет правила вида $A_3 \to A_1\alpha$, поэтому на этом шаге никаких действий не выполняем.

Шаг 5. j=1 < i—1, j:=j+1 = 2, переходим к шагу 4.

Шаг 4. Для символа A_3 во множестве правил **P'** нет правила вида $A_3 \to A_2\alpha$, поэтому на этом шаге никаких действий не выполняем.

Шаг 5. j=2 = i—1, переходим опять к шагу 2.

Шаг 2. Для A_3 имеем правила $A_3 \rightarrow (A_1)|a|b$. Эти правила не содержат левой рекурсии. Переносим их в **P'**, а символ A_3 добавляем в **VN'**. Получим **VN'**=$\{A_1,A_1',A_2,A_2',A_3\}$.

Шаг 3. i=3 = 3. Построение грамматики **G'** закончено.

В результате выполнения алгоритма преобразования получили нелеворекурсивную грамматику G({+,—,/,*,a,b}, $\{A_1,A_1',A_2,A_2',A_3\}$, **P'**, A_1) с правилами:

```
P':     A1 → A2 | A2A1'
        A1' → +A2 | —A2 | +A2A1' | —A2A1'
        A2 → A3 | A3A2'
        A2' → *A3 | /A3 | *A3A2' | /A3A2'
        A3 → (A1) | a | b
```

Синтаксические распознаватели с возвратом

Принципы работы распознавателей с возвратом

Распознаватели с возвратом — это самый примитивный тип распознавателей для КС-языков. Логика их работы основана на моделировании функционирования недетерминированного МП-автомата .

Поскольку моделируется недетерминированный МП-автомат, то на некотором шаге работы моделирующего алгоритма существует возможность возникновения нескольких возможных следующих состояний автомата. В таком случае есть два варианта реализации алгоритма [15].

В первом варианте на каждом шаге работы алгоритм должен запоминать все возможные следующие состояния МП-автомата, выбирать одно из них, переходить в это состояние и действовать так до тех пор, пока либо не будет достигнуто конечное состояние автомата, либо автомат не перейдет в такую конфигурацию, когда следующее состояние будет не определено. Если достигнуто одно из конечных состояний — входная цепочка принята, работа алгоритма завершается. В противном случае алгоритм должен вернуть автомат на несколько шагов назад, когда еще был возможен выбор одного из набора следующих состояний, выбрать другой вариант и промоделировать поведение автомата с этим условием. Алгоритм завершается с ошибкой, когда все возможные варианты работы автомата перебраны и при этом не было достигнуто ни одно из возможных конечных состояний.

Во втором варианте алгоритм моделирования МП-автомата должен на каждом шаге работы при возникновении неоднозначности с несколькими возможными следующими состояниями автомата запускать новую свою копию для обработки каждого из этих состояний. Алгоритм завершается, если хотя бы одна из выполняющихся его копий достигнет одного из конечных состояний. При этом работа всех остальных копий алгоритма прекращается. Если ни одна из копий алгоритма не достигла конечного состояния МП-автомата, то алгоритм завершается с ошибкой.

Второй вариант реализации алгоритма связан с управлением параллельными процессами в вычислительных системах, поэтому сложен в реализации. Кроме того, на каждом шаге работы МП-автомата альтернатив следующих состояний может быть

много, а количество возможных параллельно выполняющихся процессов в операционных системах ограничено, поэтому применение второго варианта алгоритма осложнено. По этим причинам большее распространение получил первый вариант алгоритма, который предусматривает возврат к ранее запомненным состояниям МП-автомата — отсюда и название «разбор с возвратами».

Следует отметить, что, хотя МП-автомат является односторонним распознавателем, алгоритм моделирования его работы предусматривает возврат назад, к уже прочитанной части цепочки символов, чтобы исключить недетерминизм в поведении автомата (который иначе невозможно промоделировать).

Есть еще одна особенность в моделировании МП-автомата: любой практически ценный алгоритм должен завершаться за конечное число шагов (успешно или неуспешно). Алгоритм моделирования работы произвольного МП-автомата в общем случае не удовлетворяет этому условию.

Чтобы избежать таких ситуаций, алгоритмы разбора с возвратами строят не для произвольных МП-автоматов, а для МП-автоматов, удовлетворяющих некоторым заданным условиям. Как правило, эти условия связаны с тем, что МП-автомат должен строиться на основе грамматики заданного языка только после того, как она подвергнется некоторым преобразованиям. Поскольку преобразования грамматик сами по себе не накладывают каких-либо ограничений на входной класс КС-языков, то они и не ограничивают применимости алгоритмов разбора с возвратами. Следовательно, эти алгоритмы применимы для любого КС-языка, заданного произвольной КС-грамматикой.

Алгоритмы разбора с возвратами обладают экспоненциальными характеристиками. Это значит, что вычислительные затраты алгоритмов экспоненциально зависят от длины входной цепочки символов: α, $\alpha \in \mathbf{VT}^*$, $n=|\alpha|$. Конкретная зависимость определяется вариантом реализации алгоритма [4 т.1, 5].

Доказано, что в общем случае при первом варианте реализации для произвольной КС-грамматики G(**VT**,**VN**,**P**,S) время выполнения данного алгоритма $T_э$ будет иметь экспоненциальную зависимость от длины входной цепочки, а необходимый объем памяти $M_э$ — линейную зависимость от длины входной цепочки: $T_э=O(e^n)$ и $M_э=O(n)$. При втором варианте реализации, наоборот, время выполнения данного алгоритма $T_э$ будет иметь линейную зависимость от длины входной цепочки, а необходимый объем памяти $M_э$ — экспоненциальную зависимость от длины входной цепочки: $T_э=O(n)$ и $M_э=O(e^n)$.

Экспоненциальная зависимость вычислительных затрат от длины входной цепочки существенно ограничивает применимость алгоритмов разбора с возвратами. Они тривиальны в реализации, но имеют неудовлетворительные характеристики, поэтому могут использоваться только для КС-языков с малой длиной входных предложений языка[1]. Для многих классов КС-языков существуют более эффективные алгоритмы распознавания, поэтому алгоритмы разбора с возвратами применяются редко.

Далее рассмотрены два основных варианта таких алгоритмов.

[1] Возможность использовать эти алгоритмы в реальных компиляторах весьма сомнительна, поскольку длина входной цепочки может достигать нескольких тысяч и даже десятков тысяч символов. Очевидно, что время работы алгоритма будет в таком варианте явно неприемлемым даже на самых современных компьютерах.

Нисходящий распознаватель с подбором альтернатив

Нисходящий распознаватель с подбором аль тернатив моделирует работу МП-автомата с одним состоянием q: R({q},**V**,**Z**,δ,q,S,{q}). Автомат распознает цепочки КС-языка, заданного КС-грамматикой G(**VT**,**VN**,**P**,S). Входной алфавит автомата содержит терминальные символы грамматики: **V**=**VT**, а алфавит магазинных символов строится из терминальных и нетерминальных символов грамматики: **Z**=**VT**∪**VN**.

Начальная конфигурация автомата определяется так: (q, α,S) — автомат пребывает в своем единственном состоянии q, считывающая головка находится в начале входной цепочки символов α∈ **VT***, в стеке лежит символ, соответствующий целевому символу грамматики S.

Конечная конфигурация автомата определяется так: (q, λ,λ) — автомат пребывает в своем единственном состоянии q,считывающая головка находится за концом входной цепочки символов, стек пуст.

Функция переходов МП-автомата строится на основе правил грамматики:

1) (q,α)∈ δ(q,λ,A), A∈ **VN**, α∈ (**VT**∪**VN**)*, если правило A→α содержится во множестве правил **P** грамматики **G**: A→α∈ **P**;
2) (q,λ)∈ δ(q,a,a) ∀a∈ **VT**.

Работу данного МП-автомата можно неформально описать следующим образом: если на верхушке стека автомата находится нетерминальный символ A, то его можно заменить на цепочку символов α, если в грамматике языка есть правило A →α, не сдвигая при этом считывающую головку автомата (этот шаг работы называется «подбор альтернативы»); если же на верхушке стека находится терминальный символ *a*, который совпадает с текущим символом входной цепочки, то этот символ можно выбросить из стека и передвинуть считывающую головку на одну позицию вправо (этот шаг работы называется «выброс»). Данный МП-автомат может быть недетерминированным, поскольку при подборе альтернативы в грамматике языка может оказаться более одного правила вида A →α, тогда функция δ(q,λ,A) будет содержать более одного следующего состояния — у МП-автомата будет несколько альтернатив.

Решение о том, выполнять ли на каждом шаге работы МП-автомата выброс или подбор альтернативы, принимается однозначно. Моделирующий алгоритм должен обеспечивать выбор одной из возможных аль тернатив и хранение информации о том, какие альтернативы на каком шаге уже были выбраны, чтобы иметь возможность вернуться к этому шагу и подобрать другие аль тернативы. Такой алгоритм разбора называется алгоритмом с подбором альтернатив.

Данный МП-автомат строит левосторонние выводы для грамматики G(**VT**,**VN**,**P**,S). Для моделирования такого автомата необходимо, чтобы грамматика G(**VT**,**VN**,**P**,S) не была леворекурсивной — в противном случае, очевидно, алгоритм может войти в бесконечный цикл. Поскольку , как было показано выше, произвольную КС-грамматику всегда можно преобразовать к нелеворекурсивному виду, этот алгоритм применим для любой КС-грамматики. Следовательно, им можно распознавать цепочки любого КС-языка.

Сам по себе алгоритм разбора с подбором аль тернатив, использующий возвраты, не находит применения в реальных компиляторах. Однако его основные принципы

лежат в основе многих нисходящих распознавателей, строящих левосторонние выводы и работающих без использования возвратов. Методы, позволяющие строить такие распознаватели для некоторых классов КС-языков, рассмотрены далее.

Восходящий распознаватель на основе алгоритма «сдвиг—свертка»

Восходящий распознаватель по алгоритму «сдвиг—свертка» строится на основе расширенного МП-автомата с одним состоянием q: R({q}, **V**,**Z**,δ,q,S,{q}). Автомат распознает цепочки КС-языка, заданного КС-грамматикой G(**VT**,**VN**,**P**,S). Входной алфавит автомата содержит терминальные символы грамматики: **V**=**VT**; а алфавит магазинных символов строится из терминальных и нетерминальных символов грамматики **Z**=**VT**∪**VN**.

Начальная конфигурация автомата определяется так: (q, α,λ) — автомат пребывает в своем единственном состоянии q, считывающая головка находится в начале входной цепочки символов α∈ **VT***, стек пуст.

Конечная конфигурация автомата определяется так: (q, λ,S) — автомат пребывает в своем единственном состоянии q,считывающая головка находится за концом входной цепочки символов, в стеке лежит символ, соответствующий целевому символу грамматики S.

Функция переходов МП-автомата строится на основе правил грамматики:

1) (q,A)∈ δ(q,λ,γ), A∈ **VN**, γ∈ (**VT**∪**VN**)*, если правило A →γ содержится во множестве правил **P** грамматики **G**: A→γ∈ **P**;
2) (q,A)∈ δ(q,a,λ) ∀a∈ **VT**.

Неформально работу этого расширенного автомата можно описать так: если на верхушке стека находится цепочка символов γ, то ее можно заменить на нетерминальный символ A, если в грамматике языка существует правило вида A →γ, не сдвигая при этом считывающую головку автомата (этот шаг работы называется «свертка»); с другой стороны, если считывающая головка автомата обозревает некоторый символ входной цепочки a, то его можно поместить в стек, сдвинув при этом головку на одну позицию вправо (этот шаг работы называется «сдвиг» или «перенос»). Алгоритм, моделирующий работу такого расширенного МП-автомата, называется алгоритмом «сдвиг—свертка» или «перенос—свертка».

Данный расширенный МП-автомат строит правосторонние выводы для грамматики G(**VT**,**VN**,**P**,S). Для моделирования такого автомата необходимо, чтобы грамматика G(**VT**,**VN**,**P**,S) не содержала λ-правил и цепных правил — в противном случае алгоритм может войти в бесконечный цикл из сверток. Поскольку , как было доказано выше, произвольную КС-грамматику всегда можно преобразовать к виду без λ-правил и цепных правил, то этот алгоритм применим для любой КС-грамматики. Следовательно, им можно распознавать цепочки любого КС-языка.

Данный расширенный МП-автомат потенциально имеет больше неоднозначностей, чем рассмотренный выше МП-автомат, основанный на алгоритме подбора альтернатив. На каждом шаге работы автомата надо решать следующие вопросы:

1) что необходимо выполнять: сдвиг или свертку;

2) если выполнять свертку, то какую цепочку γ выбрать для поиска правил (цепочка γ должна встречаться в правой части правил грамматики);
3) какое правило выбрать для свертки, если окажется, что существует несколько правил вида A→γ (несколько правил с одинаковой правой частью).

Чтобы промоделировать работу этого расширенного МП-автомата, надо на каждом шаге запоминать все предпринятые действия, чтобы иметь возможность вернуться к уже сделанному шагу и выполнить эти же действия по-другому . Этот процесс должен повторяться до тех пор, пока не будут перебраны все возможные варианты.

Сам по себе алгоритм «сдвиг—свертка» с возвратами не находит применения в реальных компиляторах. Однако его базовые принципы лежат в основе многих восходящих распознавателей, строящих правосторонние выводы и работающих без использования возвратов. Методы, позволяющие строить такие распознаватели для некоторых классов КС-языков, рассмотрены далее.

В принципе, два рассмотренных алгоритма — нисходящего и восходящего разбора с возвратами — имеют схожие характеристики по требуемым вычислительным ресурсам и одинаково просты в реализации. Т о, какой из них лучше взять для реализации простейшего распознавателя в том или ином случае, зависит прежде всего от грамматики языка.

Нисходящие распознаватели КС-языков без возвратов

Стремление улучшить алгоритм с подбором аль тернатив для нисходящего разбора заключается, в первую очередь, в определении метода, по которому на каждом шаге алгоритма можно было бы однозначно выбрать одну из всего множества возможных альтернатив. В таком случае алгоритм не требовал бы возврата на предыдущие шаги и за счет этого обладал бы линейными характеристиками. В случае неуспеха выполнения алгоритма входная цепочка не принимается, повторные итерации разбора не выполняются.

Левосторонний разбор по методу рекурсивного спуска

Наиболее очевидным методом выбора одной из множества альтернатив является выбор ее на основании символа a∈ **VT**, обозреваемого считывающей головкой автомата на каждом шаге его работы. Поскольку в процессе нисходящего разбора именно этот символ должен появиться на верхушке магазина для продвижения считывающей головки автомата на один шаг (условие $\delta(q,a,a)=\{(q,\lambda)\}$, ∀a∈ **VT** в функции переходов МП-автомата), разумно искать альтернативу, где он присутствует в начале цепочки, стоящей в правой части правила грамматики.

По такому принципу действует алгоритм разбора по методу рекурсивного спуска.

Алгоритм разбора по методу рекурсивного спуска

В реализации этого алгоритма для каждого нетерминального символа A∈ **VN** грамматики G(**VN**,**VT**,**P**,S) строится процедура разбора, которая получает на вход цепоч-

ку символов α и положение считывающей головки в цепочке i. Если для символа A в грамматике **G** определено более одного правила, то процедура разбора ищет среди них правило вида $A \to a\gamma$, $a \in \mathbf{VT}$, $\gamma \in (\mathbf{VN} \cup \mathbf{VT})^*$, первый символ правой части которого совпадал бы с текущим символом входной цепочки $a = \alpha_i$. Если такого правила не найдено, то цепочка не принимается.

Название алгоритма происходит из его реализации, которая заключается в последовательности рекурсивных вызовов процедур разбора. Для начала разбора входной цепочки нужно вызвать процедуру для символа S с параметром i=1.

Условия применимости метода можно получить из описания алгоритма — в грамматике $G(\mathbf{VN},\mathbf{VT},\mathbf{P},S)$ $\forall A \in \mathbf{VN}$ возможны только два варианта правил:

$A \to \gamma$, $\gamma \in (\mathbf{VN} \cup \mathbf{VT})^*$, и это единственное правило для A;

$A \to a_1\beta_1 | a_2\beta_2 | ... | a_n\beta_n$, $\forall i$: $a_i \in \mathbf{VT}$, $\beta_i \in (\mathbf{VN} \cup \mathbf{VT})^*$, и если $i \neq j$, то $a_i \neq a_j$.

Этим условиям удовлетворяет незначительное количество реальных грамматик.

Это достаточные, но не необходимые условия. Если грамматика не удовлетворяет этим условиям, еще не значит, что заданный ею язык не может распознаваться с помощью метода рекурсивного спуска. Возможно, над грамматикой просто необходимо выполнить ряд дополнительных преобразований.

К сожалению, не существует алгоритма, который позволил бы преобразовать произвольную КС-грамматику к указанному выше виду, равно как не существует и алгоритма, который позволил бы проверить, возможны ли такого рода преобразования. То есть для произвольной КС-грамматики нельзя сказать, анализируем ли заданный ею язык методом рекурсивного спуска или нет.

СОВЕТ

Можно рекомендовать ряд преобразований, которые способствуют приведению грамматики к требуемому виду (но не гарантируют его достижения) [4,15].

В целом алгоритм рекурсивного спуска эффективен и прост в реализации, но имеет очень ограниченную применимость.

Пример реализации метода рекурсивного спуска

Дана грамматика `G({a,b,c},{A,B,C,S},P,S)`:

```
P:      S → aA | bB
        A → a | bA | cC
        B → b | aB | cC
        C → AaBb
```

Необходимо построить распознаватель, работающий по методу рекурсивного спуска. Видно, что грамматика удовлетворяет условиям, необходимым для построения такого распознавателя.

Напишем процедуры на языке программирования C, которые будут обеспечивать разбор входных цепочек языка, заданного данной грамматикой. Согласно алгоритму необходимо построить процедуру разбора для каждого нетерминального символа грамматики, поэтому дадим процедурам соответствующие наименования. Входные данные для процедур разбора будут следующие:

- цепочка входных символов;
- положение указателя (считывающей головки МП-автомата) во входной цепочке;
- массив для записи номеров примененных правил;
- порядковый номер очередного правила в массиве.

Результатом работы каждой процедуры может быть число, отличное от нуля («истина»), или 0 («ложь»). В первом случае входная цепочка символов принимается распознавателем, во втором случае — не принимается. Для удобства реализации в том случае, если цепочка принимается распознавателем, будем возвращать текущее положение указателя в цепочке. Кроме того, потребуется еще одна дополнительная процедура для ведения записей в массиве последовательности правил (назовем ее *WriteRules*).

```
void WriteRules(int* piRul, int* iP, int iRule)
{
    piRul[*iP] = iRule;
    *iP = *iP + 1;
}

int proc_S (char* szS, int iN, int* piRul, int* iP)
{
    switch (szS[iN])
    {
        case 'a':
            WriteRules(piRul,iP,1);
            return proc_A(szS,iN+1,piRul,iP);
        case 'b':
            WriteRules(piRul,iP,2);
            return proc_B(szS,iN+1,piRul,iP);
    }
    return 0;
}

int proc_A (char* szS, int iN, int* piRul, int* iP)
{
    switch (szS[iN])
    {
        case 'a':
            WriteRules(piRul,iP,3);
            return iN+1;
        case 'b':
            WriteRules(piRul,iP,4);
            return proc_A(szS,iN+1,piRul,iP);
```

```
        case 'c':
            WriteRules(piRul,iP,5);
            return proc_C(szS,iN+1,piRul,iP);
    }
    return 0;
}

int proc_B (char* szS, int iN, int* piRul, int* iP)
{
    switch (szS[iN])
    {
        case 'b':
            WriteRules(piRul,iP,6);
            return iN+1;
        case 'a':
            WriteRules(piRul,iP,7);
            return proc_B(szS,iN+1,piRul,iP);
        case 'c':
            WriteRules(piRul,iP,8);
            return proc_B(szS,iN+1,piRul,iP);
    }
    return 0;
}

int proc_C (char* szS, int iN, int* piRul, int* iP)
{ int i;
    WriteRules(piRul,iP,9);
    i = proc_A(szS,iN,piRul,iP);
    if (i == 0) return 0;
    if (szS[i] != 'a') return 0;
    i++;
    i = proc_B(szS,i,piRul,iP);
    if (i == 0) return 0;
    if (szS[i] != 'b') return 0;
    return i+1;
}
```

Теперь для распознавания входной цепочки необходимо иметь целочисленный массив `Rules` достаточного объема для хранения номеров правил. Т огда работа распознавателя заключается в вызове процедуры `proc_S(Str,0,Rules,&N)`, где `Str` — это входная цепочка символов, `N` — переменная для запоминания количества примененных правил (первоначально `N=0`). Затем требуется обработка полученного результата: если результат на 1 превышает длину цепочки — цепочка принята, иначе — це-

почка не принята. В первом случае в массиве `Rules` будем иметь последовательность номеров правил грамматики, необходимых для вывода цепочки, а в переменной `N` — количество этих правил.

Объем массива `Rules` заранее не известен, так как заранее не известно количество шагов вывода. Чтобы избежать проблем с недостаточным объемом статического массива, приведенные выше процедуры распознавателя можно модифицировать так, чтобы они работали с динамическим распределением памяти. На логику работы распознавателя это никак не повлияет[1].

Из приведенного примера видно, что алгоритм рекурсивного спуска удобен и прост в реализации. Главным препятствием в его применении является то, что класс грамматик, допускающих разбор на основе этого алгоритма, сильно ограничен.

Расширенные варианты метода рекурсивного спуска

Метод рекурсивного спуска позволяет выбрать альтернативу, ориентируясь на текущий символ входной цепочки, обозреваемый считывающей головкой МП-автомата. Если имеется возможность просматривать не один, а несколько символов вперед от текущего положения считывающей головки, то можно расширить область применимости метода рекурсивного спуска. В этом случае уже можно искать правила на основе некоторого терминального символа, входящего в правую часть правила. Естественно, и в таком варианте выбор должен быть однозначным — для каждого нетерминального символа в левой части правила необходимо, чтобы в правой части правила не встречалось двух одинаковых терминальных символов.

Этот метод требует также анализа типа присутствующей в правилах рекурсии, поскольку один и тот же терминальный символ может встречаться во входной строке несколько раз, и в зависимости от типа рекурсии следует искать его крайнее левое или крайнее правое вхождение в строке.

Рассмотрим грамматику арифметических выражений для символов `a` и `b`:

```
G({+,-,/,*,a,b},{S,T,E},P,S):
P:      S → S+T | S-T | T
        T → T*E | T/E | E
        E → (S) | a | b
```

Это грамматика для арифметических выражений, которая уже была рассмотрена в разделе «Проблемы однозначности и эквивалентности грамматик» главы 1.

Запишем правила этой грамматики в форме с применением метасимволов . Получим

```
P:      S → T{(+T,-T)}
        T → E{(*E,/E)}
        E → (S) | a | b
```

[1] Следует помнить также, что метод рекурсивного спуска основан на рекурсивном вызове множества процедур, что при значительной длине входной цепочки символов может потребовать соответствующего объема стека вызовов для хранения адресов процедур, их параметров и локальных переменных. Более подробно этот вопрос рассмотрен в разделе «Распределение памяти» главы 5 данного учебника.

При такой форме записи процедура разбора для каждого нетерминального символа становится тривиальной.

Для символа `S` распознаваемая строка должна всегда начинаться со строки, допустимой для символа `T`, за которой может следовать любое количество символов `+` или `–`, и если они найдены, то за ними опять должна быть строка, допустимая для символа `T`. Аналогично, для символа `T` распознаваемая строка должна всегда начинаться со строки, допустимой для символа `E`, за которой может следовать любое количество символов `*` или `/`, и если они найдены, то за ними опять должна быть строка, допустимая для символа `E`. С другой стороны, для символа `E` строка должна начинаться строго с символов `(`, `a` или `b`, причем в первом случае за символом `(` должна следовать строка, допустимая для символа `S`, а за ней — обязательно символ `)`.

Исходя из этого построены процедуры разбора входной строки на языке Pascal (используется Borland Pascal или Object Pascal, которые допускают тип `string` — строка). Входными данными для них являются:

- исходная строка символов;
- текущее положение указателя в исходной строке;
- длина исходной строки (в принципе, этот параметр можно опустить, но он введен для удобства);
- результирующая строка правил.

Процедуры построены так, что в результирующую строку правил помещаются номера примененных правил в строковом формате, перечисленные через запятую (`,`). Правила нумеруются в грамматике, записанной в форме Бэкуса—Наура, в порядке слева направо и сверху вниз (всего в исходной грамматике 9 правил). Распознаватель строит левосторонний вывод, поэтому на основе строки номеров правил всегда можно получить цепочку вывода или дерево вывода.

Для начала разбора нужно вызвать процедуру `proc_S(S,1,N,Pr)`, где

- `S` — входная строка символов;
- `N` — длина входной строки (в языке Borland Pascal вместо `N` можно взять `Length(S)`);
- `Pr` — строка, куда будет помещена последовательность примененных правил.

Результатом выполнения `proc_S(S,1,N,Pr)` будет `N+1`, если строка `S` принимается, и некоторое число, меньшее `N+1`, если строка не принимается. Если строка `S` принимается, то строка `Pr` будет содержать последовательность номеров правил, которые необходимо применить для того, чтобы вывести `S`[1].

[1] Использование языка программирования Borland Pascal накладывает определенные технические ограничения на данный распознаватель — длина строки в этом языке не может превышать 255 символов. Однако данные ограничения можно снять, если реализовать свой тип данных `строка`. При использовании Borland Delphi эти ограничения отпадают. Конечно, такого рода ограничения не имеют принципиального значения при теоретическом исследовании работы распознавателя, тем не менее автор считает необходимым упомянуть о них.

```
procedure proc_S (S: string; i,n: integer; var pr: string): integer;
var s1 : string;
begin
    i := proc_T(S,i,n,s1);
    if i > 0 then
    begin
        pr := '3,' + s1;
        while (i <= n) and (i <> 0) do
        case S[i] of
            '+': begin
                if i = n then i := 0
                else
                begin
                    i := proc_T(S,i+1,n,s1);
                    pr := '1,' + pr +','+ s1;
                end;
            end;
            '-': begin
                if i = n then i := 0
                else
                begin
                    i := proc_T(S,i+1,n,s1);
                    pr := '2,' + pr +','+ s1;
                end;
            end;
            else break;
        end;{case}
    end;{if}
    proc_S := i;
end;

procedure proc_S (S: string; i,n: integer; var pr: string): integer;
var s1 : string;
begin
    i := proc_E(S,i,n,s1);
    if i > 0 then
    begin
        pr := '6,' + s1;
        while (i <= n) and (i <> 0) do
        case S[i] of
            '*': begin
                if i = n then i := 0
                else
```

```
                begin
                    i := proc_E(S,i+1,n,s1);
                    pr := '4,' + pr +','+ s1;
                end;
            end;
            '/': begin
                if i = n then i := 0
                else
                begin
                    i := proc_E(S,i+1,n,s1);
                    pr := '5,' + pr +','+ s1;
                end;
            end;
            else break;
        end;{case}
    end;{if}
    proc_S := i;
end;

procedure proc_E (S: string; i,n: integer; var pr: string): integer;
var s1 : string;
begin
    case S[i] of
        'a': begin
            pr := '8';
            proc_E := i+1;
        end;
        'b': begin
            pr := '9';
            proc_E := i+1;
        end;
        '(': begin
            proc_E := 0;
            if i < n then
            begin
                i := proc_S(S,i+1,n,s1);
                if (i > 0) and (i < n) then
                begin
                    pr := '7,' + s1;
                    if S[i] = ')' then proc_E := i+1;
                end;
            end;
        end;
```

продолжение ⇩

окончание

```
        else proc_E := 0;
    end;{case}
end;
```

Конечно, и в данном случае алгоритм рекурсивного спуска позволил построить достаточно простой распознаватель, однако, прежде чем удалось его применить, потребовался неформальный анализ правил грамматики. Далеко не всегда такого рода неформальный анализ является возможным, особенно если грамматика содержит десятки и даже сотни правил — человек не всегда в состоянии уловить их смысл и взаимосвязь. Поэтому модификации алгоритма рекурсивного спуска хотя просты и удобны, но не всегда применимы. Даже понять сам факт того, можно или нет в заданной грамматике построить такого рода распознаватель, бывает очень непросто [4 т.1, 5, 12, 15, 18, 21, 58, 59].

Далее будут рассмотрены распознаватели и алгоритмы, которые основаны на строго формальном подходе. Они предваряют построение распознавателя рядом обоснованных действий и преобразований, с помощью которых подготавливаются необходимые исходные данные. В этом случае подготовку всех исходных данных для распознавателя можно формализовать и автоматизировать. Тогда эти предварительные действия можно выполнить с помощью компьютера, в то время как расширенная трактовка рекурсивного спуска предполагает неформальный анализ грамматики человеком, и в этом серьезный недостаток метода.

LL(k)-грамматики

Логическим продолжением идеи, положенной в основу метода рекурсивного спуска, является предложение использовать для выбора одной из множества альтернатив не один, а несколько символов входной цепочки. Однако напрямую переложить алгоритм выбора альтернативы для одного символа на такой же алгоритм для цепочки символов не удастся — два соседних символа в цепочке на самом деле могут быть выведены с использованием различных правил грамматики, поэтому неверным будет напрямую искать их в одном правиле. Т ем не менее существует класс грамматик, основанный именно на этом принципе — выборе одной альтернативы из множества возможных на основе нескольких очередных символов в цепочке. Это LL(k)-грамматики. Правда, алгоритм работы распознавателя для них не так очевидно прост, как рассмотренный выше алгоритм рекурсивного спуска.

Грамматика обладает свойством LL(k), k>0, если на каждом шаге вывода для однозначного выбора очередной альтернативы достаточно знать символ на верхушке стека и рассмотреть первые k символов от текущего положения считывающей головки во входной цепочке символов.

Грамматика называется *LL(k)-грамматикой*, если она обладает свойством LL(k) для некоторого k>0[1].

[1] Требование k > 0, безусловно, является разумным — для принятия решения о выборе той или иной альтернативы МП-автомату надо рассмотреть хотя бы один символ входной цепочки. Если представить себе LL-грамматику с k = 0, то в такой грамматике вывод совсем не будет зависеть от входной цепочки. В принципе, такая грамматика возможна, но в ней будет всего одна-единственная цепочка вывода. Поэтому практическое применение языка, заданного такого рода грамматикой, представляется весьма сомнительным.

Название «LL(k)» несет определенный смысл. Первая литера «L» происходит от слова «left» и означает, что входная цепочка символов читается в направлении слева направо. Вторая литера «L» также происходит от слова «left» и означает , что при работе распознавателя используется левосторонний вывод. Вместо «k» в названии класса грамматики стоит некоторое число, которое показывает , сколько символов надо рассмотреть, чтобы однозначно выбрать одну из множества аль тернатив. Так, существуют LL(1)-грамматики, LL(2)-грамматики и другие классы.

В совокупности все LL(k)-грамматики для всех k>0 образуют класс LL-грамматик.

На рис. 4.2 схематично показано частичное дерево вывода для некоторой LL(k)-грамматики. В нем ω обозначает уже разобранную часть входной цепочки α, которая построена на основе левой части дерева y. Правая часть дерева x — это еще не разобранная часть, а A — текущий нетерминальный символ на верхушке стека МП-автомата. Цепочка x представляет собой незавершенную часть цепочки вывода, содержащую как терминальные, так и нетерминальные символы. После завершения вывода символ A раскрывается в часть входной цепочки υ, а правая часть дерева x преобразуется в часть входной цепочки τ. Свойство LL(k) предполагает, что однозначный выбор альтернативы для символа A может быть сделан на основе k первых символов цепочки $\upsilon\tau$, являющейся частью входной цепочки α.

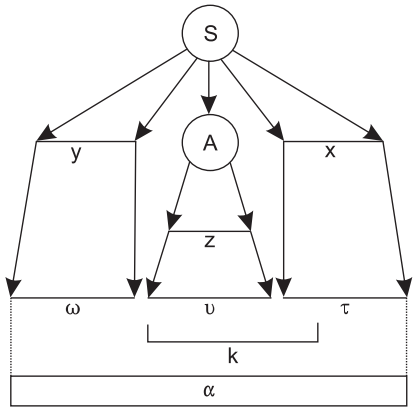

Рис. 4.2. Схема построения дерева вывода для LL(k)-грамматики

Алгоритм разбора входных цепочек для LL(k)-грамматики носит название «k-предсказывающего алгоритма». Принципы его выполнения во многом соответствуют функционированию МП-автомата с той разницей, что на каждом шаге работы этот алгоритм может просматривать k символов вперед от текущего положения считывающей головки автомата.

Для LL(k)-грамматик известны следующие полезные свойства:

- всякая LL(k)-грамматика для любого k>0 является однозначной ;
- существует алгоритм, позволяющий проверить, является ли заданная грамматика LL(k)-грамматикой для строго определенного числа k.

Кроме того, известно, что все грамматики, допускающие разбор по методу рекурсивного спуска, являются подклассом LL(1)-грамматик. То есть любая грамматика,

допускающая разбор по методу рекурсивного спуска, является LL(1)-грамматикой (но не наоборот!).

Есть, однако, неразрешимые проблемы для LL(k)-грамматик. Это общие проблемы, характерные для всех классов КС-грамматик:

- не существует алгоритма, который мог бы проверить, является ли заданная КС-грамматика LL(k)-грамматикой для некоторого произвольного числа k;
- не существует алгоритма, который мог бы преобразовать произвольную КС-грамматику к виду LL(k)-грамматики для некоторого k или доказать, что преобразование невозможно.

Для LL(k)-грамматики при k>1 совсем не обязательно, чтобы все правые части правил грамматики для каждого нетерминального символа начинались с k различных терминальных символов. Принципы распознавания предложений входного языка такой грамматики накладывают менее жесткие ограничения на правила грамматики, поскольку k соседних символов, по которым однозначно выбирается очередная альтернатива, могут встречаться в нескольких правилах грамматики.

ПРИМЕЧАНИЕ

Грамматики, у которых все правые части правил для всех нетерминальных символов начинаются с k различных терминальных символов, носят название «сильно LL(k)-грамматики». Метод построения распознавателей для них достаточно прост, алгоритм разбора очевиден, но, к сожалению, такие грамматики встречаются крайне редко.

Поскольку все LL(k)-грамматики используют левосторонний нисходящий распознаватель, основанный на алгоритме с подбором альтернатив, очевидно, что они не могут допускать левую рекурсию.

ВНИМАНИЕ

Никакая леворекурсивная грамматика не может быть LL-грамматикой. Следовательно, первым делом при попытке преобразовать грамматику к виду LL-грамматики необходимо устранить в ней левую рекурсию.

Класс LL-грамматик широк, но все же он недостаточен для того, чтобы покрыть все возможные синтаксические конструкции в языках программирования. Известно, что существуют ДКС-языки, которые не могут быть заданы LL(k)-грамматикой ни при каких k. Однако LL-грамматики удобны для использования, поскольку позволяют построить линейные распознаватели.

Методы построения распознавателей для LL(k)-грамматик при k>1 в данной книге не рассматриваются, с ними можно ознакомиться в работе [4 т.1].

Синтаксический разбор для LL(1)-грамматик

Очевидно, для каждого нетерминального символа LL(1)-грамматики не может быть двух правил, начинающихся с одного и того же терминального символа. Однако это менее жесткое условие, чем то, которое накладывает распознаватель по методу рекурсивного спуска, поскольку LL(1)-грамматика может допускать в правой части правил цепочки, начинающиеся с нетерминальных символов, а также λ-правила. LL(1)-грамматики позволяют построить достаточно простой и эффективный распознаватель.

Для построения распознавателей языков, заданных LL(k)-грамматиками, используются два множества, определяемые следующим образом:

- FIRST(k,α) — множество терминальных цепочек, выводимых из $\alpha \in (\mathbf{VT} \cup \mathbf{VN})^*$, укороченных до k символов;
- FOLLOW(k,A) — множество укороченных до k символов терминальных цепочек, которые могут следовать непосредственно за $A \in \mathbf{VN}$ в цепочках вывода.

Формально эти два множества могут быть определены следующим образом:

$FIRST(k,\alpha) = \{ \omega \in \mathbf{VT}^* \mid$ либо $|\omega| \le k$ и $\alpha \Rightarrow^* \omega$, либо $|\omega| > k$ и $\alpha \Rightarrow^* \omega x, x \in (\mathbf{VT} \cup \mathbf{VN})^*\}$, $\alpha \in (\mathbf{VT} \cup \mathbf{VN})^*$, $k>0$.

$FOLLOW(k,A) = \{\omega \in \mathbf{VT}^* \mid S \Rightarrow^* \alpha A \gamma$ и $\omega \in FIRST(\gamma,k)$, $\alpha \in \mathbf{VT}^*\}$, $A \in \mathbf{VN}$, $k>0$.

Эти множества используются не только для построения распознавателей языков, заданных LL(k)-грамматиками, но и для ряда других классов грамматик, которые будут рассматриваться далее. В случае LL(1)-грамматик используются множества FIRST(1,α) иFOLLOW(1,A) .

Для LL(1)-грамматик алгоритм работы распознавателя предельно прост. Он заключается всего в двух условиях, проверяемых на шаге выбора альтернативы. Исходными данными для этих условий являются символ $a \in \mathbf{VT}$, обозреваемый считывающей головкой, и символ $A \in \mathbf{VN}$, находящийся на верхушке стека.

Эти условия можно сформулировать так:

- необходимо выбрать в качестве альтернативы правило $A \to \alpha$, если $a \in FIRST(1,\alpha)$;
- необходимо выбрать в качестве альтернативы правило $A \to \lambda$, если оно есть и $a \in FOLLOW(1,A)$.

Если ни одно из этих условий не выполняется, то цепочка не принадлежит заданному языку и алгоритм должен сигнализировать об ошибке.

Действия алгоритма на шаге «выброса» остаются без изменений.

Кроме того, чтобы убедиться, является ли заданная грамматика G(**VT**,**VN**,**P**,S) LL(1)-грамматикой, необходимо и достаточно проверить следующее условие: для каждого символа $A \in \mathbf{VN}$, для которого в грамматике существует более одного правила вида $A \to \alpha_1|\alpha_2|...|\alpha_n$, должно выполняться требование

$FIRST(1,\alpha_i FOLLOW(1,A)) \cap FIRST(1,\alpha_j FOLLOW(1,A)) = \varnothing \ \forall i \ne j, n \ge i > 0, n \ge j > 0$.

Очевидно, что если для символа $A \in \mathbf{VN}$ отсутствует правило вида $A \to \lambda$, то, согласно этому требованию, все множества $FIRST(1,\alpha_1)$, $FIRST(1,\alpha_2)$, ..., $FIRST(1,\alpha_n)$ должны попарно не пересекаться, если же присутствует правило $A \to \lambda$, то они не должны также пересекаться с множеством FOLLOW(1,A).

ВНИМАНИЕ

Очевидно, что LL(1)-грамматика неможет содержать для любого нетерминального символа $A \in VN$ двух правил, начинающихся с одного и того же терминального символа.

Условие, накладываемое на правила LL(1)-грамматики, является довольно жестким. Очень немногие реальные грамматики могут быть отнесены к классу LL(1)-грамматик.

Например, даже довольно простая грамматика G({a}, {S}, {S→a|aS}, S) не удовлетворяет этому условию (хотя она является регулярной праволинейной грамматикой).

Иногда удается преобразовать правила грамматики так, чтобы они удовлетворяли требованию LL(1)-грамматик. Например, приведенная выше грамматика может быть преобразована к виду G'({a}, {S,A},{S →aA,A→λ|S}, S) [1]. В такой форме она уже является LL(1)-грамматикой. Но формального метода преобразовать произвольную КС-грамматику к виду LL(1)-грамматики или убедится в том, что такое преобразование невозможно, не существует.

СОВЕТ

Для приведения произвольной грамматики к виду LL(1)-грамматики можно рекомендовать преобразования, рассмотренные выше при знакомстве с методом рекурсивного спуска. Но применение этих преобразований не гарантирует , что произвольную КС-грамматику удастся привести к виду LL(1)-грамматики.

Для того чтобы запрограммировать работу МП-автомата, выполняющего разбор входных цепочек символов языка, заданного LL(1)-грамматикой, надо научиться строить множества символов FIRST(1, α) и FOLLOW(1,A). Для множества FIRST(1, α) все очевидно, если цепочка α начинается с терминального символа b (α=bβ, b∈ **VT**, α∈ (**VT**∪**VN**)$^+$, β∈ (**VT**∪**VN**)*) — в этом случае FIRST(1, α)={b}, если же она начинается с нетерминального символа B (α=Bβ, B∈ **VN**, α∈ (**VT**∪**VN**)$^+$, β∈ (**VT**∪**VN**)*), то FIRST(1,α)=FIRST(1,B). Следовательно, для LL(1)-грамматик остается только найти алгоритм построения множеств FIRST(1,B) и FOLLOW(1,A) для всех нетерминальных символов A,B∈ **VN**.

Исходными данными для этих алгоритмов служат правила грамматики.

Алгоритм построения множества FIRST(1,A)

Алгоритм строит множества FIRST(1,A) сразу для всех нетерминальных символов грамматики G(**VT**,**VN**,**P**,S), A∈ **VN**. Для выполнения алгоритма надо предварительно преобразовать исходную грамматику G(**VT**,**VN**,**P**,S) в грамматику G'(**VT**,**VN**',**P**',S'), не содержащую λ-правил. На основании полученной грамматики **G**' и выполняется построение множеств FIRST(1,A) для всех A ∈**VN** (если A∈**VN**, то, согласно алгоритму преобразования, также справедливо A ∈**VN**'). Множества строятся методом последовательного приближения. Если в результате преобразования грамматики **G** в грамматику **G**' множество **VN**' содержит новый символ S', то при построении множества FIRST(1,A) он не учитывается.

Алгоритм состоит из нескольких шагов.

Шаг 1. Для всех A ∈**VN**: $FIRST_0(1,A)=\{X \mid A \rightarrow X\alpha \in \mathbf{P}, X \in (\mathbf{VT} \cup \mathbf{VN}), \alpha \in (\mathbf{VT} \cup \mathbf{VN})^*\}$ (первоначально вносим во множество первых символов для каждого нетерминаль-

[1] Можно убедиться, что эти две грамматики задают один и тот же язык: L(G) = L(G'). Это легко сделать, поскольку обе они являются не только КС-грамматиками, но и регулярными праволинейными грамматиками. Кроме того, формальное преобразование G' в G существует — достаточно устранить в грамматике G' λ-правила и цепные правила, и будет получена исходная грамматика G. А вот формального преобразования G в G' нет . В общем случае все может быть гораздо сложнее.

ного символа A все символы, стоящие в начале правых частей правил для этого символа A); i:=0.

Шаг 2. Для всех A∈ **VN**: $FIRST_{i+1}(1,A)=FIRST_i(1,A)\cup FIRST_i(1,B)$, для всех нетерминальных символов B∈ $(FIRST_i(1,A)\cap$ **VN**).

Шаг 3. Если ∃A∈ **VN**: $FIRST_{i+1}(1,A)\neq FIRST_i(1,A)$, то i:=i+1 и вернуться к шагу 2, иначе перейти к шагу 4.

Шаг 4. Для всех A ∈ **VN**: $FIRST(1,A)=FIRST_i(1,A)\setminus$ **VN** (исключаем из построенных множеств все нетерминальные символы).

Алгоритм построения множества FOLLOW(1,A)

Алгоритм строит множества FOLLOW(1,A) сразу для всех нетерминальных символов грамматики G(**VT**,**VN**,**P**,S), A∈ **VN**. Для выполнения алгоритма предварительно надо построить все множества FIRST(1,A), ∀A∈ **VN**. Множества строятся методом последовательного приближения. Алгоритм состоит из нескольких шагов.

Шаг 1. Для всех A∈ **VN**: $FOLLOW_0(1,A)=\{X \mid \exists B\to\alpha AX\beta\in$ **P**, B∈ **VN**, X∈ (**VT**∪**VN**), α,β∈ (**VT**∪**VN**)*} (первоначально вносим во множество последующих символов для каждого нетерминального символа A все символы, которые в правых частях правил встречаются непосредственно за символом A); i:=0.

Шаг 2. $FOLLOW_0(1,S)=FOLLOW_0(1,S)\cup\{\lambda\}$ (вносим пустую цепочку во множество последующих символов для целевого символа S — это означает, что в конце разбора за целевым символом цепочка кончается, иногда для этой цели используется специальный символ конца цепочки: $\perp_к$).

Шаг 3. Для всех A∈ **VN**: $FOLLOW'_i(1,A)=FOLLOW_i(1,A)\cup FIRST(1,B)$, для всех нетерминальных символов B∈ $(FOLLOW_i(1,A)\cap$ **VN**).

Шаг 4. Для всех A ∈ **VN**: $FOLLOW''_i(1,A) = FOLLOW'_i(1,A)\cup FOLLOW'_i(1,B)$, для всех нетерминальных символов B∈ $(FOLLOW'_i(1,A)\cap$ **VN**), если существует правило B→λ.

Шаг 5 . Для всех A ∈ **VN**: $FOLL\ OW_{i+1}(1,A)=FOLLOW''_i(1,A)\cup FOLLOW''_i(1,B)$, для всех нетерминальных символов B ∈ **VN**, если существует правило B →αA, α∈ (**VT**∪**VN**)*.

Шаг 6. Если ∃A∈ **VN**: $FOLLOW_{i+1}(1,A)\neq FOLLOW_i(1,A)$, то i:=i+1 и вернуться к шагу 3, иначе перейти к шагу 7.

Шаг 7. Для всех A∈ **VN**: $FOLLOW(1,A)=FOLLOW_i(1,A)\setminus$ **VN** (исключаем из построенных множеств все нетерминальные символы).

Пример построения распознавателя для LL(1)-грамматики

Рассмотрим в качестве примера грамматику

G({+,–,/,*,a,b},{S,R,T,F,E},**P**,S) с правилами:

```
P:    S → T | TR
      R → +T | –T | +TR | –TR
      T → E | EF
      F → *E | /E | *EF | /EF
      E → (S) | a | b
```

Это нелеворекурсивная грамматика для арифметических выражений (ранее она была построена в подразделе «Устранение левой рекурсии» в данной главе).

Эта грамматика не является LL(1)-грамматикой. Чтобы убедиться в этом, достаточно обратить внимание на правила для символов R и F — для них имеется по два правила, начинающихся с одного и того же терминального символа.

Преобразуем ее в другой вид, добавив λ-правила. В результате получим новую грамматику

G'({+,–,/,*,a,b},{S,R,T,F,E},**P'**,S) с правилами:

```
P':    S → TR
       R → λ | +TR | -TR
       T → EF
       F → λ | *EF | /EF
       E → (S) | a | b
```

Построенная грамматика **G'** эквивалентна исходной грамматике **G**. В этом можно убедиться, если воспользоваться алгоритмом устранения λ-правил из подраздела «Устранение λ-правил» данной главы. Применив его к грамматике **G'**, получим грамматику **G**, а по условиям данного алгоритма L(**G'**)=L(**G**). Таким образом, мы получили эквивалентную грамматику, хотя она и построена неформальным методом (следует помнить, что не существует формального алгоритма, преобразующего произвольную КС-грамматику в LL(k)-грамматику для заданного k).

Эта грамматика является LL(1)-грамматикой. Чтобы убедиться в этом, построим множества FIRST и FOLLOW для нетерминальных символов этой грамматики (поскольку речь заведомо идет об LL(1)-грамматике, цифру 1 в обозначении множеств опустим для сокращения записи).

Для построения множества **FIRST** будем использовать исходную грамматику **G**, так как именно она получается из **G'** при устранении λ-правил.

Построение множества FIRST.

Шаг 1. $FIRST_0(S) = \{T\}$;

$FIRST_0(R) = \{+,-\}$;

$FIRST_0(T) = \{E\}$;

$FIRST_0(F) = \{*,/\}$;

$FIRST_0(E) = \{(,a,b\}$;

i = 0.

Шаг 2. $FIRST_1(S) = \{T,E\}$;

$FIRST_1(R) = \{+,-\}$;

$FIRST_1(T) = \{E,(,a,b\}$;

$FIRST_1(F) = \{*,/\}$;

$FIRST_1(E) = \{(,a,b\}$.

Шаг 3. i = 1, возвращаемся к шагу 2.

Шаг 2. $FIRST_2(S) = \{T,E,(,a,b\}$;

$FIRST_2(R) = \{+,-\}$;

FIRST$_2$(T) = {E,(,a,b};
FIRST$_2$(F) = {*,/};
FIRST$_2$(E) = {(,a,b}.

Шаг 3. i = 2, возвращаемся к шагу 2.

Шаг 2. FIRST$_3$(S) = {T,E,(,a,b};
FIRST$_3$(R) = {+,–};
FIRST$_3$(T) = {E,(,a,b};
FIRST$_3$(F) = {*,/};
FIRST$_3$(E) = {(,a,b}.

Шаг 3. i = 2, переходим к шагу 4.

Шаг 4. FIRST(S) = {(,a,b};
FIRST(R) = {+,–};
FIRST(T) = { (,a,b};
FIRST(F) = {*,/};
FIRST(E) = {(,a,b}.

Построение закончено.

Построение множества FOLLOW.

Шаг 1. FOLLOW$_0$(S) = {)};
FOLLOW$_0$(R) = ∅;
FOLLOW$_0$(T) = {R};
FOLLOW$_0$(F) = ∅;
FOLLOW$_0$(E) = {F};
i = 0.

Шаг 2. FOLLOW$_0$(S) = {),λ};
FOLLOW$_0$(R) = ∅;
FOLLOW$_0$(T) = {R};
FOLLOW$_0$(F) = ∅;
FOLLOW$_0$(E) = {F}.

Шаг 3. FOLLOW'$_0$(S) = {),λ};
FOLLOW'$_0$(R) = ∅;
FOLLOW'$_0$(T) = {R,+,–};
FOLLOW'$_0$(F) = ∅;
FOLLOW'$_0$(E) = {F,*,/}.

Шаг 4. FOLLOW"$_0$(S) = {),λ};
FOLLOW"$_0$(R) = ∅;
FOLLOW"$_0$(T) = {R,+,–};
FOLLOW"$_0$(F) = ∅;
FOLLOW"$_0$(E) = {F,*,/}.

Шаг 5. FOLLOW$_1$(S) = {),λ};

FOLLOW$_1$(R) = {),λ};

FOLLOW$_1$(T) = {R,+,–};

FOLLOW$_1$(F) = {R,+,–};

FOLLOW$_1$(E) = {F,*,/}.

Шаг 6. i = 1, возвращаемся к шагу 3.

Шаг 3. FOLLOW'$_1$(S) = {),λ};

FOLLOW'$_1$(R) = {),λ};

FOLLOW'$_1$(T) = {R,+,–};

FOLLOW'$_1$(F) = {R,+,–};

FOLLOW'$_1$(E) = {F,*,/}.

Шаг 4. FOLLOW"$_1$(S) = {),λ};

FOLLOW"$_1$(R) = {),λ};

FOLLOW"$_1$(T) = {R,+,–,),λ};

FOLLOW"$_1$(F) = {R,+,–,),λ};

FOLLOW"$_1$(E) = {F,R,*,/,+,–,),λ}.

Шаг 5. FOLLOW$_2$(S) = {),λ};

FOLLOW$_2$(R) = {),λ};

FOLLOW$_2$(T) = {R,+,–,),λ};

FOLLOW$_2$(F) = {R,+,–,),λ};

FOLLOW$_2$(E) = {F,R,*,/,+,–,),λ}.

Шаг 6. i = 2, возвращаемся к шагу 3.

Шаг 3. FOLLOW'$_2$(S) = {),λ};

FOLLOW'$_2$(R) = {),λ};

FOLLOW'$_2$(T) = {R,+,–,),λ};

FOLLOW'$_2$(F) = {R,+,–,),λ};

FOLLOW'$_2$(E) = {F,R,*,/,+,–,),λ}.

Шаг 4. FOLLOW"$_2$(S) = {),λ};

FOLLOW"$_2$(R) = {),λ};

FOLLOW"$_2$(T) = {R,+,–,),λ};

FOLLOW"$_2$(F) = {R,+,–,),λ};

FOLLOW"$_2$(E) = {F,R,*,/,+,–,),λ}.

Шаг 5. FOLLOW$_3$(S) = {),λ};

FOLLOW$_3$(R) = {),λ};

FOLLOW$_3$(T) = {R,+,–,),λ};

FOLLOW$_3$(F) = {R,+,–,),λ};

FOLLOW$_3$(E) = {F,R,*,/,+,–,),λ}.

Шаг 6. `i = 2`, переходим к шагу 7.

Шаг 7. `FOLLOW(S) = {),λ};`

`FOLLOW(R) = {),λ};`

`FOLLOW(T) = {+,–,),λ};`

`FOLLOW(F) = {+,–,),λ};`

`FOLLOW(E) = {*,/,+,–,),λ}.`

Построение закончено.

В результате выполнения построений можно увидеть, что необходимое и достаточное условие принадлежности КС-грамматики к классу LL(1)-грамматик выполняется.

Построенные множества FIRST и FOLLOW можно представить в виде таблицы. Результат выполненных построений отражает табл. 4.1.

Таблица 4.1 Множества FIRST и FOLLOW для грамматики G'

Символ A∈VN	FIRST(1,A)	FOLLOW(1,A)
S	(a b	) λ
R	+ –	) λ
T	(a b	+ –) λ
F	* /	+ –) λ
E	(a b	* / + –) λ

Рассмотрим работу распознавателя. Ход разбора будем отражать по шагам работы алгоритма в виде конфигурации МП-автомата, к которой добавлена цепочка, содержащая последовательность примененных правил грамматики. Состояние автомата `q`, указанное в его конфигурации, можно опустить, так как оно единственное: `(α,z,γ)`, где

α — непрочитанная часть входной цепочки символов;

`z` — содержимое стека (верхушка стека находится слева);

γ — последовательность номеров примененных правил (последовательность дополняется слева, так как автомат порождает левосторонний вывод).

Примем, что правила в грамматике номеруются в порядке слева направо и сверху вниз. На основе номеров примененных правил при успешном завершении разбора можно построить цепочку вывода и дерево вывода.

В качестве примера возьмем две правильные цепочки символов, `a+a*b` и `(a+a)*b` и две ошибочные цепочки символов, `a+a*` и `(+a)*b`.

Разбор цепочки `a+a*b`.

1. `(a+a*b,S,λ)`.
2. `(a+a*b,TR,1)`, так как `a∈FIRST(1,TR)`.
3. `(a+a*b,EFR,1,5)`, так как `a∈FIRST(1,EF)`.
4. `(a+a*b,aFR,1,5,10)`, так как `a∈FIRST(1,a)`.
5. `(+a*b, FR, 1,5,10)`.
6. `(+a*b,R,1,5,10,6)`, так как `+∈FOLLOW(1,F)`.

7. (+a*b,+TR,1,5,10,6,3), так как +∈FIRST(1,+TR).
8. (a*b,TR,1,5,10,6,3).
9. (a*b,EFR,1,5,10,6,3,5), так как a∈FIRST(1,EF).
10. (a*b,aFR,1,5,10,6,3,5,10), так как a∈FIRST(1,a).
11. (*b,FR,1,5,10,6,3,5,10).
12. (*b,*EFR,1,5,10,6,3,5,10,7), так как *∈FIRST(1,*EF).
13. (b,EFR,1,5,10,6,3,5,10,7).
14. (b,bFR,1,5,10,6,3,5,10,7,11), так как b∈FIRST(1,b).
15. (λ,FR,1,5,10,6,3,5,10,7,11).
16. (λ,R,1,5,10,6,3,5,10,7,11,6), так как λ∈FOLLOW(1,F).
17. (λ,λ,1,5,10,6,3,5,10,7,11,6,2), так как λ∈FOLLOW(1,R), разбор закончен. Цепочка принимается.

Получили цепочку вывода

S ⇒ TR ⇒ EFR ⇒ aFR ⇒ aR ⇒ a+TR ⇒ a+EFR ⇒ a+aFR ⇒ a+a*EFR ⇒ a+a*bFR ⇒ a+a*bR ⇒ a+a*b

Соответствующее ей дерево вывода приведено на рис. 4.3.

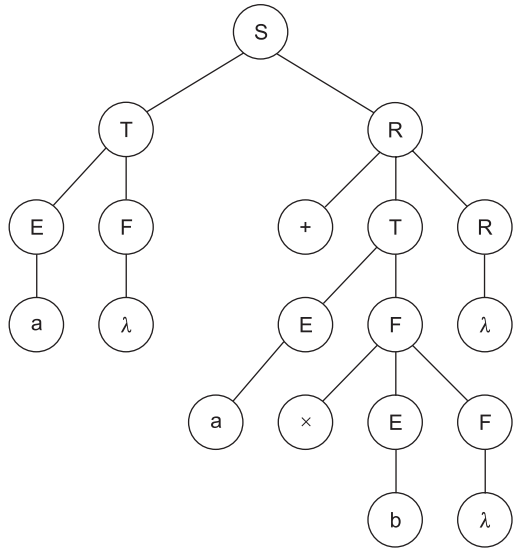

Рис. 4.3. Дерево вывода в LL(1)-грамматике для цепочки «a+a*b»

Разбор цепочки (a+a)*b.

1. ((a+a)*b,S,λ).
2. ((a+a)*b,TR,1), так как (∈FIRST(1,TR).
3. ((a+a)*b,EFR,1,5), так как (∈FIRST(1,EF).
4. ((a+a)*b,(S)FR,1,5,9), так как (∈FIRST(1,(S)).
5. (a+a)*b,S)FR,1,5,9).

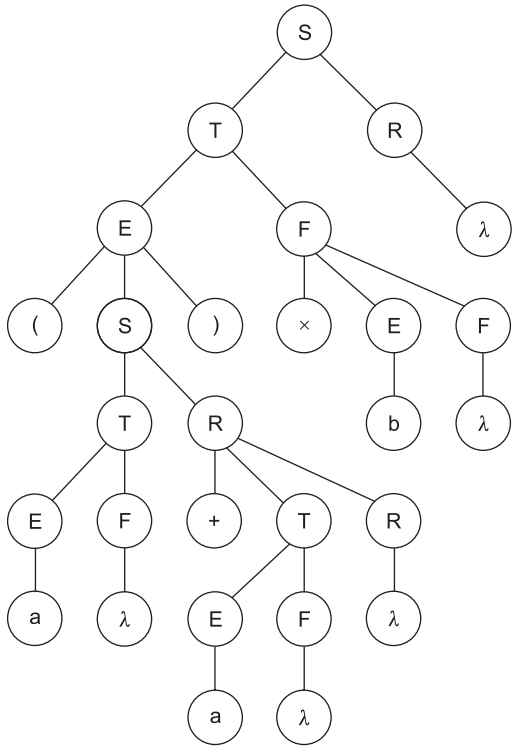

Рис. 4.4. Дерево вывода в LL(1)-грамматике для цепочки «(a+a)*b»

6. (a+a)*b,TR)FR,1,5,9,1), так как a∈FIRST(1,TR).
7. (a+a)*b,EFR)FR,1,5,9,1,5), так как a∈FIRST(1,EF).
8. (a+a)*b,aFR)FR,1,5,9,1,5,10), так как a∈FIRST(1,a).
9. (+a)*b,FR)FR,1,5,9,1,5,10).
10. (+a)*b,R)FR,1,5,9,1,5,10,6), так как +∈FOLLOW(1,F).
11. (+a)*b,+TR)FR,1,5,9,1,5,10,6,3), так как +∈FIRST(1,+TR).
12. .(a)*b,TR)FR,1,5,9,1,5,10,6,3)
13. (a)*b,EFR)FR,1,5,9,1,5,10,6,3,5), так как a∈FIRST(1,EF).
14. (a)*b,aFR)FR,1,5,9,1,5,10,6,3,5,10), так как a∈FIRST(1,a).
15. .(q,)*b,FR)FR,1,5,9,1,5,10,6,3,5,10)
16. ()*b,R)FR,1,5,9,1,5,10,6,3,5,10,6), так как)∈FOLLOW(1,F).
17. ()*b,)FR,1,5,9,1,5,10,6,3,5,10,6,2), так как)∈FOLLOW(1,R).
18. .(*b,FR,1,5,9,1,5,10,6,3,5,10,6,2)
19. (*b,*EFR,1,5,9,1,5,10,6,3,5,10,6,2,7), так как *∈FOLLOW(1,*EF).
20. .(b,EFR,1,5,9,1,5,10,6,3,5,10,6,2,7)
21. (b,bFR,1,5,9,1,5,10,6,3,5,10,6,2,7,11), так как b∈FIRST(1,b).
22. (λ,FR,1,5,9,1,5,10,6,3,5,10,6,2,7,11).

23. (λ,R,1,5,9,1,5,10,6,3,5,10,6,2,7,11,6), так как λ∈ FOLLOW(1,F).
24. (λ,λ,1,5,9,1,5,10,6,3,5,10,6,2,7,11,6,2), так как λ∈FOLLOW(1,R), разбор закончен. Цепочка принимается.

Получили цепочку вывода

S ⇒ TR ⇒ EFR ⇒ (S)FR ⇒ (TR)FR ⇒ (EFR)FR ⇒ (aFR)FR ⇒ (aR)FR ⇒ (a+TR)FR ⇒ (a+EFR)FR ⇒ (a+aFR)FR ⇒ (a+aR)FR ⇒ (a+a)FR ⇒ (a+a)*EFR ⇒ (a+a)*bFR ⇒ (a+a)*bR ⇒ (a+a)*b

Соответствующее ей дерево вывода приведено на рис. 4.4.

Разбор цепочки a+a*.

1. (a+a*,S,λ).
2. (a+a*,TR,1), так как a∈FIRST(1,TR).
3. (a+a*,EFR,1,5), так как a∈FIRST(1,EF).
4. (a+a*,aFR,1,5,10), так как a∈FIRST(1,a).
5. (+a*,FR,1,5,10).
6. (+a*,R,1,5,10,6), так как +∈FOLLOW(1,F).
7. (+a*,+TR,1,5,10,6,3), так как +∈FIRST(1,+TR).
8. .(a*,TR,1,5,10,6,3)
9. (a*,EFR,1,5,10,6,3,5), так как a∈FIRST(1,EF).
10. (a*,aFR,1,5,10,6,3,5,10), так как a∈FIRST(1,a).
11. .(*,FR,1,5,10,6,3,5,10)
12. (*,*EFR,1,5,10,6,3,5,10,7), так как *∈FIRST(1,*EF).
13. (λ,EFR,1,5,10,6,3,5,10,7).
14. Ошибка, так как λ∈FOLLOW(1,E), но нет правила вида E → λ. Цепочка не принимается.

Разбор цепочки (+a)*b.

1. ((+a)*b,S,λ).
2. ((+a)*b,TR,1), так как (∈FIRST(1,TR).
3. ((+a)*b,EFR,1,5), так как (∈FIRST(1,EF).
4. ((+a)*b,(S)FR,1,5,9), так как (∈FIRST(1,(S)).
5. .(+a)*b,S)FR,1,5,9)
6. Ошибка, так как нет правил для S вида S → α таких, чтобы +∈FIRST(1,α) и +∉FOLLOW(1,S). Цепочка не принимается.

Из рассмотренных примеров видно, что алгоритму разбора цепочек, построенному на основе распознавателя для LL(1)-грамматик, требуется гораздо меньше шагов на принятие решения о принадлежности цепочки языку, чем рассмотренному выше алгоритму разбора с возвратами. Надо отметить, что оба алгоритма распознают цепочки одного и того же языка. Данный алгоритм имеет большую эффективность, поскольку при росте длины цепочки количество шагов его растет линейно, а не экспоненциально. Кроме того, ошибка обнаруживается этим алгоритмом сразу , в то время как разбор с возвратами будет просматривать для неверной входной цепочки возможные варианты до конца, пока не переберет их все.

Очевидно, что этот алгоритм является более эффективным, но жесткие ограничения на правила для LL(1)-грамматик сужают возможности его применения.

Восходящие синтаксические распознаватели без возвратов

Восходящие распознаватели выполняют построение дерева вывода снизу вверх. Результатом их работы является правосторонний вывод. Функционирование таких распознавателей основано на модификациях алгоритма «сдвиг—свертка» (или «перенос—свертка»), который был рассмотрен выше. При их создании применяются методы, которые позволяют однозначно выбрать между выполнением «сдвига» («переноса») или «свертки» на каждом шаге алгоритма, а при выполнении свертки однозначно выбрать правило, по которому будет производиться свертка.

LR(k)-грамматики

Идея состоит в том, чтобы модифицировать алгоритм «сдвиг—свертка» таким образом, чтобы на каждом шаге его работы можно было однозначно дать ответ на следующие вопросы:

- что следует выполнять: сдвиг (перенос) или свертку;
- какую цепочку символов α выбрать из стека для выполнения свертки;
- какое правило выбрать для выполнения свертки (в том случае, если существует несколько правил вида $A_1 \to \alpha$, $A_2 \to \alpha$, ... $A_n \to \alpha$).

Тогда восходящий алгоритм распознавания цепочек КС-языка не требовал бы выполнения возвратов. Конечно, как уже было сказано, это нельзя сделать в общем случае, для всех КС-языков. Но среди всех КС-языков можно выделить такие классы языков, для которых подобная реализация распознающего алгоритма возможна.

В первую очередь, можно использовать тот же самый подход, который был положен в основу определения LL(k)-грамматик. Т огда мы получим другой класс КС-грамматик, который носит название LR(k)-грамматик.

КС-грамматика обладает свойством LR(k), $k \geq 0$, если на каждом шаге вывода для однозначного решения вопроса о выполняемом действии в алгоритме «сдвиг—свертка» («перенос—свертка») достаточно знать содержимое верхней части стека и рассмотреть первые k символов от текущего положения считывающей головки автомата во входной цепочке символов.

Грамматика называется *LR(k)-грамматикой,* если она обладает свойством LR(k) для некоторого $k \geq 0$[1].

Название «LR(k)», как и рассмотренное выше «LL(k)», несет определенный смысл. Первая литера «L» обозначает порядок чтения входной цепочки символов: слева направо. Вторая литера «R» происходит от слова «right» и означает , что в результате

[1] Существование LR(0)-грамматик уже не является бессмыслицей, в отличие от LL(0)-грамматик. Расширенный МП-автомат анализирует несколько символов, находящихся на верхушке стека. Поэтому, даже если при k = 0 автомат и не будет смотреть на текущий символ входной цепочки, построенный им вывод все равно будет зависеть от содержимого стека, а значит, и от содержимого входной цепочки.

работы распознавателя получается правосторонний вывод. Вместо «k» в названии грамматики стоит число, которое показывает, сколько символов входной цепочки надо рассмотреть, чтобы принять решение о действии на каждом шаге алгоритма «сдвиг—свертка». Существуют LR(0)-грамматики, LR(1)-грамматики и другие классы.

В совокупности все LR(k)-грамматики для всех $k \geq 0$ образуют класс LR-грамматик.

На рис. 4.5 схематично показано частичное дерево вывода для некоторой LR(k)-грамматики. В нем ω обозначает уже разобранную часть входной цепочки α, на основе которой построена левая часть дерева y. Правая часть дерева x — это еще не разобранная часть, а A — это нетерминальный символ, к которому на очередном шаге будет свернута цепочка символов z, находящаяся на верхушке стека МП-автомата. В эту цепочку уже входит прочитанная, но еще не разобранная часть входной цепочки υ. Правая часть дерева x будет построена на основе части входной цепочки τ. Свойство LR(k) предполагает, что однозначный выбор действия, выполняемого на каждом шаге алгоритма «сдвиг—свертка», может быть сделан на основе цепочки υ и k первых символов цепочки τ, являющихся частью входной цепочки α. Этим очередным действием может быть свертка цепочки z к символу A или перенос первого символа из цепочки τ и добавление его к цепочке z.

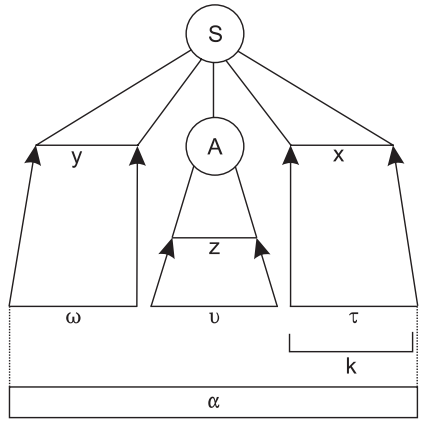

Рис. 4.5. Схема построения дерева вывода для LR(k)-грамматики

Рассмотрев схему построения дерева вывода для LR(k)-грамматики на рис. 4.5 и сравнив ее с приведенной выше на рис. 4.2 схемой для LL(k)-грамматики, можно предположить, что класс LR-грамматик является более широким, чем класс LL-грамматик. Основанием для такого предположения служит тот факт, что на каждом шаге работы распознавателя LR(k)-грамматики обрабатывается больше информации, чем на шаге работы распознавателя LL(k)-грамматики. Действительно, для принятия решения на каждом шаге алгоритма распознавания LL(k)-грамматики используются первые k символов из цепочки $\upsilon\tau$, а для принятия решения на шаге распознавания LR(k)-грамматики — вся цепочка υ и еще первые k символов из цепочки τ. Очевидно, что во втором случае можно проанализировать больший объем информации и, таким образом, построить вывод для более широкого класса КС-языков.

Приведенное выше довольно нестрогое утверждение имеет строго обоснованное доказательство. Доказано, что класс LR-грамматик является более широким, чем класс LL-грамматик [4 т.1, 5]. То есть для каждого КС-языка, заданного LL-грамматикой, может быть построена LR-грамматика, задающая тот же язык, но не наоборот[1].

Для LR(k)-грамматик известны следующие полезные свойства:

- всякая LR(k)-грамматика для любого k≥0 является однозначной;
- существует алгоритм, позволяющий проверить, является ли заданная грамматика LR(k)-грамматикой для строго определенного числа k.

Есть, однако, неразрешимые проблемы для LR(k)-грамматик. Это общие проблемы, характерные для всех классов КС-грамматик:

- не существует алгоритма, который мог бы проверить, является ли заданная КС-грамматика LR(k)-грамматикой для некоторого произвольного числа k;
- не существует алгоритма, который мог бы преобразовать (или доказать, что преобразование невозможно) произвольную КС-грамматику к виду LR(k)-грамматики для некоторого k.

Формальное определение LR(k) свойства для КС-грамматик можно найти в [4 т.1, 5]. Оно основано на понятии *пополненной КС-грамматики*. Грамматика **G'** является пополненной грамматикой, построенной на основании исходной грамматики G(**VT**,**VN**,**P**,S), если выполняются следующие условия:

- грамматика **G'** совпадает с грамматикой **G**, если целевой символ S не встречается нигде в правых частях правил грамматики **G**;
- грамматика **G'** строится как грамматика G'(**VT**,**VN**∪{S'},**P**∪{S'→S},S'), если целевой символ S встречается в правой части хотя бы одного правила из множества **P** в исходной грамматике **G**.

Фактически пополненная КС-грамматика строится таким образом, чтобы ее целевой символ не встречался в правой части ни одного правила. Если нужно, то в исходную грамматику **G** добавляется новый целевой символ S' и новое правило S'→S. Очевидно, что пополненная грамматика **G'** эквивалентна исходной грамматике **G**.

Понятие «пополненная грамматика» введено, чтобы в процессе работы алгоритма «сдвиг—свертка» выполнение свертки к целевому символу S' служило сигналом к завершению алгоритма (поскольку в пополненной грамматике символ S' в правых частях правил не встречается). Поскольку построение пополненных грамматик выполняется элементарно и не накладывает никаких ограничений на исходную КС-грамматику, то дальше будем считать, что все распознаватели для LR(k)-грамматик работают с пополненными грамматиками.

Распознаватель для LR(k)-грамматик функционирует на основе управляющей таблицы **T**. Эта таблица состоит из двух частей, называемых «Действия» и «Пе-

[1] Говоря о соотношении классов LL-грамматик и LR-грамматик, мы не затрагиваем вопрос о значениях k для этих грамматик. Если для некоторой LL(k)-грамматики всегда существует эквивалентная ей LR-грамматика, то это вовсе не значит , что она будет LR(k)-грамматикой с тем же значением k. Но существуют LR-грамматики, для которых нет эквивалентных им LL-грамматик для всех возможных значений k > 0.

реходы». По строкам таблицы распределены все цепочки символов на верхушке стека, которые могут приниматься во внимание в процессе работы распознавателя. По столбцам в части «Действия» распределены все части входной цепочки символов длиной не более k (аванцепочки), которые могут следовать за считывающей головкой автомата в процессе выполнения разбора; а в части «Переходы» — все терминальные и нетерминальные символы грамматики, которые могут появляться на верхушке стека автомата при выполнении действий (сдвигов или сверток).

Клетки управляющей таблицы **T** в части «Действия» содержат следующие данные:

- «*сдвиг*» — если в данной ситуации требуется выполнение сдвига (переноса текущего символа из входной цепочки в стек);
- «*успех*» — если возможна свертка к целевому символу грамматики S и разбор входной цепочки завершен;
- целое число (« *свертка*») — если возможно выполнение свертки (целое число обозначает номер правила грамматики, по которому должна выполняться свертка);
- «*ошибка*» — во всех других ситуациях.

Действия, выполняемые распознавателем, можно вычислять всякий раз на основе состояния стека и текущей аванцепочки. Однако этого вычисления можно избежать, если после выполнения действия сразу же определять, какая строка таблицы **T** будет использована для выбора следующего действия. Тогда эту строку можно поместить в стек вместе с очередным символом и выбрать из стека в момент , когда она понадобится. Таким образом, автомат будет хранить в стеке не только символы алфавита, но и связанные с ними строки управляющей таблицы **T**.

Клетки управляющей таблицы **T** в части «Переходы» как раз и служат для выяснения номера строки таблицы, которая будет использована для определения выполняемого действия на очередном шаге. Эти клетки содержат следующие данные:

- *целое число* — номер строки таблицы **T**;
- «*ошибка*» — во всех других ситуациях.

Для удобства работы распознаватель LR(k)-грамматики использует также два специальных символа, $\perp_н$ и $\perp_к$. Считается, что входная цепочка символов всегда начинается символом $\perp_н$ и завершается символом $\perp_к$. Тогда в начальном состоянии работы распознавателя символ $\perp_н$ находится на верхушке стека, а считывающая головка обозревает первый символ входной цепочки. В конечном состоянии в стеке должны находиться символы S (целевой символ) и $\perp_н$, а считывающая головка автомата должна обозревать символ $\perp_к$.

Алгоритм функционирования распознавателя LR(k)-грамматики можно описать следующим образом.

Шаг 1. Поместить в стек символ $\perp_н$ и начальную (нулевую) строку управляющей таблицы **T**. В конец входной цепочки поместить символ $\perp_к$. Перейти к шагу 2.

Шаг 2. Прочитать с вершины стека строку управляющей таблицы**T**. Выбрать из этой строки часть «Действие» в соответствии с аванцепочкой, обозреваемой считывающей головкой автомата. Перейти к шагу 3.

Шаг 3. В соответствии с типом действия выполнить выбор из четырех вариантов:

- *«сдвиг»* — если входная цепочка не прочитана до конца, прочитать и запомнить как «новый символ» очередной символ из входной цепочки, сдвинуть считывающую головку на одну позицию вправо, иначе прервать выполнение алгоритма и сообщить об ошибке;
- целое число (*«свертка»*) — выбрать правило в соответствии с номером, удалить из стека цепочку символов, составляющую правую часть выбранного правила, взять символ из левой части правила и запомнить его как «новый символ»;
- *«ошибка»* — прервать выполнение алгоритма, сообщить об ошибке;
- *«успех»* — выполнить свертку к целевому символу S, прервать выполнение алгоритма, сообщить об успешном разборе входной цепочки символов, если входная цепочка прочитана до конца, иначе сообщить об ошибке.

Конец выбора. Перейти к шагу 4.

Шаг 4. Прочитать с вершины стека строку управляющей таблицы**T**. Выбрать из этой строки часть «Переходы» в соответствии с символом, который был запомнен как «новый символ» на предыдущем шаге. Перейти к шагу 5.

Шаг 5. Если часть «Переходы» содержит вариант «ошибка», тогда прервать выполнение алгоритма и сообщить об ошибке, иначе (если там содержится номер строки управляющей таблицы **T**) положить в стек «новый символ» и строку таблицы **T** с выбранным номером. Вернуться к шагу 2.

Для работы алгоритма кроме управляющей таблицы **T** используется также временная переменная («новый символ»), хранящая значение терминального или нетерминального символа. В программной реализации алгоритма вовсе не обязательно помещать в стек сами строки управляющей таблицы — поскольку таблица неизменна, достаточно запоминать соответствующие ссылки.

Доказано, что данный алгоритм имеет линейную зависимость необходимых для его выполнения вычислительных ресурсов от длины входной цепочки символов. Следовательно, распознаватель для LR(k)-грамматики является линейным распознавателем.

На практике используются LR(0)-грамматики и LR(1)-грамматики. LR(k)-грамматики со значениями k>1 практически не применяются. Это вызвано двумя причинами:

- построение распознавателей для LR(k)-грамматик при k>1 связано со значительными сложностями, а управляющая таблица **T** имеет большой объем;
- доказано, что для любого детерминированного КС-языка может быть построена LR(1)-грамматика, задающая этот язык, поэтому не имеет смысла использовать LR(k)-грамматики со значениями k > 1.

ВНИМАНИЕ

Класс языков, заданных LR(1)-грамматиками, полностью совпадает с классом детерминированных КС-языков .

То есть любая LR(1)-грамматика задает ДКС-язык (это очевидно следует из однозначности всех LR-грамматик), и для любого ДКС-языка можно построить

LR(1)-грамматику, задающую этот язык. Второе утверждение уже не столь очевидно, но доказано в теории формальных языков [4 т.1, 5].

ПРИМЕЧАНИЕ

Класс LR(1)-грамматик мог бы быть универсальным для построения синтаксических распознавателей, если бы была разрешима проблема преобразования для КС-грамматик.

Синтаксические конструкции всех языков программирования относятся к классу ДКС-языков. Но ДКС-язык может быть задан грамматикой, которая не относится к классу LR(1)-грамматик. В таком случае совсем не очевидно, что для этого языка удастся построить распознаватель на основе LR(1)-грамматики, потому что в общем случае нет алгоритма, который позволил бы эту грамматику получить, хотя и известно, что она существует. То, что проблема не разрешима в общем случае, совсем не означает, что ее не удастся решить в конкретной ситуации. Факт существования LR(1)-грамматики для каждого ДКС-языка играет важную роль — всегда есть смысл попытаться построить такую грамматику.

Синтаксический разбор для LR(0)-грамматик

Построение управляющей таблицы для LR(0)-грамматик

Простейшим случаем LR(k)-грамматик являются LR(0)-грамматики. При k=0 распознающий расширенный МП-автомат совсем не принимает во внимание текущий символ, обозреваемый считывающей головкой. Решение о выполняемом действии принимается только на основании содержимого стека автомата. При этом не должно возникать конфликтов между выполняемым действием (сдвиг или свертка), а также между различными вариантами при выполнении свертки.

Для построения управляющей таблицы **T** заданной LR(0)-грамматики введем понятие *последовательности ситуаций*.

Ситуация представляет собой множество правил КС-грамматики, записанных с учетом положения считывающей головки МП-автомата, которое может возникнуть в процессе выполнения разбора сентенциальной формы этой грамматики. Положение считывающей головки автомата обозначается в правой части правила $A\to\alpha$, где $\alpha\in(\mathbf{VT}\cup\mathbf{VN})^*$, специальным символом •, который может стоять в произвольном месте цепочки α: $\alpha=\gamma\bullet\beta$ (причем любая из цепочек, γ и β, или обе эти цепочки могут быть пустыми). Этот символ не входит в алфавит грамматики (он не является ни терминальным, ни нетерминальным символом).

Если S — целевой символ грамматики **G** и правила $S\to\alpha_1|\alpha_2|...|\alpha_n$ входят во множество правил этой грамматики, то ситуация, содержащая множество правил $\{S\to\bullet\alpha_1; S\to\bullet\alpha_2; ... S\to\bullet\alpha_n\}$, называется *начальной ситуацией*. Смысл начальной ситуации очевиден: считывающая головка распознающего МП-автомата находится в начале одной из возможных сентенциальных форм грамматики.

Последовательность ситуаций строится по определенным правилам, начиная от начальной ситуации. Правила построения последовательности ситуаций следующие:

- если ситуация содержит правило вида $A\to\gamma\bullet B\beta$, где $A,B\in\mathbf{VN}$, $\gamma,\beta\in(\mathbf{VN}\cup\mathbf{VT})^*$, и при этом правила $B\to\alpha_1|\alpha_2|...|\alpha_m$ входят во множество правил грамматики, то правила вида $B\to\bullet\alpha_1; B\to\bullet\alpha_2; ... B\to\bullet\alpha_m$ должны быть добавлены в ситуацию;

- если ситуация содержит правила вида $A \to \gamma \bullet x\beta$, где $A \in \mathbf{VN}$, $\gamma,\beta \in (\mathbf{VN} \cup \mathbf{VT})^*$, а x — произвольный символ ($x \in \mathbf{VN} \cup \mathbf{VT}$), то из нее может быть построена новая ситуация, содержащая правила вида $A \to \gamma x \bullet \beta$, при этом новая ситуация должна проистекать из исходной ситуации на основании символа x.

Ситуации, проистекающие друг из друга на основании различных символов, образуют последовательность ситуаций. Последовательность ситуаций может быть изображена в виде графа взаимосвязи ситуаций, вершины которого соответствуют ситуациям, а дуги — взаимосвязям между ситуациями. Дуги графа взаимосвязи ситуаций будут помечены терминальными и нетерминальными символами грамматики, по которым ситуации связаны между собой.

Поскольку количество правил грамматики конечно, то конечно и количество их комбинаций с символом •. Следовательно, конечно и множество всех возможных подмножеств таких комбинаций, то есть и количество возможных ситуаций также конечно. Поэтому последовательность ситуаций всегда может быть построена и никогда не будет бесконечной.

Обозначим все ситуации в последовательности ситуаций $\mathbf{R}_i$, где i — номер ситуации от 0 до N−1 (где N — общее количество ситуаций в последовательности). Т огда начальная ситуация будет обозначаться $\mathbf{R}_0$.

После построения последовательности ситуаций на ее основе строится управляющая таблица **T**. Построение таблицы происходит следующим образом:

- количество строк в таблице **T** соответствует количеству N ситуаций **R** в последовательности ситуаций, причем каждой строке T_i в таблице **T** соответствует ситуация $\mathbf{R}_i$ в последовательности ситуаций;
- для всех ситуаций в последовательности от $\mathbf{R}_0$ до $\mathbf{R}_{N-1}$ выполняется следующее:
  - если ситуация $\mathbf{R}_i$ содержит правило вида $S \to \gamma \bullet$, $\gamma \in (\mathbf{VN} \cup \mathbf{VT})^*$, где S — целевой символ грамматики, то в графе *«Действия»* строки T_i таблицы **T** должно быть записано *«успех»*;
  - если ситуация $\mathbf{R}_i$ содержит правило вида $A \to \gamma \bullet$, где $A \in \mathbf{VN}$, $\gamma \in (\mathbf{VN} \cup \mathbf{VT})^*$, A не является целевым символом и грамматика содержит правило вида $A \to \gamma$ с номером *m*, то в графе *«Действия»* строки T_i таблицы **T** должно быть записано число *m* (*свертка* по правилу с номером *m*);
  - если ситуация $\mathbf{R}_i$ содержит правила вида $A \to \gamma \bullet x\beta$, где $A \in \mathbf{VN}$, а x — произвольный символ ($x \in \mathbf{VN} \cup \mathbf{VT}$), $\gamma,\beta \in (\mathbf{VN} \cup \mathbf{VT})^*$, то в графе *«Действия»* строки T_i таблицы **T** должно быть записано *«сдвиг»*;
  - если ситуация $\mathbf{R}_j$ проистекает из ситуации $\mathbf{R}_i$ и связана с нею через символ x ($x \in \mathbf{VN} \cup \mathbf{VT}$), то в графе *«Переходы»*, соответствующей символу x, строки T_i таблицы **T** должно быть записано число *j*;
- клетки таблицы, оставшиеся пустыми после заполнения таблицы T на основании последовательности ситуаций, соответствуют состоянию *«ошибка»*.

Если удается непротиворечивым образом заполнить управляющую таблицу**T** на основании последовательности ситуаций, то рассматриваемая грамматика является LR(0)-грамматикой. Иначе, когда при заполнении таблицы возникают противоречия, грам-

матика не является LR(0)-грамматикой, и для нее нельзя построить распознаватель типа LR(0) [4 т.1, 5, 12, 59].

Пример построения распознавателя для LR(0)-грамматики

Рассмотрим КС-грамматику G({a,b}, {S}, {S→aSS|b}, S). Пополненная грамматика для нее будет иметь вид G'({a,b}, {S,S'}, {S'→S,S→aSS|b}, S'). Построим последовательность ситуаций для грамматики G'.

Начальная ситуация R_0 будет содержать правило S'→ •S, но в нее также должны быть включены правила S→•aSS и S→•b, поскольку в грамматике для символа S есть правила S→aSS|b. Получим ситуацию R_0={S'→•S,S→•aSS,S→•b}.

Из начальной ситуации R_0 по символу S можно получить ситуацию R_1, содержащую правило S'→S•. Других правил в нее не может быть добавлено, поэтому получаем R_1={S'→S•}.

Из начальной ситуации R_0 по символу a можно получить ситуацию R_2, содержащую правило S→a•SS, но в нее также должны быть включены правила S→•aSS и S→•b, поскольку в грамматике для символа S есть правила S→aSS|b. Получаем R_2={S→a•SS,S→•aSS,S→•b}.

Из начальной ситуации R_0 по символу b можно получить ситуацию R_3, содержащую правило S→b•. Других правил в нее не может быть добавлено, поэтому получаем R_3={S→b•}.

Из ситуации R_2 по символу S можно получить ситуацию R_4, содержащую правило S→aS•S, но в нее также должны быть включены правила S→•aSS и S→•b, поскольку в грамматике для символа S есть правила S→aSS|b. Получаем R_4={S→aS•S,S→•aSS,S→•b}.

Из ситуации R_4 по символу S можно получить ситуацию R_5, содержащую правило S→aSS•. Других правил в нее не может быть добавлено, поэтому получаем R_5={S→aSS•}.

Из ситуации R_2 по символу a можно получить ситуацию R_6, содержащую правило S→a•SS, но в нее также должны быть включены правила S→•aSS и S→•b, поскольку в грамматике для символа S есть правила S→aSS|b. Получаем R_6={S→a•SS, S→•aSS,S→•b}. Можно заметить, что эта ситуация совпадает с ситуацией R_2: R_6=R_2.

Из ситуации R_2 по символу b можно получить ситуацию R_7, содержащую правило S→b•. Других правил в нее не может быть добавлено, поэтому получаем R_7={S→b•}. Можно заметить, что эта ситуация совпадает с ситуацией R_3: R_7=R_3.

Из ситуации R_4 по символу a можно получить ситуацию R_8, содержащую правило S→a•SS, но в нее также должны быть включены правила S→•aSS и S→•b, поскольку в грамматике для символа S есть правила S→aSS|b. Получаем R_8={S→a•SS,S→•aSS,S→•b}. Можно заметить, что эта ситуация совпадает с ситуацией R_2: R_8=R_2.

Из ситуации R_4 по символу b можно получить ситуацию R_9, содержащую правило S→b•. Других правил в нее не может быть добавлено, поэтому получаем R_9={S→b•}. Можно заметить, что эта ситуация совпадает с ситуацией R_3: R_9=R_3.

Всего получилось 6 различных ситуаций от R_0 до R_5. Граф взаимосвязи ситуаций показан на рис. 4.6.

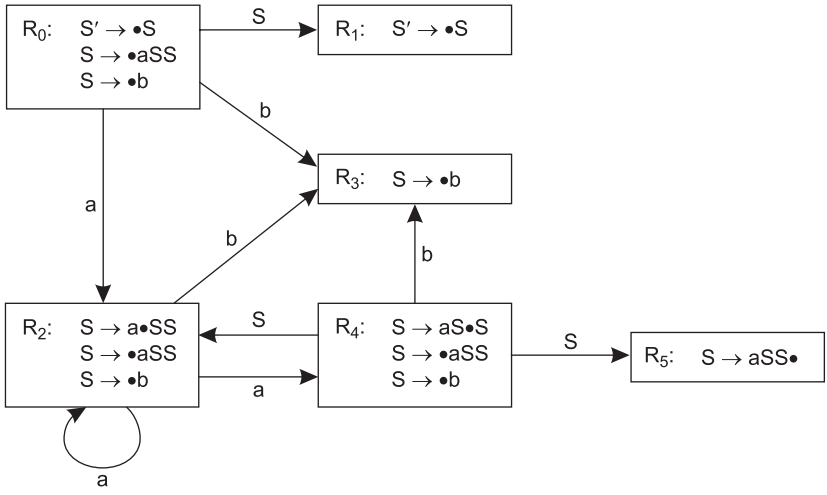

Рис. 4.6. Пример графа взаимосвязи ситуаций для LR(0)-грамматики

На основе найденной последовательности ситуаций можно построить управляющую таблицу распознавателя для LR(0)-грамматики . Поскольку при построении управляющей таблицы не возникает противоречий, то рассмотренная грамматика является LR(0)-грамматикой. Управляющая таблица для нее приведена в табл. 4.2.

Таблица 4.2. Пример управляющей таблицы для LR(0)-грамматики

№	Стек	Действия	Переходы		
			S	A	b
0	$\perp_н$	сдвиг	1	2	3
1	S	успех, 1			
2	a	сдвиг	4	2	3
3	b	свертка, 3			
4	aS	сдвиг	5	2	3
5	aSS	свертка, 2			

Колонка *«Стек»*, присутствующая в таблице, не нужна для распознавателя. Она введена для пояснения состояния стека автомата. Правила грамматики пронумерованы от 1 до 3. Распознаватель работает невзирая на текущий символ, обозреваемый считывающей головкой расширенного МП-автомата, поэтому колонка *«Действия»* в таблице имеет один столбец, не помеченный никаким символом.

Рассмотрим примеры распознавания цепочек языка, заданного этой грамматикой. Конфигурацию расширенного МП-автомата будем отображать в виде трех компонент: непрочитанная часть входной цепочки символов, содержимое стека МП-автомата, последовательность номеров примененных правил грамматики (поскольку автомат имеет только одно состояние, его можно не учитывать). В стеке МП-автомата вместе с помещенными туда символами показаны и номера строк управляющей таблицы, соответствующие этим символам в формате {*символ, номер строки*}.

Разбор цепочки ababab:

1. (abababb⊥$_к$,{⊥$_н$,0},λ).
2. (bababb⊥$_к$,{⊥$_н$,0}{a,2},λ).
3. (ababb⊥$_к$,{⊥$_н$,0}{a,2}{b,3},λ).
4. (ababb⊥$_к$,{⊥$_н$,0}{a,2}{S,4},3).
5. (babb⊥$_к$,{⊥$_н$,0}{a,2}{S,4}{a,2},3).
6. (abb⊥$_к$,{⊥$_н$,0}{a,2}{S,4}{a,2}{b,3},3).
7. (abb⊥$_к$,{⊥$_н$,0}{a,2}{S,4}{a,2}{S,4},3,3).
8. (bb⊥$_к$,{⊥$_н$,0}{a,2}{S,4}{a,2}{S,4}{a,2},3,3).
9. (b⊥$_к$,{⊥$_н$,0}{a,2}{S,4}{a,2}{S,4}{a,2}{b,3},3,3).
10. (b⊥$_к$,{⊥$_н$,0}{a,2}{S,4}{a,2}{S,4}{a,2}{S,4},3,3,3).
11. (⊥$_к$,{⊥$_н$,0}{a,2}{S,4}{a,2}{S,4}{a,2}{S,4}{b,3},3,3,3).
12. (⊥$_к$,{⊥$_н$,0}{a,2}{S,4}{a,2}{S,4}{a,2}{S,4}{S,5},3,3,3,3).
13. (⊥$_к$,{⊥$_н$,0}{a,2}{S,4}{a,2}{S,4}{S,5},3,3,3,3,2).
14. (⊥$_к$,{⊥$_н$,0}{a,2}{S,4}{S,5},3,3,3,3,2,2).
15. (⊥$_к$,{⊥$_н$,0}{S,1},3,3,3,3,2,2,2).
16. (⊥$_к$,{⊥$_н$,0}{S',*},3,3,3,3,2,2,2,1) — разбор завершен.

Соответствующая цепочка вывода будет иметь вид (используется правосторонний вывод) S'⇒S⇒aSS⇒aSaSS⇒aSaSaSS⇒aSaSaSb⇒aSaSabb⇒aSababb⇒abababb.

Разбор цепочки aabbb:

1. (aabbb⊥$_к$,{⊥$_н$,0},λ).
2. (abbb⊥$_к$,{⊥$_н$,0}{a,2},λ).
3. (bbb⊥$_к$,{⊥$_н$,0}{a,2}{a,2},λ).
4. (bb⊥$_к$,{⊥$_н$,0}{a,2}{a,2}{b,3},λ).
5. (bb⊥$_к$,{⊥$_н$,0}{a,2}{a,2}{S,4},3).
6. (b⊥$_к$,{⊥$_н$,0}{a,2}{a,2}{S,4}{b,3},3).
7. (b⊥$_к$,{⊥$_н$,0}{a,2}{a,2}{S,4}{S,5},3,3).
8. (b⊥$_к$,{⊥$_н$,0}{a,2}{S,4},3,3,2).
9. (⊥$_к$,{⊥$_н$,0}{a,2}{S,4}{b,3},3,3,2).
10. (⊥$_к$,{⊥$_н$,0}{a,2}{S,4}{S,5},3,3,2,3).
11. (⊥$_к$,{⊥$_н$,0}{S,1},3,3,2,3,2).
12. (⊥$_к$,{⊥$_н$,0}{S',*},3,3,2,3,2,1) — разбор завершен.

Соответствующая цепочка вывода будет иметь вид (используется правосторонний вывод) S'⇒S⇒aSS⇒aSb⇒aaSSb⇒aaSbb⇒aabbb.

Разбор цепочки aabb:

1. (aabb⊥$_к$,{⊥$_н$,0},λ).
2. (abb⊥$_к$,{⊥$_н$,0}{a,2},λ).

3. (bb⊥к,{⊥н,0}{a,2}{a,2},λ).
4. (b⊥к,{⊥н,0}{a,2}{a,2}{b,3},λ).
5. (b⊥к,{⊥н,0}{a,2}{a,2}{S,4},3).
6. (⊥к,{⊥н,0}{a,2}{a,2}{S,4}{b,3},3).
7. (⊥к,{⊥н,0}{a,2}{a,2}{S,4}{S,5},3,3).
8. (⊥к,{⊥н,0}{a,2}{S,4},3,3,2).
9. Ошибка, невозможно выполнить сдвиг.

Распознаватель для LR(0)-грамматики достаточно прост. Приведенный выше пример можно сравнить с методом рекурсивного спуска или с распознавателем для LL(1)-грамматики — оба этих метода применимы к описанной выше грамматике. По количеству шагов работы распознавателя эти методы сопоставимы, но по реализации нисходящие распознаватели в данном случае немного проще.

Ограничения LR(0)-грамматик

Управляющая таблица **T** для LR(0)-грамматики строится на основании понятия «левых контекстов» нетерминальных символов : после выполнения свертки для нетерминального символа A в стеке МП-автомата ниже этого символа будут располагаться только те символы, которые могут встречаться в цепочке вывода слева от A. Эти символы и составляют «левый контекст» для A. Поскольку выбор между сдвигом или сверткой, а также между типом свертки в LR(0)-грамматиках выполняется только на основании содержимого стека, то LR(0)-грамматика должна допускать однозначный выбор на основе левого контекста для каждого символа [4 т.2, 5].

Однако класс LR(0)-грамматик сильно ограничен, поскольку существует очень мало грамматик, позволяющих выполнять разбор на основании только левого контекста.

Рассмотрим КС-грамматику G({a,b}, {S}, {S→SaSb|λ}, S). Пополненная грамматика для нее будет иметь вид G'({a,b}, {S,S'}, {S'→S,S→SaSb|λ}, S').

Начнем построение последовательности ситуаций для грамматики G'.

Начальная ситуация $\mathbf{R}_0$ будет содержать правило S'→•S, но в нее также должны быть включены правила S→•SaSb и S→•, поскольку в грамматике для символа S есть правила S→SaSb|λ (в правиле S→• пустое место соответствует пустой цепочке λ, и это правило в равной степени можно рассматривать как S→•λ или S→λ•). Получим ситуацию $\mathbf{R}_0$={S'→•S,S→•SaSb,S→•}.

При построении управляющей таблицы **T** для этой грамматики уже на шаге для ситуации $\mathbf{R}_0$ возникнут противоречия: с одной стороны, ситуация содержит правило S→•, и для нее в управляющей таблице в графе «*Действия*» должно быть записано «*свертка*» с номером правила *3*: S→λ (правила в грамматике пронумерованы последовательно от 0 до 3); но, с другой стороны, ситуация содержит правила S'→•S и S→•SaSb, поэтому для нее в той же графе «*Действия*» должно быть записано «*сдвиг*». Возникшее противоречие говорит о том, что рассматриваемая грамматика не является LR(0)-грамматикой, для нее невозможно выполнить разбор входной цепочки только на основании левого контекста.

ВНИМАНИЕ

Очевидно, что любая грамматика, содержащая λ-правила, не может быть LR(0)-грамматикой.

Можно попытаться выполнить преобразования (например исключить λ-правила), чтобы привести грамматику к виду LR(0)-грамматики, но это не всегда возможно.

В тех случаях, когда невозможно построить восходящий распознаватель на основе использования только левого контекста, для построения распознавателя используется не только левый, но и правый контекст. Правым контекстом является символ входной цепочки, обозреваемый считывающей головкой расширенного МП-автомата. В этом случае мы получаем распознаватель на основе LR(1)-грамматики[1].

Синтаксический разбор для LR(1)-грамматик

Построение управляющей таблицы для LR(1)-грамматики

Как и для LR(0)-грамматик, управляющую таблицу **T** для LR(1)-грамматик можно построить на основании последовательности ситуаций [4 т. 2, 5]. Однако ситуация для LR(1)-грамматики имеет более сложную форму, чем для LR(0)-грамматики.

Правило в ситуации для LR(1)-грамматики должно иметь вид $A \to \gamma \bullet \beta / a$, где $A \in \mathbf{VN}$, $\gamma, \beta \in (\mathbf{VN} \cup \mathbf{VT})^*$, $a \in \mathbf{VT}$. Символ •, а также цепочки γ и β имеют тот же смысл, что и для LR(0)-грамматик, а символ $a \in \mathbf{VT}$ определяет правый контекст.

Начальная ситуация в последовательности ситуаций для LR(1)-грамматики должна содержать правила $\{S \to \bullet\alpha_1/\perp_к, S \to \bullet\alpha_2/\perp_к, \ldots S \to \bullet\alpha_n/\perp_к\}$, если правила $S \to \alpha_1|\alpha_2|\ldots|\alpha_n$ входят в множество правил этой грамматики, а символ S является целевым символом. Символ $\perp_к$ обозначает символ конца входной цепочки.

Правила построения последовательности ситуаций для LR(1)-грамматики следующие:

- если ситуация содержит правило вида $A \to \gamma \bullet B b \beta / a$, где $A,B \in \mathbf{VN}$, $a,b \in \mathbf{VT}$, $\gamma, \beta \in (\mathbf{VN} \cup \mathbf{VT})^*$, и при этом правила $B \to \alpha_1|\alpha_2|\ldots|\alpha_m$ входят во множество правил грамматики, то правила вида $B \to \bullet\alpha_1/b$; $B \to \bullet\alpha_2/b$; ... $B \to \bullet\alpha_m/b$ должны быть добавлены в ситуацию;
- если ситуация содержит правило вида $A \to \gamma \bullet B/a$, где $A,B \in \mathbf{VN}$, $a \in \mathbf{VT}$, $\gamma \in (\mathbf{VN} \cup \mathbf{VT})^*$, и при этом правила $B \to \alpha_1|\alpha_2|\ldots|\alpha_m$ входят во множество правил грамматики, то правила вида $B \to \bullet\alpha_1/a$; $B \to \bullet\alpha_2/a$; ... $B \to \bullet\alpha_m/a$ должны быть добавлены в ситуацию;
- если ситуация содержит правило вида $A \to \gamma \bullet BC\beta/a$, где $A,B,C \in \mathbf{VN}$, $a,b \in \mathbf{VT}$, $\gamma, \beta \in (\mathbf{VN} \cup \mathbf{VT})^*$, и при этом правила $B \to \alpha_1|\alpha_2|\ldots|\alpha_m$ входят во множество правил грамматики, тогда $\forall c \in \mathbf{VT}$, таких, что $c \in \mathrm{FIRST}(1,C)$, правила вида $B \to \bullet\alpha_1/c$; $B \to \bullet\alpha_2/c$; ... $B \to \bullet\alpha_m/c$ должны быть добавлены в ситуацию;
- если ситуация содержит правила вида $A \to \gamma \bullet x\beta/a$, где $A \in \mathbf{VN}$, $a \in \mathbf{VT}$, а x — произвольный символ ($x \in \mathbf{VN} \cup \mathbf{VT}$), $\gamma, \beta \in (\mathbf{VN} \cup \mathbf{VT})^*$, то из нее может быть построена новая ситуация, содержащая правила вида $A \to \gamma x \bullet \beta/a$, при этом новая ситуация должна проистекать из исходной ситуации на основании символа x.

[1] Как уже было сказано выше, распознаватели для LR(k)-грамматик с $k > 1$ практически не используются.

Здесь FIRST(1,C) обозначает множество первых символов для нетерминального символа C∈ **VN**. Построение этого множества было рассмотрено выше в разделе, посвященном нисходящим распознавателям.

Очевидно, что, поскольку правила в ситуациях LR(1)-грамматики учитывают еще и правый контекст, общее количество возможных ситуаций в LR(1)-грамматике будет существенно больше, чем количество ситуаций в LR(0)-грамматике.

Управляющая таблица **T** для распознавателя на основе LR(1)-грамматики строится на основе последовательности ситуаций так же, как и для LR(0)-грамматики, с учетом того, что графа *«Действия»* в этой таблице разбита на несколько подграф по количеству возможных терминальных символов, каждая из которых соответствует определенному символу правого контекста. Для всех ситуаций в последовательности от $\mathbf{R}_0$ до $\mathbf{R}_{N-1}$ с учетом правого контекста выполняются следующие действия:

- если ситуация $\mathbf{R}_i$ содержит правило вида S →γ•/a, γ∈ **(VN∪VT)***, a∈ **VT**, где S — целевой символ грамматики, то в графе *«Действия»* строки T_i таблицы **T** для символа *a* должно быть записано *«успех»*;
- если ситуация $\mathbf{R}_i$ содержит правило вида A →γ•/a, где A ∈ **VN**, γ∈ **(VN∪VT)***, a∈ **VT**, A не является целевым символом и грамматика содержит правило вида A → γ с номером *m*, то в графе *«Действия»* строки T_i таблицы **T** для символа a должно быть записано число *m* (*свертка* по правилу с номером *m*);
- если ситуация $\mathbf{R}_i$ содержит правила вида A→γ•xβ/a, где A∈ **VN**, a∈ **VT**, а x — произвольный символ (x ∈ **VN∪VT**), γ,β∈ **(VN∪VT)***, то в графе *«Действия»* строки T_i таблицы **T** для символа a должно быть записано *«сдвиг»*;
- если ситуация $\mathbf{R}_j$ проистекает из ситуации $\mathbf{R}_i$ и связана с нею через символ x (x∈ **VN∪VT**), то в графе *«Переходы»*, соответствующей символу x, строки T_i таблицы **T** должно быть записано число *j*.

Если удается непротиворечивым образом заполнить управляющую таблицу **T**, то рассматриваемая грамматика является LR(1)-грамматикой. Иначе, когда при заполнении таблицы возникают противоречия, грамматика не является LR(1)-грамматикой, и для нее нельзя построить распознаватель типа LR(1) [4 т.1,2, 5].

Пример построения распознавателя для LR(1)-грамматики

Возьмем КС-грамматику `G({a,b}, {S}, {S→SaSb|λ}, S)`, которая уже рассматривалась выше. Пополненная грамматика для нее будет иметь вид `G'({a,b}, {S,S'}, {S'→S,S→SaSb|λ}, S')`.

Построим последовательность ситуаций для грамматики `G'`.

Начальная ситуация $\mathbf{R}_0$ будет содержать правило $S' \to \bullet S/\perp_к$, но в нее также должны быть включены правила $S \to \bullet SaSb/\perp_к$ и $S \to \bullet/\perp_к$, поскольку в грамматике для символа S есть правила $S \to SaSb|\lambda$, а затем на основании правила $S \to \bullet SaSb/\perp_к$ еще и правила $S \to \bullet SaSb/a$ и $S \to \bullet/a$. Получим ситуацию $\mathbf{R}_0 = \{S' \to \bullet S/\perp_к, S \to \bullet SaSb/\perp_к, S \to \bullet/\perp_к, S \to \bullet SaSb/a, S \to \bullet/a\}$.

Из начальной ситуации $\mathbf{R}_0$ по символу S можно получить ситуацию $\mathbf{R}_1$, содержащую правила $S' \to S\bullet/\perp_к$, $S \to S\bullet aSb/\perp_к$ и $S \to S\bullet aSb/a$. Других правил в нее не может быть добавлено, поэтому получаем $\mathbf{R}_1 = \{S' \to S\bullet/\perp_к, S \to S\bullet aSb/\perp_к, S \to S\bullet aSb/a\}$.

Из ситуации R_1 по символу a можно получить ситуацию R_2, содержащую правила S→Sa•Sb/$\perp_к$ и S→Sa•Sb/a, но в нее также должны быть включены правила S→•SaSb/b и S→•/b, поскольку в грамматике для символа S есть правила S→SaSb|λ, а затем на основании правила S→•SaSb/b еще и правила S→•SaSb/a и S→•/a. Получим ситуацию R_2={S→Sa•Sb/$\perp_к$, S→Sa•Sb/a, S→•SaSb/b, S→•/b, S→•SaSb/a, S→•/a}.

Продолжив построения, получим 8 различных ситуаций от R_0 до R_7, граф взаимосвязи которых показан на рис. 4.7.

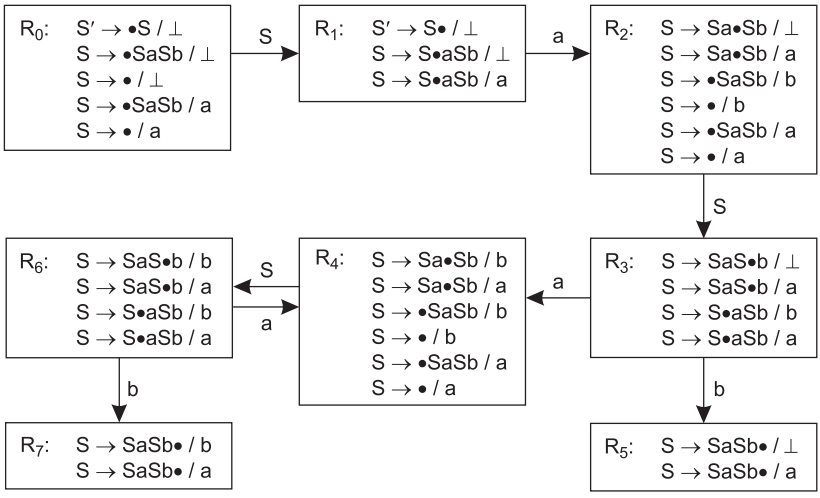

Рис. 4.7. Пример графа взаимосвязи ситуаций для LR(1)-грамматики

На основе найденной последовательности ситуаций можно построить управляющую таблицу распознавателя для LR(1)-грамматики . Поскольку при построении управляющей таблицы не возникает противоречий, то рассмотренная грамматика является LR(1)-грамматикой. Управляющая таблица для нее приведена в табл. 4.3.

Таблица 4.3. Пример управляющей таблицы для LR(1)-грамматики

№	Стек	Действия			Переходы		
		a	b	$\perp_к$	a	B	S
0	⊥н	свертка, 3		свертка, 3			1
1	S	сдвиг		успех, 1	2		
2	Sa	свертка, 3	свертка, 3				3
3	SaS	сдвиг	сдвиг		4	5	
4	SaSa	свертка, 3	свертка, 3				6
5	SaSb	свертка, 2		свертка, 2			
6	SaSaS	сдвиг	сдвиг		4	7	
7	SaSaSb	свертка, 2	свертка, 2				

Колонка *«Стек»*, присутствующая в таблице, не нужна для распознавателя. Она введена для пояснения состояния стека автомата. Пустые клетки в таблице соот-

ветствуют состоянию *«ошибка»*. Правила грамматики пронумерованы от 1 до 3. Колонка *«Действия»* в таблице содержит перечень действий, соответствующих текущему входному символу , обозреваемому считывающей головкой расширенного МП-автомата.

Рассмотрим примеры распознавания цепочек языка, заданного этой грамматикой. Конфигурацию расширенного МП-автомата будем отображать в виде трех компонентов, как это было сделано для LR(0)-грамматик.

Разбор цепочки `abababb`:

1. `(abababb⊥`$_{к}$`,{⊥`$_{н}$`,0},λ).`
2. `(abababb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1},3).`
3. `(bababb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2},3).`
4. `(bababb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2}{S,3},3,3).`
5. `(ababb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2}{S,3}{b,5},3,3).`
6. `(ababb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1},3,3,2).`
7. `(babb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2},3,3,2).`
8. `(babb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2}{S,3},3,3,2,3).`
9. `(abb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2}{S,3}{b,5},3,3,2,3).`
10. `(abb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1},3,3,2,3,2).`
11. `(bb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2},3,3,2,3,2).`
12. `(bb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2}{S,3},3,3,2,3,2,3).`
13. `(b⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2}{S,3}{b,5},3,3,2,3,2,3).`
14. `ошибка`, нет данных для `b` в строке 5.

Разбор цепочки `aababb`:

1. `(aababb⊥`$_{к}$`,{⊥`$_{н}$`,0},λ).`
2. `(aababb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1},3).`
3. `(ababb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2},3).`
4. `(ababb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2}{S,3},3,3).`
5. `(babb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2}{S,3}{a,4},3,3).`
6. `(babb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2}{S,3}{a,4}{S,6},3,3,3).`
7. `(abb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2}{S,3}{a,4}{S,6}{b,7},3,3,3).`
8. `(abb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2}{S,3},3,3,3,2).`
9. `(bb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2}{S,3}{a,4},3,3,3,2).`
10. `(bb⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2}{S,3}{a,4}{S,6},3,3,3,2,3).`
11. `(b⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2}{S,3}{a,4}{S,6}{b,7},3,3,3,2,3).`
12. `(b⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2}{S,3},3,3,3,2,3,2).`
13. `(⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1}{a,2}{S,3}{b,5},3,3,3,2,3,2).`
14. `(⊥`$_{к}$`,{⊥`$_{н}$`,0}{S,1},3,3,3,2,3,2,2).`
15. `(⊥`$_{к}$`,{⊥`$_{н}$`,0}{S',*},3,3,3,2,3,2,2,1)` — разбор завершен.

Соответствующая цепочка вывода будет иметь вид (используется правосторонний вывод) S'⇒ S⇒SaSb⇒SaSaSbb⇒SaSabb⇒SaSaSbabb⇒SaSababb⇒Saababb⇒aababb.

Сложности построения LR(1)-распознавателей

Класс языков, заданных LR(1)-грамматиками , является самым широким классом КС-языков, допускающим построение линейного восходящего распознавателя. Все детерминированные КС-языки могут быть заданы с помощью LR(1)-грамматик. Однако не всякая однозначная КС-грамматика является LR(1)-грамматикой. И если язык задан грамматикой, не являющейся LR(1)-грамматикой, то в общем случае не существует алгоритма преобразования ее к виду LR(1).

Это проблема общая для всех КС-грамматик. Но для LR(1)-грамматик можно с уверенностью сказать, что если не удалось преобразовать произвольную КС-грамматику к виду LR(1)-грамматики, то нет большого смысла пытаться преобразовать ее к виду LR(k)-грамматики с k>1.

Дело в том, что даже для LR(1)-грамматик управляющая таблица распознавателя имеет значительный объем, а для ее построения необходимо рассмотреть большое количество взаимосвязанных ситуаций. В общем случае задача имеет комбинаторную сложность, и для реальной LR(1)-грамматики, содержащей несколько десятков правил и несколько десятков терминальных символов, множество рассматриваемых ситуаций будет достаточно велико (сотни возможных ситуаций плюс все взаимосвязи между ними). Сложность построения управляющих таблиц для LR(k)-грамматик при k>1 будет увеличиваться с ростом k в геометрической прогрессии пропорционально количеству терминальных символов грамматики. Поэтому LR(k)-грамматики с k>1 практического применения не имеют.

Например, неоднократно рассмотренная выше грамматика арифметических выражений для символов a и b G({+,-,/,*,a,b}, {S',S,T,E}, P, S'):

```
P:    S' → S
        S → S+T | S-T | T
        T → T*E | T/E | E
        E → (S) | a | b
```

является LR(1)-грамматикой, но последовательность ситуаций для нее слишком велика, чтобы ее можно было проиллюстрировать в данном учебник (желающие читатели могут построить последовательность ситуаций и управляющую таблицу для данной грамматики, используя приведенный выше алгоритм).

Высокая трудоемкость построения управляющей таблицы для LR(1)-грамматик послужила причиной тому, что для реализации линейных восходящих распознавателей на основе алгоритма «сдвиг—свертка» были предложены другие классы грамматик, такие как SLR(k)-грамматики и LALR(k)-грамматики, грамматики предшествования, грамматики с ограниченным правым контекстом (ОПК) и другие. Не все из этих классов КС-грамматик определяют столь широкий класс КС-языков, как LR(1)-грамматики, но многие из них очень удобны и практичны для построения синтаксических анализаторов.

Далее в этой главе будут рассмотрены еще два класса КС-грамматик — SLR(1)-грамматики и LALR(1)-грамматики, являющиеся незначительной модификацией

LR(1)-грамматик. Отдельный раздел посвящен грамматикам предшествования. Остальные грамматики из всего множества классов КС-грамматик в данном учебнике не рассматриваются, их можно найти в [4 т.2, 5].

SLR(1)- и LALR(1)-грамматики

Распознаватели для обоих этих классов грамматик представляют собой модификацию распознавателя для LR(1)-грамматик.

Синтаксический разбор для SLR(1)-грамматик

Идея распознавателей для SLR(k)-грамматик заключается в том, чтобы сократить множество рассматриваемых ситуаций и упростить построение управляющей таблицы **T**. Ведь увеличение количества рассматриваемых ситуаций в LR(1)-грамматике по сравнению с LR(0)-грамматикой происходит из-за того, что в каждом правиле для каждой ситуации учитывается правый контекст. В то же время реально рассматривать правый контекст необходимо не во всех ситуациях, а только в тех, где возникают конфликты в алгоритме построения таблицы **T**.

Все SLR(k)-грамматики для всех k>1 образуют класс SLR-грамматик (k>1, поскольку SLR(0)-грамматика совпадает по определению с LR(0)-грамматикой). Название SLR-грамматик происходит из самой идеи их построения — SLR означает «Simple LR», то есть упрощенные LR-грамматики.

В распознавателях на основе SLR(k)-грамматик построение управляющей таблицы **T** выполняется следующим образом:

- строится последовательность ситуаций, соответствующая LR(0)-грамматике ;
- на основе построенной последовательности ситуаций заполняется управляющая таблица **T**, причем, если при заполнении таблицы возникают конфликты, то для их разрешения используют правый контекст длиной k и множества символов FIRST(k,α) и FOLLOW(k,A) для правил вида A$\rightarrow\alpha$.

Здесь множества FIRST(k,α) и FOLLOW(k,A) представляют собой, соответственно, множество k первых терминальных символов цепочки вывода изα и k первых терминальных символов, которые могут следовать в цепочках вывода за символом A. Эти множества были определены выше в разделе, посвященном LL(1)-грамматикам.

На практике чаще всего применяются SLR(1)-грамматики , поскольку для них построение множеств символов FIRST(1,α) и FOLLOW(1,A) выполняется элементарно просто.

Управляющая таблица **T** для распознавателя на основе SLR(1)-грамматики строится на основе последовательности ситуаций так же, как и для LR(0)-грамматики, но графа «*Действия*» в этой таблице разбита на несколько подграф по количеству возможных терминальных символов, аналогично LR(1)-грамматике . Для всех ситуаций в последовательности от $\mathbf{R}_0$ до $\mathbf{R}_{N-1}$ с учетом правого контекста выполняются следующие действия:

- если ситуация $\mathbf{R}_i$ содержит правило вида S $\rightarrow\gamma\bullet$, $\gamma\in(\mathbf{VN}\cup\mathbf{VT})^*$, где S — целевой символ грамматики, то в графе *«Действия»* строки T_i таблицы **T** должно быть записано *«успех»* для всех символов a$\in$ FOLLOW(1,S) — фактически таким символом является символ конца строки $\perp_к$;

- если ситуация $\mathbf{R}_i$ содержит правило вида $A \to \gamma\bullet$, где $A \in \mathbf{VN}$, $\gamma \in (\mathbf{VN} \cup \mathbf{VT})^*$, A не является целевым символом и грамматика содержит правило вида $A \to \gamma$ с номером *m*, то в графе *«Действия»* строки T_i таблицы **T** должно быть записано число *m* (*свертка* по правилу с номером *m*) для всех символов $a \in$ FOLLOW(1,A);
- если ситуация $\mathbf{R}_i$ содержит правила вида $A \to \gamma\bullet x\beta$, где $A \in \mathbf{VN}$, а x — произвольный символ ($x \in \mathbf{VN} \cup \mathbf{VT}$), $\gamma,\beta \in (\mathbf{VN} \cup \mathbf{VT})^*$, то в графе *«Действия»* строки T_i таблицы **T** должно быть записано *«сдвиг»* для всех символов $a \in$ FIRST(1,x)[1];
- если ситуация $\mathbf{R}_j$ проистекает из ситуации $\mathbf{R}_i$ и связана с нею через символ x ($x \in \mathbf{VN} \cup \mathbf{VT}$), то в графе *«Переход»*, соответствующей символу x, строки T_i таблицы **T** должно быть записано число *j*.

Если удается непротиворечивым образом заполнить управляющую таблицу **T**, то рассматриваемая грамматика является SLR(1)-грамматикой. Иначе, когда при заполнении таблицы возникают противоречия, грамматика не является SLR(1)-грамматикой и для нее нельзя построить распознаватель типа SLR(1) [4 т.2, 5, 58].

Например, рассмотрим грамматику арифметических выражений для символов a и b `G({+,-,/,*,a,b}, {S',S,T,E}, P, S')`:

`P:`

```
S' → S
S → S+T | S-T | T
T → T*E | T/E | E
E → (S) | a | b
```

Если начать строить множество ситуаций LR(0)-грамматики для этой грамматики, то начальная ситуация будет выглядеть так: $\mathbf{R}_0$=`{S'→•S, S→•S+T, S→•S-T, S→•T, T→•T*E, T→•T/E, T→•E, E→•(S), E→•a, E→•b}`. Из нее по символу `S` можно получить ситуацию $\mathbf{R}_1$=`{S'→S•, S→S•+T, S→S•-T}`, а по символу `T` — ситуацию $\mathbf{R}_2$=`{S→T•, T→T•*E, T→T•/E}`.

При построении управляющей таблицы **T** для этой грамматики на основе алгоритма для LR(0)-грамматик в состояниях $\mathbf{R}_1$ и $\mathbf{R}_2$ возникнут противоречия, поэтому данная грамматика не является LR(0)-грамматикой. Однако если воспользоваться алгоритмом для SLR(1)-грамматики, то противоречий удается избежать:

- для ситуации $\mathbf{R}_1$, приняв во внимание, что `FOLLOW(1,S')`=$\{\perp_к\}$, а `FIRST(1,+)`=`{+}` и `FIRST(1,-)`=`{-}`, получим, что для правого контекста $\perp_к$ («конец строки») необходимо сигнализировать об успехе, а для правых контекстов `+` и `-` — выполнять сдвиг (остальные варианты правого контекста в этой ситуации предполагают ошибку);
- для ситуации $\mathbf{R}_2$ найдем `FOLLOW(1,S)`, получим `FOLLOW(1,S)={+,-,)}`, `FIRST(1,*)={*}` и `FIRST(1,/)={/}`, получим, что для правых контекстов `+,-,)` необходимо выполнять свертку по правилу `S→T`, а для правых контекстов `*` и `/` — сдвиг (остальные варианты правого контекста в этой ситуации предполагают ошибку).

Построив полностью последовательность ситуаций для этой грамматики и управляющую таблицу **T** на основе данной последовательности, можно убедиться, что рассмотренная грамматика является SLR(1)-грамматикой.

[1] Напоминаем, что для терминального символа b справедливо FIRST(1,b) = {b}.

Ограничения SLR(1)-грамматик

Языки, заданные SLR(1)-грамматиками, представляют собой более широкий класс КС-языков, чем языки, заданные LR(0)-грамматиками .

С другой стороны, SLR(1)-грамматики позволяют сократить объем управляющей таблицы **T** распознавателя за счет того, что количество строк в ней соответствует количеству состояний LR(0)-грамматики, которое зачастую существенно меньше, чем количество состояний LR(1)-грамматики. В то же время они используют более простой алгоритм исключения конфликтных ситуаций при построении этой таблицы, чем LR(1)-грамматики. В этом их преимущество перед LR(1)-грамматиками .

Однако не всякая LR(1)-грамматика является SLR(1)-грамматикой. Оказывается, что класс языков, заданных SLR(1)-грамматиками, у́же, чем класс языков, заданных LR(1)-грамматиками. Это значит, что он у́же, чем класс ДКС-языков.

ВНИМАНИЕ

Не всякий детерминированный КС-язык может быть задан SLR(1)-грамматикой.

Возьмем рассмотренную ранее КС-грамматику `G({a,b}, {S}, {S→SaSb|λ}, S)` с пополненной грамматикой `G'({a,b}, {S,S'}, {S'→S,S→SaSb|λ}, S')`.

Как было показано выше, начальная ситуация для этой грамматики имеет вид $R_0=\{S'\to\bullet S, S\to\bullet SaSb, S\to\bullet\}$. При построении управляющей таблицы **T** на основе алгоритма для LR(0)-грамматики возникают противоречия. Если попытаться воспользоваться алгоритмом для SLR(1)-грамматики, то необходимо вычислить множества `FIRST(1,S)` и `FOLLOW(1,S)`. Получим `FIRST(1,S)={a}` и `FOLLOW(1,S)={a,b}`. Оказывается, что для правого контекста `a` конфликта избежать не удается: с одной стороны, в этой ситуации для него необходимо выполнять сдвиг , поскольку `S→•SaSb` входит в ситуацию и $a\in$`FIRST(1,S)`, но с другой стороны, для него надо выполнять свертку по правилу `S→λ`, поскольку `S→•` входит в ситуацию, и $a\in$`FOLLOW(1,S)`. Следовательно, эта грамматика не является SLR(1)-грамматикой.

Для того чтобы расширить класс языков, заданных упрощенными LR(1)-грамматиками, был предложен еще один класс грамматик — LALR(1)-грамматики.

Синтаксический разбор для LALR(1)-грамматик

Идея распознавателей для LALR(k)-грамматик та же самая, что и для SLR(k)-грамматик: сократить множество рассматриваемых ситуаций до множества ситуаций LR(0)-грамматики и упростить построение управляющей таблицы **T**, но при этом найти алгоритм разрешения конфликтов при построении для ситуаций, в которых эти конфликты возникают, на основании правого контекста длиной k.

Все LALR(k)-грамматики для всех k>1 образуют класс LALR-грамматик (k>1, поскольку LALR(0)-грамматика совпадает по определению с LR(0)-грамматикой). Название LALR-грамматик происходит из идеи, которая используется для разрешения конфликтов — LALR означает «Look Ahead», то есть LR-грамматики с «заглядыванием вперед».

На практике используются только LALR(1)-грамматики, так как применяемый для них метод разрешения конфликтов достаточно сложно распространить на LALR(k)-грамматики с k>1.

В распознавателях на основе LALR(1)-грамматик построение управляющей таблицы **T** выполняется так же, как и для SLR(1)-грамматик, но разрешение конфликтов выполняется не на основании множества FOLL OW(1,A), а путем анализа правого контекста для каждого конфликтного правила в ситуации на основании того, как это правило попало в данную ситуацию. Если анализ контекста допускает выполнение свертки для некоторого правого контекста, то для этого правого контекста в управляющую таблицу **T** записывается действие *«свертка»* на основании конфликтного правила. Действие *«сдвиг»* записывается для всех остальных символов правого контекста, которые входят в множество FIRST(1, β) правил вида A →α•β и не допускают выполнение свертки (более подробное описание анализа правого контекста для LALR(1)-грамматик можно найти в [4 т.2, 5, 58].

Возьмем КС-грамматику `G({a,b}, {S}, {S→SaSb|λ}, S)` с пополненной грамматикой `G'({a,b}, {S,S'}, {S'→S,S→SaSb|λ}, S')`.

Построим для нее последовательность ситуаций:

R_0 = {S'→•S, S → •SaSb, S → •};

R_1 = {S'→S•, S→S•aSb} (порождена из R_0 по символу S);

R_2 = {S→Sa•Sb, S→•SaSb, S→•} (порождена из R_1 и из R_3 по символу a);

R_3 = {S→SaS•b, S→S•aSb} (порождена из R_2 по символу S);

R_4 = {S→SaSb•} (порождена из R_3 по символу b).

Конфликты присутствуют в ситуациях $\mathbf{R}_1$ и $\mathbf{R}_2$.

Рассмотрим ситуацию $\mathbf{R}_1$. В ней возможна свертка по правилу S'→S• или сдвиг в соответствии с правилом S→S•aSb. Если проанализировать правило S'→S•, то видно, что оно появилось в ситуации $\mathbf{R}_1$ из правила S'→•S ситуации $\mathbf{R}_0$, следовательно, правым контекстом для этого правила может быть только символ $\perp_к$. Тогда для ситуации $\mathbf{R}_1$ имеем, что для правого контекста $\perp_к$ должна выполняться свертка по правилу S'→S (а поскольку в правой части правила стоит целевой символ S', то это соответствует действию *«успех»*), а для правого контекста a — сдвиг (FIRST(1,aSb)={a}).

Аналогично, для ситуации $\mathbf{R}_2$ возможна свертка по правилу S→• или сдвиг в соответствии с правилами S→Sa•Sb и S→•SaSb. Если проанализировать правило S→•, то видно, что оно могло появиться в ситуации $\mathbf{R}_2$ из правила S→Sa•Sb этой же ситуации — тогда правым контекстом для него должен быть символ b, но это же правило могло появиться в ситуации $\mathbf{R}_2$ также на основании правила S→•SaSb из этой же ситуации — тогда правым контекстом для него будет символ a. Получаем, что в ситуации $\mathbf{R}_2$ для правых контекстов a и b должна выполняться свертка по правилу S→λ. Сдвиг в этой ситуации выполняться не должен, поскольку FIRST(1,Sb)=FIRST(1,SaSb)={a,b}, но для символов a и b уже выполняется свертка.

Все конфликтные ситуации удалось разрешить. Таким образом, данная грамматика является LALR(1)-грамматикой.

Построим управляющую таблицу для данной грамматики, используя найденную последовательность ситуаций и правила разрешения конфликтов, полученные для ситуаций $\mathbf{R}_1$ и $\mathbf{R}_2$ (табл. 4.4).

Как и в ранее построенных таблицах, колонка *«Стек»* не нужна распознавателю, а введена для пояснения состояния стека. Пустые клетки в таблице соответствуют состоянию *«ошибка»*. Правила грамматики пронумерованы от 1 до 3. Колонка *«Дей-*

ствия» в таблице содержит перечень действий, соответствующих текущему входному символу, обозреваемому считывающей головкой автомата.

Таблица 4.4. Пример управляющей таблицы для LALR(1)-грамматики

№	Стек	Действия			Переходы		
		A	b	$\perp_к$	a	b	S
0	⊥н	свертка, 3		свертка, 3			1
1	S	сдвиг		успех, 1	2		
2	Sa	свертка, 3	свертка, 3				3
3	SaS	сдвиг	сдвиг		2	4	
4	SaSb	свертка, 2	свертка, 2	свертка, 2			

Рассмотрим примеры распознавания цепочек языка, заданного этой грамматикой. Конфигурацию расширенного МП-автомата будем отображать в виде трех компонентов, как это было сделано выше.

Разбор цепочки ababab:

1. (abababb$\perp_к$,{$\perp_н$,0},λ).
2. (abababb$\perp_к$,{$\perp_н$,0}{S,1},3).
3. (bababb$\perp_к$,{$\perp_н$,0}{S,1}{a,2},3).
4. (bababb$\perp_к$,{$\perp_н$,0}{S,1}{a,2}{S,3},3,3).
5. (ababb$\perp_к$,{$\perp_н$,0}{S,1}{a,2}{S,3}{b,4},3,3).
6. (ababb$\perp_к$,{$\perp_н$,0}{S,1},3,3,2).
7. (babb$\perp_к$,{$\perp_н$,0}{S,1}{a,2},3,3,2).
8. (babb$\perp_к$,{$\perp_н$,0}{S,1}{a,2}{S,3},3,3,2,3).
9. (abb$\perp_к$,{$\perp_н$,0}{S,1}{a,2}{S,3}{b,4},3,3,2,3).
10. (abb$\perp_к$,{$\perp_н$,0}{S,1},3,3,2,3,2).
11. (bb$\perp_к$,{$\perp_н$,0}{S,1}{a,2},3,3,2,3,2).
12. (bb$\perp_к$,{$\perp_н$,0}{S,1}{a,2}{S,3},3,3,2,3,2,3).
13. (b$\perp_к$,{$\perp_н$,0}{S,1}{a,2}{S,3}{b,4},3,3,2,3,2,3).
14. (b$\perp_к$,{$\perp_н$,0}{S,1},3,3,2,3,2,3,2).
15. Ошибка, нет действий для b в строке 1.

Разбор цепочки aababb:

1. (aababb$\perp_к$,{$\perp_н$,0},λ).
2. (aababb$\perp_к$,{$\perp_н$,0}{S,1},3).
3. (ababb$\perp_к$,{$\perp_н$,0}{S,1}{a,2},3).
4. (ababb$\perp_к$,{$\perp_н$,0}{S,1}{a,2}{S,3},3,3).
5. (babb$\perp_к$,{$\perp_н$,0}{S,1}{a,2}{S,3}{a,2},3,3).
6. (babb$\perp_к$,{$\perp_н$,0}{S,1}{a,2}{S,3}{a,2}{S,3},3,3,3).
7. (abb$\perp_к$,{$\perp_н$,0}{S,1}{a,2}{S,3}{a,2}{S,3}{b,4},3,3,3).
8. (abb$\perp_к$,{$\perp_н$,0}{S,1}{a,2}{S,3},3,3,3,2).

9. (bb⊥к,{⊥н,0}{S,1}{a,2}{S,3}{a,2},3,3,3,2).
10. (bb⊥к,{⊥н,0}{S,1}{a,2}{S,3}{a,2}{S,3},3,3,3,2,3).
11. (b⊥к,{⊥н,0}{S,1}{a,2}{S,3}{a,2}{S,3}{b,4},3,3,3,2,3).
12. (b⊥к,{⊥н,0}{S,1}{a,2}{S,3},3,3,3,2,3,2).
13. (⊥к,{⊥н,0}{S,1}{a,2}{S,3}{b,4},3,3,3,2,3,2).
14. (⊥к,{⊥н,0}{S,1},3,3,3,2,3,2,2).
15. (⊥к,{⊥н,0}{S',*},3,3,3,2,3,2,2,1) — разбор завершен.

Соответствующая цепочка вывода будет иметь вид (используется правосторонний вывод)

S'⇒S⇒SaSb⇒SaSaSbb⇒SaSabb⇒SaSaSbabb⇒SaSababb⇒Saababb⇒aababb.

Можно убедиться, что вывод, найденный с помощью распознавателя на основе LALR(1)-грамматики, соответствует выводу, найденному распознавателем на основе LR(1)-грамматики.

Особенности LALR(1)-грамматик

Как и SLR(1)-грамматики, LALR(1)-грамматики позволяют сократить объем управляющей таблицы **T** распознавателя за счет того, что количество строк в ней соответствует количеству состояний LR(0)-грамматики . Но по сравнению с SLR(1)-грамматиками они используют более сложный алгоритм исключения конфликтных ситуаций при построении этой таблицы. В целом построить распознаватель на основе LALR(1)-грамматики проще, чем распознаватель на основе LR(1)-грамматики, но немного сложнее, чем распознаватель на основе SLR(1)-грамматики.

Класс LALR(1)-грамматик шире, чем класс SLR(1)-грамматик. Например, рассмотренная выше пополненная КС-грамматика G'({a,b}, {S,S'}, {S'→S, S→SaSb|λ}, S') является LALR(1)-грамматикой, но не является SLR(1)-грамматикой. Однако важно, что и языки, заданные LALR(1)-грамматиками, представляют собой более широкий класс КС-языков, чем языки, заданные SLR(1)-грамматиками.

ПРИМЕЧАНИЕ

Всякая SLR(1)-грамматика является LALR(1)-грамматикой, но не наоборот.

Доказано, что любой ДКС-язык может быть задан LALR(1)-грамматикой, а это значит, что класс языков, заданных LALR(1)-грамматиками, совпадает с классом языков, заданных LR(1)-грамматиками [4 т.2, 5]. Хотя известно, что класс LR(1)-грамматик шире, чем класс LALR(1)-грамматик.

ВНИМАНИЕ

Всякий язык, заданный LR(1)-грамматикой, может быть задан LALR(1)-грамматикой, но не всякая LR(1)-грамматика является LALR(1)-грамматикой.

LALR(1)-грамматики представляют собой более удобный инструмент для построения распознавателей, чем LR(1)-грамматики. Кроме того, этот класс грамматик достаточно мощный, чтобы на его основе можно было построить синтаксический анали-

затор для любого языка программирования, представляющего практический интерес (а такой интерес представляют только ДКС-языки). Однако этот класс грамматик у́же, чем класс LR(1)-грамматик, а значит, существуют LR(1)-грамматики, которые не являются LALR(1)-грамматиками. Как и для других классов КС-грамматик, не всегда удается найти преобразование, позволяющее построить LALR(1)-грамматику для языка, заданного LR(1)-грамматикой. В ряде случаев этот факт может ограничить применение распознавателей на основе LALR(1)-грамматик.

Но распознаватели на основе LALR(1)-грамматик имеют еще один недостаток по сравнению с распознавателями на основе LR(1)-грамматик. Дело в том, что метод анализа правого контекста, применяемый при построении распознавателя для LALR(1)-грамматики несколько ограничивает возможности распознавателя по обнаружению ошибок во входной цепочке символов. Ошибка, конечно же, будет обнаружена, но распознавателю на основе LALR(1)-грамматики потребуется для этого больше шагов, чем распознавателю на основе LR(1)-грамматики. Это значит, что ошибка будет найдена на более поздней стадии синтаксического анализа[1]. В таком случае у компилятора будет меньше возможностей по диагностике ошибки. Поэтому можно сказать, что распознаватели на основе LALR(1)-грамматик упрощают выполнение синтаксического анализа, но при этом уменьшают возможности компилятора по диагностике ошибок, по сравнению с распознавателями на основе LR(1)-грамматик.

Тем не менее, несмотря на указанные недостатки, распознаватели на основе LALR(1)-грамматик являются одним из самых мощных и эффективных средств для построения синтаксических анализаторов. Именно поэтому данный тип распознавателя используется при решении задачи автоматизации построения синтаксического анализатора.

Примеры построения LR(0)-, SLR(1)- и LALR(1)- распознавателей можно найти в книгах [5, 58].

Автоматизация построения синтаксических анализаторов (программа YACC)

При разработке различных прикладных программ часто возникает задача синтаксического разбора некоторого входного текста. Конечно, ее можно всегда решить, полностью самостоятельно построив соответствующий анализатор. И хотя задача выполнения синтаксического разбора встречается не столь часто, как задача выполнения лексического разбора, но все-таки и для ее решения были предложены соответствующие программные средства.

Автоматизированное построение синтаксических анализаторов может быть выполнено с помощью программы YACC.

Программа YACC (Yet Another Compiler Compiler) предназначена для построения синтаксического анализатора КС-языка. Анализируемый язык описывается с помощью грамматики в виде, близком форме Бэкуса—Наура (нормальная форма Бэкуса—Наура — НФБН). Результатом работы YACC является исходный текст программы синтаксического анализатора. Анализатор, который порождается YACC, реализует восходящий распознаватель на основе LALR(1)-грамматики [3, 9, 31, 55, 58].

[1] Этот факт можно заметить даже на примере разбора простейшей ошибочной входной цепочки, который был рассмотрен выше для LR(1)-грамматики, а затем и для LALR(1)-грамматики.

Как и программа LEX , служащая для автоматизации построения лексических анализаторов, программа Y ACC тесно связана с историей операционных систем типа UNIX. Эта программа входит в поставку многих версий ОС UNIX или Linux. Поэтому чаще всего результатом работы YACC является исходный текст синтаксического распознавателя на языке C. Однако существуют версии YACC, выполняющиеся под управлением ОС, отличных от UNIX, и порождающие исходный код на других языках программирования (например, Pascal).

Принцип работы YACC похож на принцип работы LEX: на вход поступает файл, содержащий описание грамматики заданного КС-языка, а на выходе получаем текст программы синтаксического распознавателя , который, естественно, можно дополнять и редактировать, как и любую другую программу на заданном языке программирования .

Исходная грамматика для YACC состоит из трех секций, разделенных двойным символом `%` — `%%`: секции описаний, секции правил, в которой описывается грамматика, и секции программ, содержимое которой просто копируется в выходной файл.

Например, ниже приведено описание простейшей грамматики для Y ACC, которая соответствует грамматике арифметических выражений , многократно использовавшейся в качестве примера в данном пособии:

```
%token a
%token b
%start e
%%
e : e '+' m | e '—' m | m ;
m : m '*' t | m '/' t | t ;
t : a | b | '(' e ')' ;
%%
```

Секция описаний содержит информацию о том, что идентификатор `a` является лексемой (терминальным символом) грамматики, а символ `e` — ее начальным нетерминальным символом.

Грамматика записана обычным образом — идентификаторы обозначают терминальные и нетерминальные символы; символьные константы типа `'+'` и `'—'` считаются терминальными символами. Символы `:`, `|`, `;` принадлежат к метаязыку YACC и читаются согласно НФБН «есть по определению», «или» и «конец правила» соответственно.

В отличие от LEX, который всегда способен построить лексический распознаватель, если входной файл содержит правильное регулярное выражение, Y ACC не всегда может построить распознаватель, даже если входной язык задан правильной КС-грамматикой. Ведь заданная грамматика может и не принадлежать к классу LALR(1)-грамматик. В этом случае YACC выдаст сообщение об ошибке (о наличии неразрешимого конфликта в LALR(1)-грамматике) при построении синтаксического анализатора. Т огда пользователь должен либо преобразовать грамматику , либо задать Y ACC некоторые дополнительные правила, которые могут облегчить построение анализатора. Например, YACC позволяет указать правила, явно задающие приоритет операций и порядок их выполнения (слева направо или справа налево).

С каждым правилом грамматики может быть связано действие, которое будет выполнено при свертке по данному правилу . Оно записывается в виде заключенной в фигурные скобки последовательности операторов языка, на котором порождается исходный текст программы распознавателя (обычно это язык C). Последовательность должна располагаться после правой части соответствующего правила. Т акже YACC позволяет управлять действиями, которые будут выполняться распознавателем в том случае, если входная цепочка не принадлежит заданному языку . Распознаватель имеет возможность выдать сообщение об ошибке, остановиться либо же продолжить разбор, предприняв некоторые действия, связанные с попыткой локализовать либо устранить ошибку во входной цепочке.

YACC в настоящее время не является единственным программным продуктом, предназначенным для автоматизации построения синтаксических анализаторов, так же как и LEX не является единственным программным продуктом для автоматизации построения лексических анализаторов. Сейчас на рынке средств разработки программного обеспечения присутствует целый ряд программных продуктов, ориентированных на решение такого рода задач. Некоторые из них распространяются бесплатно или же бесплатно, но с некоторыми ограничениями, другие являются коммерческими программными продуктами. Многие средства такого рода входят в состав ОС и систем программирования (стоит отметить, что LEX и YACC входят в состав ОС типа UNIX).

СОВЕТ

Если у читателей существует потребность в программном обеспечении для автоматизированного построения синтаксических анализаторов, автор прежде всего рекомендует обратиться для поиска таких средств во Всемирную сеть Интернет. По ключевым словам «YACC» и «LALR» можно получить достаточное количество полезных ссылок.

Более подробные сведения о программе автоматизированного построения синтаксических распознавателей YACC можно получить в [22, 31, 53].

Синтаксические распознаватели на основе грамматик предшествования

Общие принципы грамматик предшествования

Еще одним распространенным классом КС-грамматик, для которых можно построить восходящий распознаватель без возвратов, являются грамматики предшествования. Так же как и распознаватель для рассмотренных выше LR-грамматик, распознаватель для грамматик предшествования строится на основе алгоритма «сдвиг—свертка» («перенос—свертка»), который в общем виде был рассмотрен в разделе «Синтаксические распознаватели с возвратом».

Принцип организации распознавателя на основе грамматики предшествования исходит из того, что для каждой упорядоченной пары символов в грамматике устанавливается отношение, называемое отношением предшествования. В процессе разбора расширенный МП-автомат сравнивает текущий символ входной цепочки с одним из символов, находящихся на верхушке стека автомата. В процессе сравнения проверяется, какое из возможных отношений предшествования существует между этими двумя символами. В зависимости от найденного отношения выполняется либо сдвиг

либо свертка. При отсутствии отношения предшествования между символами алгоритм сигнализирует об ошибке.

Задача заключается в том, чтобы иметь возможность непротиворечивым образом определить отношения предшествования между символами грамматики. Если это возможно, то грамматика может быть отнесена к одному из классов грамматик предшествования.

Существует несколько видов грамматик предшествования. Они различаются по тому, какие отношения предшествования в них определены и между какими типами символов (терминальными или нетерминальными) могут быть установлены эти отношения. Кроме того, возможны незначительные модификации функционирования самого алгоритма «сдвиг—свертка» в распознавателях для таких грамматик (в основном на этапе выбора правила для выполнения свертки, когда возможны неоднозначности) [4 т.1,2, 5].

Выделяют следующие виды грамматик предшествования:

- простого предшествования ;
- расширенного предшествования ;
- слабого предшествования ;
- смешанной стратегии предшествования ;
- операторного предшествования .

Далее будут рассмотрены два наиболее простых и распространенных типа — грамматики простого и операторного предшествования.

Грамматики простого предшествования

Грамматикой простого предшествования называют такую приведенную КС-грамматику[1] $G(\mathbf{VN},\mathbf{VT},\mathbf{P},S)$, $\mathbf{V} = \mathbf{VT} \cup \mathbf{VN}$, в которой

- для каждой упорядоченной пары терминальных и нетерминальных символов выполняется не более чем одно из трех отношений предшествования:
  - $B_i \doteq B_j$ ($\forall B_i, B_j \in \mathbf{V}$), если и только если $\exists$ правило $A \to xB_iB_jy \in \mathbf{P}$, где $A \in \mathbf{VN}$, $x,y \in \mathbf{V}^*$;
  - $B_i \lessdot B_j$ ($\forall B_i, B_j \in \mathbf{V}$), если и только если $\exists$ правило $A \to xB_iDy \in \mathbf{P}$ и вывод $D \Rightarrow^* S_jz$, где $A,D \in \mathbf{VN}$, $x,y,z \in \mathbf{V}^*$;
  - $B_i \gtrdot B_j$ ($\forall B_i, B_j \in \mathbf{V}$), если и только если $\exists$ правило $A \to xCB_jy \in \mathbf{P}$ и вывод $C \Rightarrow^* zB_i$ или $\exists$ правило $A \to xCDy \in \mathbf{P}$ и выводы $C \Rightarrow^* zB_i$ и $D \Rightarrow^* B_jw$, где $A,C,D \in \mathbf{VN}$, $x,y,z,w \in \mathbf{V}^*$;
- различные правила в грамматике имеют разные правые части (в грамматике не должно быть двух различных правил с одной и той же правой частью).

Отношения $\doteq$, $\lessdot$ и $\gtrdot$ называют отношениями предшествования для символов. Отношение предшествования единственно для каждой упорядоченной пары символов. При этом между какими-либо двумя символами может и не быть отношения пред-

[1] Напоминаем, что КС-грамматика называется приведенной, если она не содержит циклов, бесплодных и недостижимых символов и λ-правил.

шествования. Отношения предшествования зависят от порядка, в котором стоят символы, и их нельзя путать со знаками математических операций — они не обладают ни свойством коммутативности, ни свойством ассоциативности. Например, если известно, что $B_i \cdot > B_j$, то не обязательно выполняется $B_j < \cdot B_i$ (поэтому знаки предшествования иногда помечают специальной точкой: $=\cdot$, $<\cdot$, $\cdot>$).

Для грамматик простого предшествования известны следующие полезные свойства:

- всякая грамматика простого предшествования является однозначной ;
- легко проверить, является или нет произвольная КС-грамматика грамматикой простого предшествования.

Как и для многих других классов грамматик, для грамматик простого предшествования не существует алгоритма, который мог бы преобразовать произвольную КС-грамматику в грамматику простого предшествования или доказать, что преобразование невозможно.

Метод предшествования основан на том факте, что отношения предшествования между двумя соседними символами распознаваемой строки соответствуют трем следующим вариантам:

- $B_i < \cdot B_{i+1}$, если символ B_{i+1} — крайний левый символ некоторой основы (это отношение между символами можно назвать «предшествует основе» или просто «предшествует»);
- $B_i \cdot > B_{i+1}$, если символ B_i — крайний правый символ некоторой основы (это отношение между символами можно назвать «следует за основой» или просто «следует»);
- $B_i = \cdot B_{i+1}$, если символы B_i и B_{i+1} принадлежат одной основе (это отношение между символами можно назвать «составляют основу»).

Исходя из этих соотношений выполняется разбор строки для грамматики предшествования.

Суть принципа такого разбора можно понять из рис. 4.8. На нем изображена входная цепочка символов $\alpha\gamma\beta\delta$ в тот момент, когда выполняется свертка цепочки γ. Символ a является последним символом подцепочки α, а символ b — первым символом подцепочки β. Тогда, если в грамматике удастся установить непротиворечивые отношения предшествования, в процессе выполнения разбора по алгоритму «сдвиг—свертка» можно всегда выполнять сдвиг до тех пор, пока между символом на верхушке стека и текущим символом входной цепочки существует отношение $<\cdot$ или $=\cdot$. А как только между этими символами будет обнаружено отношение $\cdot>$, так сразу надо выполнять свертку. Причем для выполнения свертки из стека надо выбирать все символы, связанные отношением $=\cdot$. То, что все различные правила в грамматике предшествования имеют различные правые части, гарантирует непротиворечивость выбора правила при выполнении свертки.

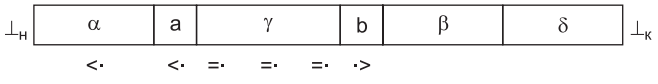

Рис. 4.8. Отношения между символами входной цепочки в грамматике предшествования

Таким образом, установление непротиворечивых отношений предшествования между символами грамматики в комплексе с несовпадающими правыми частями различных правил дает ответы на все вопросы, которые надо решить для организации работы алгоритма «сдвиг—свертка» без возвратов.

На основании отношений предшествования строят матрицу предшествования грамматики. Строки матрицы предшествования помечаются первыми (левыми) символами, столбцы — вторыми (правыми) символами отношений предшествования. В клетки матрицы на пересечении соответствующих столбца и строки помещаются знаки отношений. При этом пустые клетки матрицы говорят о том, что между данными символами нет ни одного отношения предшествования.

Матрицу предшествования грамматики сложно построить, опираясь непосредственно на определения отношений предшествования. У добнее воспользоваться двумя дополнительными множествами — множеством крайних левых и множеством крайних правых символов относительно нетерминальных символов грамматики G(**VN**,**VT**,**P**,S), **V** = **VT**∪**VN**. Эти множества определяются следующим образом:

- **L**(A)={X | ∃A⇒*Xα}, A∈ **VN**, X∈ **V**, α∈ **V*** — множество крайних левых символов относительно нетерминального символа A (цепочка α может быть пустой);
- **R**(A)={X |∃A⇒*αX}, A∈ **VN**, X∈ **V**, α∈ **V*** — множество крайних правых символов относительно нетерминального символа A.

Иными словами, множество крайних левых символов относительно нетерминального символа A — это множество всех крайних левых символов в цепочках, которые могут быть выведены из символа A. Аналогично, множество крайних правых символов относительно нетерминального символа A — это множество всех крайних правых символов в цепочках, которые могут быть выведены из символа A.

Тогда отношения предшествования можно определить так:

- $B_i = \cdot B_j$ (∀B_i,B_j∈ **V**), если ∃ правило A→αB_iB_jβ∈ **P**, где A∈ **VN**, α,β∈ **V***;
- $B_i < \cdot B_j$ (∀B_i,B_j∈ **V**), если ∃ правило A →αB_iDβ∈ **P** и B_j∈ **L**(D), где A,D ∈ **VN**, α,β∈ **V***;
- $B_i \cdot > B_j$ (∀B_i,B_j∈ **V**) , если ∃ правило A →αCB_jβ∈ **P** и B_i∈ **R**(C) или ∃ правило A→αCDβ∈ **P** и B_i∈ **R**(C), B_j∈ **L**(D), где A,C,D∈ **VN**, α,β∈ **V***.

Такое определение отношений удобнее на практике, так как не требует построения выводов, а множества **L**(A) и **R**(A) могут быть построены для каждого нетерминального символа A∈ **VN** грамматики G(**VN**,**VT**,**P**,S), **V**=**VT**∪**VN** по очень простому алгоритму.

Шаг 1. ∀A∈ **VN**:

R_0(A) = {X | A→βX, X∈ V, β∈ V*};

L_0(A) = {X | A→Xβ, X∈ V, β∈ V*}, i:=1.

Для каждого нетерминального символа A ищем все правила, содержащие A в левой части. Во множество **L**(A) включаем самый левый символ из правой части правил, а во множество **R**(A) — самый крайний правый символ из правой части. Переходим к шагу 2.

Шаг 2. $\forall A \in \mathbf{VN}$:

$R_i(A) = R_{i-1}(A) \cup R_{i-1}(B), \forall B \in (R_{i-1}(A) \cap VN)$;

$L_i(A) = L_{i-1}(A) \cup L_{i-1}(B), \forall B \in (L_{i-1}(A) \cap VN)$.

Для каждого нетерминального символа A: если множество $\mathbf{L}(A)$ содержит нетерминальные символы грамматики A', A", ..., то его надо дополнить символами, входящими в соответствующие множества $\mathbf{L}(A')$, $\mathbf{L}(A")$, ... и не входящими в $\mathbf{L}(A)$. Ту же операцию надо выполнить для $\mathbf{R}(A)$.

Шаг 3. Если $\exists\ A \in \mathbf{VN}$: $\mathbf{R}_i(A) \neq \mathbf{R}_{i-1}(A)$ или $\mathbf{L}_i(A) \neq \mathbf{L}_{i-1}(A)$, то i := i + 1 и вернуться к шагу 2, иначе построение закончено: $\mathbf{R}(A) = \mathbf{R}_i(A)$ и $\mathbf{L}(A) = \mathbf{L}_i(A)$.

Если на предыдущем шаге хотя бы одно множество, $\mathbf{L}(A)$ или $\mathbf{R}(A)$, для некоторого символа грамматики изменилось, то надо вернуться к шагу 2, иначе построение закончено.

После построения множеств $\mathbf{L}(A)$ и $\mathbf{R}(A)$ по правилам грамматики создается матрица предшествования. Матрицу предшествования дополняют символами $\perp_н$ и $\perp_к$ (начало и конец цепочки). Для них определены следующие отношения предшествования:

$\perp_н \lessdot X$, $\forall a \in \mathbf{V}$, если $\exists\ S \Rightarrow^* X\beta$, где $S \in \mathbf{VN}$, $\beta \in \mathbf{V}^*$, или (с другой стороны) если $X \in \mathbf{L}(S)$;

$\perp_к \cdot\!> X$, $\forall a \in \mathbf{V}$, если $\exists\ S \Rightarrow^* \beta X$, где $S \in \mathbf{VN}$, $\beta \in \mathbf{V}^*$, или (с другой стороны) если $X \in \mathbf{R}(S)$.

Здесь S — целевой символ грамматики.

Матрица предшествования служит основой для работы распознавателя языка, заданного грамматикой простого предшествования.

Алгоритм «сдвиг—свертка» для грамматики простого предшествования

Отношения предшествования служат для того, чтобы определить в процессе выполнения алгоритма, какое действие — сдвиг или свертка — должно выполняться на каждом шаге алгоритма, и однозначно выбрать цепочку для свертки. В начальном состоянии автомата считывающая головка обозревает первый символ входной цепочки, в стеке МП-автомата находится символ $\perp_н$ (начало цепочки), в конец цепочки помещен символ $\perp_к$ (конец цепочки). Символы $\perp_н$ и $\perp_к$ введены для удобства работы алгоритма, в язык, заданный исходной грамматикой, они не входят.

Разбор считается законченным, если считывающая головка автомата обозревает символ $\perp_к$ и при этом больше не может быть выполнена свертка. Решение о принятии цепочки зависит от содержимого стека. Автомат принимает цепочку , если в результате завершения алгоритма в стеке находятся начальный символ грамматики S и символ $\perp_н$. Выполнение алгоритма может быть прервано, если на одном из его шагов возникнет ошибка. Тогда входная цепочка не принимается.

Алгоритм состоит из следующих шагов.

Шаг 1. Поместить в верхушку стека символ $\perp_н$, считывающую головку — в начало входной цепочки символов.

Шаг 2. Сравнить с помощью отношения предшествования символ, находящийся на вершине стека (левый символ отношения), с текущим символом входной цепочки, обозреваемым считывающей головкой (правый символ отношения).

Шаг 3. Если имеет место отношение $<\cdot$ или $=\cdot$, то произвести сдвиг (перенос текущего символа из входной цепочки в стек и сдвиг считывающей головки на один шаг вправо) и вернуться к шагу 2. Иначе перейти к шагу 4.

Шаг 4. Если имеет место отношение $\cdot>$, то произвести свертку. Для этого надо найти на вершине стека все символы, связанные отношением $=\cdot$ («основу»), удалить эти символы из стека. Затем выбрать из грамматики правило, имеющее правую часть, совпадающую с основой, и поместить в стек левую часть выбранного правила (если символов, связанных отношением $=\cdot$, на верхушке стека нет, то в качестве основы используется один, самый верхний символ стека). Если правило, совпадающее с основой, найти не удалось, то необходимо прервать выполнение алгоритма и сообщить об ошибке, иначе, если разбор не закончен, вернуться к шагу 2.

Шаг 5. Если не установлено ни одно отношение предшествования между текущим символом входной цепочки и символом на верхушке стека, то надо прервать выполнение алгоритма и сообщить об ошибке.

Ошибка в процессе выполнения алгоритма возникает , когда невозможно выполнить очередной шаг — например, если не установлено отношение предшествования между двумя сравниваемыми символами (на шагах 2 и 4) или если не удается найти нужное правило в грамматике (на шаге 4). Тогда выполнение алгоритма прерывается.

Грамматики простого предшествования являются удобным механизмом для анализа входных цепочек КС-языков. Распознаватель для этого класса грамматик строить легче, чем для рассмотренных выше LR-грамматик. Однако класс языков, заданных грамматиками простого предшествования у ´же, чем класс языков, заданных LR-грамматиками. Отсюда ясно, что не всякий ДКС-язык может быть задан грамматикой простого предшествования.

У грамматик простого предшествования есть еще один недостаток — при большом количестве терминальных и нетерминальных символов в грамматике матрица предшествования будет иметь значительный объем (при этом значительная часть ее ячеек может оставаться пустой). Поиск в такой матрице может занять некоторое время, что существенно при работе распознавателя — фактически время поиска линейно зависит от числа символов грамматики, а объем матрицы — квадратично. Для того чтобы избежать хранения и обработки таких матриц, можно выполнить «линеаризацию матрицы предшествования». Тогда каждый раз, чтобы установить отношение предшествования между двумя символами, будет выполняться не поиск по матрице, а вычисление некой специально организованной функции. Вопросы линеаризации матриц предшествования здесь не рассматриваются, с применяемыми при этом методами можно ознакомиться в [3, 4 т.1,2, 18, 24].

Грамматики операторного предшествования

Операторной грамматикой называется КС-грамматика без λ-правил, в которой правые части всех правил не содержат смежных нетерминальных символов. Для операторной грамматики отношения предшествования можно задать на множестве терминальных символов (включая символы $\perp_н$ и $\perp_к$).

Грамматикой операторного предшествования называется операторная КС-грамматика G(**VN**,**VT**,**P**,S), **V**=**VT**∪**VN**, для которой выполняются следующие условия:

- для каждой упорядоченной пары терминальных символов выполняется не более чем одно из трех отношений предшествования:
  - a=·b, если и только если существует правило $A\to\alpha ab\beta\in\mathbf{P}$ или правило $A\to\alpha aCb\beta$, где $a,b\in\mathbf{VT}$, $A,C\in\mathbf{VN}$, $\alpha,\beta\in\mathbf{V}^*$;
  - a<·b, если и только если существует правило $A\to\alpha aC\beta\in\mathbf{P}$ и вывод $C\Rightarrow^*b\gamma$ или вывод $C\Rightarrow^*Db\gamma$, где $a,b\in\mathbf{VT}$, $A,C,D\in\mathbf{VN}$, $\alpha,\beta,\gamma\in\mathbf{V}^*$;
  - a·>b, если и только если существует правило $A\to\alpha Cb\beta\in\mathbf{P}$ и вывод $C\Rightarrow^*\gamma a$ или вывод $C\Rightarrow^*\gamma aD$, где $a,b\in\mathbf{VT}$, $A,C,D\in\mathbf{VN}$, $\alpha,\beta,\gamma\in\mathbf{V}^*$[1].
- различные порождающие правила имеют разные правые части, λ-правила отсутствуют.

Отношения предшествования для грамматик операторного предшествования определены таким образом, что для них выполняется еще одна особенность — правила грамматики операторного предшествования не могут содержать двух смежных нетерминальных символов в правой части. Т о есть в грамматике операторного предшествования $G(\mathbf{VN},\mathbf{VT},\mathbf{P},S)$, $\mathbf{V}=\mathbf{VT}\cup\mathbf{VN}$ не может быть ни одного правила вида $A\to\alpha BC\beta$, где $A,B,C\in\mathbf{VN}$, $\alpha,\beta\in\mathbf{V}^*$.

Для грамматик операторного предшествования также известны следующие свойства:

- всякая грамматика операторного предшествования задает ДКС-язык(но не всякая грамматика операторного предшествования при этом является однозначной!);
- легко проверить, является или нет произвольная КС-грамматика грамматикой операторного предшествования.

Как и для многих других классов грамматик, для грамматик операторного предшествования не существует алгоритма, который мог бы преобразовать произвольную КС-грамматику в грамматику операторного предшествования или доказать, что преобразование невозможно.

Принцип работы распознавателя для грамматики операторного предшествования аналогичен грамматике простого предшествования, но отношения операторного предшествования проверяются в процессе разбора только между терминальными символами.

Для грамматики данного вида на основе установленных отношений предшествования также строится матрица предшествования, но она содержит только терминальные символы грамматики.

Для построения этой матрицы удобно ввести множества крайних левых и крайних правых терминальных символов относительно нетерминального символа A — $\mathbf{L}^t(A)$ или $\mathbf{R}^t(A)$:

- $\mathbf{L}^t(A)=\{t\mid\exists A\Rightarrow^*t\gamma$ или $\exists A\Rightarrow^*Ct\gamma\}$, где $t\in\mathbf{VT}$, $A,C\in\mathbf{VN}$, $\gamma\in\mathbf{V}^*$;
- $\mathbf{R}^t(A)=\{t\mid\exists A\Rightarrow^*\gamma t$ или $\exists A\Rightarrow^*\gamma tC\}$, где $t\in\mathbf{VT}$, $A,C\in\mathbf{VN}$, $\gamma\in\mathbf{V}^*$.

[1] В литературе отношения операторного предшествования иногда обозначают другими символами, отличными от «< ·», «·>» и «= ·», чтобы не путать их с отношениями простого предшествования. Например, встречаются обозначения «<˚», «˚>» и «=˚». В данном пособии путаница исключена, поэтому будут использоваться одни и те же обозначения, хотя, по сути, отношения предшествования несколько различны.

Тогда определения отношений операторного предшествования будут выглядеть так:

- a=·b, если ∃ правило $A \to \alpha ab\beta \in \mathbf{P}$ или правило $A \to \alpha aCb\beta$, где $a,b \in \mathbf{VT}$, $A,C \in \mathbf{VN}$, $\alpha,\beta \in \mathbf{V}^*$;
- a<·b, если ∃ правило $A \to \alpha aC\beta \in \mathbf{P}$ и $b \in \mathbf{L}^t(C)$, где $a,b \in \mathbf{VT}$, $A,C \in \mathbf{VN}$, $\alpha,\beta \in \mathbf{V}^*$;
- a·>b, если ∃ правило $A \to \alpha Cb\beta \in \mathbf{P}$ и $a \in \mathbf{R}^t(C)$, где $a,b \in \mathbf{VT}$, $A,C \in \mathbf{VN}$, $\alpha,\beta \in \mathbf{V}^*$.

Для нахождения множеств $\mathbf{L}^t(A)$ и $\mathbf{R}^t(A)$ предварительно необходимо выполнить построение множеств $\mathbf{L}(A)$ и $\mathbf{R}(A)$, как это было рассмотрено ранее. Далее для построения $\mathbf{L}^t(A)$ и $\mathbf{R}^t(A)$ используется следующий алгоритм.

Шаг 1. $\forall A \in \mathbf{VN}$:

$\mathbf{R}^t_0(A) = \{t \mid A \to \beta tB$ или $A \to \beta t,\ t \in \mathbf{VT},\ B \in \mathbf{VN},\ \beta \in \mathbf{V}^*\}$;

$\mathbf{L}^t_0(A) = \{t \mid A \to Bt\beta$ или $A \to t\beta,\ t \in \mathbf{VT},\ B \in \mathbf{VN},\ \beta \in \mathbf{V}^*\}$.

Для каждого нетерминального символа A ищем все правила, содержащие A в левой части. Во множество $\mathbf{L}^t_0(A)$ включаем самый левый терминальный символ из правой части правил, игнорируя нетерминальные символы, а во множество $\mathbf{R}^t_0(A)$ — самый крайний правый терминальный символ из правой части правил. Переходим к шагу 2.

Шаг 2. $\forall A \in \mathbf{VN}$:

$\mathbf{R}^t(A) = \mathbf{R}^t_0(A) \cup \mathbf{R}^t_0(B),\ \forall B \in (\mathbf{R}(A) \cap \mathbf{VN})$;

$\mathbf{L}^t(A) = \mathbf{L}^t_0(A) \cup \mathbf{L}^t_0(B),\ \forall B \in (\mathbf{L}(A) \cap \mathbf{VN})$.

Для каждого нетерминального символа A: если множество $\mathbf{L}(A)$ содержит нетерминальные символы грамматики A', A", ..., то его надо дополнить символами, входящими в соответствующие множества $\mathbf{L}^t(A')$, $\mathbf{L}^t(A")$, ... и не входящими в $\mathbf{L}^t(A)$. Ту же операцию надо выполнить для множеств $\mathbf{R}(A)$ и $\mathbf{R}^t(A)$. Построение закончено.

Для практического использования матрицу предшествования дополняют символами $\perp_н$ и $\perp_к$ (начало и конец цепочки). Для них определены следующие отношения предшествования:

$\perp_н$ <· a, $\forall a \in \mathbf{VT}$, если $\exists S \Rightarrow^* a\alpha$ или $\exists S \Rightarrow^* Ca\alpha$, где $S,C \in \mathbf{VN}$, $\alpha \in \mathbf{V}^*$ или если $a \in \mathbf{L}^t(S)$;

$\perp_к$ ·> a, $\forall a \in \mathbf{VT}$, если $\exists S \Rightarrow^* \alpha a$ или $\exists S \Rightarrow^* \alpha aC$, где $S,C \in \mathbf{VN}$, $\alpha \in \mathbf{V}^*$ или если $a \in \mathbf{R}^t(S)$.

Здесь S — целевой символ грамматики.

Матрица предшествования служит основой для работы распознавателя языка, заданного грамматикой операторного предшествования.

ПРИМЕЧАНИЕ

Размер матрицы для грамматики операторного предшествования всегда будет меньше, чем размер матрицы эквивалентной ей грамматики простого предшествования.

Все, что было сказано выше о способах хранения матриц для грамматик простого предшествования, в равной степени относится также и к грамматикам операторного предшествования.

Алгоритм «сдвиг—свертка» для грамматики операторного предшествования

Этот алгоритм в целом похож на алгоритм для грамматик простого предшествования, рассмотренный выше. Он также выполняется расширенным МП-автоматом

и имеет те же условия завершения и обнаружения ошибок. Основное различие состоит в том, что при определении отношения предшествования этот алгоритм не принимает во внимание находящиеся в стеке нетерминальные символы и при сравнении ищет ближайший к верхушке стека терминальный символ. Однако после выполнения сравнения и определения границ основы при поиске правила в грамматике, безусловно, следует принимать во внимание нетерминальные символы.

Алгоритм состоит из следующих шагов.

Шаг 1. Поместить в верхушку стека символ $\perp_{\text{н}}$, считывающую головку — в начало входной цепочки символов.

Шаг 2. Сравнить с помощью отношения предшествования терминальный символ, ближайший к вершине стека (левый символ отношения), с текущим символом входной цепочки, обозреваемым считывающей головкой (правый символ отношения). При этом из стека надо выбрать самый верхний терминальный символ, игнорируя все возможные нетерминальные символы.

Шаг 3. Если имеет место отношение $<\cdot$ или $=\cdot$, то произвести сдвиг (перенос текущего символа из входной цепочки в стек и сдвиг считывающей головки на один шаг вправо) и вернуться к шагу 2. Иначе перейти к шагу 4.

Шаг 4. Если имеет место отношение $\cdot>$, то произвести свертку. Для этого надо найти на вершине стека все терминальные символы, связанные отношением $=\cdot$ («основу»), а также все соседствующие с ними нетерминальные символы (при определении отношения нетерминальные символы игнорируются). Если терминальных символов, связанных отношением $=\cdot$, на верхушке стека нет, то в качестве основы используется один, самый верхний в стеке терминальный символ стека. Все (и терминальные, и нетерминальные) символы, составляющие основу, надо удалить из стека, а затем выбрать из грамматики правило, имеющее правую часть, совпадающую с основой, и поместить в стек левую часть выбранного правила. Если правило, совпадающее с основой, найти не удалось, то необходимо прервать выполнение алгоритма и сообщить об ошибке, иначе, если разбор не закончен, вернуться к шагу 2.

Шаг 5. Если не установлено ни одно отношение предшествования между текущим символом входной цепочки и самым верхним терминальным символом в стеке, то надо прервать выполнение алгоритма и сообщить об ошибке.

Конечная конфигурация данного МП-автомата совпадает с конфигурацией при распознавании цепочек грамматик простого предшествования.

Пример построения распознавателя для грамматики операторного предшествования

Рассмотрим в качестве примера грамматику для арифметических выражений над символами `a` и `b` `G({+,—,/,*,a,b},{S,T,E},P,S)`:

```
P:    S → S+T | S—T | T
      T → T*E | T/E | E
      E → (S) | a | b
```

Видно, что эта грамматика является грамматикой операторного предшествования.

Построим множества крайних левых и крайних правых символов **L**(A), **R**(A) относительно всех нетерминальных символов грамматики.

Шаг 1:

$L_0(S) = \{S,T\}$, $R_0(S) = \{T\}$,

$L_0(T) = \{T,E\}$, $R_0(T) = \{E\}$,

$L_0(E) = \{(,a,b\}$, $R_0(E) = \{),a,b\}$, $i = 1$.

Шаг 2:

$L_1(S) = \{S,T,E\}$, $R_1(S) = \{T,E\}$,

$L_1(T) = \{T,E,(,a,b\}$, $R_1(T) = \{E,),a,b\}$,

$L_1(E) = \{(,a,b\}$, $R_1(E) = \{),a,b\}$.

Шаг 3: так как $L_0(S) \neq L_1(S)$, то $i=2$ и возвращаемся к шагу 2.

Шаг 2:

$L_2(S) = \{S,T,E,(,a,b\}$, $R_2(S) = \{T,E,),a,b\}$,

$L_2(T) = \{T,E,(,a,b\}$, $R_2(T) = \{E,),a,b\}$,

$L_2(E) = \{(,a,b\}$, $R_2(E) = \{),a,b\}$.

Шаг 3: так как $L_1(S) \neq L_2(S)$, то $i=3$ и возвращаемся к шагу 2.

Шаг 2:

$L_3(S) = \{S,T,E,(,a,b\}$, $R_3(S) = \{T,E,),a,b\}$,

$L_3(T) = \{T,E,(,a,b\}$, $R_3(T) = \{E,),a,b\}$,

$L_3(E) = \{(,a,b\}$, $R_3(E) = \{),a,b\}$.

Построение закончено. Получили результат:

$L(S) = \{S,T,E,(,a,b\}$, $R(S) = \{T,E,),a,b\}$,

$L(T) = \{T,E,(,a,b\}$, $R(T) = \{E,),a,b\}$,

$L(E) = \{(,a,b\}$, $R(E) = \{),a,b\}$.

На основе полученных множеств построим множества крайних левых и крайних правых терминальных символов $L^t(A)$, $R^t(A)$.

Шаг 1:

$L^t_0(S) = \{+,-\}$, $R^t_0(S) = \{+,-\}$,

$L^t_0(T) = \{*,/\}$, $R^t_0(T) = \{*,/\}$,

$L^t_0(E) = \{(,a,b\}$, $R^t_0(E) = \{),a,b\}$.

Шаг 2:

$L^t(S) = \{+,-,*,/,(,a,b\}$, $R^t(S) = \{+,-,*,/,),a,b\}$,

$L^t(T) = \{*,/,(,a,b\}$, $R^t(T) = \{*,/,),a,b\}$,

$L^t(E) = \{(,a,b\}$, $R^t(E) = \{),a,b\}$.

Построение закончено. На основе этих множеств и правил грамматики G построим матрицу предшествования грамматики (табл. 4.5).

Поясним, как заполняется матрица предшествования в таблице на примере символа +. В правиле грамматики S→S+T (правило 1) этот символ стоит слева от нетерминального символа T. В множество $L^t(T)$ входят символы: *, /, (, a, b. Ставим знак $\lessdot$ в клетках матрицы, соответствующих этим символам, в строке для символа +. В то же время в этом же правиле символ + стоит справа от нетерминального символа S.

Во множество $R^t(S)$ входят символы: +, –, *, /,), a, b. Ставим знак $\cdot>$ в клетках матрицы, соответствующим этим символам, в столбце для символа +. Больше символ + ни в каком правиле не встречается, значит, заполнение матрицы для него закончено, берем следующий символ и продолжаем заполнять матрицу таким же методом, пока не переберем все терминальные символы.

Таблица 4.5. Матрица предшествования грамматики

Символы	+	-	*	/	(	)	A	b	$\perp_к$
+	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$<\cdot$	$<\cdot$	$\cdot>$
–	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$<\cdot$	$<\cdot$	$\cdot>$
*	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$<\cdot$	$\cdot>$	$<\cdot$	$<\cdot$	$\cdot>$
/	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$<\cdot$	$\cdot>$	$<\cdot$	$<\cdot$	$\cdot>$
(	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$=\cdot$	$<\cdot$	$<\cdot$	
)	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$		$\cdot>$			$\cdot>$
A	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$		$\cdot>$			$\cdot>$
B	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$		$\cdot>$			$\cdot>$
$\perp_н$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$		$<\cdot$	$<\cdot$	

Отдельно рассмотрим символы $\perp_н$ и $\perp_к$. В строке символа $\perp_н$ ставим знак $<\cdot$ в клетках символов, входящих во множество $L^t(S)$. Это символы +, –, *, /, (, a, b. В столбце символа $\perp_к$ ставим знак $\cdot>$ в клетках символов, входящих во множество $R^t(S)$. Это символы +, –, *, /,), a, b.

Еще можно отметить, что в клетке, соответствующей открывающей скобке (символ () слева и закрывающей скобке (символ)) справа, помещается знак $=\cdot$ («составляют основу»). Так происходит, поскольку в грамматике присутствует правило $E \rightarrow (S)$, где эти символы стоят рядом (через нетерминальный символ) в его правой части. Следует отметить, что понятия «справа» и «слева» здесь имеют важное значение: в клетке, соответствующей закрывающей скобке (символ)) слева и открывающей скобке (символ () справа, знак отсутствует — такое сочетание символов недопустимо (отношение $(=\cdot)$ верно, а отношение $)=\cdot($ — неверно).

Алгоритм разбора цепочек грамматики операторного предшествования игнорирует нетерминальные символы. Поэтому имеет смысл преобразовать исходную грамматику таким образом, чтобы оставить в ней только один нетерминальный символ. Тогда получим следующий вид правил:

```
P:      S → S+S | S–S | S (правила 1, 2 и 3)
        S → S*S | S/S | S (правила 4, 5 и 6)
        S → (S) | a | b (правила 7, 8 и 9)
```

Если теперь исключить бессмысленные правила вида $S \rightarrow S$, то получим следующее множество правил (нумерацию правил сохраним):

```
P:      S → S+S | S–S (правила 1, 2)
        S → S*S | S/S (правила 4, 5)
        S → (S) | a | b (правила 7, 8 и 9)
```

Такое преобразование не ведет к созданию эквивалентной грамматики и выполняется только для упрощения работы алгоритма после построения матрицы

предшествования. Полученная в резуль тате преобразования грамматика не является однозначной, но в алгоритм распознавания уже заложены все необходимые данные, поэтому распознаватель остается детерминированным. Построенная таким способом грамматика называется «*остовной*» грамматикой. Вывод, полученный при разборе на основе остовной грамматики, называют результатом «остовного» разбора или «остовным» выводом [4 т.1,2, 5].

По результатам остовного разбора можно построить соответствующий ему вывод на основе правил исходной грамматики. Однако эта задача не представляет практического интереса, поскольку остовный вывод отличается от вывода на основе исходной грамматики только тем, что в нем отсутствуют шаги, связанные с применением цепных правил, и не учитываются типы нетерминальных символов. Для компиляторов же распознавание цепочек входного языка заключается не в нахождении того или иного вывода, а в выявлении основных синтаксических конструкций исходной программы с целью построения на их основе цепочек языка результирующей программы. В этом смысле типы нетерминальных символов и цепные правила не несут никакой полезной информации, а только усложняют обработку цепочки вывода. Поэтому для реального компилятора нахождение остовного вывода является даже более полезным, чем нахождение вывода на основе исходной грамматики. Найденный остовный вывод в дальнейших преобразованиях уже не нуждается[1].

Рассмотрим работу алгоритма распознавания на примерах. Последовательность разбора будем записывать в виде последовательности конфигураций расширенного МП-автомата из трех составляющих:

1) не просмотренная автоматом часть входной цепочки;
2) содержимое стека;
3) последовательность примененных правил грамматики.

Так как автомат имеет только одно состояние, то для определения его конфигурации достаточно двух составляющих — положения считывающей головки во входной цепочке и содержимого стека. Последовательность номеров правил несет дополнительную полезную информацию, по которой можно построить цепочку вывода или дерево вывода.

Будем обозначать такт автомата: $\div_{п}$, если на данном такте выполнялся перенос, и $\div_{с}$, если выполнялась свертка. Последовательности разбора цепочек входных символов будут, таким образом, иметь вид, приведенный ниже.

Пример 1. Входная цепочка `a+a*b`.

1. $\{$`a+a*b`$\perp_{к}$`;`$\perp_{н}$`;`$\varnothing\}\div_{п}$.
2. $\{$`+a*b`$\perp_{к}$`;`$\perp_{н}$`a;`$\varnothing\}\div_{с}$.
3. $\{$`+a*b`$\perp_{к}$`;`$\perp_{н}$`S;8`$\}\div_{п}$.
4. $\{$`a*b`$\perp_{к}$`;`$\perp_{н}$`S+;8`$\}\div_{п}$.
5. $\{$`*b`$\perp_{к}$`;`$\perp_{н}$`S+a;8`$\}\div_{с}$.

[1] Из цепочки (и дерева) вывода удаляются цепные правила, которые, как будет показано далее, все равно не несут никакой полезной семантической (смысловой) нагрузки, а потому для компилятора являются бесполезными. Это положительное свойство распознавателя.

6. {*b⊥к;⊥нS+S;8,8}÷п.
7. {b⊥к;⊥нS+S*;8,8}÷п.
8. {⊥к;⊥нS+S*b;8,8}÷с.
9. {⊥к;⊥нS+S*S;8,8,9}÷с.
10. {⊥к;⊥нS+S;8,8,9,4}÷с.
11. {⊥к;⊥нS;8,8,9,4,1} — разбор завершен, цепочка принята.

Соответствующая цепочка вывода будет иметь вид (используется правосторонний вывод) S⇒S+S⇒S+S*S⇒S+S*b⇒S+a*b⇒a+a*b.

Дерево вывода, соответствующее этой цепочке, приведено на рис. 4.9.

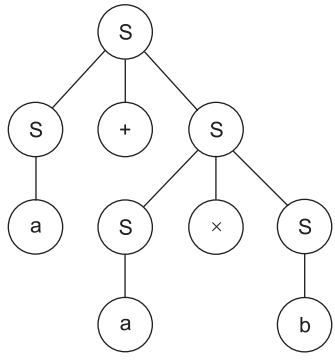

Рис. 4.9. Пример дерева вывода цепочки a+a*b для грамматики операторного предшествования

Пример 2. Входная цепочка (a+a)*b.

1. {(a+a)*b⊥к;⊥н;∅}÷п.
2. {a+a)*b⊥к;⊥н(;∅}÷п.
3. {+a)*b⊥к;⊥н(a;∅}÷с.
4. {+a)*b⊥к;⊥н(S;8}÷п.
5. {a)*b⊥к;⊥н(S+;8}÷п.
6. {)*b⊥к;⊥н(S+a;8}÷с.
7. {)*b⊥к;⊥н(S+S;8,8}÷с.
8. {)*b⊥к;⊥н(S;8,8,1}÷п.
9. {*b⊥к;⊥н(S);8,8,1}÷с.
10. {*b⊥к;⊥нS;8,8,1,7}÷п.
11. {b⊥к;⊥нS*;8,8,1,7}÷п.
12. {⊥к;⊥нS*b;8,8,1,7}÷с.
13. {⊥к;⊥нS*S;8,8,1,7,9}÷с.
14. {⊥к;⊥нS;8,8,1,7,9,4} — разбор завершен, цепочка принята.

Соответствующая цепочка вывода будет иметь вид (используется правосторонний вывод) S⇒S*S⇒S*b⇒(S)*b⇒(S+S)*b⇒(S+a)*b⇒(a+a)*b.

Дерево вывода, соответствующее этой цепочке, приведено на рис. 4.10.

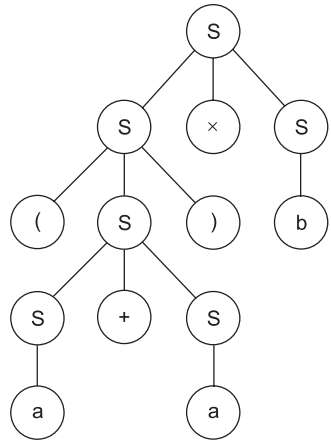

Рис. 4.10. Пример дерева вывода цепочки (a+a)*b для грамматики операторного предшествования

Пример 3. Входная цепочка `a+a*`.

1. $\{$`a+a*`$\perp_к$`;`$\perp_н$`;`$\varnothing\}\div_п$.
2. $\{$`+a*`$\perp_к$`;`$\perp_н$`a;`$\varnothing\}\div_с$.
3. $\{$`+a*`$\perp_к$`;`$\perp_н$`S;8`$\}\div_п$.
4. $\{$`a*`$\perp_к$`;`$\perp_н$`S+;8`$\}\div_п$.
5. $\{$`*`$\perp_к$`;`$\perp_н$`S+a;8`$\}\div_с$.
6. $\{$`*`$\perp_к$`;`$\perp_н$`S+S;8,8`$\}\div_п$.
7. $\{\perp_к$`;`$\perp_н$`S+S*;8,8`$\}\div_с$.
8. `Ошибка!` (Нет правила для выполнения свертки на этом шаге.)

Пример 4. Входная цепочка `a+a)*b`.

1. $\{$`a+a)*b`$\perp_к$`;`$\perp_н$`;`$\varnothing\}\div_п$.
2. $\{$`+a)*b`$\perp_к$`;`$\perp_н$`a;`$\varnothing\}\div_с$.
3. $\{$`+a)*b`$\perp_к$`;`$\perp_н$`S;8`$\}\div_п$.
4. $\{$`a)*b;`$\perp_н$`S+;7`$\}\div_п$.
5. $\{$`)*b`$\perp_к$`;`$\perp_н$`S+a;8`$\}\div_с$.
6. $\{$`)*b`$\perp_к$`;`$\perp_н$`S+S;8,8`$\}\div_с$.
7. $\{$`)*b`$\perp_к$`;`$\perp_н$`S;8,8,1`$\}$.
8. `Ошибка!` (Нет отношений предшествования между символами $\perp_н$ и `)`.)

Более подробно построение распознавателя для грамматик операторного предшествования рассмотрено в книге [42].

Как было сказано выше, матрица для грамматики операторного предшествования всегда имеет меньший объем, чем матрица эквивалентной ей грамматики простого предшествования. Кроме того, распознаватель грамматики операторного пред-

шествования игнорирует нетерминальные символы, не учитывает цепные правила, делает меньше шагов и порождает более короткую цепочку вывода. Поэтому распознаватель для грамматики операторного предшествования всегда проще, чем для эквивалентной ей грамматики простого предшествования.

Интересно, что, поскольку распознаватель на основе грамматик операторного предшествования не учитывает типы нетерминальных символов, он может работать даже с неоднозначными грамматиками, в которых есть правила, различающиеся только типами нетерминальных символов. Примером такой грамматики может служить грамматика G"({a,b},{S,A,B},**P**,S) с правилами:

```
P:      S → A | B
        A → aAb | ab
        B → aBb | ab
```

Как и для любой другой грамматики операторного предшествования, распознаватель для этой грамматики будет детерминированным. Остовная грамматика, построенная на ее основе, будет иметь только два правила вида S→aSb|ab. Неоднозначность заключается в том, что каждому найденному остовному выводу будет соответствовать не один, а несколько выводов в исходной грамматике (в данном случае — всегда два вывода в зависимости от того, какое правило из S→A|B будет применено на первом шаге вывода).

ПРИМЕЧАНИЕ

Грамматики, содержащие правила, различающиеся только типами нетерминальных символов, практического значения не имеют , а потому интереса для построения компиляторов не представляют.

Хотя классы грамматик простого и операторного предшествования несопоставимы[1], класс языков операторного предшествования у´же, чем класс языков простого предшествования. Поэтому не всегда возможно для языка, заданного грамматикой простого предшествования, построить грамматику операторного предшествования. Поскольку класс языков, заданных грамматиками операторного предшествования, еще более узок, чем даже класс языков, заданных грамматиками простого предшествования, с помощью этих грамматик можно определить далеко не каждый ДКС-язык. Грамматики операторного предшествования —очень удобный инструмент для построения распознавателей, но они имеют ограниченную область применения.

Контрольные вопросы и задачи

Вопросы

1. Какие из следующих утверждений справедливы:
   - если язык задан КС-грамматикой, то он может быть задан с помощью МП-автомата;
   - если язык задан КС-грамматикой, то он может быть задан с помощью ДМП-автомата;

[1] В том, что эти два класса грамматик несопоставимы, можно убедиться, рассмотрев два приведенных выше примера — в них взяты различные по своей сути и классам грамматики, хотя они и являются эквивалентными — задают один и тот же язык.

- если язык задан КА, то он может быть задан КС-грамматикой;
- если язык задан ДМП-автоматом, то он может быть задан КС-грамматикой;
- если язык задан однозначной КС-грамматикой, то для него можно построить КА;
- если язык задан однозначной КС-грамматикой, то он может быть задан с помощью ДМП-автомата;
- если язык задан расширенным МП-автоматом, то он может быть задан КС-грамматикой?

2. Какие из приведенных ниже МП-автоматов являются детерминированными:

$R_1(\{q_1,q_2\},\{a,b\},\{a,b,A\},\delta_1,q_1,A,\{q_2\}),\delta_1(q_1,a,A) = \{(q_1,AA)\},\delta_1(q_2,a,A) = \{(q_2,\lambda)\},\delta_1(q_1,b,A) = \{(q_2,A)\},\delta_1(q_2,b,A) = \{(q_2,Aab)\},\delta_1(q_1,\lambda,a) = \{(q_1,\lambda)\},\delta_1(q_1,\lambda,b) = \{(q_1,\lambda)\};$

$R_2(\{q\},\{a,b\},\{a,b,A\},\delta_2,q,A,\{q\}),\delta_2(q,a,A) = \{(q,AA)\},\delta_2(q,b,A) = \{(q,A)\},\delta_2(q,\lambda,a) = \{(q,\lambda)\},\delta_2(q,\lambda,b) = \{(q,\lambda)\},\delta_2(q,\lambda,A) = \{(q,A)\};$

$R_3(\{q\},\{a,b\},\{a,b,A\},\delta_3,q,A,\{q\}),\delta_3(q,a,A) = \{(q,AA),(q,a),(q,\lambda)\},\delta_3(q,b,A) = \{(q,A)\},\delta_3(q,\lambda,a) = \{(q,A)\};$

$R_4(\{q\},\{a,b\},\{a,b,A\},\delta_4,q,A,\{q\}),\delta_4(q,a,A) = \{(q,AAb)\},\delta_4(q,b,A) = \{(q,a)\},\delta_4(q,\lambda,a) = \{(q,\lambda)\},\delta_4(q,\lambda,b) = \{(q,\lambda)\}?$

3. Почему синтаксические конструкции языков программирования могут быть распознаны с помощью ДМП-автоматов?
4. Можно ли полностью проверить структуру входной программы с помощью ДМП-автоматов для большинства языков программирования? Если нет , то можно ли решить ту же задачу, используя МП-автоматы или расширенные МП-автоматы?
5. Всегда ли преобразование правил КС-грамматик ведет к упрощению правил?
6. Почему при преобразовании КС-грамматики к приведенному виду сначала необходимо удалить бесплодные символы, а потом — недостижимые символы?
7. Почему для языка, заданного КС-грамматикой, содержащей λ-правила, существуют трудности с моделированием работы расширенного МП-автомата, выполняющего восходящий разбор с правосторонним выводом?
8. Почему необходимо устранить именно левую рекурсию из правил грамматики? Для моделирования работы какого (восходящего или нисходящего) распознавателя представляет сложности левая рекурсия?
9. Можно ли полностью устранить рекурсию из правил грамматики, записанных в форме Бэкуса—Наура?
10. На алгоритме работы какого распознавателя основан метод рекурсивного спуска?
11. Можно ли реализовать распознаватель по методу расширенного рекурсивного спуска для грамматики, содержащей левую рекурсию?
12. За счет чего класс LL(1)-грамматик является более широким, чем класс КС-грамматик, для которых можно построить распознаватель по методу рекурсивного спуска?

13. На каком алгоритме основана работа распознавателя для LL(k)-грамматики?
14. На каком алгоритме основана работа распознавателя для LR(k)-грамматики?
15. Почему в грамматиках простого предшествования и операторного предшествования не могут присутствовать λ-правила?
16. Класс языков, заданных LR(1)-грамматиками, совпадает с классом ДКС-языков. Зачем же тогда используются другие классы грамматик?

Задачи

1. Задана грамматика G({".",+,-,0,1}, {<число>,<часть>,<цифра>,<осн>}, **P**, <число>):

   P: <число> → +<осн> | -<осн> | <осн>

   <осн> → <часть>.<часть> | <часть>. | <часть>

   <часть> → <цифра> | <часть><цифра>

   <цифра> → 0 | 1

   Постройте для нее обычный и расширенный МП-автоматы.

   Являются ли построенные автоматы детерминированными?

2. Дана грамматика: G({a,b,c},{A,B,C,D,E,F,G,S},**P**,S):

   P: S → aAbB | E

   A → BCa | a | λ

   B → ACb | b | λ

   C → A | B | bA | aB | cC | aE | bE

   E → Ea | Eb | Ec | ED | FG | DG

   D → c | Fb | Fa | λ

   F → BC | AC | DC | EC

   G → Ga | Gb | Gc | GD

   Преобразуйте ее к приведенному виду.

3. Преобразуйте грамматику, построенную в задаче 2, к виду без λ-правил.

4. Дана грамматика G({" ",(,),o,r,a,n,d,t,b},{S,T,E,F},**P**,S):

   P: S → S or T | T

   T → T and E | E

   E → not E | F

   F → (S) | b

   - Преобразуйте ее к виду без цепных правил.
   - Исключите левую рекурсию в правилах грамматики.

5. Дана грамматика: G({if,then,else,a,b},{S,T,E,F},**P**,S):

   P: S → if b then T else S | if b then S | a

   T → if b then T else T | a

   Выполните для нее разбор цепочки символов `if b then if b then if b then a else a`.

   К первому или к последнему `if` будет отнесено `else` в этой цепочке?

Упражнения

1. Постройте нисходящий распознаватель с возвратом для грамматики, заданной в задаче 3. Выполните разбор цепочки символов «a or a and not a and (a or not not a)».
2. Постройте восходящий распознаватель с возвратом для грамматики, заданной в задаче 3. Выполните разбор цепочки символов «a or a and not a and (a or not not a)».
3. Постройте распознаватель, основанный на методе рекурсивного спуска для грамматики `G({a,b,c},{S,A,B,C},P,S)`:

   P: S → aABb | bBAa | cCc

   A → aA | bB | cC

   B → b | aAC

   C → aA | bA | cC

   Выполните разбор цепочки символов `aabbabbcabbb`.
4. Постройте распознаватель, основанный на расширенном методе рекурсивного спуска для грамматики `G({a,<,=,>,+,-,/,*},{S,T,E},P,S)`:

   P: S → T<T | T>T | T<=T | T>=T

   T → T+E | T-E | E

   E → a*a | a/a | a

   Выполните разбор цепочки символов `a*a+a<a/a-a*a`.
5. Дана грамматика `G({(,),+,*,a},{S,T,F},P,S)`:

   P: S → S+T | T

   T → T*E | F

   F → (S) | a

   Постройте для нее распознаватель на основе SLR(1)-грамматики.
6. Дана грамматика `G({(,),^,&,~,a},{S,T,E,F},P,S)`:

   P: S → S^T | T

   T → T&E | E

   E → ~E | F

   F → (S) | a

   Постройте для нее распознаватель на основе грамматики операторного предшествования. Выполните разбор цепочки символов `a^a&~a&(a^~~a)`.
7. Дана грамматика `G({if,then,else,a,b},{S,T,E,F},P,S)`:

   P: S → if b then T else S | if b then S | a

   T → if b then T else T | a

   Постройте для нее распознаватель на основе грамматики операторного предшествования, считая `if`, `then` и `else` едиными терминальными символами.

ГЛАВА 5 Генерация и оптимизация кода

Семантический анализ и подготовка к генерации кода

Назначение семантического анализа

Здесь уже неоднократно упоминалось, что практически все языки программирования, строго говоря, не являются КС-языками . Поэтому полный разбор исходной программы компилятор не может выполнить в рамках КС-языков с помощью КС-грамматик и МП-автоматов. Полный распознаватель для большинства языков программирования может быть построен в рамках КЗ-языков , поскольку все реальные языки программирования контекстно-зависимы[1].

Итак, полный распознаватель для языка программирования можно построить на основе распознавателя КЗ-языка. Однако известно, что такой распознаватель имеет экспоненциальную зависимость требуемых для выполнения разбора исходной программы вычислительных ресурсов от длины входной цепочки [4 т.1, 5, 15, 58]. Компилятор, построенный на основе такого распознавателя, будет неэффективным с точки зрения скорости работы (либо объема необходимой памяти). Поэтому такие компиляторы практически не используются, а все реально существующие компиляторы выполняют анализ исходной программы в два этапа: первый — синтаксический анализ на основе распознавателя для одного из известных классов КС-языков; второй — семантический анализ.

Для проверки семантической правильности исходной программы необходимо иметь всю информацию о найденных лексических единицах языка . Эта информация помещается в таблицу идентификаторов на основе конструкций, найденных синтаксическим распознавателем. Примерами таких конструкций являются блоки описаний констант и идентификаторов (если они предусмотрены семантикой языка) или операторы, где тот или иной идентификатор встречается впервые (если семантика языка предусматривает описание идентификатора по факту его первого использования). Поэтому семантический анализ входной программы может быть произведен только после завершения ее синтаксического анализа.

---

[1] Примером контекстной зависимости, часто встречающейся во многих языках программирования, может служить необходимость предварительно описать идентификатор до его первого использования.

Таким образом, входными данными для семантического анализа служат:

- таблица идентификаторов ;
- результаты разбора синтаксических конструкций входного языка.

Результаты выполнения синтаксического анализа могут быть представлены в одной из форм внутреннего представления программы в компиляторе, которые рассмотрены далее в этой главе.

Семантический анализ обычно выполняется на двух этапах компиляции: на этапе синтаксического разбора и в начале этапа подготовки к генерации кода. В первом случае всякий раз по завершении анализа определенной синтаксической конструкции входного языка выполняется ее семантическая проверка на основе данных, имеющихся в таблице идентификаторов (такими конструкциями, как правило, являются процедуры, функции и блоки операторов входного языка). Во втором случае после завершения всей фазы синтаксического анализа выполняется полный семантический анализ программы на основании данных в таблице идентификаторов (сюда попадает , например, поиск неописанных идентификаторов). Иногда семантический анализ выделяют в отдельный этап (фазу) компиляции.

В каждом компиляторе обычно присутствуют оба варианта семантического анализатора. Конкретная их реализация зависит от версии компилятора и семантики входного языка [4 т.2, 5, 33, 42, 58, 59, 63].

Этапы семантического анализа

Семантический анализатор выполняет следующие основные действия:

- проверка соблюдения в исходной программе семантических соглашений входного языка ;
- дополнение внутреннего представления программы в компиляторе операторами и действиями, неявно предусмотренными семантикой входного языка;
- проверка элементарных семантических (смысловых) норм языков программирования, напрямую не связанных с входным языком.

Проверка соблюдения во входной программе семантических соглашений

Эта проверка заключается в сопоставлении входных цепочек исходной программы с требованиями семантики входного языка программирования. Каждый язык программирования имеет четко специфицированные семантические соглашения, которые не могут быть проверены на этапе синтаксического разбора. Именно их в первую очередь проверяет семантический анализатор.

Примерами таких соглашений являются следующие требования:

- каждая метка, на которую есть ссылка, должна один раз, присутствовать в программе;
- каждый идентификатор должен быть описан один раз и ни один идентификатор не может быть описан более одного раза (с учетом блочной структуры описаний);
- все операнды в выражениях и операциях должны иметь типы, допустимые для данного выражения или операции;

- типы переменных в выражениях должны быть согласованы между собой;
- при вызове процедур и функций число и типы фактических параметров должны быть согласованы с числом и типами формальных параметров.

Это только примерный перечень такого рода требований. Конкретный состав требований, которые должен проверять семантический анализатор, жестко связан с семантикой входного языка (например, некоторые языки допускают не описывать идентификаторы определенных типов). Варианты реализаций такого рода семантических анализаторов детально рассмотрены в [4 т.2, 5, 58, 59].

Например, если мы возьмем оператор языка Pascal, имеющий вид

```
a := b + c;
```

то с точки зрения синтаксического разбора это будет абсолютно правильный оператор. Однако нельзя сказать, является ли этот оператор правильным с точки зрения входного языка (Pascal), пока не будут проверены семантические требования для всех входящих в него лексических элементов. Такими элементами здесь являются идентификаторы `a`, `b` и `c`. Не зная, что они собой представляют , невозможно не только окончательно утверждать правильность приведенного выше оператора, но и понять его смысл. Фактически, необходимо знать описание этих идентификаторов.

В том случае, если хотя бы один из них не описан, имеет место явная ошибка. Если это числовые переменные и константы, то это оператор сложения, если же это строковые переменные и константы — оператор конкатенации строк. Кроме того, идентификатор `a`, например, ни в коем случае не может быть константой — иначе нарушена семантика оператора присваивания. Также невозможно, чтобы одни из идентификаторов были числами, а другие — строками или, скажем, идентификаторами массивов или структур — такое сочетание аргументов для операции сложения недопустимо. И это только некоторая часть соглашений, которые должен проверить компилятор с точки зрения семантики входного языка (в данном примере — Pascal).

ВНИМАНИЕ

Следует отметить, что от семантических соглашений зависит не только правильность оператора, но и его смысл.

Действительно, операции алгебраического сложения и конкатенации строк имеют различный смысл, хотя и обозначаются в рассмотренном примере одним знаком операции — `+`. Следовательно, от семантического анализа зависит также код результирующей программы.

Если какое-либо из семантических требований входного языка не выполняется, то компилятор выдает сообщение об ошибке и процесс компиляции на этом, как правило, прекращается.

Дополнение внутреннего представления программы

Это дополнение внутреннего представления программы связано с добавлением в него операторов и действий, неявно предусмотренных семантикой входного языка. Как правило, эти операторы и действия связаны с преобразованием типов операндов в выражениях и при передаче параметров в процедуры и функции.

Если вернуться к рассмотренному выше элементарному оператору языка Pascal

```
a := b + c;
```

то можно отметить, что здесь выполняются две операции: одна операция сложения (или конкатенации, в зависимости от типов операндов) и одна операция присвоения результата. Соответствующим образом должен быть порожден и код результирующей программы.

Однако не все так очевидно просто. Допустим, что где-то перед рассмотренным оператором мы имеем описание его операндов в виде

```
var
    a : double;
    b : integer;
    c : real;
```

из этого описания следует, что `c` — вещественная переменная языка Pascal, `b` — целочисленная переменная, `a` — вещественная переменная с двойной точностью. Тогда смысл рассмотренного оператора с точки зрения входной программы существенным образом меняется, поскольку в языке Pascal нельзя напрямую выполнять операции над операндами различных типов. Существуют правила преобразования типов, принятые для данного языка. Кто должен выполнять эти преобразования?

Это может сделать разработчик программы — но тогда преобразования типов в явном виде должны будут присутствовать в тексте входной программы. Для рассмотренного примера это выглядело бы примерно так:

```
a := double(real(b) + c);
```

В ряде случаев явное указание операций преобразования типов для входного языка является обязательным. Кроме того, разработчик исходной программы может использовать преобразования типов, отличные от предусмотренных входным языком по умолчанию, и в этом случае он также должен явно указать их. Например, оператор, описанный ниже:

```
a := double(b + trunc(c));
```

выполняет целочисленное сложение, в отличие от ранее приведенного оператора, который выполнял сложение с плавающей точкой. При этом он использует преобразования типов, отличные от преобразований, предусмотренных языком Pascal. Результат выполнения операторов исходной программы может кардинально различаться в зависимости от используемых преобразований типов. Например, если предположить, что перед выполнением рассматриваемого оператора его операнды будут иметь значения `b = 1` и `c = 2.5`, то в результате выполнения первого варианта оператора (операция сложения с плавающей точкой) переменная `a` получит значение `3.5`, а в результате выполнения второго варианта оператора (целочисленное сложение) — `3`.

Однако разработчик исходной программы может не указывать явно используемые преобразования типов. Тогда необходимые преобразования типов выполняет код, порождаемый компилятором, если эти преобразования предусмотрены семантическими соглашениями языка. Для этого в составе библиотек функций, доступных компилятору, должны быть функции преобразования типов (более подробно состав библиотек компилятора описан в главе «Современные системы программирова-

ния»). Вызовы этих функций будут включены компилятором в текст результирующей программы для удовлетворения семантических соглашений о преобразованиях типов входного языка, хотя в тексте исходной программы в явном виде они не присутствуют. Чтобы это произошло, эти функции должны быть добавлены и во внутреннее представление программы в компиляторе. За это также отвечает семантический анализатор.

При отсутствии явно указанных преобразований типов, в рассмотренном примере будет не две, а четыре операции: преобразование целочисленной переменной `b` в формат вещественных чисел; сложение двух вещественных чисел; преобразование результата в вещественное число с двойной точностью; присвоение результата переменной `a`. Количество операций возросло вдвое, причем добавились два вызова нетривиальных функций преобразования типов. Разработчик программы должен помнить это, если хочет добиться высокой эффективности результирующего кода.

СОВЕТ

Явное указание преобразований типов, особенно для принципиально важных операторов, является хорошим стилем программирования и позволяет избежать трудно обнаруживаемых семантических ошибок.

Преобразование типов — это только один вариант операций, неявно добавляемых компилятором в код результирующей программы на основе семантических соглашений. Существуют и другие варианты такого рода операций (преобразование типов — самый распространенный пример).

Таким образом, и в этом случае действия, выполняемые семантическим анализатором, существенным образом влияют на порождаемый компилятором код результирующей программы.

Проверка смысловых норм языков программирования

Проверка элементарных смысловых норм языков программирования, напрямую не связанных с входным языком, — это сервисная функция, которую предоставляет разработчикам большинство современных компиляторов. Эта функция обеспечивает проверку компилятором соглашений, выполнение которых связано со смыслом как всей исходной программы в целом, так и отдельных ее фрагментов. Эти соглашения применимы к большинству современных языков программирования.

Примерами таких соглашений являются следующие требования:

- каждая переменная или именованная константа должна хотя бы один раз использоваться в программе;
- каждая переменная должна быть определена до ее первого использования при любом ходе выполнения программы (первому использованию переменной должно всегда предшествовать присвоение ей какого-либо значения);
- результат функции должен быть определен при любом ходе ее выполнения;
- каждый оператор в исходной программе должен иметь возможность хотя бы один раз выполниться;

- операторы условия и выбора должны предусматривать возможность хода выполнения программы по каждой из своих ветвей;
- операторы цикла должны предусматривать возможность завершения цикла.

Конечно, это только примерный перечень основных соглашений. Конкретный состав проверяемых соглашений зависит от семантики языка.

ПРИМЕЧАНИЕ

В отличие от семантических требований языка, строго проверяемых семантическим анализатором, выполнение данных соглашений не является обязательным.

То, какие конкретно соглашения будут проверяться и как они будут обрабатываться, зависит от качества компилятора, от функций, заложенных в него разработчиками. Простейший компилятор вообще может не выполнять этот этап семантического анализа и не проверять ни одного такого соглашения[1].

Необязательность соглашений такого типа накладывает еще одну особенность на их обработку: их несоблюдение не может трактоваться как ошибка. Даже если компилятор полностью уверен в своей «правоте», тот факт , что какое-то из указанных соглашений не соблюдается, не должен приводить к прекращению компиляции исходной программы. Обычно факт обнаружения несоблюдения такого рода соглашений трактуется компилятором как «предупреждение» (warning). Компилятор выдает пользователю сообщение об обнаружении несоблюдения одного из соглашений, не прерывая процесс компиляции — то есть он просто обращает внимание пользователя на то или иное место в исходной программе. Т о, как реагировать на «предупреждение» (вносить изменения в исходную программу или нет), — это забота и ответственность разработчика исходной программы.

Необязательность указанных соглашений объясняется тем, о чем уже говорилось выше в главе «Основные принципы построения трансляторов» — ни один компилятор не способен полностью понять и оценить смысл исходной программы. А поскольку смысл программы доступен только человеку (для плохо написанной программы — только ее разработчику), то он и должен нести ответственность за выполнение семантических соглашений[2].

Задача проверки выполнения семантических соглашений входного языка во многом связана с проблемой верификации программ. Эта проблема детально рассмотрена в [1, 10, 20, 32, 36, 39, 44].

---

[1] Конечно, современные компиляторы, создаваемые известными фирмами-разработчиками, стремятся выполнить проверку максимально возможного числа такого рода соглашений. Эта функция обычно преподносится как одно из достоинств компилятора и способствует завоеванию им хороших позиций на рынке.

[2] Обычно хорошим стилем считается построить программу так, чтобы она компилировалась «без предупреждений» — то есть так, чтобы компилятор не обнаруживал в ней ни одного несоблюдения соглашений о смысле операторов входного языка. В большинстве случаев это возможно. Тем не менее практически все современные компиляторы позволяют отключить в процессе компиляции программы проверку того или иного соглашения, вплоть до полного исключения всех возможных вариантов такого рода проверки. Эти соглашения входят в состав «опций» компиляторов. Следует отметить, что отключать можно только необязательные соглашения — ни одну проверку семантических требований языка отключить невозможно.

Рассмотрим в качестве примера функцию, представляющую собой фрагмент входной программы на языке C:

```
int f_test(int a)
{    int b,c;
        b=0;
        c=0;
        if(b=1) { return a; }
        c=a+b;
}
```

Практически любой современный компилятор языка C обнаружит в данном месте входной программы массу «неточностей». Например, переменная `c` описана, ей присваивается значение, но она нигде не используется. Значение переменной `b`, присвоенное в операторе `b=0;`, тоже никак не используется. Наконец, условный оператор лишен смысла, так как всегда предусматривает ход выполнения только по одной своей ветке, а значит , и оператор `c=a+b;` никогда выполнен не будет. Скорее всего, компилятор выдаст еще одно предупреждение, характерное именно для языка C — в операторе `if(b=1)` присвоение стоит в условии (это не запрещено ни синтаксисом, ни семантикой языка C, но является очень распространенной семантической ошибкой в языке C). В принципе, смысл (а точнее, бессмысленность) этого фрагмента будет правильно воспринят и обработан компилятором.

Однако если взять аналогичный по смыслу, но синтаксически более сложный фрагмент программы, то картина будет несколько иная:

```
int f_test_add(int* a, int* b)
{
        *a=1;
        *b=0;
        return *a;
}

int f_test(int a)
{    int b,c;
        b=0;
        if (f_test(&b,&c)!=0) { return a; }
        c=a+b;
}
```

Здесь компилятор уже вряд ли сможет выяснить порядок изменения значений переменных и выполнение условий в данном фрагменте из двух функций (обе они сами по себе независимо вполне осмысленны!). Единственное предупреждение, которое, скорее всего, получит в данном случае разработчик, — это то, что функция`f_test` не всегда корректно возвращает результат (отсутствует оператор `return` перед концом функции). И то это предупреждение на самом деле не будет соответствовать истинному положению вещей!

ПРИМЕЧАНИЕ

Проверка выполнения дополнительных семантических соглашений является весьма полезной функцией компиляторов, но ответственность за смысл исходной программы по-прежнему остается на ее разработчике.

Идентификация лексических единиц языков программирования

Идентификация переменных, типов, процедур, функций и других лексических единиц языков программирования — это установление однозначного соответствия между лексическими единицами и их именами в тексте исходной программы. Идентификация лексических единиц языка чаще всего выполняется на этапе семантического анализа .

Как правило, большинство языков программирования требуют, чтобы в исходной программе имена лексических единиц не совпадали как между собой, так и с ключевыми словами синтаксических конструкций языка. Но чаще всего этого бывает недостаточно, чтобы установить однозначное соответствие между лексическими единицами и их именами, поскольку существуют дополнительные смысловые (семантические) ограничения, накладываемые языком на употребление этих имен.

СОВЕТ

Существуют языки программирования (например, PL/1), которые допускают , чтобы имена идентификаторов совпадали с ключевыми словами языка. В таких языках возможны весьма интересные синтаксические конструкции. Однако такая возможность не предоставляет ничего хорошего, кроме лишних хлопот разработчикам компилятора. Даже если такая особенность присуща входному языку, пользоваться ею настоятельно не рекомендуется — читаемость программы будет сильнозатруднена, что говорит о плохом стиле программирования .

Например, локальные переменные в большинстве языков программирования имеют так называемую «область видимости» , которая ограничивает употребление имени переменной рамками того блока исходной программы, где эта переменная описана. Это значит, что, с одной стороны, такая переменная не может быть использована вне пределов своей области видимости. С другой стороны, имя переменной может быть не уникальным, поскольку в двух различных областях видимости допускается существование двух различных переменных с одинаковым именем (причем в большинстве языков программирования, допускающих блочные структуры, области видимости переменных могут перекрываться). Другой пример такого рода ограничений на уникальность имен — это тот факт , что в языке программирования С две разные функции или процедуры с различными аргументами могут иметь одно и то же имя.

Безусловно, полный перечень таких ограничений зависит от семантики языка программирования. Все они четко заданы в описании языка и не могут допускать неоднозначности в толковании, но они также не могут быть полностью определены на этапе лексического анализа , а потому требуют от компилятора дополнительных действий на этапах синтаксического разбора и семантического анализа. Общая направленность этих действий такова, чтобы дать каждой лексической единице языка уникальное имя в пределах всей исходной программы и потом использовать это имя при синтезе результирующей программы.

Можно дать примерный перечень действий компиляторов для идентификации переменных, констант, функций, процедур и других лексических единиц языка:

- имена локальных переменных дополняются именами тех блоков (функций, процедур), в которых эти переменные описаны;
- имена внутренних переменных и функций модулей исходной программы дополняются именами самих модулей (это касается только внутренних имен);
- имена процедур и функций, принадлежащих объектам (классам) в объектно-ориентированных языках программирования дополняются наименованиями типов объектов (классов), которым они принадлежат;
- имена процедур и функций модифицируются в зависимости от типов их формальных аргументов.

Конечно, это далеко не полный перечень возможных действий компилятора, каждая реализация компилятора может предполагать свой набор действий. То, какие из них будут использоваться и как они будут реализованы на практике, зависит от языка исходной программы и разработчиков компилятора.

Как правило, уникальные имена, которые компилятор присваивает лексическим единицам языка, используются только во внутреннем представлении исходной программы, и разработчик исходной программы не сталкивается с ними. Но они могут потребоваться ему в некоторых случаях — например, при отладке программы , при порождении текста результирующей программы на языке ассемблера или при использовании библиотеки, созданной на другом языке программирования (или даже с помощью другой версии компилятора). Тогда разработчик должен знать, по каким правилам компилятор порождает уникальные имена для лексических единиц исходной программы[1].

СОВЕТ

Правила, по которым происходит модификация имен, достаточно просты. Их можно выяснить для конкретной версии компилятора и использовать при разработке. Однако лучше этого не делать, так как нет гарантии, что при переходе к другой версии компилятора эти правила не изменятся — тогда код станет неработоспособным. Правильным средством будет аккуратное использование механизма отключения именования лексических единиц, предоставляемое синтаксисом исходного языка.

Во многих современных языках программирования предусмотрены специальные настройки и ключевые слова, которые позволяют отключить процесс порождения компилятором уникальных имен для лексических единиц языка. Эти слова входят в специальные синтаксические конструкции языка (как правило, это конструкции, содержащие слова `export` или `external`). Если пользователь использует эти средства, то компилятор не применяет механизм порождения уникальных имен для указанных лексических единиц. В этом случае разработчик программы сам отвечает

[1] Очень часто бывает, что пользователь не может вызвать и использовать по имени функцию (или процедуру), которая содержится в некоторой библиотеке. Сообщения компилятора вида «Функция *такая-то* не найдена» вызывают немало удивления, особенно когда пользователь сам создал библиотеку и точно знает, что эта функция в ней есть. Чаще всего проблема связана с наименованием функций — компилятор дает функции в библиотеке не совсем то имя, которое дал ей пользователь в исходном коде.

за уникальность имени данной лексической единицы в пределах всей исходной программы или даже в пределах всего проекта (если используется несколько исходных модулей). Если требование уникальности не будет выполняться, могут возникнуть синтаксические или семантические ошибки на стадии компиляции либо же другие ошибки на более поздних этапах разработки программного обеспечения (эти этапы описаны далее в главе «Современные системы программирования»).

Чаще всего эта проблема относится к именам процедур или функций в библиотеках. Если корректно описать их, то использование объектного кода библиотеки не будет зависеть от типа компилятора и даже от входного языка (тут важно еще учесть соглашения о передаче параметров, которые рассмотрены далее).

Распределение памяти

Принципы распределения памяти

Распределение памяти — это процесс, который ставит в соответствие лексическим единицам исходной программы адрес, размер и атрибуты области памяти, необходимой для этой лексической единицы. *Область памяти* — это блок ячеек памяти, выделяемый для данных, каким-то образом объединенных логически. Логика таких объединений задается семантикой исходного языка.

Распределение памяти работает с лексическими единицами языка — переменными, константами, функциями и т. п. — и с информацией об этих единицах, полученной на этапах лексического и синтаксического анализа. Как правило, исходными данными для процесса распределения памяти в компиляторе служат таблица идентификаторов, построенная лексическим анализатором, и декларативная часть программы (так называемая «область описаний»), полученная в результате синтаксического анализа. Не во всех языках программирования декларативная часть программы присутствует явно, некоторые языки предусматривают дополнительные семантические правила для описания констант и переменных.

Процесс распределения памяти в современных компиляторах, как правило, работает с относительными, а не с абсолютными (физическими) адресами ячеек памяти (разница между абсолютными и относительными адресами указана в главе «Современные системы программирования»). Распределение памяти выполняется перед генерацией кода результирующей программы, потому что его результаты должны быть использованы в процессе генерации кода.

Каждую область памяти можно классифицировать по двум параметрам: в зависимости от ее роли в результирующей программе и в зависимости от способа ее распределения в ходе выполнения результирующей программы.

По роли области памяти в результирующей программе она бывает *глобальная* или *локальная*.

По способам распределения область памяти бывает *статическая* или *динамическая*. Динамическая память, в свою очередь, может распределяться либо разработчиком исходной программы (по командам разработчика), либо компилятором (автоматически).

Классификация областей памяти представлена на рис. 5.1.

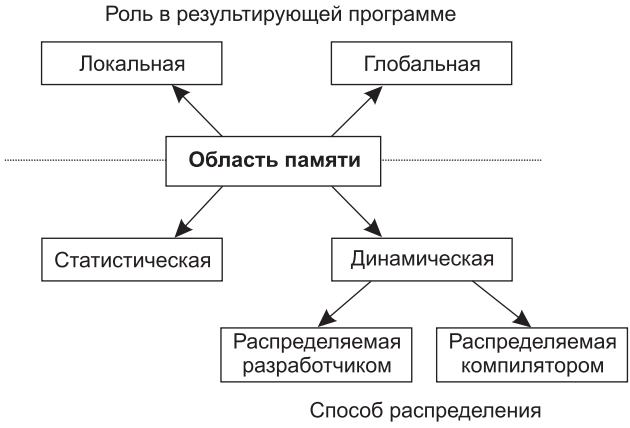

Рис. 5.1. Область памяти в зависимости от ее роли и способа распределения

Далеко не все лексические единицы языка требуют для себя выделения памяти. То, под какие элементы языка нужно выделять области памяти, а под какие нет, определяется исключительно реализацией компилятора и архитектурой используемой вычислительной системы. Так, целочисленные константы можно разместить в статической памяти, а можно непосредственно в тексте результирующей программы (это позволяют практически все современные вычислительные системы), то же самое относится и к константам с плавающей точкой, но их размещение в тексте программы допустимо не всегда. Кроме того, в целях экономии памяти, занимаемой результирующей программой, под различные элементы языка компилятор может выделить одни и те же ячейки памяти. Например, в одной и той же области памяти могут быть размещены одинаковые строковые константы или две различные локальные переменные, которые никогда не используются одновременно.

Простые и сложные структуры данных. Выравнивание границ данных

Распределение памяти для переменных скалярных типов

Во всех языках программирования существует понятие так называемых «базовых типов данных», которые также называют основными или *скалярными* типами. Размер области памяти, необходимый для лексемы скалярного типа, считается известным. Он определяется семантикой языка и архитектурой целевой вычислительной системы, на которой должна выполняться результирующая программа.

Идеальным вариантом для разработчиков программ был бы такой компилятор, у которого размер памяти для базовых типов зависел бы только от семантики языка. Но чаще всего зависимость результирующей программы от архитектуры целевой вычислительной системы полностью исключить не удается. Создатели компиляторов и языков программирования предлагают механизмы, позволяющие свести эту зависимость к минимуму.

СОВЕТ

Размер области памяти каждого скалярного типа данных фиксирован и известен для определенной целевой вычислительной системы. Однако не рекомендуется непосредственно использовать его в тексте исходной программы, так как это ограничивает переносимость программы. Вместо этого нужно использовать функции определения размера памяти, предоставляемые входным языком программирования.

Например, в языке программирования C базовыми типами данных являются типы `char`, `int`, `long int` и т. п. (реально этих типов, конечно, больше, чем три), а в языке программирования Pascal — типы `byte`, `char`, `word`, `integer` и т. п. Размер базового типа `int` в языке C для архитектуры компьютера на базе 16-разрядных процессоров составляет 2 байта, а для 32-разрядных процессоров — 4 байта. Разработчики исходной программы на этом языке, конечно, могут узнать данную информацию, но если ее использовать в исходной программе напрямую, то такая программа будет жестко привязана к конкретной архитектуре целевой вычислительной системы. Чтобы исключить эту зависимость, лучше использовать механизм определения размера памяти для типа данных, предоставляемый языком программирования, — в языке C это функция `sizeof`.

Распределение памяти для сложных структур данных

Для более сложных структур данных используются правила распределения памяти, определяемые семантикой (смыслом) этих структур. Эти правила достаточно просты и, в принципе, одинаковы во всех языках программирования.

Вот правила распределения памяти под основные виды структур данных:

- для массивов — произведение числа элементов в массиве на размер памяти для одного элемента (то же правило применимо и для строк, но во многих языках строки содержат еще и дополнительную служебную информацию фиксированного объема);
- для структур (записей с именованными полями) — сумма размеров памяти по всем полям структуры;
- для объединений (союзов, общих областей, записей с вариантами) — размер максимального поля в объединении;
- для реализации объектов (классов) — размер памяти для структуры с такими же именованными полями плюс память под служебную информацию объектно-ориентированного языка (как правило, фиксированного объема).

Формулы для вычисления объема памяти можно записать следующим образом:

для массивов: $V_{\text{мас}} = \prod_{i=1,n}(m_i)V_{\text{эл}}$,
где n — размерность массива, m_i — количество элементов i-й размерности, $V_{\text{эл}}$ — объем памяти для одного элемента;

для структур: $V_{\text{стр}} = \sum_{i=1,n} V_{\text{поля}_i}$,
где n — общее количество полей в структуре, $V_{\text{поля}_i}$ — объем памяти для i-го поля структуры;

для объединений: $V_{\text{стр}} = \max_{i=1,n} V_{\text{поля}_i}$,

где n — общее количество полей в объединении, $V_{\text{поля}_i}$ — объем памяти для i-го поля объединения.

Для более сложных структур данных входного языка объем памяти, отводимой под эти структуры данных, вычисляется рекурсивно. Например, если имеется массив структур, то при вычислении объема отводимой под этот массив памяти для вычисления объема памяти, необходимой для одного элемента массива, будет вызвана процедура вычисления памяти структуры. Такой подход определения объема занимаемой памяти очень удобен, если декларативная часть языка представлена в виде дерева типов. Тогда для вычисления объема памяти, занимаемой типом из каждой вершины дерева, нужно вычислить объем памяти для всех потомков этой вершины, а потом применить формулу , связанную непосредственно с самой вершиной (этот механизм подобен механизму СУ-перевода, применяемому при генерации кода, который описан далее в разделе «Г енерация кода. Методы генерации кода» в данной главе). Как раз такого типа древовидные конструкции строит синтаксический анализатор для декларативной части языка.

Выравнивание границ областей памяти

Говоря об объеме памяти, занимаемой различными лексемами языка, следует упомянуть еще один момент, связанный с выравниванием границ областей памяти, отводимых для различных лексических единиц. Архитектура многих современных вычислительных систем предусматривает, что обработка данных выполняется более эффективно, если адрес, по которому выбираются данные, кратен определенному числу байт [1] (как правило, это 2, 4, 8 или 16 байт). Современные компиляторы учитывают особенности целевых вычислительных систем. При распределении данных они могут размещать области памяти для лексем наиболее оптимальным образом. Поскольку не всегда размер памяти, отводимой под лексическую единицу, кратен указанному числу байт, то в общем объеме памяти, отводимой под резуль тирующую программу, могут появляться неиспользуемые области.

Например, если мы имеем описание переменных на языке С:

```
static char c1,c2,c3;
```

то, зная, что под одну переменную типа `char` отводится 1 байт памяти, можем ожидать, что для описанных выше переменных потребуется всего 3 байта памяти. Однако если кратность адресов для доступа к памяти составляет 4 байта, то под эти переменные будет отведено в целом 12 байт памяти, из которых 9 не будут использоваться.

Как правило, разработчику исходной программы не нужно знать, каким образом компилятор распределяет адреса под отводимые области памяти. Чаще всего компилятор сам выбирает оптимальный метод, и, изменяя границы выделенных областей, он всегда корректно осуществляет доступ к ним. Вопрос об этом может возникнуть при работе с указателями на ячейки памяти, а также если с данными программы, написанной на одном языке, работают программы, написанные на другом языке программирования (чаще всего на языке ассемблера). Реже такие проблемы

[1] Некоторые вычислительные системы даже вообще не могут работать с адресами, не кратными определенному числу байт.

возникают при использовании двух различных компиляторов с одного и того же входного языка.

Например, если мы имеем рассмотренное выше описание переменных на языке C и один указатель:

```
static char c1,c2,c3;
static char* p1;
```

то не рекомендуется для доступа к переменным использовать операторы, аналогичные приведенным ниже:

```
p1=&c1;
*p1='a'; p1++;
*p1='b'; p1++;
*p1='c'; p1++;
```

Результат выполнения приведенных здесь операторов существенно зависит от того, какова кратность адресов целевой вычислительной системы и как компилятор учтет эту кратность. При кратности адресов 1 байт приведенный выше фрагмент кода последовательно заполнит переменные c1,c2,c3 значениями 'a','b','c' (как, видимо, это и предполагается). Но при любых других значениях кратности адресов результат выполнения этого фрагмента кода будет отличаться от ожидаемого, так как при увеличении значения указателя он будет указывать на неиспользуемые области памяти.

Большинство современных компиляторов позволяют пользователю самому указать, использовать или нет кратные адреса и какую границу кратности установить (если это возможно для архитектуры целевой вычислительной системы). Такие настройки возможны как для всей исходной программы в целом, так и для отдельных структур данных, входящих в нее.

СОВЕТ

Не рекомендуется без крайней необходимости менять значения настройки кратности адресов, установленные компилятором. И тем более делать это для выполнения операций с указателями, подобных приведенной выше, — это не только является плохим стилем программирования, но может также существенно повлиять на переносимость программы.

Статическое и динамическое связывание. Менеджеры памяти

Глобальная и локальная память

Глобальная область памяти — это область памяти, которая выделяется один раз при инициализации результирующей программы и действует все время выполнения результирующей программы.

Как правило, глобальная область памяти может быть доступна из любой части исходной программы, но многие языки программирования позволяют налагать синтаксические и семантические ограничения на доступность глобальных областей памяти. При этом сами области памяти и связанные с ними лексемы остаются гло-

бальными, ограничения налагаются только на возможность их использования в тексте исходной программы (и эти ограничения, в принципе, можно обойти).

Локальная область памяти — это область памяти, которая выделяется в начале выполнения некоторого фрагмента результирующей программы (блока, функции, процедуры или оператора) и может быть освобождена по завершении выполнения данного фрагмента.

Доступ к локальной области памяти всегда запрещен за пределами того фрагмента программы, в котором она выделяется. Это определяется как синтаксическими и семантическими правилами языка, так и кодом результирующей программы. Даже если удастся обойти ограничения, налагаемые входным языком, использование таких областей памяти вне их области видимости приведет к катастрофическим последствиям для результирующей программы.

Распределение памяти на локальные и глобальные области целиком определяется семантикой входного языка. Только зная смысл синтаксических конструкций входного языка, можно четко сказать, какая из них будет отнесена в глобальную область памяти, а какая — в локальную. Иногда в исходном языке для некоторых конструкций нет четкого разграничения, тогда решение об их отнесении в ту или иную область памяти принимается разработчиками компилятора и может зависеть от используемой версии компилятора. При этом разработчики исходной программы не должны полагаться на тот факт, что один раз принятое решение будет неизменным во всех версиях компилятора.

Рассмотрим для примера фрагмент текста модуля программы на языке Pascal:

```
...
const
        Global_1 = 1;
        Global_2 : integer = 2;
var
        Global_I : integer;
...
function Test (Param: integer): pointer;
const
        Local_1 = 1;
        Local_2 : integer = 2;
var
        Local_I : integer;
begin
  ...
end;
```

Согласно семантике языка Pascal, переменная `Global_I` является глобальной переменной языка и размещается в глобальной области памяти, константа `Global_1` также является глобальной, но язык не требует, чтобы компилятор обязательно размещал ее в памяти — значение константы может быть непосредственно вставлено в код результирующей программы там, где она используется. Типизированная константа `Global_2` является глобальной, но, семантика языка предполагает, что в отличие

от константы Global_1, эта константа обязательно будет размещена в памяти, а не в коде программы. Доступность идентификаторов Global_I, Global_2 и Global_1 из других модулей зависит от того, где и как они описаны. Например, для компилятора Borland Pascal переменные и константы, описанные в заголовке модулей, доступны из других модулей, а переменные и константы, описанные в теле модулей, недоступны, хотя и те и другие являются глобальными.

Параметр Param функции Test, переменная Local_I и константа Local_1 являются локальными элементами этой функции. Они не доступны вне пределов данной функции и располагаются в локальной области памяти. Типизированная константа Local_2 представляет собой очень интересный элемент программы. С одной стороны, она недоступна вне пределов функции Test и потому является ее локальным элементом. С другой стороны, как типизированная константа она будет располагаться в глобальной области памяти, хотя ниоткуда извне не может быть доступна (аналогичными конструкциями языка C, например, являются локальные переменные функций, описанные как static).

Семантические особенности языка должны учитывать создатели компилятора, когда разрабатывают модуль распределения памяти. Безусловно, это должен знать и разработчик исходной программы, чтобы не допускать семантических (смысловых) ошибок — этот тип ошибок сложнее всего поддается обнаружению. Так, в приведенном выше примере в качестве результата функции Test может выступать адрес переменной Global_I или типизированной константы Global_2. Результатом функции не может быть адрес констант Global_1 или Local_1 — это запрещено семантикой языка. Гораздо сложнее вопрос с переменной Local_I или параметром функции Param. Будучи элементами локальной области памяти функции Test, они никак не могут выступать в качестве результата функции, потому что он будет использован вне самой функции, когда область ее локальной памяти может уже не существовать. Но ни синтаксис, ни семантика языка не запрещают использовать адрес этих элементов в качестве результата функции (и далеко не всегда компилятор способен хотя бы обнаружить такой факт). Эти особенности языка должен учитывать разработчик программы. А вот адрес типизированной константы Local_2 в принципе может быть результатом функции Test. Ведь хотя она и является локальной константой, но размещается в глобальной области памяти[1].

Статическая и динамическая память

Статическая область памяти — это область памяти, размер которой известен на этапе компиляции.

Поскольку для статической области памяти известен размер, компилятор всегда может выделить эту область памяти и связать ее с соответствующим элементом программы. Поэтому для статической области памяти компилятор порождает некото-

[1] Рассуждения о возможности или невозможности использовать адреса тех или иных элементов программы в качестве результата функции в приведенном примере носят чисто теоретический характер. Вопрос о практическом использовании такого рода функций автор оставляет открытым, поскольку он касается проблем читаемости и наглядности исходного текста программ, а также стиля программирования. Эти проблемы представляют большой интерес, но выходят за рамки данного учебника.

рый адрес (как правило, это относительный адрес — смысл относительного адреса описан в главе «Современные системы программирования»).

Статические области памяти обрабатываются компилятором самым простейшим образом, поскольку напрямую связаны со своим адресом. В этом случае говорят о *статическом связывании* области памяти и лексической единицы входного языка.

Динамическая область памяти — это область памяти, размер которой на этапе компиляции программы не известен.

Размер динамической области памяти будет известен только в процессе выполнения результирующей программы. Поэтому для динамической области памяти компилятор не может определить адрес — для нее он порождает фрагмент кода, который отвечает за распределение памяти (ее выделение и освобождение). Как правило, с динамическими областями памяти связаны операции с указателями и с экземплярами объектов (классов) в объектно-ориентированных языках программирования. При использовании динамических областей памяти говорят о *динамическом связывании* области памяти и лексической единицы входного языка.

Динамические области памяти, в свою очередь, можно разделить на динамические области памяти, выделяемые пользователем, и динамические области памяти, выделяемые непосредственно компилятором.

Динамические области памяти, выделяемые пользователем, появляются в тех случаях, когда разработчик исходной программы явно использует в тексте программы функции, связанные с распределением памяти (примером таких функций являются `New` и `Dispose` в языке Pascal, `malloc` и `free` в языке C, `new` и `delete` в языке C++ и др.). Функции распределения памяти могут использовать либо напрямую средства ОС, либо средства исходного языка (которые, в конце концов, все равно основаны на средствах ОС). В этом случае за своевременное выделение и освобождение памяти отвечает разработчик, а не компилятор. Компилятор должен только построить код вызова соответствующих функций — в принципе, для него работа с динамической памятью, выделяемой пользователем, ничем не отличается от работы с любыми другими функциями и дополнительных сложностей не вызывает.

Другое дело — динамические области памяти, выделяемые компилятором. Они появляются, когда пользователь использует типы данных, операции над которыми предполагают перераспределение памяти, в явном виде не присутствующее в тексте исходной программы[1]. Примерами таких типов данных могут служить строки в некоторых языках программирования, динамические массивы и, конечно, многие операции над экземплярами объектов (классов) в объектно-ориентированных языках (характерный пример — тип данных `string` в языке Object Pascal). В этом случае сам компилятор отвечает за порождение кода, который будет всегда обеспечивать своевременное выделение памяти под элементы программы и освобождать ее по мере использования.

[1] В принципе, в этом случае разработчик исходной программы может даже и не подозревать о том, что используемая им операция предполагает перераспределение памяти. Однако это не лучшим образом будет сказываться на эффективности созданной разработчиком программы.

Как статические, так и динамические области памяти сами по себе могут быть глобальными или локальными.

Менеджеры памяти

Многие компиляторы объектно-ориентированных языков программирования используют для работы с динамической памятью специальный *менеджер памяти*, к которому обращаются как при выделении памяти по команде пользователя, так и при выделении памяти самим компилятором. Менеджер памяти обеспечивает выполнение функций выделения и освобождения используемой оперативной памяти и следит за ее наиболее рациональным использованием.

Как правило, роль менеджера памяти заключается в том, что при первом требовании на выделение оперативной памяти он запрашивает у ОС область памяти намного большего объема, чем это первоначально необходимо результирующей программе. Зато для всех последующих требований на выделение оперативной памяти менеджер памяти стремится выделить фрагменты из уже имеющейся в его распоряжении области памяти, не обращаясь к функциям ОС до тех пор, пока вся эта область памяти не будет исчерпана. Когда вся имеющаяся в его распоряжении память распределена, менеджер памяти вновь запрашивает у ОС область памяти заведомо большего размера, чем требуется результирующей программе. По мере выполнения результирующей программы менеджер памяти следит за использованием всех находящихся в его распоряжении областей памяти и их фрагментов. Если какая-то из запрошенных им областей памяти полностью перестает использоваться, менеджер памяти освобождает ее с помощью функций ОС. В более сложных случаях менеджер памяти обладает также функциями перераспределения уже используемой памяти и сборки мусора — поиска в выделенной памяти фрагментов, которые не освобождены, но в то же время уже не используются.

При создании менеджера памяти разработчики компилятора преследуют две основные цели:

1) сокращается количество обращений результирующей программы к системным функциям ОС, обеспечивающим выделение и освобождение оперативной памяти, а поскольку это довольно сложные функции, то в целом увеличивается быстродействие результирующей программы;
2) сокращается фрагментация оперативной памяти, характерная именно для объектно-ориентированных языков, поскольку менеджер памяти запрашивает у ОС оперативную память укрупненными фрагментами и освобождает ее также укрупненными фрагментами.

Главным недостатком при использовании менеджера памяти является тот факт, что результирующая программа всегда запрашивает у ОС оперативной памяти больше, чем необходимо для ее выполнения. Однако этот недостаток окупается за счет более рационального использования уже выделенной памяти.

Код менеджера памяти включается в текст результирующей программы или поставляется в виде отдельной динамически загружаемой библиотеки. Компилятор автоматически включает вызовы функций менеджера памяти в текст результирующей программы там, где это необходимо. В таких языках программирования, как Pascal или C++, менеджер памяти не является обязательной составляющей библиотеки

функций компилятора, но, например, в языке Java, где не предусмотрена возможность выделения динамической памяти по запросам пользователя и вся память автоматически выделяется и освобождается компилятором, менеджер памяти необходим, и функции его достаточно сложны.

Дисплей памяти процедуры (функции). Стековая организация дисплея памяти

Понятие дисплея памяти процедуры (функции)

Дисплей памяти процедуры (функции) — это область данных, доступных для обработки в этой процедуре (функции).

Как правило, дисплей памяти процедуры включает следующие составляющие:

- глобальные данные (переменные и константы) всей программы;
- формальные аргументы процедуры;
- локальные данные (переменные и константы) данной процедуры.

Также в дисплей памяти часто включают адрес возврата.

Адрес возврата — это адрес того фрагмента кода резуль тирующей программы, куда должно быть передано управление после того, как завершится выполнение вызванной процедуры или функции. Фактически адрес возврата — это адрес машинной команды, следующей за командой обращения (вызова) к процедуре или функции. Адрес возврата никак не обрабатывается в процедуре или функции, но должен быть сохранен на все время ее выполнения.

Процедура должна работать со своими данными однотипным образом вне зависимости от того, из какого места результирующей программы она была вызвана. Поэтому компилятор должен соответствующим образом организовать обращение к данным, доступным процедуре.

С глобальными данными никаких проблем не возникает — их адреса известны, они все устанавливаются на этапе распределения памяти. Даже если память под эти данные распределяется динамически, компилятору однозначно известно, как к этой памяти обратиться, и способ обращения одинаков для всей программы. Поэтому он всегда может создать код для обращения к этим данным вне зависимости от того, откуда это обращение происходит.

Сложнее с локальными параметрами и переменными процедуры. Они относятся к локальной памяти процедуры и потому должны быть как-то связаны с нею. Существует несколько вариантов связывания локальных переменных и параметров с кодом соответствующей процедуры или функции [15]. Современные вычислительные системы ориентированы, главным образом, на стековую организацию дисплея памяти процедуры (функции). Поэтому компиляторы при создании резуль тирующей программы используют именно этот вариант.

Стековая организация дисплея памяти процедуры (функции)

Стековая организация дисплея памяти процедуры (функции) основана на том, что для хранения параметров (аргументов) процедур и функций, их локальных пере-

менных, а также адреса возврата в результирующей программе выделяется специальная область памяти, одна на всю программу, организованная в виде стека. Этот стек называют *стеком передачи параметров*, или просто *стеком параметров*.

При стековой организации дисплея памяти процедуры или функции в момент ее вызова все параметры и адрес возврата помещаются в стек параметров. При завершении выполнения процедуры или функции параметры извлекаются («выталкиваются») из стека, а управление передается по адресу возврата.

Внутри вызываемой процедуры или функции на верхушке стека может быть выделено место для хранения всех ее локальных переменных. Объектный код процедуры или функции адресуется к параметрам и своим локальным переменным по смещениям относительно верхушки стека.

В этой схеме компилятор организует единый для всей программы стек параметров. Верхушка стека адресуется одним из регистров процессора — регистром стека. Для доступа к данным удобно использовать еще один регистр процессора, называемый базовым регистром. В начале выполнения результирующей программы стек пуст.

Тогда при вызове процедуры или функции необходимо выполнить следующие действия:

- поместить в стек все параметры процедуры или функции;
- запомнить в стеке адрес возврата и передать управление вызываемой процедуре.

В начале выполнения процедуры или функции она должна выполнить следующие действия:

- запомнить в стеке значение базового регистра;
- запомнить состояние регистра стека в базовом регистре;
- увеличить значение регистра стека на размер памяти, необходимый для хранения локальных переменных процедуры или функции.

После выполнения этих действий может выполняться код вызванной процедуры или функции. Доступ ко всем локальным переменным и параметрам (аргументам) при этом осуществляется через базовый регистр (параметры лежат в стеке ниже места, указанного базовым регистром, а локальные переменные и константы — выше места, указанного базовым регистром, но ниже места, указанного регистром стека).

При возврате из процедуры или функции необходимо выполнить следующие действия:

- присвоить регистру стека значение базового регистра;
- выбрать из стека значение базового регистра;
- выбрать из стека адрес возврата;
- передать управление по адресу возврата и выбрать из стека все параметры процедуры.

После этого можно продолжить выполнение кода результирующей программы, следующего за вызовом процедуры или функции.

Если происходит несколько вложенных вызовов процедур или функций, то их параметры и локальные переменные помещаются в стек последовательно, одни за други-

ми. При этом локальные переменные и параметры процедур и функций, вызванных ранее, не мешают выполнению процедур и функций, вызванных позже.

На рис. 5.2 показано, как изменяется содержание стека параметров при выполнении трех последовательных вызовов процедур: сначала процедуры A, затем процедуры B и снова процедуры A. При вызовах стек заполняется, а при возврате из кода процедур — освобождается в обратном порядке.

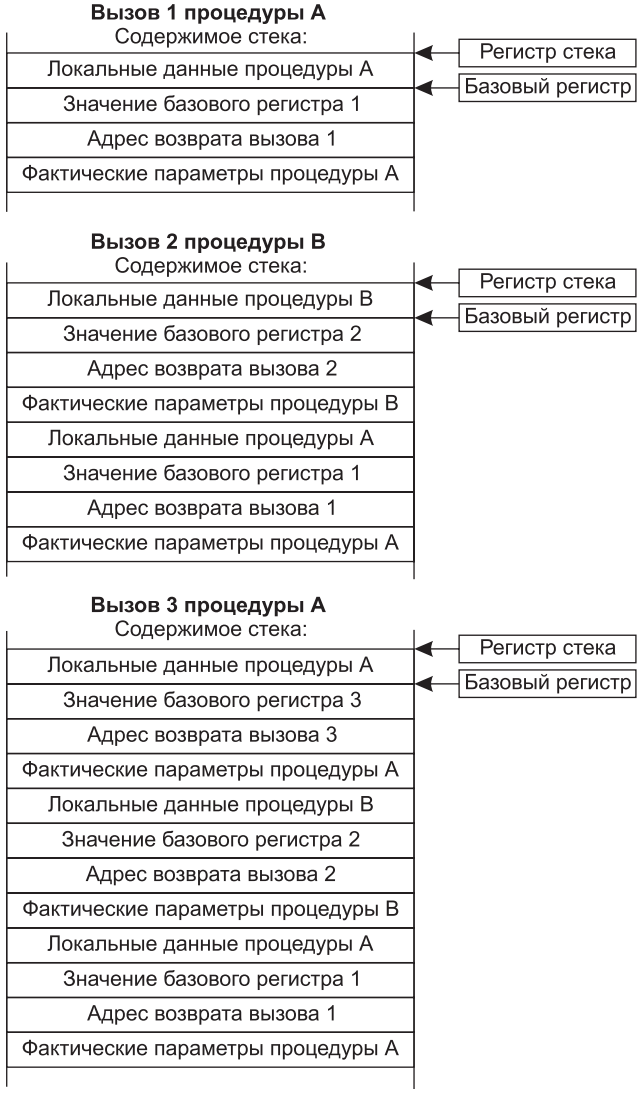

Рис. 5.2. Содержание стека параметров при выполнении трех последовательных вызовов процедур

Из рис. 5.2 видно, что при рекурсивном вызове все локальные данные последовательно размещаются в стеке и при этом каждая процедура работает только со своими

данными. Более того, такая схема легко обеспечивает поддержку вызовов вложенных процедур, которые допустимы, например, в языке Pascal.

Стековая организация памяти получила широкое распространение в современных компиляторах. Практически все существующие ныне языки программирования предполагают именно такую схему организации памяти, ее также используют и в программах на языках ассемблера. Широкое распространение стековой организации дисплея памяти процедур и функций тесно связано также с тем, что большинство операций, выполняемых при вызове процедуры в этой схеме, реализованы в виде соответствующих машинных команд во многих современных вычислительных системах. Это, как правило, все операции со стеком параметров, а также команды вызова процедуры (с автоматическим сохранением адреса возврата в стеке) и возврата из вызванной процедуры (с выборкой адреса возврата из стека). Такой подход значительно упрощает и ускоряет выполнение вызовов при стековой организации дисплея памяти процедуры (функции).

Стековая организация дисплея памяти процедуры позволяет организовать рекурсию вызовов. Однако у этой схемы есть один существенный недостаток — память, отводимая под стек параметров, используется неэффективно. Очевидно, что стек параметров должен иметь размер, достаточный для хранения всех параметров, локальных данных и адресов возврата при самом глубоком вложенном вызове процедуры в результирующей программе. Как правило, ни компилятор, ни разработчик программы не могут знать точно, какая максимальная глубина вложенных вызовов процедур возможна в программе. Следовательно, не известно заранее, какая глубина стека потребуется результирующей программе во время ее выполнения. Размер стека обычно выбирается разработчиком программы «с запасом»[1], так, чтобы гарантированно обеспечить любую возможную глубину вложенных вызовов процедур и функций. С другой стороны, бóльшую часть времени своей работы результирующая программа не использует значительную часть стека, а значит, выделенная для этой цели часть оперативной памяти не используется, так как стек параметров сам по себе не может быть использован для других целей.

Стековая организация дисплея памяти процедур и функций присутствует практически во всех современных компиляторах. Тем не менее процедуры и функции, построенные с помощью одного входного языка высокого уровня, не всегда могут быть использованы в программах, написанных на другом языке. Дело в том, что стековая организация определяет основные принципы организации дисплея памяти процедуры, но не определяет точно механизм его реализации. В разных входных языках детали реализации этой схемы могут различаться порядком помещения параметров процедуры в стек и извлечения из него. Эти вопросы определяются в соглашениях о вызове и передаче параметров, принятых в различных языках программирования.

---

[1] К сожалению, нет точных расчетных методов для определения требуемого размера стека. Разработчик обычно оценивает его приближенно, исходя из логики результирующей программы. Когда во время отладки программы возникают сообщения о недостатке объема стека (а в них имеется в виду именно стек параметров), то размер стремятся значительно увеличить. С другой стороны, очень большой объем необходимого стека параметров может говорить о невысоком качестве программы. Если объем доступной памяти под стек ограничен (например, в MS-DOS — не более 64 Кбайт), это ведет к ограничению допустимой вложенности рекурсий, а бесконечная рекурсия всегда ведет к переполнению стека параметров.

Параметры могут помещаться в стек в прямом порядке (в порядке их следования в описании процедуры) или в обратном порядке. Извлекаться из стека они могут либо в момент возврата из процедуры (функции), либо непосредственно после возврата[1]. Кроме того, для повышения эффективности программ и увеличения скорости вызова процедур и функций, компиляторы могут предусматривать передачу части параметров не через стек, а через свободные регистры процессора.

Обычно разработчику нет необходимости знать о том, какое соглашение о передаче параметров принято во входном языке программирования. При обработке вызовов внутри текста исходной программы компилятор сам корректно формирует код для выполнения вызовов процедур и функций. Наиболее развитые компиляторы могут варьировать код вызова в зависимости от типа и количества параметров процедуры (функции). Однако если разработчику необходимо использовать процедуры и функции, созданные на основе одного входного языка, в программе на другом входном языке, то компилятор не может знать заранее соглашение о передаче параметров, принятое для этих процедур и функций. Такие проблемы возникают чаще всего тогда, когда необходимо выполнить вызов процедур и функций из внешних библиотек. В этом случае разработчик должен сам позаботиться о том, чтобы соглашения о передаче параметров в двух языках были согласованы, поскольку несогласованность в передаче параметров может привести к полной неработоспособности результирующей программы.

СОВЕТ

Для того чтобы корректно выполнять вызовы процедур и функций из библиотек, необходимо использовать специальные ключевые слова языка программирования, которые позволяют разработчику явно указать, какое соглашение о передаче параметров будет использовано для той или иной процедуры (функции).

Еще одна проблема, которая может возникать при вызове функции, — это проблема возврата результата ее выполнения. Как правило, для этой цели компилятор использует определенный регистр процессора, чаще всего — регистр аккумулятора, если он существует в целевой вычислительной архитектуре. Но если результатом выполнения функции является массив или структура данных (многие языки программирования допускают это), которые не могут быть размещены в регистре процессора, то возникает вопрос, как передать их в место вызова. В этом случае способ передачи результата выполнения функции может зависеть от языка программирования и даже от версии компилятора.

Исключительные ситуации и их обработка

Понятие исключительной ситуации

Понятие *исключительной ситуации* (exception) появилось в современных объектно-ориентированных языках программирования.

[1] Существует два общепринятых соглашения о вызове и передаче параметров: соглашение языка Pascal и соглашение языка C. В первом случае параметры помещаются в стек в прямом порядке и извлекаются из стека в момент выполнения возврата из процедуры (функции) — то есть внутри самой процедуры (функции); во втором случае параметры помещаются в стек в обратном порядке, а извлекаются из стека после выполнения возврата из функции — то есть непосредственно после выполнения вызова в вызывающем коде.

Проблема заключалась в том, что в таких языках есть специального вида функции — конструкторы (constructor) и специального вида процедуры — деструкторы (destructor), которые выполняют действия по созданию объектов (классов) и уничтожению их соответственно. При создании объектов выделяется необходимая для этого область памяти, а при освобождении объектов эта область памяти должна освобождаться. Как и при выполнении любых процедур и функций, при выполнении этих специальных процедур и функций могут произойти нештатные ситуации, вызванные кодом результирующей программы или кодом вызываемых ею библиотек ОС (например, ситуация недостатка оперативной памяти при ее выделении). Но в отличие от всех остальных процедур и функций, которые обычно возвращают в таких ситуациях предусмотренный код ошибки, конструктор и деструктор не могут вернуть код ошибки при возникновении нештатной ситуации, поскольку имеют строго определенный и неизменный смысл.

С другой стороны, разработчику желательно иметь средства обнаружения и обработки нештатных ситуаций и для этих двух специальных случаев — конструктора и деструктора.

В объектно-ориентированных языках программирования для этой цели был предложен механизм исключительных ситуаций и работа с ними. Идея оказалась удачной и затем была распространена не только на специальные, но и на все остальные функции языков программирования. В настоящее время все объектно-ориентированные языки программирования предоставляют разработчику механизм работы с исключительными ситуациями. Синтаксис и реализация этого механизма различаются в зависимости от используемого языка программирования, но его смысл (семантика) является общим для всех языков.

Исключительная ситуация — это нештатная ситуация, возникающая в ходе выполнения результирующей программы, предусматривающая, что в момент ее возникновения выполнение результирующей программы будет прервано в месте возникновения исключительной ситуации и управление будет передано обработчику исключительной ситуации. Может быть несколько типов (вариантов) исключительных ситуаций, возникающих в ходе выполнения результирующей программы. Каждый тип исключительной ситуации может предусматривать свой обработчик исключительной ситуации. Если для какого-то типа исключительной ситуации обработчик исключительной ситуации отсутствует , то в этом случае управление должно быть передано системному обработчику исключительной ситуации.

Исключительные ситуации могут возникать в следующих случаях:

- при обнаружении ошибки в аппаратно-программных средствах вычислительного комплекса, на котором исполняется результирующая программа (примерами таких ошибок являются: деление на 0, доступ к закрытой области памяти, запись данных в файл, закрытый для записи, и т. п.);
- при обнаружении на этапе выполнения результирующей программы действий, запрещенных семантикой входного языка (примерами таких ошибок являются: ошибка при динамическом преобразовании типов, вызов неопределенной виртуальной функции, доступ за границы массива или структуры данных и т. п.);
- по команде разработчика исходной программы, явно указывающей на необходимость порождения исключительной ситуации.

Причина возникновения исключительной ситуации влияет на то, какой программный код будет ответственным за порождение исключительной ситуации и вызов ее обработчика: в первом случае это должен сделать программный код ОС или входящих в ее состав библиотек, во втором и третьем случаях этот код должен входить в состав результирующей программы, но если во втором случае компилятор должен сам включить его в результирующую программу без явного на то указания, то в последнем случае он должен сделать это по явному указанию разработчика.

Таким образом, в механизме порождения и обработки исключительных ситуаций может быть задействован не только объектный код результирующей программы, но и код других компонентов целевой вычислительной системы (ОС, системных библиотек, драйверов устройств), где выполняется результирующая программа. Следовательно, обработка исключительных ситуаций зависит от типа целевой вычислительной системы. То есть компилятор должен включать в результирующую программу код порождения и обработки исключительных ситуаций в зависимости от типа вычислительной системы, на которую она ориентирована.

Обработчики исключительных ситуаций

За обработку исключительных ситуаций отвечают специальные синтаксические конструкции, предусмотренные в объектно-ориентированных языках программирования. Примерами таких конструкций являются блоки типа `try ... except ...` и `try ... finally ...` в языке программирования Pascal, блоки типа `throw ... catch ...` в языке программирования C++ и др. Считается, что текст исходной программы, помещенный внутрь таких блоков, может вызвать возникновение исключительных ситуаций определенного типа. Текст исходной программы, помещенный в синтаксически выделенную часть таких блоков, считается текстом обработчика исключительных ситуаций, специальные синтаксические конструкции и семантические правила в каждом языке программирования служат для определения типа обрабатываемых исключительных ситуаций. Компилятор анализирует исходный текст таких блоков и порождает соответствующий объектный код результирующей программы.

Поскольку считается, что в результирующей программе может произойти любая исключительная ситуация вне зависимости от того, предусмотрел или нет ее возникновение разработчик программы, то весь код исходной программы как бы находится внутри одного блока обработки исключительных ситуаций. Компилятор сам порождает код обработчика исключительных ситуаций для всей результирующей программы в целом — этот обработчик является системным обработчиком исключительных ситуаций. Он обрабатывает все типы исключительных ситуаций. Как правило, он получает управление в тех случаях, когда появляется исключительная ситуация, возникновение которой не предусмотрел разработчик результирующей программы. Обычно системный обработчик исключительных ситуаций формирует сообщение об ошибке, которое видит пользователь, но не прекращает выполнение результирующей программы.

На обработку исключительных ситуаций влияет модель дисплея памяти процедуры (функции) , используемая данным компилятором, поскольку исключительная ситуация может произойти в момент выполнения любой процедуры или функции. Все современные компиляторы используют стековую модель дисплея памяти процедуры (функции) для обработки исключительных ситуаций.

С точки зрения компилятора важно построить объектный код обработчика исключительной ситуации таким образом, чтобы он, с одной стороны, корректно вызывался при возникновении исключительной ситуации, а с другой стороны, не разрушал структуры данных результирующей программы — прежде всего стек параметров .

Обработка исключительных ситуаций осложняется тем, что заранее не известно, в каком месте объектного кода и в какой момент выполнения результирующей программы может произойти исключительная ситуация. Вне зависимости от этого, управление всегда должен получить обработчик данной исключительной ситуации, а если его нет — системный обработчик исключительных ситуаций.

При возникновении исключительной ситуации происходят следующие действия:

- прерывается выполнение результирующей программы;
- ищется адрес обработчика исключительной ситуации;
- управление передается объектному коду обработчика исключительной ситуации, при этом должны быть выполнены следующие условия:
  - все локальные данные, помещенные в стек параметров, должны быть изъяты из него вне зависимости от глубины вложенности вызовов процедур и функций, где произошла исключительная ситуация (стек параметров должен быть приведен в состояние для той процедуры или функции, где находится обработчик исключительной ситуации);
  - для всех данных, для которых компилятором (не разработчиком!) были выделены динамические области памяти , эти области памяти должны быть освобождены;
- обработчик исключительной ситуации проверяет, соответствует ли ему тип произошедшей исключительной ситуации, и если нет , то его выполнение прерывается и все начинается с самого начала;
- начинает выполняться объектный код обработчика исключительной ситуации.

Поскольку исключительная ситуация может произойти в любом месте и в любой момент выполнения результирующей программы, то адрес ее обработчика должен находиться в фиксированной заранее известной статической области памяти. Более того, поскольку исключительная ситуация может быть порождена не только непосредственно в коде результирующей программы, но и при ее обращении к системным функциям (ОС, драйверам, динамически загружаемым библиотекам), то область памяти для хранения адреса обработчика должна быть известна ОС — это должна быть системная область памяти.

В общем случае объектный код, порождаемый компилятором для обработки исключительных ситуаций, достаточно сложен. Но принцип, лежащий в его основе, может быть рассмотрен на самом тривиальном уровне.

В простейшем случае необходимо иметь одну системную ячейку памяти E_1 для хранения адреса. В этой ячейке памяти хранится адрес регистра стека, указывающий на ячейку в стеке, хранящую адрес обработчика исключительных ситуаций.

Тогда для начала блока обработки исключительных ситуаций компилятор должен порождать объектный код, который выполняет следующее:

- запоминает в стеке параметров текущее значение базового регистра ;
- запоминает в стеке параметров адрес обработчика исключительной ситуации для данного блока;
- запоминает в стеке параметров текущий адрес, хранящийся в ячейке E_1;
- помещает в ячейку E_1 текущее значение регистра стека.

Для конца блока обработки исключительных ситуаций (если исключительная ситуация не произошла) компилятор должен порождать объектный код, который выполняет следующее:

- извлекает из стека предыдущий адрес, хранившийся в ячейке E_1, и помещает его в ячейку E_1;
- извлекает из стека два адреса (адрес обработчика для данного блока и базовый регистр), далее они уже не понадобятся.

При возникновении исключительной ситуации происходит следующее:

- текущий адрес из ячейки E_1 помещается в регистр стека;
- из стека извлекается предыдущий адрес, хранившийся в ячейке E_1 и записывается в ячейку E_1;
- из стека извлекается адрес обработчика исключительной ситуации и ему передается управление;
- получив управление, обработчик исключительной ситуации извлекает из стека базовый регистр — теперь значение регистра стека и базового регистра соответствует уровню вложенности процедуры или функции, в которой находится код обработчика исключительной ситуации;
- если опять произойдет исключительная ситуация (или если произошедшая ситуация не относится к данному обработчику) — необходимый адрес регистра стека уже находится в ячейке E_1, и все действия опять повторятся.

Такой механизм позволяет организовать как бы стек обработчиков исключительных ситуаций, построенный на основе ссылок внутри стека параметров . Общая схема ссылок при использовании простейшего механизма обработки исключительных ситуаций приведена на рис. 5.3.

Как уже было сказано выше, реальный механизм обработки исключительных ситуаций гораздо сложнее и может зависеть от типа ОС. Но схема, приведенная на рис. 5.3, в общем виде отражает технологию, используемую системами программирования для порождения объектного кода обработки исключительных ситуаций.

Такая схема обработки исключительных ситуаций обеспечивает корректное изменение содержимого стека параметров при возникновении исключительной ситуации на любой глубине вложенности вызовов процедур и функций. Однако она никак не учитывает необходимость освобождать области памяти , динамически выделенные для структур данных компилятором. Чаще всего эта проблема решается следующим образом: для каждого блока, содержащего динамические области памяти, выделяемые компилятором, компилятор автоматически организует блок обработки исключительных ситуаций, в котором обработчик исключительных ситуаций выполняет действия по освобождению выделенных компилятором динамических областей памяти для всех типов исключительных ситуаций.

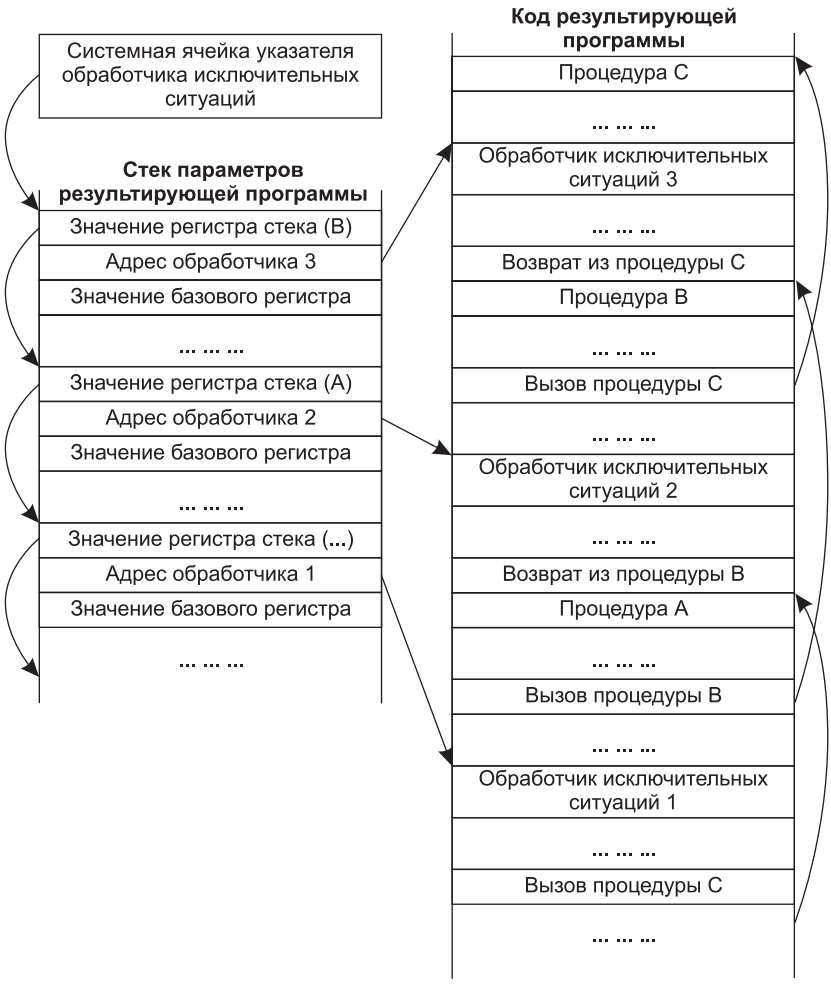

Рис. 5.3. Простейший механизм обработки исключительных ситуаций через стек параметров

Например, в языке Object P ascal область памяти для типа данных `string` автоматически выделяется и освобождается компилятором. Поэтому два блока описания процедуры на этом языке, приведенные ниже, в принципе, эквивалентны:

```
procedure A;
var s: string;
begin
  …
  {код процедуры}
end;
```

```
procedure A;
  var s: string;
 begin
     try
     …
        {код процедуры}
       finally s := '';
   end;{try}
 end;
```

ВНИМАНИЕ

Следует помнить, что компилятор не порождает автоматических обработчиков исключительных ситуаций для освобождения динамических областей памяти, выделяемой разработчиком. Разработчик результирующей программы сам должен позаботиться об этом!

Не все нештатные ситуации, возникающие в ходе выполнения результирующей программы, обрабатываются как исключительные ситуации. Некоторые аппаратные ошибки и ошибки по защите памяти, возникающие в ходе выполнения, обрабатываются ОС и результирующей программе не сообщаются. Также невозможна обработка исключительных ситуаций в тех случаях, когда повреждены данные, хранящиеся в стеке параметров (например, адрес обработчика исключительной ситуации). В этих случаях управление получает специальный код ОС, предназначенный для обработки исключений (прерываний). Как правило, в таких ситуациях пользователю выдается системное сообщение об ошибке и выполнение результирующей программы прекращается.

Разработчикам следует помнить, что для порождения и обработки исключительных ситуаций компилятор порождает достаточно сложный объектный код, который, кроме всего прочего, должен взаимодействовать с функциями ОС. Поэтому не рекомендуется пользоваться исключительными ситуациями там, где в этом нет необходимости. Кроме того, не стоит применять обработку исключительных ситуаций в тех случаях, когда надо найти и исправить логическую ошибку в исходной программе (например, если возникает непредвиденная исключительная ситуация защиты памяти или выхода за границы области данных).

СОВЕТ

Рекомендуется использовать механизм исключительных ситуаций в исходной программе только тогда, когда нет других средств обнаружить и обработать нештатную ситуацию.

Например, два фрагмента исходной программы на языке Object Pascal, приведенные ниже, выполняют практически одни и те же действия:

```
if i <> 0 then f := k / i
else
begin
…
{код обработки
 нештатной ситуации}
end;
```

```
try
f := k / i;
except
on EZeroDivide do
begin
…
{код обработки
нештатной ситуации}
end;
end;{try}
```

Но для первого из приведенных фрагментов компилятор построит более эффективный объектный код, чем для второго. Кроме того, логика выполнения и наглядность у первого фрагмента лучше, чем у второго.

Память для типов данных (RTTI-информация)

В ранее существовавших языках программирования все вызовы процедур и функций исходной программы в результирующей программе транслировались в команду вызова подпрограммы с известным адресом. Адрес вызываемой процедуры или функции был известен на этапе компиляции. Этот вариант вызова получил название «раннее связывание» и оставался единственным вариантом до появления объектно-ориентированных языков программирования.

В объектно-ориентированных языках программирования существует понятие виртуальных (virtual) функций (или процедур) . При обращении к таким функциям и процедурам адрес вызываемой функции или процедуры становится известным только в момент выполнения результирующей программы. Такой вариант вызова носит название «позднее связывание» . Поскольку адрес вызываемой процедуры или функции зависит от того типа данных, для которого она вызывается, в современных компиляторах для объектно-ориентированных языков программирования предусмотрены специальные структуры данных для организации таких вызовов [2, 7, 16, 19, 26, 27, 28, 37, 47, 48, 57, 60]. Эти структуры данных напрямую недоступны разработчику исходной программы, но должны обрабатываться компилятором.

Современные компиляторы с объектно-ориентированных языков программирования предусматривают, что результирующая программа может обрабатывать не только переменные, константы и другие структуры данных, но и информацию о типах данных, описанных в исходной программе . Эта информация получила название RTTI — Run Time Type Information — «информация о типах во время выполнения».

Состав RTTI-информации определяется семантикой входного языка и реализацией компилятора. Как правило, для каждого типа данных в объектно-ориентированном языке программирования создается уникальный идентификатор типа данных, который используется для сопоставления типов. Для него может храниться наименование типа, а также другая служебная информация, которая используется компилятором в коде результирующей программы. Вся эта информация может быть в той или иной мере доступна пользователю. Еще одна цель хранения RTTI-информации — обеспечить корректный механизм вызова виртуальных процедур и функций (так называемое «позднее связывание»), предусмотренный во всех объектно-ориентированных языках программирования [26, 27, 28, 40, 43, 47, 48, 60].

RTTI-информация хранится в результирующей программе в виде RTTI-таблицы. RTTI-таблица представляет собой глобальную статическую структуру данных, которая создается и заполняется в момент начала выполнения результирующей программы. Компилятор с объектно-ориентированного входного языка отвечает за порождение в результирующей программе кода, ответственного за заполнение RTTI-таблицы.

Каждому типу данных в RTTI-таблице соответствует своя область данных, в которой хранится вся необходимая информация об этом типе данных, его взаимосвязи с другими типами данных в общей иерархии типов данных программы, а также все указатели на код виртуальных процедур и функций, связанных с этим типом. Всю эту информацию и указатели для каждого типа данных в RTTI-таблице размещает код, автоматически порождаемый компилятором.

В принципе, всю эту информацию можно было бы разместить непосредственно в памяти, статически или динамически отводимой для каждого экземпляра объекта

(класса) того или иного типа. Но поскольку данная информация для всех объектов одного и того же типа совпадает , то это привело бы к нерациональному использованию оперативной памяти. Поэтому компилятор размещает всю информацию по типу данных в одном месте, а каждому экземпляру объекта (класса) при выделении памяти для него добавляет только небольшой объем служебной информации, связывающей этот объект с соответствующим ему типом данных (как правило, эта служебная информация является указателем на нужную область данных в RTTI-таблице). При статическом распределении памяти под экземпляры объектов (классов) компилятор сам помещает в нее необходимую служебную информацию, при динамическом распределении — порождает код, который заполнит эту информацию во время выполнения программы (поскольку RTTI-таблица является статической областью данных, адрес ее известен и фиксирован).

На рис. 5.4 приведена схема, иллюстрирующая построение RTTI-таблицы и ее связь с экземплярами объектов в результирующей программе.

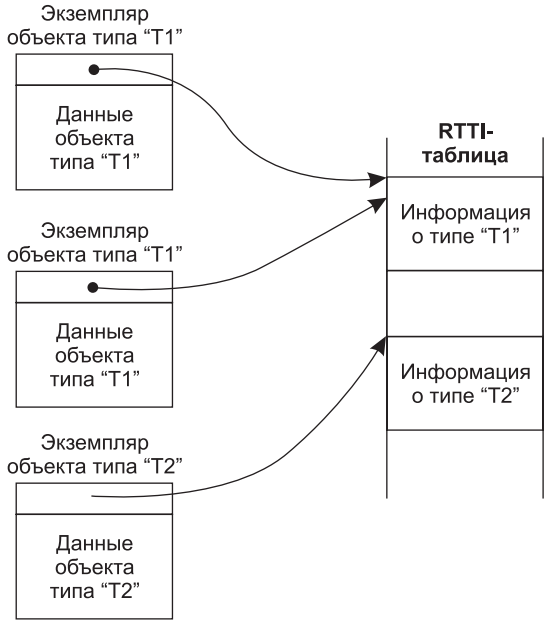

Рис. 5.4. Взаимосвязь RTTI-таблицы с экземплярами объектов в результирующей программе

Компилятор автоматически строит код, ответственный в результирующей программе за создание и заполнение RTTI-таблицы и за ее взаимосвязь с экземплярами объектов (классов) различных типов. Разработчику не надо заботиться об этом, тем более что он, как правило, не может воздействовать на служебную информацию, помещаемую компилятором в RTTI-таблицу.

Проблемы могут появиться в том случае, когда некоторая программа взаимодействует с динамически загружаемой библиотекой. При этом могут возникнуть ситуации, вызванные несоответствием типов данных в программе и библиотеке, хотя и та и другая могут быть построены на одном и том же входном языке с помощью одного

и того же компилятора. Дело в том, что код инициализации программы, порождаемый компилятором, ответствен за создание и заполнение своей RTTI-таблицы, а код инициализации библиотеки, порождаемый тем же компилятором, — своей RTTI-таблицы. Если программа и динамически загружаемая библиотека работают с одними и теми же типами данных, то эти RTTI-таблицы будут полностью совпадать, но при этом они не будут одной и той же таблицей. Это уже должен учитывать разработчик программы при проверке соответствия типов данных.

Еще бóльшие сложности могут возникнуть, если программа, построенная на основе входного объектно-ориентированного языка программирования , будет взаимодействовать с библиотекой или программой, построенной на основе другого объектно-ориентированного языка (или даже построенной на том же языке, но с помощью другого компилятора). Поскольку формат RTTI-таблиц и состав хранимой в них служебной информации не специфицированы для всех языков программирования, то они полностью зависят от используемой версии компилятора. Соответствующим образом различается и порождаемый компилятором код резуль тирующей программы.

В общем случае организовать такого рода взаимодействие между двумя фрагментами кода, порожденного двумя различными компиляторами, напрямую практически невозможно. Для этого необходимо использовать механизмы, предусмотренные для взаимодействия между различными программами (о возможностях такого рода взаимодействия см. в разделах «Разработка приложений в двухзвенной архитектуре» и «Разработка программ в многоуровневой архитектуре» главы 6).

Генерация кода. Методы генерации кода

Общие принципы генерации кода. Синтаксически управляемый перевод

Принципы генерации объектного кода

Генерация объектного кода — это перевод компилятором внутреннего представления исходной программы в цепочку символов выходного языка.

Генерация объектного кода порождает результирующую объектную программу на языке ассемблера или непосредственно на языкемашинных кодов. Внутреннее представление программы может иметь любую структуру в зависимости от реализации компилятора, в то время как результирующая программа всегда представляет собой линейную последовательность команд. Поэтому генерация объектного кода (объектной программы) в любом случае должна выполнять действия, связанные с преобразованием сложных синтаксических структур в линейные цепочки.

Генерацию кода можно считать функцией, определенной на синтаксическом дереве, построенном в результате синтаксического анализа, и на информации, содержащейся в таблице идентификаторов. Поэтому генерация объектного кода выполняется после того, как выполнены синтаксический анализ программы и все необходимые действия по подготовке к генерации кода. Характер отображения входной программы в последовательность команд, выполняемых генерацией, зависит от входного языка, архитектуры целевой вычислительной системы, а также от качества желаемого объектного кода.

В идеале компилятор должен выполнить синтаксический анализ всей входной программы, затем провести ее семантический анализ, после чего приступать к подготовке генерации и непосредственно к генерации кода. Однако такая схема работы компилятора практически почти никогда не применяется. Дело в том, что в общем случае ни один семантический анализатор и ни один компилятор не способны проанализировать и оценить смысл всей исходной программы в целом. Формальные методы анализа семантики применимы только к очень незначительной части возможных исходных программ. Поэтому у компилятора нет практической возможности порождать эквивалентную результирующую программу на основе всей исходной программы.

Как правило, компилятор выполняет генерацию результирующего кода поэтапно, на основе законченных синтаксических конструкций входной программы. Компилятор выделяет законченную синтаксическую конструкцию из текста исходной программы, порождает для нее фрагмент результирующего кода и помещает его в текст результирующей программы. Затем он переходит к следующей синтаксической конструкции. Так продолжается до тех пор, пока не будет разобрана вся исходная программа. В качестве анализируемых законченных синтаксических конструкций выступают операторы, блоки операторов, описания процедур и функций. Их конкретный состав зависит от входного языка и реализации компилятора.

Смысл (семантику) каждой такой синтаксической конструкции входного языка можно определить исходя из ее типа, а тип определяется синтаксическим анализатором на основании грамматики входного языка. Примерами типов синтаксических конструкций могут служить операторы цикла, условные операторы, операторы выбора и т. д. Одни и те же типы синтаксических конструкций характерны для различных языков программирования, при этом они различаются синтаксисом (который задается грамматикой языка), но имеют схожий смысл (который определяется семантикой). В зависимости от типа синтаксической конструкции выполняется генерация кода результирующей программы, соответствующего данной синтаксической конструкции. Для семантически схожих конструкций различных входных языков программирования может порождаться схожий результирующий код.

Синтаксически управляемый перевод

Чтобы компилятор мог построить код результирующей программы для всей синтаксической конструкции исходной программы, часто используется метод, называемый синтаксически управляемым переводом — *СУ-переводом*.

Идея СУ-перевода основана на том, что синтаксис и семантика языка взаимосвязаны. Это значит, что смысл предложения языка зависит от синтаксической структуры этого предложения. Теория СУ-перевода была предложена американским лингвистом Ноамом Хомским [1]. Она справедлива как для формальных языков, так и для языков естественного общения — например, смысл предложения русского языка зависит от входящих в него частей речи (подлежащего, сказуемого, дополнений и др.) и от взаимосвязи между ними. Однако естественные языки допускают неоднозначности в грамматиках — отсюда происходят различные двусмысленные фразы,

[1] Фамилия этого известного лингвиста уже встречалась, когда речь шла о типах языков и грамматик и их классификации.

значение которых человек обычно понимает из того контекста, в котором эти фразы встречаются (если он может это сделать). В языках программирования неоднозначности в грамматиках исключены, поэтому любое предложение языка имеет четко определенную структуру и однозначный смысл, напрямую связанный с этой структурой.

Входной язык компилятора имеет бесконечное множество допустимых предложений, поэтому невозможно задать смысл каждого предложения. Но все входные предложения строятся на основе конечного множества правил грамматики, которые всегда можно найти. Так как этих правил конечное число, то для каждого правила можно определить его семантику (значение). То, как это выполняется для входного языка компилятора, было рассмотрено выше в главах, посвященных лексическому и синтаксическому анализу.

Но абсолютно то же самое можно утверждать и для выходного языка компилятора вне зависимости от того, является ли этот язык языком машинных кодов или другим языком низкого уровня. Выходной язык содержит бесконечное множество допустимых предложений, но все они строятся на основе конечного множества известных правил, каждое из которых имеет определенную семантику (смысл). Если по отношению к исходной программе компилятор выступает в роли распознавателя, то для результирующей программы он является генератором предложений выходного языка. Задача заключается в том, чтобы найти порядок правил выходного языка, по которым необходимо выполнить генерацию.

Грубо говоря, идея СУ-перевода заключается в том, что каждому правилу входного языка компилятора сопоставляется одно или несколько (или ни одного) правил выходного языка в соответствии с семантикой входных и выходных правил. Т о есть при сопоставлении надо выбирать правила выходного языка, которые несут тот же смысл, что и правила входного языка.

СУ-перевод — это основной метод порождения кода резуль тирующей программы на основании резуль татов синтаксического анализа. Для удобства понимания сути метода можно считать, что резуль тат синтаксического анализа представлен в виде дерева синтаксического анализа, хотя в реальных компиляторах это не всегда так.

Суть принципа СУ-перевода заключается в следующем: с каждой вершиной дерева синтаксического разбора N связывается цепочка некоторого промежуточного кода C(N). Код для вершины N строится путем сцепления (конкатенации) в фиксированном порядке последовательности кода C(N) и последовательностей кодов, связанных со всеми вершинами, являющимися прямыми потомками N. В свою очередь, для построения последовательностей кода прямых потомков вершины N потребуется найти последовательности кода для их потомков — потомков второго уровня вершины N и т. д. Процесс перевода идет, таким образом, снизу вверх в строго установленном порядке, определяемом структурой дерева.

Для того чтобы построить СУ-перевод по заданному дереву синтаксического разбора, необходимо найти последовательность кода для корня дерева. Поэтому для каждой вершины дерева порождаемую цепочку кода надо выбирать таким образом, чтобы код, приписываемый корню дерева, оказался искомым кодом для всего оператора, представленного этим деревом. В общем случае необходимо иметь единообразную интерпретацию кода C(N), которая встречалась бы во всех ситуациях, где присутствует вершина N. В принципе, эта задача может оказаться нетривиальной,

так как требует оценки смысла (семантики) каждой вершины дерева. При применении СУ-перевода задача интерпретации кода для каждой вершины дерева решается только разработчиком компилятора.

Возможна модель компилятора, в которой синтаксический анализ исходной программы и генерация кода результирующей программы объединены в одну фазу. Такую модель можно представить в виде компилятора, у которого операции генерации кода совмещены с операциями выполнения синтаксического разбора. Для описания компиляторов такого типа часто используется термин «СУ-компиляция» (синтаксически управляемая компиляция).

Схему СУ-компиляции можно реализовать не для всякого входного языка программирования. Если принцип СУ-перевода применим ко всем входным КС-языкам, то применить СУ-компиляцию оказывается не всегда возможно. Однако известно, что схемы перевода на основе СУ-компиляции можно построить для многих широко распространенных классов КС-языков, в частности для LR- и LL- языков [4 т.1,2, 5].

В процессе СУ-перевода и СУ-компиляции не только вырабатываются цепочки текста выходного языка, но и совершаются некоторые дополнительные действия, выполняемые самим компилятором. В общем случае схемы СУ-перевода могут предусматривать выполнение следующих действий:

- помещение в выходной поток данных команд результирующей программы, представляющих собой результат работы (выход) компилятора;
- выдача пользователю сообщений об обнаруженных ошибках и предупреждениях (которые должны помещаться в другой выходной поток, отличный от потока, используемого для команд результирующей программы);
- порождение и выполнение команд, указывающих, что некоторые действия должны быть произведены самим компилятором (например, операции, выполняемые над данными, размещенными в таблице идентификаторов).

Ниже рассмотрены основные технические вопросы, позволяющие реализовать схемы СУ-перевода и СУ-компиляции. Но прежде чем рассматривать их, необходимо разобраться со способами внутреннего представления программы в компиляторе. От того, как исходная программа представляется внутри компилятора, во многом зависят методы, используемые для обработки команд этой программы.

Способы внутреннего представления программ

Виды внутреннего представления программы

Результатом работы синтаксического анализатора на основе КС-грамматики входного языка является последовательность правил грамматики, примененных для построения входной цепочки. По этой последовательности, зная тип распознавателя, можно построить цепочку вывода или дерево вывода . В этом случае дерево вывода выступает в качестве дерева синтаксического анализа и представляет собой результат работы синтаксического анализатора в компиляторе.

Однако ни цепочка вывода, ни дерево синтаксического разбора не являются целью работы компилятора. Дерево вывода содержит массу избыточной информации, которая для дальнейшей работы компилятора не требуется. Эта информация включает

в себя все нетерминальные символы, содержащиеся в узлах дерева, — после того как дерево построено, они не представляют интереса.

Для полного представления о типе и структуре найденной и разобранной синтаксической конструкции входного языка в принципе достаточно знать последовательность номеров правил грамматики, примененных для ее построения. Однако форма представления этой достаточной информации может быть различной в зависимости как от реализации самого компилятора, так и от фазы компиляции. Эта форма называется *внутренним представлением программы* (иногда используются также термины «промежуточное представление» или «промежуточная программа»).

Все внутренние представления программы обычно содержат в себе две принципиально различные вещи: операторы и операнды. Различия между формами внутреннего представления заключаются лишь в том, как операторы и операнды соединяются между собой. Также операторы и операнды должны отличаться друг от друга вне зависимости от того, в каком порядке они встречаются во внутреннем представлении программы. За различение операндов и операторов, как уже было сказано выше, отвечает разработчик компилятора, который руководствуется семантикой входного языка.

Существует большое количество различных форм внутреннего представления программ. В данной книге будут рассмотрены следующие формы внутреннего представления программ[1]:

- связные списочные структуры, представляющие синтаксические деревья ;
- многоадресный код с явно именуемым результатом (тетрады);
- многоадресный код с неявно именуемым результатом (триады);
- постфиксная (обратная польская) запись операций;
- ассемблерный код;
- машинные команды .

Первые четыре из перечисленных форм внутреннего представления программы не зависят от архитектуры целевой вычислительной системы — они являются *машинно-независимыми* формами внутреннего представления программы. Последние две из перечисленных форм внутреннего представления программы являются *машинно-зависимыми*, так как зависят от архитектуры целевой вычислительной системы.

В каждом конкретном компиляторе может использоваться одна из форм внутреннего представления, выбранная разработчиками. Но чаще всего компилятор не ограничивается использованием только одной формы для внутреннего представления программы. На различных фазах компиляции могут использоваться различные формы, которые по мере выполнения проходов компилятора преобразуются одна в другую.

---

[1] Существует три формы записи выражений — префиксная, инфиксная и постфиксная. При префиксной записи операция записывается перед своими операндами, при инфиксной — между операндами, а при постфиксной — после операндов. Общепринятая запись арифметических выражений является примером инфиксной записи. Запись математических функций и функций в языках программирования является префиксной (другие примеры префиксной записи — команды ассемблера, триады и тетрады в том виде, как они рассмотрены далее). Постфиксная запись в повседневной жизни встречается редко. С нею сталкиваются разве что пользователи стековых калькуляторов и программисты на языке Forth.

Не все из перечисленных форм широко используются в современных компиляторах, и об этом будет сказано по мере их рассмотрения.

Некоторые компиляторы, незначительно оптимизирующие результирующий код, генерируют объектный код по мере разбора исходной программы. В этом случае применяется схема СУ-компиляции, когда фазы синтаксического разбора, семантического анализа, подготовки и генерации объектного кода совмещены в одном проходе компилятора. Тогда внутреннее представление программы существует только условно в виде последовательности шагов алгоритма разбора. В любом случае компилятор всегда будет работать с представлением программы в форме машинных команд или команд ассемблера — иначе он не сможет построить результирующую программу.

Далее будут рассмотрены указанные выше формы внутреннего представления программы, начиная от связных списочных структур.

Синтаксические деревья. Преобразование дерева разбора в дерево операций

Связные списочные структуры, представляющие собой синтаксические деревья, наиболее просто и эффективно строить на этапе синтаксического анализа, руководствуясь правилами и типом вывода, порожденного синтаксическим распознавателем.

В синтаксическом дереве внутренние узлы (вершины) соответствуют операциям, а листья представляют собой операнды. Как правило, листья синтаксического дерева связаны с записями в таблице идентификаторов. Структура синтаксического дерева отражает синтаксис языка, на котором написана исходная программа.

Синтаксические деревья могут быть построены компилятором для любой части исходной программы. Не всегда синтаксическому дереву должен соответствовать фрагмент кода результирующей программы — например, возможно построение синтаксических деревьев для декларативной части языка. В этом случае операции, имеющиеся в дереве, не требуют порождения объектного кода, но несут информацию о действиях, которые должен выполнить сам компилятор над соответствующими элементами. В том случае, когда синтаксическому дереву соответствует некоторая последовательность операций, влекущая порождение фрагмента объектного кода, говорят о дереве операций.

Дерево операций можно непосредственно построить из дерева вывода, порожденного синтаксическим анализатором. Для этого достаточно исключить из дерева вывода цепочки нетерминальных символов, а также узлы, не несущие семантической (смысловой) нагрузки при генерации кода. Примером таких узлов могут служить различные скобки, которые меняют порядок выполнения операций и операторов, но после построения дерева никакой смысловой нагрузки уже не несут, так как им не соответствует никакой объектный код.

То, какой узел в дереве является операцией, а какой — операндом, никак невозможно определить из грамматики, описывающей синтаксис входного языка. Также ниоткуда не следует, каким операциям должен соответствовать объектный код в результирующей программе, а каким — нет. Все это определяется только исходя из семантики — «смысла» — входного языка. Поэтому только разработчик компилятора может четко определить, как при построении дерева операций должны различаться

операнды и сами операции, а также то, какие операции являются семантически незначащими для порождения объектного кода.

Алгоритм преобразования дерева семантического разбора в дерево операций можно представить следующим образом.

Шаг 1. Если в дереве больше не содержится узлов, помеченных нетерминальными символами, то выполнение алгоритма завершено; иначе перейти к шагу 2.

Шаг 2. Выбрать крайний левый узел дерева, помеченный нетерминальным символом грамматики, и сделать его текущим. Перейти к шагу 3.

Шаг 3. Если текущий узел имеет только один нижележащий узел, то текущий узел необходимо удалить из дерева, а связанный с ним узел присоединить к узлу вышележащего уровня (исключить из дерева цепочку) и вернуться к шагу 1; иначе перейти к шагу 4.

Шаг 4. Если текущий узел имеет нижележащий узел (лист дерева), помеченный терминальным символом, который не несет семантической нагрузки, тогда этот лист нужно удалить из дерева и вернуться к шагу 3; иначе перейти к шагу 5.

Шаг 5. Если текущий узел имеет один нижележащий узел (лист дерева), помеченный терминальным символом, обозначающим знак операции, а остальные узлы помечены как операнды, то лист, помеченный знаком операции, надо удалить из дерева, текущий узел пометить этим знаком операции и перейти к шагу 1; иначе перейти к шагу 6.

Шаг 6. Если среди нижележащих узлов для текущего узла есть узлы, помеченные нетерминальными символами грамматики, то необходимо выбрать крайний левый среди этих узлов, сделать его текущим узлом и перейти к шагу 3; иначе выполнение алгоритма завершено.

Если семантика языка задана корректно, то в результате работы алгоритма из дерева будут исключены все нетерминальные символы.

Возьмем в качестве примеров синтаксические деревья, построенные для цепочки `(a+a)*b` из языка, заданного различными вариантами грамматики арифметических выражений. Эти деревья приведены выше в качестве примеров на рис. 4.6, 4.12 и 4.14. Семантически незначащими символами в этой грамматике являются круглые скобки (они задают порядок операций и влияют на синтаксический разбор, но не порождают результирующий код) и пустые строки. Знаки операций заданы символами `+`, `–`, `/` и `*`, остальные символы (`a` и `b`) являются операндами.

В результате применения алгоритма преобразования деревьев синтаксического разбора в дерево операций к деревьям, представленным на рис. 4.6, 4.12 и 4.14, получим дерево операций, представленное на рис. 5.5. Причем, несмотря на то что исходные синтаксические деревья имели различную структуру, зависящую от используемой грамматики, результирующее дерево операций в резуль тате всегда имеет одну и ту же структуру, зависящую только от семантики входного языка.

Дерево операций является формой внутреннего представления программы, которой удобно пользоваться на этапах синтаксического и семантического анализа, а также подготовки к генерации кода, когда еще нет необходимости работать непосредственно с кодами команд резуль тирующей программы. Дерево операций четко отражает связь всех операций между собой, поэтому его удобно использовать также для преобразований, связанных с перестановкой и переупорядочиванием операций без изменения конечного результата (таких как арифметические преобразования). Более подробно эти вопросы рассмотрены в [4 т.2, 5, 15].

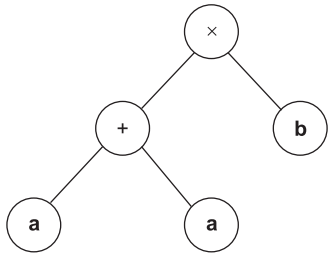

Рис. 5.5. Пример дерева операций для языка арифметических выражений

Недостаток синтаксических деревьев заключается в том, что они представляют собой сложные связные структуры, а потому не могут быть тривиальным образом преобразованы в линейную последовательность команд результирующей программы.

Синтаксические деревья могут быть преобразованы в другие формы внутреннего представления программы, представляющие собой линейные списки, с учетом семантики входного языка. Алгоритмы такого рода преобразований рассмотрены далее. Эти преобразования выполняются на основе принципов СУ-компиляции.

Многоадресный код с явно именуемым результатом (тетрады)

Тетрады представляют собой запись операций в форме из четырех составляющих: операции, двух операндов и результата операции. Например, тетрады могут выглядеть так: `<операция>(<операнд1>,<операнд2>,<результат>)`.

Тетрады представляют собой линейную последовательность команд. При вычислении выражения, записанного в форме тетрад, они вычисляются одна за другой последовательно. Каждая тетрада в последовательности вычисляется так: операция, заданная тетрадой, выполняется над операндами и результат ее выполнения помещается в переменную, заданную результатом тетрады. Если какой-то из операндов (или оба операнда) в тетраде отсутствует (например, если тетрада представляет собой унарную операцию), то он может быть опущен или заменен пустым операндом (в зависимости от принятой формы записи). Результат вычисления тетрады никогда не может быть опущен, иначе тетрада полностью теряет смысл. Порядок вычисления тетрад может быть изменен, но только если допустить наличие тетрад, целенаправленно изменяющих этот порядок (например, тетрады, вызывающие переход на несколько шагов вперед или назад при каком-то условии).

Тетрады представляют собой линейную последовательность, а потому для них несложно написать тривиальный алгоритм, который будет преобразовывать последовательность тетрад в последовательность команд результирующей программы. В этом их преимущество перед синтаксическими деревьями. Тетрады не зависят от архитектуры целевой вычислительной системы, а потому представляют собой машинно-независимую форму внутреннего представления программы.

Тетрады требуют больше памяти для своего представления, чем триады, они также не отражают явно взаимосвязь операций между собой. Кроме того, есть сложности с преобразованием тетрад в машинный код, так как они плохо отображаются

в команды ассемблера и машинные коды, поскольку в наборах команд большинства современных компьютеров редко встречаются операции с тремя операндами.

Например, выражение `A:=B*C+D–B*10`, записанное в виде тетрад, будет иметь вид

```
* (B, C, T1)
+ (T1, D, T2)
* (B, 10, T3)
– (T2, T3, T4)
:= (T4, 0, A)
```

Здесь все операции обозначены соответствующими знаками (при этом присвоение также является операцией). Идентификаторы `T1`,...,`T4` обозначают временные переменные, используемые для хранения результатов вычисления тетрад. Следует обратить внимание, что в последней тетраде (присвоение), которая требует только одного операнда, в качестве второго операнда выступает незначащий операнд `0`.

Многоадресный код с неявно именуемым результатом (триады)

Триады представляют собой запись операций в форме из трех составляющих: операции и двух операндов. Например, триады могут иметь вид`<операция>(<операнд1>, <операнд2>)`. Особенностью триад является то, что один или оба операнда могут быть ссылками на другую триаду в том случае, если в качестве операнда данной триады выступает резуль тат выполнения другой триады. Поэтому триады последовательно номеруют для удобства указания ссылок одних триад на другие (в реализации компилятора в качестве ссылок можно использовать не номера, а ссылки в виде указателей — тогда при изменении порядка следования триад менять ссылки не требуется).

Триады представляют собой линейную последовательность команд. При вычислении выражения, записанного в форме триад, они вычисляются одна за другой последовательно. Каждая триада в последовательности вычисляется так: операция, заданная триадой, выполняется над операндами, а если в качестве одного из операндов (или обоих операндов) выступает ссылка на другую триаду, то берется результат вычисления той триады. Результат вычисления триады нужно сохранять во временной памяти, так как он может быть затребован последующими триадами. Если какой-то из операндов в триаде отсутствует (например, если триада представляет собой унарную операцию), то он может быть опущен или заменен пустым операндом (в зависимости от принятой формы записи). Порядок вычисления триад, как и тетрад, может быть изменен, но только если допустить наличие триад, целенаправленно изменяющих этот порядок (например, триады, вызывающие переход на несколько шагов вперед или назад при каком-то условии).

Триады представляют собой линейную последовательность, а потому , как и для тетрад, для них несложно написать тривиальный алгоритм, который будет преобразовывать последовательность триад в последовательность команд резуль тирующей программы. В этом их преимущество перед синтаксическими деревьями. Однако требуется еще и алгоритм, отвечающий за распределение памяти, необходимой для хранения промежуточных результатов, так как временные переменные для этой цели не используются. В этом отличие триад от тетрад.

Так же как и тетрады, триады представляют собой машинно-независимую форму внутреннего представления программы.

Триады требуют меньше памяти для своего представления, чем тетрады, они также явно отражают взаимосвязь операций между собой, что делает их применение более удобным. Необходимость иметь алгоритм, отвечающий за распределение памяти для хранения промежуточных результатов, не является недостатком, так как это дает возможность эффективно распределять резуль таты не только по доступным ячейкам памяти, но и по имеющимся регистрам процессора. В этом есть определенные преимущества. Кроме того, триады ближе к двухадресным машинным командам, чем тетрады, а именно такие команды более всего распространены в наборах команд большинства современных компьютеров.

Например, выражение `A:=B*C+D—B*10`, записанное в виде триад, будет иметь вид

```
* (B, C)
+ (^1, D)
* (B, 10)
— (^2, ^3)
:= (A, ^4)
```

Здесь операции обозначены соответствующим знаком (присвоение также является операцией), а знак ^ означает ссылку операнда одной триады на результат другой.

Постфиксная запись операций

Постфиксная (обратная польская) запись — очень удобная для вычисления выражений форма записи операций и операндов. Эта форма предусматривает , что знаки операций записываются после операндов.

Более подробно постфиксная запись операций рассмотрена далее в разделе «Обратная польская запись операций».

Ассемблерный код и машинные команды

Машинные команды удобны тем, что при их использовании внутреннее представление программы полностью соответствует объектному коду и сложные преобразования не требуются. Команды ассемблера представляют собой лишь форму записи машинных команд (см. раздел «Трансляторы, компиляторы и интерпретаторы — общая схема работы», глава 2), а потому в качестве формы внутреннего представления программы практически ничем не отличаются от них.

Однако использование команд ассемблера или машинных команд для внутреннего представления программы требует дополнительных структур для отображения взаимосвязи операций. Очевидно, что в этом случае внутреннее представление программы получается зависимым от архитектуры целевой вычислительной системы. Значит, при ориентации компилятора на другой результирующий код потребуется перестраивать как само внутреннее представление программы, так и методы его обработки (при использовании триад или тетрад этого не требуется).

Тем не менее машинные команды — это язык, на котором должна быть записана результирующая программа. Поэтому компилятор так или иначе должен рабо-

тать с ними. Кроме того, только обрабатывая машинные команды (или их представление в форме команд ассемблера), можно добиться наиболее эффективной результирующей программы (см. далее в разделе «Оптимизация кода. Основные методы оптимизации»). Отсюда следует, что любой компилятор работает с представлением результирующей программы в форме машинных команд, однако их обработка происходит , как правило, на завершающих этапах фазы генерации кода.

Обратная польская запись операций

Описание и свойства обратной польской записи операций

Обратная польская запись — это постфиксная запись операций. Она была предложена польским математиком Я. Лукашевичем, откуда и происходит ее название [4 т .2, 5, 18, 58, 63][1].

В этой записи знаки операций записываются непосредственно за операндами. По сравнению с обычной (инфиксной) записью операций в польской записи операнды следуют в том же порядке, а знаки операций — строго в порядке их выполнения. Тот факт, что в этой форме записи все операции выполняются в том порядке, в котором они записаны, делает ее чрезвычайно удобной для вычисления выражений на компьютере. Польская запись не требует учитывать приоритет операций, в ней не употребляются скобки, и в этом ее основное преимущество.

Она чрезвычайно эффективна в тех случаях, когда для вычислений используется стек. Ниже будет рассмотрен алгоритм вычисления выражений в форме обратной польской записи с использованием стека.

Главный недостаток обратной польской записи также проистекает из метода вычисления выражений в ней: поскольку используется стек, то для работы с ним всегда доступна только верхушка стека, а это делает крайне затруднительной оптимизацию выражений в форме обратной польской записи. Практически выражения в форме обратной польской записи почти не поддаются оптимизации.

Но там, где оптимизация вычисления выражений не требуется или не имеет большого значения, обратная польская запись оказывается очень удобным методом внутреннего представления программы.

Обратная польская запись была предложена первоначально для записи арифметических выражений. Однако этим ее применение не ограничивается. В компиляторе можно порождать код в форме обратной польской записи для вычисления практически любых выражений[2]. Для этого достаточно ввести знаки, предусматривающие вычисление соответствующих операций. В том числе можно ввести операции условного и безусловного перехода, предполагающие изменение последовательности хода вычислений и перемещение вперед или назад на некоторое количество ша-

[1] Ошибочно думать, что польская запись выражений используется для записи арифметических выражений в Польше — это не так. В Польше, как и во многих других странах мира, пользуются привычной всем инфиксной формой записи, когда знаки математических операций ставятся между операндами.

[2] Язык программирования Forth является хорошим примером языка, где для вычисления любых выражений явно используются стек и обратная польская запись операций.

гов в зависимости от результата на верхушке стека [4 т.2, 5, 58, 59, 63]. Такой подход позволяет очень широко применять форму обратной польской записи.

Преимущества и недостатки обратной польской записи определяют и сферу ее применения. Так, она очень широко используется для вычисления выражений в интерпретаторах и командных процессорах, где оптимизация вычислений либо отсутствует вовсе, либо не имеет существенного значения.

Вычисление выражений с помощью обратной польской записи

Вычисление выражений в обратной польской записи выполняется элементарно просто с помощью стека. Для этого выражение просматривается в порядке слева направо, и встречающиеся в нем элементы обрабатываются по следующим правилам:

1) если встречается операнд, то он помещается в стек (на верхушку стека);
2) если встречается знак унарной операции (требующей одного операнда), то операнд выбирается с верхушки стека, операция выполняется и резуль тат помещается в стек (на верхушку стека);
3) если встречается знак бинарной операции (требующей двух операндов), то сначала с верхушки стека выбирается второй операнд, затем — первый операнд, операция выполняется и результат помещается в стек (на верхушку стека).

Вычисление выражения заканчивается, когда достигается конец записи выражения. Результат вычисления при этом всегда находится на верхушке стека.

Очевидно, что данный алгоритм можно легко расширить и для более сложных операций, требующих трех и более операндов.

На рис. 5.6 рассмотрены примеры вычисления выражений в обратной польской записи.

Рис. 5.6. Вычисление выражений в обратной польской записи с использованием стека

Схема СУ-компиляции для перевода выражений в обратную польскую запись

Существует множество алгоритмов, которые позволяют преобразовывать выражения из обычной (инфиксной) формы записи в обратную польскую запись.

Далее рассмотрен алгоритм, построенный на основе схемы СУ-компиляции для языка арифметических выражений с операциями +, –, * и /. В качестве основы алгоритма выбрана грамматика арифметических выражений, которая уже многократно рассматривалась ранее в качестве примера.

Эта грамматика приведена далее:

```
G({+,–,/,*,a,b},{S,T,E},P,S):
P:
S → S+T | S–T | T
T → T*E | T/E | E
E → (S) | a | b
```

Схему СУ-компиляции будем строить с таким расчетом, что имеется выходная цепочка символов R и известно текущее положение указателя в этой цепочке p. Распознаватель, выполняя свертку или подбор альтернативы по правилу грамматики, может записывать символы в выходную цепочку и менять текущее положение указателя в ней.

Построенная таким образом схема СУ-компиляции для преобразования арифметических выражений в форму обратной польской записи оказывается исключительно простой [4 т.2, 5]. Она приведена ниже. В ней с каждым правилом грамматики связаны некоторые действия, которые записаны через ; (точку с запятой) сразу за правой частью каждого правила. Если никаких действий выполнять не нужно, в записи следует пустая цепочка (λ).

```
S → S+T; R(p)="+", p=p+1
S → S-T; R(p)="–", p=p+1
S → T; λ
T → T*E; R(p)="*", p=p+1
T → T/E; R(p)="/", p= p+1
T → E; λ
E → (S); λ
E → a; R(p)="a", p=p+1
E → b; R(p)="b", p=p+1
```

Эту схему СУ-компиляции можно использовать, например, для распознавателя на основе грамматики операторного предшествования, рассмотренного ранее в разделе «Восходящие синтаксические распознаватели без возвратов» главы 4. Тогда, в соответствии с принципом СУ-перевода, каждый раз выполняя свертку на основе некоторого правила, распознаватель будет выполнять также и действия, связанные с этим правилом. В результате его работы будет построена цепочка R, содержащая представление исходного выражения в форме обратной польской записи (строго говоря, в данном случае автомат будет представлять собой уже не только КС-распознаватель, но и КС-преобразователь цепочек [4 т.1,2, 5]).

Представленная схема СУ-компиляции, с одной стороны, иллюстрирует , насколько просто выполнить преобразование выражения в форму обратной польской записи, а с другой стороны, демонстрирует , как на практике работает принцип СУ-компиляции.

Более подробно реализация различных схем СУ-перевода рассмотрена в книге [42].

Оптимизация кода. Основные методы оптимизации

Общие принципы оптимизации кода

Как уже говорилось выше, в подавляющем большинстве случаев генерация кода выполняется компилятором не для всей исходной программы в целом, а последовательно для отдельных ее конструкций. Полученные для каждой синтаксической конструкции входной программы фрагменты результирующего кода также последовательно объединяются в общий текст результирующей программы. При этом связи между различными фрагментами результирующего кода в полной мере не учитываются.

В свою очередь, для построения результирующего кода различных синтаксических конструкций входного языка используется метод СУ-перевода. Он также объединяет цепочки построенного кода по структуре дерева без учета их взаимосвязей (что видно из примеров, приведенных выше).

Построенный таким образом код результирующей программы может содержать лишние команды и данные. Это снижает эффективность выполнения результирующей программы. В принципе, компилятор может завершить на этом генерацию кода, поскольку результирующая программа построена и она является эквивалентной по смыслу (семантике) программе на входном языке. Однако эффективность результирующей программы важна для ее разработчика, поэтому большинство современных компиляторов выполняют еще один этап компиляции — оптимизацию результирующей программы (или просто «оптимизацию»), чтобы повысить ее эффективность, насколько это возможно.

Важно отметить два момента: во-первых, выделение оптимизации в отдельный этап генерации кода — это вынужденный шаг. Компилятор вынужден производить оптимизацию построенного кода, поскольку он не может выполнить семантический анализ всей входной программы в целом, оценить ее смысл и, исходя из него, построить результирующую программу. Оптимизация нужна, поскольку результирующая программа строится не вся сразу, а поэтапно. Во-вторых, оптимизация — это необязательный этап компиляции. Компилятор может вообще не выполнять оптимизацию, и при этом результирующая программа будет правильной, а сам компилятор будет полностью выполнять свои функции. Однако практически все компиляторы так или иначе выполняют оптимизацию, поскольку их разработчики стремятся завоевать хорошие позиции на рынке средств разработки программного обеспечения. Оптимизация, которая существенно влияет на эффективность результирующей программы, является здесь немаловажным фактором.

Теперь дадим определение понятию «оптимизация».

Оптимизация программы — это обработка, связанная с переупорядочиванием и изменением операций в компилируемой программе с целью получения более эффективной результирующей объектной программы. Оптимизация выполняется на этапах подготовки к генерации и непосредственно при генерации объектного кода.

В качестве показателей эффективности результирующей программы можно использовать два критерия: объем памяти, необходимый для выполнения результирующей программы (для хранения всех ее данных и кода), и скорость выполнения (быстродействие) программы. Далеко не всегда удается выполнить оптимизацию так, чтобы удовлетворить обоим этим критериям. Зачастую сокращение необходимого программе объема данных ведет к уменьшению ее быстродействия и наоборот. Поэтому для оптимизации обычно выбирается либо один из упомянутых критериев, либо некий комплексный критерий, основанный на них. Выбор критерия оптимизации обычно выполняет непосредственно пользователь в настройках компилятора.

Но, даже выбрав критерий оптимизации, в общем случае практически невозможно построить код результирующей программы, который являлся бы самым коротким или самым быстрым кодом, соответствующим входной программе. Дело в том, что нет алгоритмического способа нахождения самой короткой или самой быстрой результирующей программы, эквивалентной заданной исходной программе. Эта задача в принципе неразрешима. Существуют алгоритмы, которые можно ускорять сколь угодно много раз для большого числа возможных входных данных, и при этом для других наборов входных данных они окажутся неоптимальными [4, т .2, 5]. К тому же компилятор обладает весьма ограниченными средствами анализа семантики всей входной программы в целом. Все, что можно сделать на этапе оптимизации, — это выполнить над заданной программой последовательность преобразований в надежде сделать ее более эффективной.

Чтобы оценить эффективность результирующей программы, полученной с помощью того или иного компилятора, часто прибегают к сравнению ее с эквивалентной программой (программой, реализующей тот же алгоритм), полученной из исходной программы, написанной на языке ассемблера. Лучшие оптимизирующие компиляторы могут получать результирующие объектные программы из сложных исходных программ, написанных на языках высокого уровня, почти не уступающие по качеству программам на языке ассемблера. Обычно соотношение эффективности программ, построенных с помощью компиляторов с языков высокого уровня, к эффективности программ, построенных с помощью ассемблера, составляет 1,1—1,3. То есть объектная программа, построенная с помощью компилятора с языка высокого уровня, обычно содержит на 10—30 % больше команд, чем эквивалентная ей объектная программа, построенная с помощью ассемблера, а также выполняется на 10—30 % медленнее[1].

Это очень неплохие результаты, достигнутые компиляторами с языков высокого уровня, если сравнить трудозатраты на разработку программ на языке ассемблера и языке высокого уровня. Далеко не каждую программу можно реализовать на языке ассемблера в приемлемые сроки (а значит, и выполнить напрямую приведенное выше сравнение можно только для узкого круга программ).

[1] Обычно такое сравнение выполняют для специальных тестовых программ, для которых код на языке ассемблера уже заранее известен. Полученные результаты распространяют на все множество входных программ, поэтому их можно считать очень усредненными.

Оптимизацию можно выполнять на любой стадии генерации кода, начиная от завершения синтаксического разбора и вплоть до последнего этапа, когда порождается код результирующей программы. Если компилятор использует несколько различных форм внутреннего представления программы, то каждая из них может быть подвергнута оптимизации, причем различные формы внутреннего представления ориентированы на различные методы оптимизации [4 т .2, 5, 42, 58, 59, 63]. Т аким образом, оптимизация в компиляторе может выполняться несколько раз на этапе генерации кода.

Принципиально различаются два основных вида оптимизирующих преобразований:

- преобразования исходной программы (в форме ее внутреннего представления в компиляторе), не зависящие от результирующего объектного языка;
- преобразования результирующей объектной программы.

Первый вид преобразований не зависит от архитектуры целевой вычислительной системы, на которой будет выполняться результирующая программа. Обычно он основан на выполнении хорошо известных и обоснованных математических и логических преобразований, производимых над внутренним представлением программы (некоторые из них будут рассмотрены ниже).

Второй вид преобразований может зависеть не только от свойств объектного языка (что очевидно), но и от архитектуры вычислительной системы, на которой будет выполняться результирующая программа. Т ак, например, при оптимизации могут учитываться объем кэш-памяти и методы организации конвейерных операций центрального процессора. В большинстве случаев эти преобразования сильно зависят от реализации компилятора и являются ноу-хау производителей компилятора. Именно этот тип оптимизирующих преобразований позволяет существенно повысить эффективность результирующего кода.

Используемые методы оптимизации ни при каких условиях не должны приводить к изменению «смысла» исходной программы (то есть к таким ситуациям, когда результат выполнения программы изменяется после ее оптимизации). Для преобразований первого вида проблем обычно не возникает. Преобразования второго вида могут вызывать сложности, поскольку не все методы оптимизации, используемые создателями компиляторов, могут быть теоретически обоснованы и доказаны для всех возможных видов исходных программ. Именно эти преобразования могут повлиять на «смысл» исходной программы. Поэтому большинство компиляторов предусматривает возможность отключать те или иные из возможных методов оптимизации.

Нередко оптимизация ведет к тому , что смысл программы оказывается не совсем таким, каким его ожидал увидеть разработчик программы, но не по причине наличия ошибки в оптимизирующей части компилятора, а потому , что пользователь не принимал во внимание некоторые аспекты программы, связанные с оптимизацией. Например, компилятор может исключить из программы вызов некоторой функции с заранее известным результатом, но, если эта функция имела «побочный эффект» — изменяла некоторые значения в глобальной памяти, — смысл программы может измениться. Чаще всего это говорит о плохом стиле программирования исходной программы. Такие ошибки трудноуловимы, для их нахождения следует обратить внимание на предупреждения разработчику программы, выдаваемые семантическим

анализом, или отключить оптимизацию. Применение оптимизации также нецелесообразно в процессе отладки исходной программы.

У современных компиляторов существуют возможности выбора не только общего критерия оптимизации, но и отдельных методов, которые будут использоваться при выполнении оптимизации.

Методы преобразования программы зависят от типов синтаксических конструкций исходного языка. Теоретически разработаны методы оптимизации для многих типовых конструкций языков программирования.

Оптимизация может выполняться для следующих типовых синтаксических конструкций:

- линейных участков программы ;
- логических выражений ;
- циклов ;
- вызовов процедур и функций ;
- других конструкций входного языка.

Во всех случаях могут использоваться как машинно-зависимые , так и машинно-независимые методы оптимизации.

Далее будут рассмотрены только некоторые машинно-независимые методы оптимизации, касающиеся, в первую очередь, линейных участков программы. Перечень их здесь далеко не полный. С остальными машинно-независимыми методами можно более подробно ознакомиться в [4 т.2, 5]. Что касается машинно-зависимых методов, то они, как правило, редко упоминаются в литературе. Некоторые из них рассматриваются в технических описаниях компиляторов. Здесь эти методы будут рассмотрены только в самом общем виде.

Оптимизация линейных участков программы

Принципы оптимизации линейных участков

Линейный участок программы — это выполняемая по порядку последовательность операций, имеющая один вход и один выход. Чаще всего линейный участок содержит последовательность вычислений, состоящих из арифметических операций и операторов присвоения значений переменным.

Любая программа предусматривает выполнение вычислений и присваивание значений, поэтому линейные участки встречаются в любой программе. В реальных программах они составляют существенную часть программного кода. Поэтому для линейных участков разработан широкий спектр методов оптимизации кода.

Кроме того, характерной особенностью любого линейного участка является последовательный порядок выполнения операций, входящих в его состав. Ни одна операция в составе линейного участка программы не может быть пропущена, ни одна операция не может быть выполнена большее числораз, чем соседние с нею операции (иначе этот фрагмент программы просто не будет линейным участком). Это существенно упрощает задачу оптимизации линейных участков программ. Поскольку все операции линейного участка выполняются последовательно, их можно пронумеровать в порядке их выполнения.

Для операций, составляющих линейный участок программы, могут применяться следующие виды оптимизирующих преобразований:

- удаление бесполезных присваиваний;
- исключение избыточных вычислений (лишних операций);
- свертка операций объектного кода;
- перестановка операций;
- арифметические преобразования.

Удаление бесполезных присваиваний заключается в том, что если в составе линейного участка программы имеется операция присвоения значения некоторой произвольной переменной `A` с номером `i`, а также операция присвоения значения той же переменной `A` с номером `j`, `i<j`, и ни в одной операции между `i` и `j` не используется значение переменной `A`, то операция присвоения значения с номером `i` является бесполезной. Фактически бесполезная операция присваивания значения дает переменной значение, которое нигде не используется. Такая операция может быть исключена без ущерба для смысла программы.

В общем случае бесполезными могут оказаться не только операции присваивания, но и любые другие операции линейного участка, результат выполнения которых нигде не используется.

Например, во фрагменте программы:

```
A := B * C;
D := B + C;
A := D * C;
```

операция присвоения `A:=B*C;` является бесполезной и может быть удалена. Вместе с удалением операции присвоения здесь может быть удалена и операция умножения, которая в результате также окажется бесполезной.

Обнаружение бесполезных операций присваивания далеко не всегда столь очевидно, как было показано в примере выше. Проблемы могут возникнуть, если между двумя операциями присваивания в линейном участке выполняются действия над указателями (адресами памяти) или вызовы функций, имеющих так называемый «побочный эффект».

Например, в следующем фрагменте программы:

```
P := @A;
A := B * C;
D := P^ + C;
A := D * C;
```

операция присвоения `A:=B*C;` уже не является бесполезной, хотя это и не столь очевидно. В этом случае неверно следовать простому принципу о том, что если переменная, которой присвоено значение в операции с номером `i`, не встречается ни в одной операции между `i` и `j`, то операция присвоения с номером `i` является бесполезной.

Не всегда удается установить, используется или нет присвоенное переменной значение, только на основании факта упоминания переменной в операциях. Тогда устранение лишних присваиваний становится достаточно сложной задачей, требующей учета операций с адресами памяти и ссылками.

Исключение избыточных вычислений (лишних операций) заключается в нахождении и удалении из объектного кода операций, которые повторно обрабатывают одни и те же операнды. Операция линейного участка с порядковым номером `i` считается лишней, если существует идентичная ей операция с порядковым номером `j`, `j<i`, и никакой операнд, обрабатываемый этой операцией, не изменялся никакой операцией, имеющей порядковый номер между `i` и `j`.

Свертка объектного кода — это выполнение во время компиляции тех операций исходной программы, для которых значения операндов уже известны. Тривиальным примером свертки является вычисление выражений, все операнды которых являются константами. Более сложные варианты алгоритма свертки принимают во внимание также и переменные, и функции, значения для которых уже известны.

Не всегда компилятору удается выполнить свертку, даже если выражение допускает ее выполнение. Например, выражение `A:=2*B*C*3;` может быть преобразовано к виду `A:=6*B*C;`, но при порядке вычислений `A:=2*(B*(C*3));` это не столь очевидно. Для более эффективного выполнения свертки объектного кода можно совместить ее выполнение с другим методом — перестановкой операций.

Хорошим стилем программирования является объединение вместе операций, производимых над константами, чтобы облегчить компилятору выполнение свертки. Например, если имеется константа `PI=3.14`, представляющая соответствующую математическую постоянную, то операцию `b=sin(2πa);` лучше записать в виде `B:=sin(2*PI*A);` или даже `B:=sin((2*PI)*A);`, чем в виде `B:=sin(2*A*PI)`.

Перестановка операций заключается в изменении порядка следования операций, которое может повысить эффективность программы, но не будет влиять на конечный результат вычислений.

Например, операции умножения в выражении `A:=2*B*3*C;` можно переставить без изменения конечного результата и выполнить в порядке `A:=(2*3)*(B*C);`. Тогда представляется возможным выполнить свертку и сократить количество операций.

Другое выражение, `A:=(B+C)+(D+E);`, может потребовать как минимум одной ячейки памяти (или регистра процессора) для хранения промежуточного результата. Но при вычислении его в порядке `A:=B+(C+(D+E));` можно обойтись одним регистром, в то время как результат будет тем же.

Особое значение перестановка операций приобретает в тех случаях, когда порядок их выполнения влияет на эффективность программы в зависимости от архитектурных особенностей целевой вычислительной системы (таких как использование регистров процессора, конвейеров обработки данных и т. п.). Эти особенности рассмотрены далее в подразделе, посвященном машинно-зависимым методам оптимизации.

Арифметические преобразования представляют собой выполнение изменения характера и порядка следования операций на основании известных алгебраических и логических тождеств.

Например, выражение `A:=B*C+B*D;` может быть заменено на `A:=B*(C+D);`. Конечный результат при этом не изменится, но объектный код будет содержать на одну операцию умножения меньше.

К арифметическим преобразованиям можно отнести и такие действия, как замена возведения в степень на умножение; а целочисленного умножения на константы,

кратные 2, — на выполнение операций сдвига. В обоих случаях удается повысить быстродействие программы заменой сложных операций более простыми.

Далее подробно рассмотрены два метода: свертка объектного кода и исключение лишних операций.

Свертка объектного кода

Свертка объектного кода — это выполнение во время компиляции тех операций исходной программы, для которых значения операндов уже известны. Нет необходимости многократно выполнять эти операции в результирующей программе — вполне достаточно один раз выполнить их при компиляции программы.

Простейший вариант свертки — выполнение в компиляторе операций, операндами которых являются константы. Несколько более сложен процесс определения тех операций, значения которых могут быть известны в результате выполнения других операций. Для этой цели при оптимизации линейных участков программы используется специальный алгоритм свертки объектного кода.

Алгоритм свертки для линейного участка программы работает со специальной таблицей **T**, которая содержит пары (<переменная>,<константа>) для всех переменных, значения которых уже известны. Кроме того, алгоритм свертки помечает те операции во внутреннем представлении программы, для которых в результате свертки уже не требуется генерация кода. Так как при выполнении алгоритма свертки учитывается взаимосвязь операций, то удобной формой представления для него являются триады, так как в других формах представления операций (таких как тетрады или команды ассемблера) требуются дополнительные структуры, чтобы отразить связь результатов одних операций с операндами других.

Рассмотрим выполнение алгоритма свертки объектного кода для триад. Для пометки операций, не требующих порождения объектного кода, будем использовать триады специального вида `C(K,0)`.

Алгоритм свертки триад последовательно просматривает триады линейного участка и для каждой триады делает следующее:

1) если операнд есть переменная, которая содержится в таблице **T**, то операнд заменяется соответствующим значением константы;
2) если операнд есть ссылка на особую триаду типа `C(K,0)`, то операнд заменяется значением константы `K`;
3) если все операнды триады являются константами, то триада может быть свернута. Тогда данная триада выполняется, и вместо нее помещается особая триада вида `C(K,0)`, где `K` — константа, являющаяся результатом выполнения свернутой триады (при генерации кода для особой триады объектный код не порождается, а потому она в дальнейшем может быть просто исключена);
4) если триада является присваиванием типа `A := B`, тогда
   - если `B` — константа, то `A` со значением константы заносится в таблицу **T** (если там уже было старое значение для `A`, то это старое значение исключается);
   - если `B` не константа, то `A` вообще исключается из таблицы **T**, если оно там есть.

Рассмотрим пример выполнения алгоритма.

Пусть фрагмент исходной программы (записанной на языке типа Pascal) имеет вид:

```
I := 1 + 1;
I := 3;
J := 6*I + I;
```

Ее внутреннее представление в форме триад будет иметь вид:

1. `.+ (1,1)`
2. `.:= (I,^1)`
3. `.:= (I,3)`
4. `.* (6,I)`
5. `.+ (^4,I)`
6. `.:= (J,^5)`

Процесс выполнения алгоритма свертки показан в табл. 5.1.

Таблица 5.1. Пример работы алгоритма свертки

Триада	Шаг 1	Шаг 2	Шаг 3	Шаг 4	Шаг 5	Шаг 6
1	C (2, 0)	C (2, 0)	C (2, 0)	C (2, 0)	C (2, 0)	C (2, 0)
2	:= (I, ^1)	:= (I, 2)	:= (I, 2)	:= (I, 2)	:= (I, 2)	:= (I, 2)
3	:= (I, 3)	:= (I, 3)	:= (I, 3)	:= (I, 3)	:= (I, 3)	:= (I, 3)
4	* (6, I)	* (6, I)	* (6, I)	C (18, 0)	C (18, 0)	C (18, 0)
5	+ (^4, I)	+ (^4, I)	+ (^4, I)	+ (^4, I)	C (21, 0)	C (21, 0)
6	:= (J, ^5)	:= (J, ^5)	:= (J, ^5)	:= (J, ^5)	:= (J, ^5)	:= (J, 21)
T	(,)	(I, 2)	(I, 3)	(I, 3)	(I, 3)	(I, 3) (J, 21)

Если исключить особые триады типа `C(K,0)` (которые не порождают объектного кода), то в результате выполнения свертки получим следующую последовательность триад:

7. `.:= (I,2)`
8. `.:= (I,3)`
9. `.:= (J,21)`

Алгоритм свертки объектного кода позволяет исключить из линейного участка программы операции, для которых на этапе компиляции уже известен результат. За счет этого сокращается время выполнения [1], а также объем кода результирующей программы.

[1] Даже если принять во внимание, что время на выполнение операции будет потрачено компилятором при порождении результирующей программы, то все равно мы имеем выигрыш во времени. Во-первых, любая результирующая программа создается (а значит, и компилируется окончательно) только один раз, а выполняется многократно; во-вторых, оптимизируемый линейный участок может входить в состав оператора цикла или вызова функции, и тогда выигрыш очевиден.

Свертка объектного кода в принципе может выполняться не только для линейных участков программы. Когда операндами являются константы, логика выполнения программы значения не имеет — свертка может быть выполнена в любом случае. Если же необходимо учитывать известные значения переменных, то нужно принимать во внимание и логику выполнения резуль тирующей программы. Поэтому для нелинейных участков программы (ветвлений и циклов) алгоритм будет более сложным, чем последовательный просмотр линейного списка триад.

Исключение лишних операций

Алгоритм исключения лишних операций просматривает операции в порядке их следования. Так же как и алгоритму свертки, алгоритму исключения лишних операций проще всего работать с триадами, потому что они полностью отражают взаимосвязь операций.

Рассмотрим алгоритм исключения лишних операций для триад.

Чтобы следить за внутренней зависимостью переменных и триад, алгоритм присваивает им некоторые значения, называемые числами зависимости, по следующим правилам:

- изначально для каждой переменной ее число зависимости равно 0, так как в начале работы программы значение переменной не зависит ни от какой триады;
- после обработки `i`-й триады, в которой переменной `A` присваивается некоторое значение, число зависимости `A (dep(A))` получает значение `i`, так как значение `A` теперь зависит от данной `i`-й триады;
- при обработке `i`-й триады ее число зависимости `(dep(i))` принимается равным значению `1+(максимальное из чисел зависимости операндов)`.

Таким образом, при использовании чисел зависимости триад и переменных можно утверждать, что если `i`-я триада идентична `j`-й триаде (`j<i`), то `i`-я триада считается лишней в том и только в том случае, когда `dep(i)=dep(j)`.

Алгоритм исключения лишних операций использует в своей работе триады особого вида `SAME(j,0)`. Если такая триада встречается в позиции с номером `i`, то это означает, что в исходной последовательности триад некоторая триада `i` идентична триаде `j`.

Алгоритм исключения лишних операций последовательно просматривает триады линейного участка. Он состоит из следующих шагов, выполняемых для каждой триады:

1) если какой-то операнд триады ссылается на особую триаду вида `SAME(j,0)`, то он заменяется на ссылку на триаду с номером `j (^j)`;
2) вычисляется число зависимости текущей триады с номером `i` исходя из чисел зависимости ее операндов;
3) если в просмотренной части списка триад существует идентичная `j`-я триада, причем `j<i` и `dep(i)=dep(j)`, то текущая триада `i` заменяется на триаду особого вида `SAME(j,0)`;
4) если текущая триада есть присвоение, то вычисляется число зависимости соответствующей переменной.

Рассмотрим работу алгоритма на примере:

```
D := D + C*B;
A := D + C*B;
C := D + C*B;
```

Этому фрагменту программы будет соответствовать следующая последовательность триад:

1. .* (C,B)
2. .+ (D,^1)
3. .:= (D,^2)
4. .* (C,B)
5. .+ (D,^4)
6. .:= (A,^5)
7. .* (C,B)
8. .+ (D,^7)
9. .:= (C,^8)

Видно, что в данном примере некоторые операции вычисляются дважды над одними и теми же операндами, а значит , они являются лишними и могут быть исключены. Работа алгоритма исключения лишних операций отражена в табл. 5.2.

Таблица 5.2. Пример работы алгоритма исключения лишних операций

Обрабатываемая триада i	Числа зависимости переменных				Числа зависимости триад	Триады, полученные после выполнения
	A	B	C	D	dep(i)	Алгоритма
1) * (C, B)	0	0	0	0	1	1) * (C, B)
2) + (D, ^1)	0	0	0	0	2	2) + (D, ^1)
3) := (D, ^2)	0	0	0	3	3	3) := (D, ^2)
4) * (C, B)	0	0	0	3	1	4) SAME (1, 0)
5) + (D, ^4)	0	0	0	3	4	5) + (D, ^1)
6) := (A, ^5)	6	0	0	3	5	6) := (A, ^5)
7) * (C, B)	6	0	0	3	1	7) SAME (1, 0)
8) + (D, ^7)	6	0	0	3	4	8) SAME (5, 0)
9) := (C, ^8)	6	0	9	3	5	9) := (C, ^5)

Теперь, если исключить триады особого вида `SAME(j,0)`, то в результате выполнения алгоритма получим следующую последовательность триад:

1. .* (C,B)
2. .+ (D,^1)
3. .:= (D,^2)
4. .+ (D,^1)
5. .:= (A,^4)
6. .:= (C,^4)

Обратите внимание, что в итоговой последовательности изменилась нумерация триад и номера в ссылках одних триад на другие. Если в компиляторе в качестве ссылок использовать не номера триад, а непосредственно указатели на них, то изменения ссылок в таком варианте не потребуется.

Алгоритм исключения лишних операций позволяет избежать повторного выполнения одних и тех же операций над одними и теми же операндами. В результате оптимизации по этому алгоритму сокращается и время выполнения, и объем кода результирующей программы.

Другие методы оптимизации программ

Оптимизация вычисления логических выражений

Особенность оптимизации логических выражений заключается в том, что не всегда необходимо полностью выполнять вычисление всего выражения для того, чтобы знать его результат. Иногда по результату первой операции или даже по значению одного операнда можно заранее определить результат вычисления всего выражения.

Операция называется предопределенной для некоторого значения операнда, если ее результат зависит только от этого операнда и остается неизменным (инвариантным) относительно значений других операндов.

Операция логического сложения (`or`) является предопределенной для логического значения «истина» (true), а операция логического умножения (`and`) — предопределена для логического значения «ложь» (false).

Эти факты могут быть использованы для оптимизации логических выражений. Действительно, получив значение «истина» в последовательности логических сложений или значение «ложь» в последовательности логических умножений, нет никакой необходимости далее производить вычисления — результат уже определен и известен.

Например, выражение `A or B or C or D` не имеет смысла вычислять, если известно, что значение переменной `A` есть `True` («истина»).

Компиляторы строят объектный код вычисления логических выражений таким образом, что вычисление выражения прекращается сразу же, как только его значение становится предопределенным. Это позволяет ускорить вычисления при выполнении результирующей программы. В сочетании с преобразованиями логических выражений на основе тождеств булевой алгебры и перестановкой операций эффективность данного метода может быть несколько увеличена.

Не всегда такие преобразования инвариантны к смыслу программы. Например, при вычислении выражения `A or F(B) or G(C)` функции `F` и `G` не будут вызваны и выполнены, если значением переменной `A` является `True`. Это не важно, если результатом этих функций является только возвращаемое ими значение, но если они обладают «побочным эффектом» (о котором было сказано выше в замечаниях), то семантика программы может измениться.

В большинстве случаев современные компиляторы позволяют исключить такого рода оптимизацию, и тогда все логические выражения будут вычисляться до конца, без учета предопределенности результата. Однако лучше избегать такого рода «побочных эффектов», которые говорят, как правило, о плохом стиле программирования.

Хорошим стилем считается также принимать во внимание эту особенность вычисления логических выражений. Тогда операнды в логических выражениях следует стремиться располагать таким образом, чтобы в первую очередь вычислялись те из них, которые чаще определяют все значение выражения. Кроме того, значения функций лучше вычислять в конце, а не в начале логического выражения, чтобы избежать лишних обращений к ним. Так, рассмотренное выше выражение лучше записывать и вычислять в порядке `A or F(B) or G(C)`, чем в порядке `F(B) or G(C) or A`.

Не только логические операции могут иметь предопределенный результат. Некоторые математические операции и функции также обладают этим свойством. Например, умножение на `0` не имеет смысла выполнять.

Другой пример такого рода преобразований — это исключение вычислений для инвариантных операндов .

Операция называется инвариантной относительно некоторого значения операнда, если ее результат не зависит от этого значения операнда и определяется другими операндами.

Например, логическое сложение (`or`) инвариантно относительно значения «ложь» (False), логическое умножение (`and`) — относительно значения «истина»; алгебраическое сложение инвариантно относительно `0`, а алгебраическое умножение — относительно `1`.

Выполнение такого рода операций можно исключить из текста программы, если известен инвариантный для них операнд. Этот метод оптимизации имеет смысл в сочетании с методом свертки операций.

Оптимизация передачи параметров в процедуры и функции

Выше рассмотрен метод передачи параметров в процедуры и функции через стек. Этот метод применим к большинству языков программирования и используется практически во всех современных компиляторах для различных архитектур целевых вычислительных систем (на которых выполняется результирующая программа).

Данный метод прост в реализации и имеет хорошую аппаратную поддержку во многих архитектурах. Однако он является неэффективным в том случае, если процедура или функция выполняет несложные вычисления над небольшим количеством параметров. Тогда всякий раз при вызове процедуры или функции компилятор будет создавать объектный код для размещения ее фактических параметров в стеке, а при выходе из нее — код для освобождения ячеек, занятых параметрами. При выполнении результирующей программы этот код будет задействован. Если процедура или функция выполняет небольшие по объему вычисления, код для размещения параметров в стеке и доступа к ним может составлять существенную часть относительно всего кода, созданного для этой процедуры или функции. А если эта функция вызывается многократно, то этот код будет заметно снижать эффективность ее выполнения.

Существуют методы, позволяющие снизить затраты кода и времени выполнения на передачу параметров в процедуры и функции и повысить в результате эффективность результирующей программы:

- передача параметров через регистры процессора ;
- подстановка кода функции в вызывающий объектный код.

Метод передачи параметров через регистры процессора позволяет разместить все или часть параметров, передаваемых в процедуру или функцию, непосредственно в регистрах процессора, а не в стеке. Это позволяет ускорить обработку параметров функции, поскольку работа с регистрами процессора всегда выполняется быстрее, чем с ячейками оперативной памяти, где располагается стек. Если все параметры удается разместить в регистрах, то сокращается также и объем кода, поскольку исключаются все операции со стеком при размещении в нем параметров.

Понятно, что регистры процессора, используемые для передачи параметров в процедуру или функцию, не могут быть задействованы напрямую для вычислений внутри этой процедуры или функции. Поскольку программно доступных регистров процессора всегда ограниченное количество, то при выполнении сложных вычислений могут возникнуть проблемы с их распределением. Тогда компилятор должен выбрать: либо использовать регистры для передачи параметров и снизить эффективность вычислений (часть промежуточных результатов, возможно, потребуется размещать в оперативной памяти или в том же стеке), либо использовать свободные регистры для вычислений и снизить эффективность передачи параметров. Поэтому реализация данного метода зависит от количества программно доступных регистров процессора в целевой вычислительной системе и от используемого в компиляторе алгоритма распределения регистров. В современных процессорах число программно доступных регистров постоянно увеличивается с разработкой все новых и новых их вариантов, поэтому и значение этого метода постоянно возрастает.

Этот метод имеет ряд недостатков. Во-первых, очевидно, он зависит от архитектуры целевой вычислительной системы. Во-вторых, процедуры и функции, оптимизированные таким методом, не могут быть использованы в качестве процедур или функций библиотек подпрограмм ни в каком виде. Это вызвано тем, что методы передачи параметров через регистры процессора не стандартизованы (в отличие от методов передачи параметров через стек) и зависят от реализации компилятора. Наконец, этот метод не может быть использован, если где бы то ни было в процедуре или функции требуется выполнить операции с адресами (указателями) на параметры.

В большинстве современных компиляторов метод передачи параметров в процедуры и функции выбирается самим компилятором автоматически в процессе генерации кода для каждой процедуры и функции. Однако разработчик имеет, как правило, возможность запретить передачу параметров через регистры процессора либо для определенных функций (для этого используются специальные ключевые слова входного языка), либо для всех функций в программе (для этого используются настройки компилятора).

Некоторые языки программирования (такие, например, как C и C++) позволяют разработчику исходной программы явно указать, какие параметры или локальные переменные процедуры он желал бы разместить в регистрах процессора. Тогда компилятор стремится распределить свободные регистры, в первую очередь, именно для этих параметров и переменных.

Метод подстановки кода функции в вызывающий объектный код (так называемая inline-подстановка) основан на том, что объектный код функции непосредственно включается в вызывающий объектный код всякий раз в месте вызова функции.

Для разработчика исходной программы такая функция (называемая inline-функцией) ничем не отличается от любой другой функции, но для вызова ее в результирующей

программе используется принципиально другой механизм. По сути, вызов функции в результирующем объектном коде вовсе не выполняется — просто все вычисления, производимые функцией, выполняются непосредственно в самом вызывающем коде в месте ее вызова.

Очевидно, что в общем случае такой метод оптимизации ведет не только к увеличению скорости выполнения программы, но и к увеличению объема объектного кода. Скорость увеличивается за счет отказа от всех операций, связанных с вызовом функции — это не только сама команда вызова, но и все действия, связанные с передачей параметров. Вычисления при этом идут непосредственно с фактическими параметрами функции. Объем кода растет , так как приходится всякий раз включать код функции в место ее вызова. Тем не менее, если функция очень проста и включает в себя лишь несколько машинных команд, можно даже добиться сокращения кода результирующей программы, так как при включении кода самой функции в место ее вызова оттуда исключаются операции, связанные с передачей ей параметров.

Как правило, этот метод применим к очень простым функциям или процедурам, иначе объем резуль тирующего кода может существенно возрасти. Кроме того, он применим только к функциям, вызываемым непосредственно по адресу , без применения косвенной адресации через таблицы RTTI (см. раздел «Семантический анализ и подготовка к генерации кода»). Некоторые компиляторы допускают его применение только к функциям, предполагающим последовательные вычисления и не содержащим циклов.

Ряд языков программирования (например, C++) позволяют разработчику явно указать, для каких функций он желает использовать inline-подстановку. В C++, например, для этой цели служит ключевое слово входного языка `inline`.

Оптимизация циклов

Циклом в программе называется любая последовательность участков программы, которая может выполняться повторно.

Циклы присущи подавляющему большинству программ. Во многих программах есть циклы, выполняемые многократно. Большинство языков программирования имеют синтаксические конструкции, специально ориентированные на организацию циклов. Очевидно, такие циклы легко обнаруживаются на этапе синтаксического разбора.

Однако понятие цикла с точки зрения объектной программы , определенной выше, является более общим. Оно включает в себя не только циклы, явно описанные в синтаксисе языка. Циклы могут быть организованы в исходной программе с помощью любых конструкций, допускающих передачу управления (прежде всего, с помощью условных операторов и операторов безусловного перехода). Для резуль тирующей объектной программы не имеет значения, с помощью каких конструкций организован цикл в исходной программе.

Чтобы обнаружить все циклы в исходной программе, используются методы, основанные на построении графа управления программы [4 т.2, 5, 23].

Циклы обычно содержат в себе один или несколько линейных участков , где производятся вычисления. Поэтому методы оптимизации линейных участков позволяют повысить также и эффективность выполнения циклов, причем они оказыва-

ются тем более эффективными, чем больше кратность выполнения цикла. Но есть методы оптимизации программ, специально ориентированные на оптимизацию циклов.

Для оптимизации циклов используются следующие методы:

- вынесение инвариантных вычислений из циклов;
- замена операций с индуктивными переменными ;
- слияние и развертывание циклов .

Вынесение инвариантных вычислений из циклов заключается в вынесении за пределы циклов тех операций, операнды которых не изменяются в процессе выполнения цикла. Очевидно, что такие операции могут быть выполнены один раз до начала цикла, а полученные результаты потом могут использоваться в теле цикла.

Например, цикл[1]

```
for i:=1 to 10 do
begin
A[i] := B * C * A[i];
…
end;
```

может быть заменен последовательностью операций

```
D := B * C;
for i:=1 to 10 do
begin
A[i] := D * A[i];
…
end;
```

если значения `B` и `C` не изменяются нигде в теле цикла. При этом операция умножения `B*C` будет выполнена только один раз, в то время как в первом варианте она выполнялась 10 раз над одними и теми же операндами.

Неизменность операндов в теле цикла может оказаться не столь очевидной. Отсутствие присвоения значений переменным не может служить достоверным признаком их неизменности. В общем случае компилятору придется принимать во внимание все операции, которые так или иначе могут повлиять на инвариантность переменных. Такими операциями являются операции над указателями, вызовы функций (даже если сами переменные не передаются в функцию в качестве параметров, она может изменять их значения через «побочные эффекты»). Поскольку невозможно отследить все изменения указателей (адресов) и все «побочные эффекты» на этапе компиляции, компилятор вынужден отказаться от вынесения из цикла инвариантных переменных всякий раз, когда в теле цикла появляются «подозрительные» операции, которые могут поставить под сомнение инвариантность той или иной переменной.

[1] Конечно, во внутреннем представлении компилятора циклы не могут быть записаны в таком виде, но для наглядности здесь мы их будем представлять в синтаксической записи входного языка (в данном случае — языка Pascal, как и во всех примерах далее). На суть выполняемых операций оптимизации это никак не влияет.

Замена операций с индуктивными переменными заключается в изменении сложных операций с индуктивными переменными в теле цикла на более простые операции. Как правило, выполняется замена умножения на сложение.

Переменная называется индуктивной в цикле, если ее значения в процессе выполнения цикла образуют арифметическую прогрессию. Таких переменных в цикле может быть несколько, тогда в теле цикла их иногда можно заменить одной единственной переменной, а реальные значения для каждой переменной будут вычисляться с помощью соответствующих коэффициентов соотношения (всем переменным должны быть за пределами цикла присвоены значения, если, конечно, они используются).

Простейшей индуктивной переменной является переменная-счетчик цикла с перечислением значений (цикл типа `for`, который встречается в синтаксисе многих современных языков программирования). Более сложные случаи присутствия индуктивных переменных в цикле требуют специального анализа тела цикла. Не всегда выявление таких переменных является тривиальной задачей.

После того как индуктивные переменные выявлены, необходимо проанализировать те операции в теле цикла, где они используются. Часть таких операций может быть упрощена. Как правило, речь идет о замене умножения на сложение [4 т.2, 5, 23, 63].

Например, цикл

```
S := 10;
for i:=1 to N do A[i] := i*S;
```

может быть заменен последовательностью операций

```
S := 10;
T := S; i := 1;
while i <= 10 do
begin
    A[i] := T; T := T + 10; i := i + 1;
end;
```

Здесь использован синтаксис языка Pascal, а `T` — это некоторая новая временная переменная (использовать для той же цели уже существующую переменную `S` не вполне корректно, так как ее значение может быть использовано и после завершения этого цикла). В итоге удалось отказаться от выполнения N операций умножения, выполнив вместо них N операций сложения (которые обычно выполняются быстрее). Индуктивной переменной в первом варианте цикла являлась `i`, а во втором варианте — `i` и `T`.

В другом примере

```
S := 10;
for i:=1 to N do R := R + F(S); S := S + 10;
```

две индуктивные переменные — `i` и `S`. Если заменить одной, то выяснится, что переменная `i` вовсе не имеет смысла, тогда этот цикл можно заменить на последовательность операций

```
S := 10; M := 10 + N*10;
while S <= M do begin R := R + F(S); S := S + 10; end;
```

Здесь удалось исключить N операций сложения для переменной `i` за счет добавления новой временной переменной `M` (как и в предыдущем примере, использован синтаксис языка Pascal).

В современных реальных компиляторах такие преобразования используются достаточно редко, поскольку они требуют довольно сложного анализа программы, в то время как достигаемый выигрыш невелик — разница в скорости выполнения сложения и умножения, равно как и многих других операций, в современных вычислительных системах не столь существенна. Кроме того, существуют варианты циклов, для которых эффективность указанных методов преобразования является спорной [4 т.2, 5, 23].

Слияние и развертывание циклов предусматривает два различных варианта преобразований: слияния двух вложенных циклов в один и замена цикла на линейную последовательность операций.

Слияние двух циклов можно проиллюстрировать на примере циклов

```
for i:=1 to N do
for j:=1 to M do A[i,j] := 0;
```

Здесь происходит инициализация двумерного массива. Но в объектном коде двумерный массив — это всего лишь область памяти размером `N * M`, поэтому (с точки зрения объектного кода, но не входного языка!) эту операцию можно представить так:

```
K := N*M;
for i:=1 to K do A[i] := 0;
```

Развертывание циклов можно выполнить для циклов, кратность выполнения которых известна уже на этапе компиляции. Т огда цикл кратностью N можно заменить линейной последовательностью N операций, содержащих тело цикла.

Например, цикл

```
for i:=1 to 3 do A[i] := i;
```

можно заменить операциями

```
A[1] := 1;
A[2] := 2;
A[3] := 3;
```

Незначительный выигрыш в скорости достигается за счет исключения всех операций с индуктивной переменной, однако объем программы может существенно возрасти.

Машинно-зависимые методы оптимизации

Машинно-зависимые методы оптимизации ориентированы на конкретную архитектуру целевой вычислительной системы, на которой будет выполняться результирующая программа. Как правило, каждый компилятор ориентирован на одну определенную архитектуру целевой вычислительной системы. Иногда можно в настройках компилятора явно указать одну из допустимых целевых архитектур. В любом случае результирующая программа всегда порождается для четко заданной целевой архитектуры.

Архитектура вычислительной системы есть представление аппаратной и программной составляющих частей системы и взаимосвязи между ними с точки зрения систе-

мы как единого целого. Понятие «архитектура» включает в себя особенности и аппаратных, и программных средств целевой вычислительной системы. При выполнении машинно-зависимой оптимизации компилятор может принимать во внимание те или иные ее составляющие. То, какие конкретно особенности архитектуры будут приняты во внимание, зависит от реализации компилятора и определяется его разработчиками.

Количество существующих архитектур вычислительных систем к настоящему времени очень велико. Поэтому не представляется возможным рассмотреть все ориентированные на них методы оптимизации даже в форме краткого обзора. Интересующиеся этим вопросом могут обратиться к специализированной литературе [4 т .2, 5, 33, 58, 59, 63]. Далее будут рассмотрены только два основных аспекта машинно-зависимой оптимизации: распределение регистров процессора и порождение кода для параллельных вычислений.

Распределение регистров процессора

Процессоры, на базе которых строятся современные вычислительные системы, имеют, как правило, несколько программно-доступных регистров. Часть из них может быть предназначена для выполнения каких-либо определенных целей (например регистр — указатель стека или регистр — счетчик команд), другие могут быть использованы практически произвольным образом при выполнении различных операций (так называемые «регистры общего назначения»).

Использование регистров общего назначениядля хранения значений операндов и результатов вычислений позволяет добиться увеличения быстродействия программы, так как действия над регистрами процессора всегда выполняются быстрее, чем над ячейками памяти. Кроме того, в ряде процессоров не все операции могут быть выполнены над ячейками памяти, а потому часто требуется предварительная загрузка операнда в регистр. Резуль тат выполнения операции чаще всего тоже оказывается в регистре, и, если необходимо, его надо выгрузить (записать) в ячейку памяти.

Программно-доступных регистров процессора всегда ограниченное количество. Поэтому встает вопрос об их распределении при выполнении вычислений. Этим занимается алгоритм распределения регистров, который присутствует практически в каждом современном компиляторе в части генерации кода результирующей программы.

При распределении регистров под хранение промежуточных и окончательных результатов вычислений может возникнуть ситуация, когда значение той или иной переменной (связанной с нею ячейки памяти) необходимо загрузить в регистр для дальнейших вычислений, а все имеющиеся доступные регистры уже заняты. Т огда компилятору перед созданием кода по загрузке переменной в регистр необходимо сгенерировать код для выгрузки одного из значений из регистра в ячейку памяти (чтобы освободить регистр). Причем выгружаемое значение затем, возможно, придется снова загружать в регистр. Встает вопрос о выборе того регистра, значение которого нужно выгрузить в память.

При этом возникает необходимость выработки стратегии замещения регистров процессора. Она чем-то сродни стратегиям замещения процессов и страниц в памяти компьютера, которые используются в операционных системах. Однако разница состоит в том, что, в отличие от диспетчера памяти в ОС, компилятор может проана-

лизировать код и выяснить, какое из выгружаемых значений ему понадобится для дальнейших вычислений и когда (следует помнить, что компилятор сам вычислений не производит — он только порождает код для них, иное дело — интерпретатор). Как правило, стратегии замещения регистров процессора предусматривают, что выгружается тот регистр, значение которого будет использовано в последующих операциях позже всех (хотя не всегда эта стратегия является оптимальной).

Кроме общего распределения регистров, могут использоваться алгоритмы распределения регистров специального характера. Например, во многих процессорах есть регистр-аккумулятор, который ориентирован на выполнение различных арифметических операций (операции с ним выполняются либо быстрее, либо имеют более короткую запись). Поэтому в него стремятся всякий раз загрузить чаще всего используемые операнды; он же используется, как правило, при передаче результатов функций и отдельных операторов. Могут быть также регистры, ориентированные на хранение счетчиков циклов, базовых указателей и т. п. Тогда компилятор должен стремиться распределить их именно для тех целей, на выполнение которых они ориентированы.

Оптимизация кода для процессоров, допускающих распараллеливание вычислений

Многие современные процессоры допускают возможность параллельного выполнения нескольких операций. Как правило, речь идет об арифметических операциях.

Тогда компилятор может порождать объектный код таким образом, чтобы в нем содержалось максимально возможное количество соседних операций, все операнды которых не зависят друг от друга. Решение такой задачи в глобальном объеме для всей программы в целом не представляется возможным, но для конкретного оператора оно, как правило, заключается в порядке выполнения операций. В этом случае нахождение оптимального варианта сводится к выполнению перестановки операций (если она возможна, конечно). Причем оптимальный вариант зависит как от характера операции, так и от количества имеющихся в процессоре конвейеров для выполнения параллельных вычислений.

Например, операцию `A+B+C+D+E+F` на процессоре с одним потоком обработки данных лучше выполнять в порядке `(((A+B)+C)+D)+E)+F`. Тогда потребуется меньше ячеек для хранения промежуточных результатов, а скорость выполнения от порядка операций в данном случае не зависит.

Та же операция на процессоре с двумя потоками обработки данных в целях увеличения скорости выполнения может быть обработана в порядке `((A+B)+C)+((D+E)+F)`. Тогда по крайней мере операции `A+B` и `D+E`, а также сложение с их результатами могут быть обработаны в параллельном режиме. Конкретный порядок команд, а также распределение регистров для хранения промежуточных результатов будут зависеть от типа процессора.

На процессоре с тремя потоками обработки данных ту же операцию можно уже разбить на части в виде `(A+B)+(C+D)+(E+F)`. Теперь уже три операции `A+B`, `C+D` и `E+F` могут быть выполнены параллельно. Правда, их результаты уже должны быть обработаны последовательно, но тут следует принять во внимание соседние операции для нахождения наиболее оптимального варианта.

Контрольные вопросы и задачи

Вопросы

1. Справедливы ли следующие утверждения:
   - любой язык программирования является КС-языком;
   - синтаксис любого языка программирования может быть описан с помощью КС-грамматики;
   - любой язык программирования является КЗ-языком;
   - семантика любого языка программирования может быть описана с помощью КС-грамматики?
2. Можно ли построить компилятор без семантического анализатора? Если да, то какие условия должны при этом выполняться?
3. Цепочка символов, принадлежащая любому языку программирования, может быть распознана с помощью распознавателя для КЗ-языков. При этом не будет требоваться дополнительный семантический анализ цепочки. Почему такой подход не применяется в компиляторах на практике?
4. На языке C описаны переменные:

```
int i,j,k; /* целочисленные переменные */
double x,y,z; /* вещественные переменные */
```

   Совпадут ли значения, присвоенные переменным, в каждом случае:
   - `i = x / y;` и `i = (int)(x / y);`
   - `i = x / y;` и `i = ((int)x) / ((int)y);`
   - `i = j / k;` и `i = ((double)j) / (double)k);`
   - `z = x / y;` и `z = ((int)x) / ((int)y);`
   - `z = j / k;` и `z = (double)(j / k);`
   - `z = j / k;` и `z = ((double)j) / ( (double)k);`

   Объясните, какие функции неявного преобразования типов должен будет включить в объектный код компилятор в каждом из перечисленных случаев.
5. Функция ftest, описанная в исходной программе на языке C++ как `void ftest(int i)`, помещается в библиотеку. В объектном коде библиотеки она получит имя, присвоенное компилятором. Это имя может иметь вид, подобный `ftest@i@v`. Если разработчик пользуется одним и тем же компилятором, то имя функции будет присваиваться одно и то же. Почему тем не менее не стоит вызывать функцию из библиотеки по этому имени? Что должен предпринять разработчик, чтобы функция в библиотеке имела имя `ftest`?
6. Синтаксис языка Pascal предусматривает возможность вложения процедур и функций. Как это должно отразиться на организации дисплея памяти, если учесть, что, согласно семантике языка, вложенные процедуры и функции могут работать с локальными данными объемлющих процедур и функций?
7. От чего зависит состав информации, хранящейся в таблице RTTI? Регламентируется ли состав информации в этой таблице синтаксисом или семантикой ис-

ходного языка программирования? Зависит ли информация в таблице RTTI от архитектуры целевой вычислительной системы?

8. Можно ли построить компилятор, исключив фазу оптимизации кода?
9. От чего зависит эффективность объектного кода, построенного компилятором? Влияют ли на эффективность результирующего кода синтаксис и семантика исходного языка программирования?
10. Является ли СУ-перевод наиболее эффективным методом порождения результирующего кода?
11. Какие (или какой) из способов внутреннего представления программы обязательно должен уметь обрабатывать компилятор?
12. Почему имеются трудности с оптимизацией выражений, представленных в форме обратной польской записи?
13. Какой из двух основных методов оптимизации: машинно-зависимый или машинно-независимый может порождать более эффективный результирующий код? Как сочетаются эти два метода оптимизации в компиляторе?
14. Массив `A` содержит 10 элементов (индексы от 0 до 9). Будет ли условный оператор языка Pascal `if (i<10) and (A[i]=0) then ...` правильным с точки зрения порождаемого компилятором объектного кода? Будет ли правильным с этой же точки зрения оператор `if (i<=0) or (i>=10) or (A[i]=0) then ...`? Что можно сказать об объектном коде, порождаемом компилятором для каждого из этих операторов, если компилятор не будет выполнять оптимизацию логических выражений?
15. Какое значение будет иметь переменная `i` после выполнения следующего оператора языка C:

```
if (++i>10 || i++>20 && i++<50 || i++<0) i--;
```

если переменная `i` имеет перед его выполнением следующие значения:

- 9;
- 11;
- 21;
- 0;
- –3?

Как изменится значение переменной при выполнении этого оператора, если предположить, что при порождении объектного кода компилятор не выполняет оптимизацию логических выражений?

Задачи

1. Функции описаны на языке Pascal следующим образом:

```
procedure A(i : integer; j: integer);
var t: float;
begin
…
end;
…
```

продолжение ⇩

окончание

```
procedure B;
var a,b,c: integer;
begin
…
end;
```

Функция A вызывает функцию B, которая затем рекурсивно вызывает функцию A. Отобразите содержимое стека, после выполнения одного такого вызова, двух таких вызовов.

2. На языке Pascal описаны следующие типы данных:

```
type
    TestArray = array[1..4] of integer;
    TestRecord = record
        A : TestArray;
        B : char;
        C : integer;
    end;
    TestUnion = record
        J : byte;
        case I : integer of
            1: (A : TestRecord);
            2: (B : TestArray);
            3: (C : integer);
    end;
```

а также переменные:

```
var
    c1,c2 : char;
    i1,i2 : integer;
    a1 : TestArray;
    r1,r2,r3 : TestRecord;
    u1 : TestUnion;
```

Какой размер памяти потребуется для этих переменных, если считать, что размер памяти, необходимой для типов данных `char` и `byte`, составляет 1 байт, а размер памяти для типа данных `integer` — 2 байта?

Как изменится размер необходимой памяти, если предположить, что архитектура целевой вычислительной системы предполагает выравнивание данных по границе 2 байта? 4 байта?

3. Вычислите следующие выражения, записанные в форме обратной польской записи:
   - `1 2 * 3 5 4 * + +;`
   - `1 2 3 5 4 * * + +;`
   - `1 2 + 3 * 5 + 4 -;`
   - `1 2 + 3 * 5 4 + -.`

   Напишите для каждого из них выражение в форме инфиксной (обычной) записи.

4. Дана последовательность операций:

```
A := 2;
B := A * 3;
C := 4 * 3 + 1;
A := D;
D := B + C;
C := A;
B := A + 4 * C + 3 * (D + 12);
```

   - Преобразуйте ее в форму записи на основе триад.
   - Выполните свертку операций.

5. Дана последовательность операций:

```
A := B * C + D * E;
B := B * C + D * E;
C := B * C + D * E;
D := B * C - (D * E + 2);
E := B * C - (D * E + 3);
A := A * E + B * E + 1;
B := A * E + B * E + 1;
```

   - Преобразуйте ее в форму записи на основе триад.
   - Выполните исключение лишних операций.

Упражнения

1. Даны две грамматики (см. задачи к предыдущей главе):

   G_1({ (,),^,&,~,a},{S,R,T,G,E },$\mathbf{P}_1$, S):

   $\mathbf{P}_1$: S → TR

   R → λ | ^T | ^TR

   T → EG

   G → λ | &E | &EG

   E → ~E | (S) | a;

   G_2({(,),^,&,~,a},{S,T,E},$\mathbf{P}_2$,S)

   $\mathbf{P}_2$: S → S^T | T

   T → T&E | E

   E → ~E | (S) | a.

   Постройте для каждой из них цепочку вывода для цепочки символов a^a&~a&(a^~~a), постройте деревья вывода для каждой из полученных цепочек символов. Преобразуйте полученные деревья вывода в дерево операций. Проделайте то же самое для других цепочек, допустимых в языке, заданном этими грамматиками.

2. Дана грамматика G({(,),^,&,~,a},{S,T,E},**P**,S):

   P: S → S^T | T

   T → T&E | E

   E → ~E | (S) | a.

Постройте схему СУ-компиляции выражений входных цепочек языка, заданного этой грамматикой, во внутреннее представление в форме обратной польской записи.

3. Для грамматики, приведенной выше в упражнении 2, постройте схему СУ-перевода во внутреннее представление программы в форме триад. Выполните перевод в форму триад цепочки символов `a^a&~a&(a^~~a)`.
4. Для грамматики, приведенной выше в упражнении 4, постройте схему СУ-перевода во внутреннее представление программы в форме команд ассемблера (например, Intel 80x86). Выполните перевод в ассемблерный код цепочки символов `a^a&~a&(a^~~a)`.
5. Постройте алгоритм распределения регистров для хранения промежуточных результатов на основе внутреннего представления программы в форме триад.
6. Постройте алгоритм перевода внутреннего представления программы в форме триад в последовательность команд ассемблера.

ГЛАВА 6 Современные системы программирования

Понятие и структура системы программирования

Понятие о системе программирования

Любой компилятор не существует сам по себе, а решает свои задачи в рамках всего системного программного обеспечения. Основная цель компиляторов — обеспечивать разработку новых прикладных и системных программ с помощью языков высокого уровня.

Всякая программа, как системная, так и прикладная, проходит этапы жизненного цикла, начиная от проектирования и вплоть до внедрения и сопровождения [20, 36]. При этом компиляторы — это средства, служащие для создания программного обеспечения на этапах кодирования, тестирования и отладки.

Однако сам по себе компилятор не решает полностью всех задач, связанных с разработкой новой программы. И хотя компилятор является главной составляющей средств разработки, одного только лишь компилятора недостаточно для того, чтобы обеспечить прохождение программой всех указанных этапов жизненного цикла. Поэтому компиляторы — это программное обеспечение, которое функционирует в тесном взаимодействии с другими техническими средствами, применяемыми для разработки программного обеспечения.

Основные технические средства, используемые в комплексе с компиляторами, включают в себя следующие программные модули:

- текстовые редакторы, служащие для создания текстов исходных программ;
- компоновщики, позволяющие объединять несколько объектных модулей, порождаемых компилятором, в единое целое;
- библиотеки прикладных программ , содержащие в себе наиболее часто используемые функции и подпрограммы в виде готовых объектных модулей;
- загрузчики, обеспечивающие подготовку готовой программы к выполнению;
- отладчики, выполняющие программу в заданном режиме с целью поиска, обнаружения и локализации ошибок;
- другие программные средства, служащие для разработки программ и их компонентов (о некоторых из этих средств будет рассказано далее).

Все эти средства разработки функционируют не отдельно, каждое само по себе, а в тесном взаимодействии друг с другом. Данные, полученные одним модулем, поступают на вход другого и наоборот . В современных средствах разработки интеграция модулей столь высока, что пользователь часто даже и не представляет, что он работает с несколькими программными средствами — для него вся система разработки представляет собой единое целое.

Весь этот комплекс программно-технических средств составляет новое понятие, которое здесь названо «системой программирования».

Возникновение систем программирования

Первоначально компиляторы разрабатывались и поставлялись вне связи с другими техническими средствами, с которыми им приходилось взаимодействовать. Единственным средством разработки, которое поставлялось вместе с компилятором, были библиотеки стандартных функций языка программирования, потому что без их наличия разработка программ была просто невозможна. В задачу разработчика входило обеспечить взаимосвязь всех используемых технических средств:

- подготовить тексты исходной программы на входном языке компилятора;
- подать данные в виде текста исходной программы на вход компилятора;
- получить от компилятора результаты его работы в виде набора объектных файлов ;
- подать весь набор полученных объектных файлов вместе с необходимыми библиотеками подпрограмм на вход компоновщику;
- получить от компоновщика единый файл программы (исполняемый файл) и подготовить его к выполнению с помощью загрузчика;
- поставить программу на выполнение, при необходимости использовать отладчик для проверки правильности выполнения программы.

Все эти действия выполнялись с помощью последовательности команд, инициировавших запуск соответствующих программных модулей с передачей им всех необходимых параметров. Параметры передавались каждому модулю в командной строке и представляли собой набор имен файлов и настроек, реализованных в виде специальных «ключей». Пользователи могли выполнять эти команды последовательно вручную, а с развитием средств командных процессоров ОС они стали объединять их в командные файлы.

Со временем разработчики компиляторов постарались облегчить труд пользователей, предоставив им все необходимое множество программных модулей в составе одной поставки компилятора. Теперь компиляторы поставлялись уже вкупе со всеми необходимыми сопровождающими техническими средствами. Кроме того, были унифицированы форматы объектных файлов и файлов библиотек подпрограмм. Теперь разработчики, имея компилятор от одного производителя, могли в принципе пользоваться библиотеками и объектными файлами, полученными от другого производителя компиляторов или от другой команды разработчиков.

Поскольку процессу компиляции всегда соответствует типичная последовательность команд, то разрабатывать для этой цели командный файл в ОС неэффективно и неудобно. Для написания командных файлов компиляции пользователям компи-

ляторов был предложен специальный командный язык, который обрабатывался интерпретатором команд компиляции — программой make [1]. Вся последовательность действий, необходимых для порождения исполняемого файла, представляла собой специального вида интерпретируемую программу на этом языке — Makefi le [22, 31, 53, 55, 66]. В этой программе перечислялись все используемые входные модули, библиотеки подпрограмм, все порождаемые объектные файлы, «ключи» и правила обработки для каждого типа файлов. Makefile позволял в достаточно гибкой и удобной форме описать весь процесс создания программы от порождения исходных текстов до подготовки ее к выполнению.

Появление Makefile можно назвать первым шагом по созданию систем программирования. Язык Makefi le стал стандартным средством, единым для компиляторов всех разработчиков. Это было удобное, но достаточно сложное техническое средство, требующее от разработчика высокой степени подготовки и профессиональных знаний, поскольку сам командный язык Makefile был по сложности сравним с простым языком программирования. Для того чтобы понять, насколько непросто работать с такими системами программирования, в качестве наглядного примера приводится немного сокращенный текст Makefi le, построенного автоматизированным способом с помощью современных средств разработки.

```
# Microsoft Developer Studio Generated NMAKE File
CFG=CiMain - Win32 Debug
!IF "$(OS)" == "Windows_NT"
NULL=
!ELSE
NULL=nul
!ENDIF
CPP=cl.exe
RSC=rc.exe
OUTDIR=.\Debug
INTDIR=.\Debug
# Begin Custom Macros
OutDir=.\Debug
# End Custom Macros
ALL : "$(OUTDIR)\CiMain.exe"
CLEAN :
 -@erase "$(INTDIR)\CiMain.obj"
 -@erase "$(INTDIR)\vc60.idb"
 -@erase "$(INTDIR)\vc60.pdb"
 -@erase "$(OUTDIR)\CiMain.exe"
 -@erase "$(OUTDIR)\CiMain.ilk"
 -@erase "$(OUTDIR)\CiMain.pdb"
"$(OUTDIR)" :
    if not exist "$(OUTDIR)/$(NULL)" mkdir "$(OUTDIR)"
```

продолжение ⇗

[1] От английского глагола «make» — строить, создавать.

окончание

```
CPP_PROJ=/nologo /MLd /W3 /Gm /GX /ZI /Od /D "WIN32" /D "_DEBUG" /D
"_CONSOLE" /D "_MBCS" /Fp"$(INTDIR)\CiMain.pch" /YX /Fo"$(INTDIR)\\" /
Fd"$(INTDIR)\\" /FD /GZ /c

BSC32=bscmake.exe

BSC32_FLAGS=/nologo /o"$(OUTDIR)\CiMain.bsc"

BSC32_SBRS= \

LINK32=link.exe

LINK32_FLAGS=kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.
lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.
lib odbccp32.lib /nologo /subsystem:console /incremental:yes /
pdb:"$(OUTDIR)\CiMain.pdb" /debug /machine:I386 /out:"$(OUTDIR)\
CiMain.exe" /pdbtype:sept

LINK32_OBJS= \

 "$(INTDIR)\CiMain.obj"

"$(OUTDIR)\CiMain.exe" : "$(OUTDIR)" $(DEF_FILE) $(LINK32_OBJS)

    $(LINK32) @<< $(LINK32_FLAGS) $(LINK32_OBJS) <<


.c{$(INTDIR)}.obj::

   $(CPP) @<< $(CPP_PROJ) $< <<


.cpp{$(INTDIR)}.obj::

   $(CPP) @<< $(CPP_PROJ) $< <<


!IF "$(NO_EXTERNAL_DEPS)" != "1"

!IF EXISTS("CiMain.dep")

!INCLUDE "CiMain.dep"

!ELSE

!MESSAGE Warning: cannot find "CiMain.dep"

!ENDIF

!ENDIF

SOURCE=.\CiMain.cpp

"$(INTDIR)\CiMain.obj" : $(SOURCE) "$(INTDIR)"
```

Такая структура средств разработки существовала достаточно долгое время, а в некоторых случаях она используется и по сей день (особенно системными программистами в ОС типа UNIX и Linux [72]).

Появление интегрированных сред разработки

Командная строка — эффективное, но не всегда удобное средство управления компиляцией (хотя среди программистов под ОС типа UNIX обязательно найдутся желающие поспорить с этим). Фактически для каждого сложного проекта разработчикам приходилось писать еще одну дополнительную программу, описывающую на языке Makefile, как следует строить (собирать, компилировать) этот проект . А программирование всегда есть потенциальный источник ошибок. Поэтому развитие систем программирования на этом не завершилось.

Следующим шагом в развитии систем программирования стало появление так называемой «интегрированной среды разработки». Интегрированная среда объединила в себе возможности текстовых редакторов и командный язык компиляции. Пользователь (разработчик исходной программы) теперь не должен был отвечать за передачу данных с выхода одного средства разработки на вход другого, от него также не требовалось описывать этот процесс с помощью языка Makefile. Теперь ему было достаточно только указать в удобной интерфейсной форме состав необходимых для создания программы исходных модулей и библиотек. Команды, библиотеки и «ключи», необходимые компилятору и другим средствам разработки, также задавались в виде интерфейсных форм настройки.

После этого интегрированная среда разработки сама автоматически готовила всю необходимую последовательность команд Makefile, выполняла их, получала результат и сообщала о возникших ошибках при их наличии.

Но главным преимуществом интегрированной среды разработки было то, что все действия пользователь мог выполнять непосредственно в окне редактирования исходного текста программы. Он мог исправлять текст исходных модулей, получать информацию об ошибках непосредственно в исходном тексте программы, не прерывая работу с интегрированной средой. Кроме удобства работы, объединение текстовых редакторов, компиляторов и компоновщиков в единую среду дало системам программирования целый ряд достоинств, о которых будет сказано далее.

Создание интегрированных сред разработки можно назвать вторым шагом в развитии систем программирования. Оно стало возможным благодаря бурному развитию персональных компьютеров и появлению развитых средств интерфейса пользователя (сначала текстовых, а потом и графических). Пожалуй, первой удачной средой такого рода можно признать интегрированную среду программирования Turbo Pascal на основе языка Pascal производства фирмы Borland [43, 54, 56, 70]. Ее популярность на рынке определила тот факт, что со временем все разработчики компиляторов обратились к созданию интегрированных средств разработки для своих продуктов.

Развитие интегрированных сред снизило требования к профессиональным навыкам разработчиков исходных программ. Теперь в простейшем случае от разработчика требовалось только знание исходного языка. Разработчик программы, работающий в интегрированной среде, мог даже не иметь представления о структуре системы программирования, которой он пользуется[1].

Дальнейшее развитие средств разработки также тесно связано с повсеместным распространением развитых средств графического интерфейса пользователя (GUI — Graphical User Interface). Такой интерфейс стал неотъемлемой составной частью многих современных ОС и так называемых графических оболочек [25, 54, 62, 68]. Со временем он стал стандартом де-факто во всех современных прикладных программах.

Это не могло не сказаться на требованиях, предъявляемых к средствам разработки программного обеспечения. В их состав были сначала включены соответствующие библиотеки, обеспечивающие поддержку GUI и взаимодействие с функциями

[1] Конечно, это не плюс такому разработчику, но раньше такая ситуация была принципиально невозможной.

API (Application Program Interface, Прикладной программный интерфейс) операционных систем [14, 15, 62, 68]. А затем для работы с ними потребовались дополнительные средства, обеспечивающие разработку внешнего вида интерфейсных модулей. Такая работа была характерна уже более для дизайнера, чем для программиста.

Для описания графических элементов программ потребовались соответствующие языки. На их основе сложилось понятие «ресурсы» (resources) прикладных программ.

Ресурсами пользовательского интерфейса будем называть множество данных, обеспечивающих внешний вид интерфейса пользователя результирующей программы и не связанных напрямую с логикой ее выполнения. Характерными примерами ресурсов являются: тексты сообщений, выдаваемых программой; цветовая гамма элементов интерфейса; надписи на элементах управления и т. п.

ПРИМЕЧАНИЕ

Термин «ресурсы» следует признать не очень удачным, так как этим словом обозначаются очень многие понятия, связанные с вычислительными системами (например, ресурсы вычислительного процесса). Однако он уже сложился и применяется при работе со средствами разработки, поэтому придется принять его.

Для формирования структуры ресурсов, в свою очередь, потребовались редакторы ресурсов, а затем и компиляторы ресурсов, обрабатывающие результат их работы [1]. Ресурсы, полученные с выхода компиляторов ресурсов, стали обрабатываться компоновщиками и загрузчиками.

Структура современной системы программирования

Системой программирования будем называть весь комплекс программных средств, предназначенных для кодирования, тестирования и отладки программного обеспечения. Нередко системы программирования взаимосвязаны и с другими техническими средствами, служащими целям создания программного обеспечения на более ранних этапах жизненного цикла (от формулировки требований и анализа до проектирования). Однако рассмотрение таких систем выходит за рамки данного учебника.

Системы программирования в современном мире доминируют на рынке средств разработки. Как правило, разработчики компиляторов поставляют свои продукты в составе соответствующей системы программирования. Отдельные компиляторы являются редкостью и, как правило, служат узко специализированным целям. Достаточно подробный обзор современного состояния языков программирования и связанных с ними систем программирования можно найти в [58].

На рис. 6.1 приведена общая структура современной системы программирования. На ней выделены основные составляющие системы программирования и их взаимосвязь. Отдельные составляющие разбиты по группам в соответствии с этапами развития средств разработки.

[1] Наверное, с точки зрения терминологии компиляторы ресурсов правильнее было бы назвать «трансляторы», так как в результате своей работы они обычно порождают не объектный файл, а некий промежуточный код ресурсов. Однако термин «компилятор ресурсов» стал уже общепринятым.

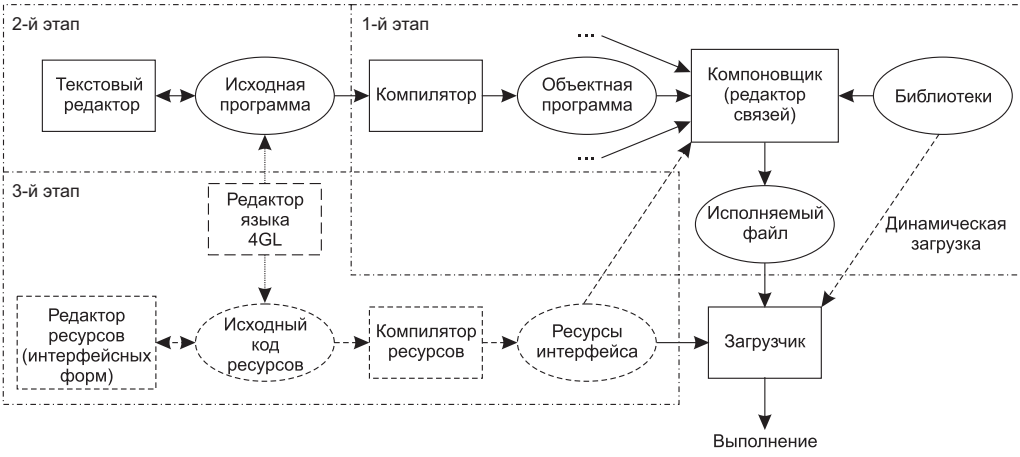

Рис. 6.1. Общая структура и этапы развития систем программирования

Из рис. 6.1 видно, что современная система программирования — это достаточно сложный комплекс различных программно-технических средств. Все они служат цели создания прикладного и системного программного обеспечения.

Текстовый редактор позволяет готовить и вносить изменения в тексты исходных программ, но в современных системах программирования его функции не ограничиваются только этим — с ним связаны практически все сервисные возможности.

Редактор ресурсов дает возможность разработчику готовить ресурсы пользовательского интерфейса для результирующей программы. Как правило, подготовка ресурсов выполняется в графическом виде (в форме графических образов), а результатом является описание ресурсов интерфейса на языке описания ресурсов, которое, в свою очередь, может быть обработано с помощью обычного текстового редактора.

Компиляторы являются главной составляющей системы программирования. В состав системы программирования может входить несколько компиляторов. Основным, конечно, является компилятор с исходного языка, на работу с которым ориентирована данная система. Также необходимым является компилятор ресурсов , обеспечивающий обработку описания ресурсов. Остальные компиляторы включаются в состав системы программирования по мере необходимости (например, в состав многих систем программирования входит компилятор с языка ассемблера).

Библиотеки подпрограмм обеспечивают работоспособность системы программирования и ее конкурентоспособность на рынке средств разработки. Для разработки необходимо как минимум две библиотеки подпрограмм и функций: библиотека функций исходного языка и библиотека функций целевой ОС, на которой будет выполняться результирующая программа. Но, как правило, система программирования предлагает пользователю обширный перечень дополнительных библиотек и функций, и чем шире этот перечень, тем богаче возможности системы программирования.

Компоновщик обеспечивает объединение всех исходных модулей в единый файл. Функции этого средства разработки практически не изменились за всю историю развития систем программирования.

Загрузчик обеспечивает подготовку результирующей программы к выполнению. В современных ОС он не входит в состав систем программирования, а является частью самой ОС.

Отладчик способствует поиску и локализации ошибок в программе. Обычно речь идет о семантических ошибках, так как подавляющее большинство синтаксических ошибок обнаруживается компилятором.

Далее будут более подробно описаны функции основных перечисленных составляющих современных систем программирования.

Тенденция такова, что развитие систем программирования сейчас идет в направлении неуклонного повышения их дружественности и сервисных возможностей. Это связано с тем, что на рынке, в первую очередь, лидируют те системы программирования, которые позволяют существенно снизить трудозатраты, необходимые для создания программного обеспечения на этапах жизненного цикла, связанных с кодированием, тестированием и отладкой программ. Показатель снижения трудозатрат в настоящее время считается более существенным, чем показатели, определяющие эффективность результирующей программы, построенной с помощью системы программирования.

В качестве основных тенденций в развитии современных систем программирования следует указать внедрение в них средств разработки на основе так называемых «языков четвертого поколения» — 4GL (Fourth Generation Languages), — а также поддержку систем «быстрой разработки программного обеспечения» — RAD (Rapid Application Development).

Языки четвертого поколения — 4GL — представляют собой широкий набор средств, ориентированных на проектирование и разработку программного обеспечения. Они строятся на основе оперирования не синтаксическими структурами языка и описаниями элементов интерфейса, а представляющими их графическими образами. На таком уровне проектировать и разрабатывать прикладное программное обеспечение может пользователь, не являющийся квалифицированным программистом, зато имеющий представление о предметной области, на работу в которой ориентирована прикладная программа. Языки четвертого поколения являются следующим (четвертым по счету) этапом в развитии систем программирования.

Описание программы, построенное на основе языков 4GL, транслируется затем в исходный текст и файл описания ресурсов интерфейса, представляющие собой обычный текст на соответствующем входном языке высокого уровня. С этим текстом уже может работать профессиональный программист-разработчик — он может корректировать и дополнять его необходимыми функциями. Дальнейший ход создания программного обеспечения идет уже традиционным путем, как это показано на рис. 6.1.

Такой подход позволяет разделить работу проектировщика, ответственного за общую концепцию всего проекта создаваемой системы, дизайнера, отвечающего за внешний вид интерфейса пользователя, и профессионального программиста, отвечающего непосредственно за создание исходного кода программного обеспечения.

В целом языки четвертого поколения решают уже более широкий класс задач, чем традиционные системы программирования. Они составляют часть средств автоматизированного проектирования и разработки программного обеспечения, поддер-

живающих все этапы жизненного цикла — CASE-систем (Computer Aided Software Engineering). Их рассмотрение выходит за рамки данного учебника.

Более подробную информацию о языках 4GL, технологии RAD и CASE-системах в целом можно получить в [10, 40, 67].

Принципы функционирования систем программирования

Функции текстовых редакторов в системах программирования

Текстовый редактор в интегрированной среде разработки

В принципе, текстовые редакторы появились вне какой-либо связи со средствами разработки. Они решали задачи создания, редактирования, обработки и хранения на внешнем носителе любых текстов, которые необязательно должны были быть исходными текстами программ на языках высокого уровня. Эти функции многие текстовые редакторы выполняют и по сей день.

Появление интегрированных сред разработки на определенном этапе развития средств разработки программного обеспечения позволило непосредственно включить текстовые редакторы в состав этих средств. Первоначально такой подход привел к тому, что пользователь (разработчик исходной программы) работал исключительно в среде текстового редактора, не отрываясь от него для выполнения компиляции, компоновки, загрузки и запуска программы на выполнение. Для этого потребовалось создать средства, позволяющие отображать ход всего процесса разработки программы в среде текстового редактора — такие, например, как метод отображения ошибок в исходной программе, обнаруженных на этапе компиляции, с позиционированием на место в тексте исходной программы, содержащее ошибку.

Можно сказать, что с появлением интегрированных сред разработки ушло в прошлое то время, когда разработчики исходных текстов вынуждены были первоначально готовить тексты программ на бумаге с последующим вводом их в компьютер. Процессы написания текстов и собственно создание программного обеспечения стали единым целым.

Интегрированные среды разработки оказались очень удобным средством. Они стали завоевывать рынок средств разработки программного обеспечения. А с их развитием расширялись и возможности, предоставляемые разработчику в среде текстового редактора. Со временем появились средства пошаговой отладки программ непосредственно по их исходному тексту, объединившие в себе возможности отладчика и редактора исходного текста. Другим примером может служить очень удобное средство, позволяющее графически выделить в исходном тексте программы все лексемы исходного языка по их типам — оно сочетает в себе возможности редактора исходных текстов и лексического анализатора компилятора.

В итоге в современных системах программирования текстовый редактор стал важной составной частью, которая не только позволяет пользователю подготавливать исходные тексты программ, но и выполняет все интерфейсные и сервисные функции,

предоставляемые пользователю системой программирования. И хотя современные разработчики по-прежнему могут использовать любые средства для подготовки исходных текстов программ, как правило, они предпочитают пользоваться тем текстовым редактором, который включен в состав системы программирования.

Лексический анализ «на лету»

Лексический анализ «на лету» — это функция текстового редактора в составе системы программирования. Она заключается в поиске и выделении лексем входного языка в тексте программы непосредственно в процессе ее создания разработчиком.

Реализуется это следующим образом: в то время, когда разработчик готовит исходный текст программы (набирает его вручную или получает из некоторого другого источника), система программирования параллельно выполняет поиск лексем в этом тексте. Поскольку скорость подготовки исходного текста даже самым квалифицированным пользователем намного уступает скорости его обработки лексическим анализатором на современном компьютере, для пользователя процесс поиска лексем происходит незаметно и не создает задержек — создается полная иллюзия параллельной работы[1].

В простейшем случае обнаруженные лексемы просто выделяются в тексте с помощью графических средств интерфейса текстового редактора — цветом, шрифтом и т. п. Это облегчает труд разработчика программы, делает исходный текст более наглядным и способствует обнаружению ошибок на самом раннем этапе — на этапе подготовке исходного кода.

В более развитых системах программирования найденные лексемы не просто выделяются по ходу подготовки исходного текста, но и помещаются в таблицу идентификаторов компилятора, входящего в состав системы программирования. Такой подход позволяет экономить время на этапе компиляции, поскольку первая ее фаза — лексический анализ — уже выполнена на этапе подготовки исходного текста программы. Но преимущество такого подхода в основном заключается не в этом, а в том, что таблица идентификаторов, созданная на этапе создания исходного кода, может помочь разработчику в дальнейшей подготовке исходного кода. Эта возможность используется в системах подсказок и гиперссылок, о которых говорится в следующем подразделе.

Системы гиперссылок подсказок и справок

Следующей сервисной возможностью, предоставляемой разработчику системой программирования за счет лексического анализа «на лету», является возможность обращения разработчика к таблице идентификаторов в ходе подготовки исходного текста программы. Разработчик может дать компилятору команду найти нужную ему лексему в таблице. Поиск может выполняться по типу или по какой-то части информации лексемы (например, по нескольким первым буквам). Причем поиск может быть контекстно-зависимым — система программирования предоставит разработчику возможность найти лексему именно того типа, который может быть использован в данном месте исходного текста. Кроме самой лексемы, разработчику

[1] Конечно, о реальной параллельной работе текстового редактора и лексического анализатора может идти речь, только если они работают в многопроцессорной вычислительной системе.

может быть предоставлена некоторая информация о ней — например, типы и состав формальных параметров для функции, перечень доступных методов для типа или экземпляра класса. Это опять же облегчает труд разработчика, поскольку избавляет его от необходимости помнить состав функций и типов многих модулей или обращаться лишний раз к документации и справочной информации.

Обращение к таблице идентификаторов для разработчика реализуется в виде двух сервисных функций: подсказок и гиперссылок.

Подсказка возникает в том случае, когда разработчик подготовил какую-то часть исходного кода и сделал паузу перед подготовкой следующей части исходного кода. В этот момент система программирования имеет возможность подсказать ему в текстовом редакторе, какой код может следовать далее. Подсказка может иметь вид пояснения или варианта кода:

- *пояснение* дает разработчику представление о том, какой код должен следовать далее, но не предлагает этот код (разработчик должен набрать код самостоятельно) — например, после ввода наименования функции появляется подсказка, содержащая перечень и типы параметров функции;
- *вариант кода* предлагает разработчику часть исходного текста или шаблон исходного текста, который может следовать за уже подготовленным фрагментом (разработчик может выбрать один из вариантов заранее подготовленного кода) — например, после ввода переменной, соответствующей некоторому классу, подсказка предлагает перечень методов и свойств этого класса, из которых можно выбрать один нужный.

Конечно, подсказки — это лишь сервисные возможности, облегчающие труд программиста, но не подменяющие его. Кроме того, поскольку система программирования в данном случае судит об исходном коде на основании результатов предварительного лексического анализа, она может ошибаться. Поэтому разработчик всегда может ввести исходный код, отличный от предложенного подсказкой, или вообще отказаться от системы подсказок.

Гиперссылка позволяет сразу же перейти от одной взаимосвязанной части уже готового исходного кода к другой. Например, по имени функции получить ее описание, по имени переменной — тип этой переменной, а по имени сложного типа данных — его описание.

Другой удобной сервисной функцией в современных системах программирования является система *справок*. Как правило, она содержит три основные части:

1) справку по семантике и синтаксису используемого входного языка;
2) руководство по работе с самой системой программирования;
3) справку о функциях библиотек, входящих в состав системы программирования.

Система справок (Help) в настоящее время является составной частью многих прикладных и системных программ. Как правило, она поддерживается соответствующими утилитами ОС. Поэтому нет ничего удивительного в том, что на рынке нет систем программирования, которые не были бы снабжены справками. Главной особенностью справок в системе программирования является тот факт , что, как и подсказки, они должны быть контекстно-зависимыми.

Более того, разработчик, создающий программу с помощью системы программирования, конечно, тоже будет желать, чтобы его результирующая программа была оснащена справкой. Для этого системы программирования включают в свой состав сервисные функции, позволяющие создавать и дополнять систему подсказок и справок. Это делается таким образом, чтобы разработчик мог создавать и распространять вместе со своими прикладными программами соответствующие им подсказки и справки.

Компилятор как составная часть системы программирования

Компиляторы являются, безусловно, базовыми модулями в составе любой системы программирования. И основным, конечно же, будет компилятор с исходного языка. Поэтому не случайно то, что они стали главным предметом рассмотрения в данном учебнике. Без компилятора никакая система программирования не имеет смысла, а все остальные ее составляющие на самом деле служат лишь целям обеспечения работы компилятора и выполнения им своих функций.

От первых этапов развития систем программирования вплоть до появления интегрированных сред разработки пользователи (разработчики исходных программ) всегда, так или иначе, имели дело с компилятором. Они непосредственно взаимодействовали с ним как с отдельным программным модулем.

Сейчас, работая с системой программирования, пользователь, как правило, имеет дело только с ее интерфейсной частью, которую обычно представляет текстовый редактор с расширенными функциями. Запуск модуля компилятора и вся его работа происходят автоматически и скрыты от пользователя — разработчик видит только конечные результаты выполнения компилятора. Хотя многие современные системы программирования сохранили прежнюю возможность непосредственного взаимодействия разработчика с компилятором, но пользуется этими средствами только узкий круг профессионалов. Большинство пользователей редко непосредственно сталкиваются с компиляторами.

Поэтому компилятор, функционирующий в составе системы программирования, должен быть создан таким образом, чтобы иметь возможность взаимодействовать с другими частями этой системы. Во-первых, он должен представлять информацию об обнаруженных ошибках и предупреждениях таким образом, чтобы она могла быть обработана в текстовом редакторе для наглядного отображения обнаруженных ошибок и предупреждений в тексте исходной программы. Во-вторых, он должен иметь возможность динамически отображать ход процесса компиляции, чтобы у разработчика не создавалось впечатления «зависания» системы программирования (большинство реальных программ содержат десятки и сотни тысяч строк исходного кода, поэтому процесс компиляции занимает некоторое время).

На самом деле, кроме основного компилятора, выполняющего перевод исходного текста на входном языке в язык машинных команд, большинство систем программирования могут содержать в своем составе целый ряд других компиляторов и трансляторов. Так, большинство систем программирования содержат в своем составе и компилятор с языка ассемблера, и компилятор (транслятор) с входного языка описания ресурсов.

Тем не менее, работая с любой системой программирования, следует помнить, что основным модулем ее всегда является компилятор. Именно технические характеристики компилятора прежде всего влияют на эффективность резуль тирующих программ, порождаемых системой программирования.

Компоновщик. Назначение и функции компоновщика

Компоновщик (или редактор связей) предназначен для связывания между собой объектных файлов, порождаемых компилятором, а также файлов библиотек, входящих в состав системы программирования. Название «компоновщик» происходит от английского «link» (связывать, сцеплять). Во многих системах программирования файл компоновщика носит имя «linker», а разработчики на жаргонном языке нередко называют его «линковщик», или «линкер».

ВНИМАНИЕ

Автору встречались в технической литературе примеры, когда компоновщик называли загрузчиком. С точки зрения автора, это принципиально неправильно. Функции компоновщика и загрузчика существенно различаются. В современных системах программирования загрузчик, как правило, отсутствует — его функции выполняет ОС.

Объектный файл (или набор объектных файлов) не может быть исполнен до тех пор, пока все модули и секции не будут в нем увязаны между собой. Это и делает редактор связей (компоновщик). Резуль татом его работы является единый файл (часто называемый «исполняемым файлом»), который содержит весь текст резуль тирующей программы на языке машинных кодов.

Задача компоновщика проста — он должен пройти весь код резуль тирующей программы, начиная от места вызова ее главной исполняемой функции (точки входа) и до конца (точки выхода), найти все вызовы внешних процедур и функций, обращения к внешним переменным и увязать их с кодом других модулей, где описаны эти процедуры, функции и переменные. Причем в этих модулях, в свою очередь, могут быть обращения к другим внешним функциям — и т . д. Функции, описанные в разных модулях исходной программы, могут вызывать друг друга сколь угодно раз — компоновщик должен найти соответствие каждому из этих вызовов и определить («разрешить») соответствующую ссылку (адрес). Отсюда и происходит другое название компоновщика — «редактор связей».

Ссылки (вызовы процедур или функций, обращения к переменным), которым компоновщик не смог найти соответствие, называются «неразрешенными» [1]. В готовой к выполнению результирующей программе не должно быть неразрешенных ссылок. Компоновщик порождает сообщение об ошибке, если при попытке собрать объектные файлы в единое целое он не смог обнаружить какой-либо необходимой составляющей.

Компоновщик начинает свою работу с того, что выбирает из первого объектного модуля программную секцию и присваивает ей начальный адрес. Программные секции остальных объектных модулей получают адреса относительно начального адреса в порядке следования. При этом может выполняться также функция выравнивания начальных адресов программных секций. Одновременно с объединением текстов

[1] Происходит от слова «решать», а не «разрешать».

программных секций объединяются секции данных, таблицы идентификаторов и внешних имен. Разрешаются межсекционные ссылки.

Процедура разрешения ссылок сводится к вычислению значений адресных констант процедур, функций и переменных с учетом перемещений секций относительно начала собираемого программного модуля. Если при этом обнаруживаются ссылки на внешние переменные, отсутствующие в списке объектных модулей, редактор связей организует их поиск в библиотеках, доступных в системе программирования. Если же и в библиотеке необходимую составляющую найти не удается, формируется сообщение об ошибке. То же самое может произойти, если описание некоторой переменной, процедуры или функции будет обнаружено более чем в одном объектном файле среди обрабатываемых компоновщиком. Тогда компоновщик не сможет определить, к какому из них относится ссылка[1].

В современных системах программирования компоновщик включает в состав исполняемого файла не только код объектных модулей, который подготовил компилятор исходной программы, но и описание ресурсов пользовательского интерфейса, которое готовит компилятор ресурсов.

Как и компилятор при распределении памяти, компоновщик работает с относительными адресами переменных, процедур и функций — условными единицами, отсчитываемыми от начала программы. Так же как и компилятор, он не может знать адресов памяти, в которых будет выполняться результирующая программа.

Компоновщик работает с теми именами процедур и функций, которые им присвоил компилятор в таблице идентификаторов. Эти имена, как уже было сказано ранее, могут отличаться от имен, данных пользователем. Поэтому , если два или более обращения к одной и той же функции или переменной в исходном тексте программы выполнены по-разному, компилятор может дать им разные имена и ссылка останется неразрешенной, что приведет к сообщению об ошибке от компоновщика (см. раздел «Семантический анализ и подготовка к генерации кода» главы 5). Так, исходный код программы может влиять на работу компоновщика и порождать ошибки, обнаруживаемые системой программирования только на этапе компоновки резуль тирующей программы.

Другая возможная ошибка на этапе компоновки — несоответствие между объектным файлом и файлом описания процедуры или функции. Дело в том, что компилятор порождает вызовы процедур и функций, исходя из их описания. Если функция описана в одном из исходных файлов, то он может проверить соответствие ее описания и исходного кода, но если обращение выполняется к библиотечной функции, то такая возможность отсутствует (библиотеки функций поставляются, как правило, в виде объектного кода). В том случае, когда имена функций совпадают, но описание не соответствует реальному объектному коду, может возникнуть трудно обнаруживаемая ошибка — такую ошибку можно отследить только при отладке результирующей программы. Конечно же, такие ошибки крайне редко встречаются в библиотеках систем программирования и других библиотеках процедур и функций, поставляемых известными производителями.

[1] В некоторых системах программирования такие ссылки разрешаются по порядку следования файлов — ссылка относится к тому объектному файлу или библиотеке, которые идут первыми по порядку среди всех возможных. Однако такие ссылки — потенциальный источник трудно обнаруживаемых ошибок.

Компоновщик — это достаточно простая компонента в составе системы программирования. От него не зависит эффективность результирующей программы, но ее объем может зависеть от типа компоновщика. В простейшем случае при обнаружении ссылки на некоторый объектный файл компоновщик подключает к исполняемому файлу весь код этого объектного файла и все его данные. Т огда алгоритм работы компоновщика прост, но в этом варианте мы получаем исполняемый файл результирующей программы, который может содержать некоторую часть объектного кода, к которой никогда не будет обращений при выполнении программы. В более сложном случае компоновщик при обнаружении ссылки подключает к исполняемому файлу не весь код объектного файла, а только ту его часть, которая связана с разрешаемой ссылкой. В этом случае исполняемый файл резуль тирующей программы будет иметь меньший объем, но код компоновщика усложнится, так как ему нужно будет отслеживать не только внешние, но и внутренние ссылки. Однако это незначительное усложнение, и, как правило, все современные системы программирования оснащены компоновщиками второго типа.

Обычно компоновщик формирует простейший программный модуль, создаваемый как единое целое. Однако в общем случае компоновщик может создавать и другие модули: программные модули с оверлейной структурой, объектные модули библиотек и модули динамически подключаемых библиотек.

Загрузчики и отладчики. Функции загрузчика

Трансляция адресов. Настраивающий загрузчик

Объектные модули строятся на основе так называемых относительных адресов . Компилятор, порождающий объектные файлы, а затем и компоновщик, объединяющий их в единое целое, не могут знать точно, в какой физической области памяти компьютера будет располагаться программа в момент ее выполнения. Поэтому они работают не с реальными адресами ячеек ОЗУ, а с некоторыми относительными адресами. Такие адреса отсчитываются от условной точки, принятой за начало области памяти, занимаемой резуль тирующей программой (обычно это точка начала первого модуля программы).

Конечно, ни одна программа не может быть исполнена в этих относительных адресах. Поэтому требуется модуль, который выполнял бы преобразование относительных адресов в реальные (абсолютные) адреса непосредственно в момент запуска программы на выполнение. Этот процесс называется трансляцией адресов и выполняет его специальный модуль, называемый загрузчиком.

Однако загрузчик не всегда является составной частью системы программирования, поскольку выполняемые им функции зависят непосредственно от архитектуры целевой вычислительной системы, в которой выполняется резуль тирующая программа, созданная системой программирования. На первых этапах развития ОС загрузчики существовали в виде отдельных модулей, которые выполняли трансляцию адресов и готовили программу к выполнению — создавали так называемый «образ задачи». Такая схема была характерна для многих ОС (например, для ОСРВ на ЭВМ типа СМ-1, ОС RSX/11 или RAFOS на ЭВМ типа СМ-4 и т . п. [17, 46]). Образ задачи можно было сохранить на внешнем носителе или же создавать его вновь всякий раз при подготовке программы к выполнению.

С развитием вычислительных систем появилась возможность выполнять трансляцию адресов непосредственно в момент запуска программы на выполнение. Для этого в состав исполняемого файла включается таблица, содержащая перечень ссылок на адреса, которые необходимо подвергнуть трансляции. В момент запуска исполняемого файла ОС обрабатывает эту таблицу и преобразовывает относительные адреса в абсолютные. Т акая схема, например, была характерна для ОС типа MS-DOS, которые были широко распространены в среде персональных компьютеров. В этой схеме модуль загрузчика как таковой отсутствует (фактически он входит в состав ОС), а система программирования ответственна за подготовку таблицы трансляции адресов — эту функцию выполняет компоновщик.

В современных ОС существуют сложные методы преобразования адресов, которые работают непосредственно уже во время выполнения программы. Эти методы основаны на возможностях, аппаратно заложенных в архитектуру вычислительных комплексов. Методы трансляции адресов могут быть основаны на сегментной, страничной и сегментно-страничной организации памяти [14, 15, 62]. Тогда для выполнения трансляции адресов в момент запуска программы должны быть подготовлены соответствующие системные таблицы. Эти функции целиком ложатся на модули ОС, поэтому они не выполняются в системах программирования.

Загрузчик, который выполняет трансляцию адресов в момент запуска программы, называется *настраивающим загрузчиком*. В современных вычислительных системах такой загрузчик входит в состав ОС, а компоновщик, входящий в состав системы программирования, готовит для настраивающего загрузчика таблицу трансляции адресов, входящую в состав исполняемого файла. Каждая ОС содержит в себе свой настраивающий загрузчик и предусматривает свой формат таблицы трансляции адресов — система программирования при создании исполняемого файла должна учитывать это. Это одна из причин, почему исполняемые файлы для различных ОС несовместимы между собой, даже если они предназначены для одной и той же архитектуры вычислительной системы (например, исполняемый файл для ОС типа UNIX не будет напрямую исполняться в среде MS-DOS, даже если обе ОС установлены на Intel-совместимом персональном компьютере).

Динамические загрузчики

Новые задачи встали перед загрузчиками с развитием ОС. В современных ОС появилась возможность создавать исполняемые программы, имеющие оверлейную структуру, использовать динамически загружаемые библиотеки и ресурсы пользовательского интерфейса.

Первоначально ОС выполняла загрузку в оперативную память компьютера полностью всей исполняемой программы вместе с кодом всех используемых в ней библиотек и модулей в момент запуска этой программы на выполнение. При этом, если объем программы превышал объем памяти, доступной для ее выполнения, программа не могла быть выполнена. Также, если код какой-нибудь библиотеки использовался в нескольких программах, всякий раз он заново загружался в оперативную память. Это вело к неэффективному использованию памяти.

С развитием оверлейных структур и динамических библиотек появилась возможность загружать исполняемый код модуля программы или библиотеки в оператив-

ную память не сразу же в момент запуска программы на выполнение, а в тот момент, когда произойдет обращение к функции или процедуре, содержащейся в этом коде [14, 15, 62]. При этом, если обращение к модулю или библиотеке не происходит, они в оперативную память компьютера не загружаются — и память на хранение их кода не расходуется. Кроме того, если код функций какой-нибудь библиотеки используется несколькими программами (особенно это относится к системным библиотекам), то нет необходимости несколько раз загружать один и тот же фрагмент кода — можно по мере необходимости обращаться к уже загруженному в оперативную память фрагменту. Напротив, код библиотеки, который не используется в данный момент ни одной программой, может быть удален из оперативной памяти (при возникновении обращения к нему он будет заново подгружен в память).

В предельном случае вся исполняемая программа может представлять собой незначительный по объему базовый модуль, загружаемый в память, а остальные модули и библиотеки будут тогда вызываться и подгружаться по мере необходимости. Такая структура позволяет существенно повысить эффективность использования оперативной памяти.

За обеспечение работы с динамически загружаемыми библиотеками и модулями отвечает *динамический загрузчик*.

Динамический загрузчик всегда есть в составе ОС. В его задачи входит следить за обращениями к динамически загружаемым библиотекам и модулям, загружать в оперативную память используемые библиотеки, следить за количеством обращений к каждой библиотеке, освобождать память, занимаемую неиспользуемыми библиотеками. При этом каждая динамически загружаемая библиотека или модуль должны представлять собой исполняемый файл определенной структуры, содержащий в себе данные, необходимые для трансляции адресов. При обращении к ней она загружается в оперативную память с помощью настраивающего загрузчика. При этом код модуля или библиотеки может использоваться несколькими исполняемыми программами, а данные должны дублироваться всякий раз при новом обращении, поскольку они должны быть уникальными для каждой исполняемой программы. Более подробно о работе с динамическими библиотеками вы можете узнать в [2, 14, 15, 25, 27, 28, 37, 48, 54, 68].

Как и формат исполняемых файлов, формат динамически загружаемых библиотек и модулей зависит от типа ОС. При создании динамической библиотеки система программирования должна учитывать, в архитектуре какой вычислительной системы она будет выполняться. В остальном создание таких библиотек мало чем отличается от создания исполняемых файлов.

Другая задача, решаемая динамическим загрузчиком, — обеспечение доступа исполняемой программы к используемым ею ресурсам пользовательского интерфейса. Ресурсы пользовательского интерфейса могут содержаться внутри самого исполняемого файла или же в специально разработанных для этого файлах с данными. Они предоставляются исполняемой программе по мере обращения к ним. Как правило, каждый ресурс имеет уникальное имя в пределах данной программы. Задача динамического загрузчика — найти ресурс по имени, загрузить в оперативную память и предоставить программе доступ к нему, а при завершении использования — освободить память, занятую ресурсом. Функции работы с ресурсами пользовательского интерфейса зависят от типа ОС. Система программирования сама формирует файлы

ресурсов или включает необходимые ресурсы непосредственно в исполняемый файл в нужном формате в зависимости от типа целевой вычислительной системы.

Отладчик. Функции отладчика

Еще одним модулем системы программирования, функции которого тесно связаны с выполнением программы, является отладчик.

Отладчик — это программный модуль, который позволяет выполнять основные задачи, связанные с мониторингом процесса выполнения результирующей прикладной программы. Этот процесс называется отладкой и включает в себя следующие основные возможности:

- последовательное пошаговое выполнение результирующей программы на основе шагов по машинным командам или по операторам входного языка;
- выполнение результирующей программы до достижения ею одной из заданных точек останова (адресов останова);
- выполнение результирующей программы до наступления некоторых заданных условий, связанных с данными и адресами, обрабатываемыми этой программой;
- просмотр содержимого областей памяти, занятых командами или данными результирующей программы.

Первоначально отладчики представляли собой отдельные программные модули, которые могли обрабатывать результирующую программу в терминах языка машинных команд. Их возможности в основном сводились к моделированию выполнения результирующих программ в архитектуре соответствующей вычислительной системы. Выполнение могло идти непрерывно либо по шагам.

Дальнейшее развитие отладчиков связано со следующими принципиальными моментами:

- появлением интегрированных сред разработки;
- появлением возможностей аппаратной поддержки средств отладки во многих вычислительных системах.

Первый из этих шагов дал возможность разработчикам программ работать не в терминах машинных команд, а в терминах исходного языка программирования, что значительно сократило трудозатраты на отладку программного обеспечения. При этом отладчики перестали быть отдельными модулями и стали интегрированной частью систем программирования, поскольку они должны были теперь поддерживать работу с таблицами идентификаторов (см. раздел «Таблицы идентификаторов. Организация таблиц идентификаторов» главы 2) и выполнять задачу, обратную идентификации лексических единиц языка (см. раздел «Семантический анализ и подготовка к генерации кода» главы 5). Это связано с тем, что в такой среде отладка программы идет в терминах имен, данных пользователем, а не в терминах внутренних имен, присвоенных компилятором. Соответствующие изменения потребовались также в функциях компиляторов и компоновщиков, поскольку они должны были включать таблицу имен в состав объектных и исполняемых файлов для ее обработки отладчиком.

Второй шаг позволил значительно расширить возможности средств отладки. Теперь для них не требовалось моделировать работу и архитектуру соответствующей вы-

числительной системы. Выполнение результирующей программы в режиме отладки стало возможным в той же среде, что и в обычном режиме. В задачу отладчика входили только функции перевода вычислительной системы в соответствующий режим перед запуском результирующей программы на отладку. Во многом эти функции являются приоритетными, поскольку зачастую требуют установки системных таблиц и флагов процессора вычислительной системы (мы рассмотрели бóльшую часть этих вопросов в начале данного учебника).

Отладчики в современных системах программирования представляют собой модули с развитым интерфейсом пользователя, работающие непосредственно с текстом и модулями исходной программы. Многие их функции интегрированы с функциями текстовых редакторов исходных текстов, входящих в состав систем программирования.

Библиотеки подпрограмм

Библиотеки подпрограмм как составная часть систем программирования

Библиотеки подпрограмм составляют существенную часть систем программирования. Наряду с дружественностью пользовательского интерфейса, состав доступных библиотек подпрограмм во многом определяет возможности системы программирования и ее позиции на рынке средств разработки программного обеспечения.

Библиотеки подпрограмм входили в состав средств разработки, начиная с самых ранних этапов их развития. Даже когда компиляторы еще представляли собой отдельные программные модули, они уже были связаны с соответствующими библиотеками, поскольку компиляция так или иначе предусматривает связь программ со стандартными функциями исходного языка. Эти функции обязательно должны входить в состав библиотек.

С точки зрения системы программирования библиотеки подпрограмм состоят из двух основных компонентов. Это собственно файл (или множество файлов) библиотеки, содержащий объектный код, и набор файлов описаний функций, подпрограмм, констант и переменных, составляющих библиотеку. Описания оформляются на соответствующем входном языке (например, для языка C или C++ это будет набор заголовочных файлов). Иногда эти файлы могут быть совмещены (например, в системе программирования Turbo Pascal стандартные библиотеки представлены в виде объектных файлов специального формата — TPU — Turbo Pascal Units). Кроме того, в современных системах программирования в состав библиотеки входит также описание ее на естественном языке в виде файла подсказок и справок.

Набор файлов описания библиотеки служит для информирования компилятора о составе входящих в библиотеку функций. Обрабатывая эти файлы, компилятор получает всю необходимую информацию о составе библиотеки с точки зрения входного языка программирования. Эти файлы предназначены только для того, чтобы избавить разработчика от необходимости постоянного описания библиотечных функций, подпрограмм, констант и переменных.

В состав системы программирования может входить большое количество разнообразных библиотек. Среди них всегда можно выделить основную библиотекусодержащую

обязательные функции входного языка программирования. Т акая библиотека называется обычно стандартной библиотекой языка. Эта библиотека всегда используется компилятором, поскольку без нее разработка программ на данном входном языке невозможна. Все остальные библиотеки необязательны и подключаются к результирующей программе только по прямому указанию разработчика.

В процессе развития систем программирования состав библиотек постоянно расширялся. В них включалось все большее и большее число функций. Это было вызвано тем, что система программирования, предоставляющая пользователю более широкий выбор библиотечных функций, получала лучшие позиции на рынке средств разработки программного обеспечения. Сами по себе библиотеки подпрограмм и функций, созданных для того или иного языка, также становились товаром на рынке средств разработки. Этот естественный процесс продолжается до сих пор, и чем длиннее история существования того или иного языка программирования, тем шире диапазон существующих для него библиотек.

Системы программирования для некоторых языков продолжают существовать во многом благодаря тому, что для них создан мощный аппарат библиотечных функций. Так, язык FORTRAN существует благодаря широчайшему набору функций для работы с математическими и физическими процессами, а язык COBOL — благодаря набору функций из финансовой сферы.

В ходе развития систем программирования принципы создания и использования библиотечных функций претерпели мало изменений. Принципиально новые возможности предоставили только современные ОС, которые позволили подключать к результирующим программам не статические, а динамические библиотеки.

Статические библиотеки подпрограмм

Статические библиотеки подпрограмм и функций представляют собой часть объектного кода, которая встраивается (подключается) к резуль тирующей программе на этапе ее разработки.

Все статические библиотеки подключаются к резуль тирующей программе один раз на этапе ее создания, после чего они являются ее неотъемлемой частью. Код статических библиотек входит в состав исполняемого файла программы точно так же, как и объектный код, созданный компилятором на основе исходного текста программы. При выполнении программы нет никакого различия между объектным кодом, построенным на основании исходного текста, созданного разработчиком, и объектным кодом библиотеки.

Объектный код статической библиотеки подключается компоновщиком к результирующей программе при создании исполняемого модуля. Поэтому по своей структуре статическая библиотека мало чем отличается от обычных объектных файлов, порождаемых компилятором. Чаще всего система программирования хранит объектный код входящих в ее состав библиотек в некотором упакованном виде. При подключении библиотеки к исполняемому файлу компоновщик распаковывает ее код и встраивает его в общий объектный код результирующей программы в обычном порядке.

Будучи один раз созданной с использованием статических библиотек, результирующая программа в дальнейшем уже никак не зависит от изменений в этих библиотеках, а также от их наличия в вычислительной системе, где она будет выполняться.

Весь необходимый объектный код уже входит в состав такой программы. В этом очевидное преимущество использования статических библиотек.

В то же время статические библиотеки обладают двумя недостатками, один из которых является весьма существенным.

- Во-первых, при наличии ошибки в библиотеке она будет проявляться во всех программах, которые используют эту библиотеку , а после ее исправления все такие программы нужно будет построить (пересобрать) заново с использованием обновленной библиотеки. Конечно, если учесть, что в современных системах программирования все новые библиотеки появляются только после тщательной всесторонней отладки, этим недостатком можно пренебречь.
- Во-вторых, поскольку объектный код статических библиотек встраивается в исполняемый файл, это ведет к увеличению объема резуль тирующей программы. В современных прикладных программах более 50 % объектного кода может составлять код из файлов библиотек. При наличии многих программ, использующих код одной и той же библиотеки, это ведет к неэффективному использованию как места для хранения исполняемых файлов программ, так и оперативной памяти вычислительной системы во время их выполнения. В особенности это относится к системным библиотекам — библиотекам ОС, которые используются если не всеми, то очень многими программами.

Именно второй недостаток статических библиотек послужил толчком к созданию другого типа библиотек подпрограмм и функций — динамически загружаемых (или просто динамических) библиотек.

Динамические библиотеки подпрограмм

Динамические (динамически загружаемые) библиотеки подключаются к результирующей программе в момент ее выполнения. В этом основное отличие динамических библиотек от обычных — статических библиотек.

Возможны два основных варианта загрузки динамических библиотек при выполнении результирующей программы. В первом варианте динамическая библиотека загружается в оперативную память компьютера сразу же, как только начинает выполняться программа, использующая данную библиотеку, вне зависимости от того, будет ли выполняться обращение к функциям библиотеки или нет. Во втором варианте библиотека загружается в оперативную память только тогда, когда происходит непосредственное обращение к одной из ее подпрограмм или функций. Соответственно, существует два варианта освобождения памяти, занимаемой динамической библиотекой: она освобождается либо только по завершении выполнения всей программы, либо по команде основной программы, сигнализирующей о том, что данная библиотека более использоваться не будет.

Второй вариант является более предпочтительным с точки зрения экономии объема занимаемой оперативной памяти. Но он сложнее в реализации и требует указаний разработчика программы, поскольку только разработчик может явно определить моменты выполнения программы, которые требуют загрузки в память той или иной библиотеки или, наоборот, освобождения памяти, когда библиотека более не будет использоваться. Первый вариант менее экономично расходует оперативную память, но проще в реализации, поскольку компоновщик сам может определить перечень

всех используемых в программе библиотек и обеспечить загрузку их в память в момент начала выполнения программы. Современные системы программирования предусматривают, как правило, оба варианта работы с динамически загружаемыми библиотеками — решение о том, какой вариант будет использован, принимает разработчик программы. В принципе, одна программа может предполагать различные варианты работы с различными библиотеками — выбор зависит от объема кода библиотеки и от частоты обращений к ней.

За загрузку динамических библиотек в оперативную память, освобождение памяти, занятой динамическими библиотеками, а также за связь кода и данных библиотек с кодом и данными исполняемой программы отвечает динамический загрузчик , который входит в состав ОС. Он же обеспечивает повторное использование кода динамических библиотек. Повторное использование кода заключается в том, что, если две различные программы обращаются к одной и той же динамической библиотеке, нет необходимости загружать ее объектный код в оперативную память дважды — достаточно обеспечить доступ обеим программам к одной и той же копии объектного кода при условии разделения данных. Более подробно эти вопросы освещены в [14, 25, 27, 28, 37, 48, 54, 62, 68].

Формат файлов динамических библиотек может быть различным. Как правило, он строго определяется требованиями соответствующей ОС и близок к формату исполняемых файлов. Как и статические библиотеки, динамические библиотеки предусматривают описание входящих в них функций в виде текста на соответствующем входном языке, чтобы дать информацию о них компилятору в ходе обработки исходного текста программы и избавить разработчика от создания таких описаний. Но для динамических библиотек компилятор не включает в код программы обращения к функциям, как он это делает для статических библиотек. Вместо этого в код вставляются вызовы соответствующих функций ОС, обеспечивающих обращение к коду динамической библиотеки, если связь с библиотекой осуществляется по первому варианту работы. Если же связь с динамической библиотекой осуществляется по второму варианту, то разработчик сам должен организовать вызов соответствующих функций ОС в исходном коде.

Преимущества динамических библиотек очевидны — они не требуют включения в результирующую программу объектного кода часто используемых функций, чем существенно сокращают ее объем. Конечно, динамические библиотеки требуют наличия в ОС специального механизма, позволяющего подключать часть объектного кода непосредственно по ходу выполнения программы. Однако для современных ОС, выполняющихся в вычислительных системах, которые поддерживают широкий набор методов адресации, это не является проблемой [14, 15, 62].

Недостатки динамически загружаемых библиотек заключаются в том, что результирующие программы оказываются связаны с объектным кодом, непосредственно не входящим в их состав. Тогда для выполнения такой программы обязательно требуется наличие всех используемых ею библиотек в составе вычислительной системы, где эта программа выполняется. Кроме того, логика работы результирующей программы становится зависимой от кода всех входящих в нее библиотек. Изменение библиотеки, которое может происходить независимо от разработчиков программы, может повлиять на функционирование самой программы. Т акое развитие событий порой оказывается неожиданным для пользователя.

Еще более проблематичной оказывается ситуация, в которой результирующая программа использует для своего выполнения две динамические библиотеки (обозначим их DLL_1 и DLL_2). При этом библиотека DLL_1 для своего выполнения также требует наличия библиотеки DLL_2. Конфликт может заключаться в том, что результирующей программе и библиотеке DLL_1 для выполнения нужны различные версии библиотеки DLL_2. При этом все динамические библиотеки загружаются в адресное пространство результирующей программы. Но, как правило, ОС не может обеспечить загрузку в адресное пространство одной программы двух различных версий одной и той же библиотеки. Конфликт становится неразрешимым.

Поэтому использование динамических библиотек накладывает определенные обязательства как на разработчика программы, так и на создателей библиотек. Разработчик прикладной программы должен использовать библиотеку только так, как это определено ее создателем, а создатель библиотеки в случае ее модификации должен организовывать изменения таким образом, чтобы они не сказывались на логике работы программ, ориентированных на предыдущие версии его библиотеки. Подавляющее большинство разработчиков (как программ, так и библиотек) стремится придерживаться этих правил, но не всегда им это удается[1].

Широкий набор динамических библиотек поддерживается всеми современными ОС. Как правило, они содержат системные функции ОС и общедоступные функции программного интерфейса (API). Кроме того, многие независимые разработчики предоставляют для различных систем программирования свои динамические библиотеки как отдельные товары на рынке средств разработки прикладных программ.

Ресурсы пользовательского интерфейса. Редакторы ресурсов

Как уже было сказано выше, *ресурсами пользовательского интерфейса* будем называть множество данных, обеспечивающих внешний вид интерфейса пользователя результирующей программы, не связанных напрямую с логикой ее выполнения. Примерами ресурсов являются: тексты сообщений, выдаваемых программой; цветовая гамма элементов интерфейса; надписи на элементах управления и т. п.

Ресурсы пользовательского интерфейса могут храниться в различном виде: либо в составе результирующей программы, либо в составе динамически загружаемой библиотеки, либо в виде отдельного файла. Формат хранения ресурсов также может быть различным. Это может быть специального вида двоичный код описания данных, близкий к объектному коду, или же структурированный текстовый файл. Формат и способ хранения ресурсов пользовательского интерфейса зависит от ОС. Как правило, каждая ОС предоставляет разработчику несколько вариантов, из которых он выбирает удобный для себя.

Работа с ресурсами пользовательского интерфейса включает в себя загрузку в память требуемого ресурса, отображение его на экране компьютера (в том или ином виде), а также освобождение памяти, занятой ресурсом. Эти действия выполняются с по-

[1] В современных ОС появились средства, позволяющие поддерживать не только файлы, но и версии динамических библиотек. Они во многом используют методы, применяемые в системах управления версиями исходных файлов. Тогда приложение, обращаясь к библиотеке, может указать не только имя, но и требуемую версию вызываемой функции библиотеки.

мощью специальных системных функций, предоставляемых ОС. Как правило, для удобства работы с ними эти функции входят в состав одной или нескольких динамически загружаемых библиотек, поставляемых ОС. Кроме того, в составе ОС обычно предусмотрен специальный набор наиболее часто используемых ресурсов интерфейса, называемых системными. Пользователь не должен разрабатывать и включать в состав результирующей программы такой ресурс — он может воспользоваться ресурсом, входящим в состав ОС. Примерами системных ресурсов являются: внешний вид текста системных сообщений (системный шрифт, или фонт), основной фон окон и меню прикладных программ, внешний вид наиболее употребительных органов управления (кнопок, списков и т. п.).

В каком бы виде и формате ни хранились ресурсы в результирующей программе, порядок их создания практически всегда один и тот же: сначала создается описание ресурсов пользовательского интерфейса на входном языке описания ресурсов, затем оно обрабатывается компилятором ресурсов , входящим в состав системы программирования. После этого компоновщик подключает ресурсы к исполняемому файлу результирующей программы, или оформляет их в виде динамически загружаемой библиотеки, или оставляет в виде отдельного файла ресурсов.

Язык описания ресурсов — это, как правило, простой язык, построенный на основе регулярной грамматики. Но строить описания ресурсов в таком виде неэффективно, так как любому разработчику внешний вид интерфейса программы удобнее формировать в виде графических образов. Это тем более важно, поскольку часто интерфейс создают не программисты, а дизайнеры. Поэтому современные системы программирования имеют в своем составе графические средства редактирования ресурсов пользовательского интерфейса. Эти средства позволяют создавать интерфейс в виде графических образов, на основе которых потом строится его описание на языке описания ресурсов. В системах программирования, предлагающих средства быстрой разработки приложений (RAD), средства разработки графического интерфейса позволяют также создать простейший шаблон исходного кода для работы с этим интерфейсом.

Преимущество использования ресурсов заключается в том, что пользовательский интерфейс результирующей программы можно проектировать, создавать и изменять независимо от логики работы самой программы. Эту работу можно поручить дизайнерам, в то время как исходный код создают программисты. Например, чтобы перевести программу на новый язык[1], часто бывает достаточно перевести на этот язык все текстовые сообщения и строковые элементы форм, и это гораздо удобнее выполнять, если они находятся отдельно, а не внутри исходного кода.

Мобильность и переносимость программного обеспечения

Условия мобильности программного обеспечения

Мобильностью программного обеспечения будем называть способность программного обеспечения выполнять свои функции на различных вычислительных систе-

[1] Имеется в виду естественный язык общения программы с пользователем, а не язык программирования.

мах вне зависимости от их архитектуры. В понятие «архитектура вычислительной системы» в данном случае включаются как аппаратные, так и все программные средства, необходимые для функционирования программного обеспечения.

Как уже отмечалось выше, на возможность выполнения программным обеспечением своих функций влияют очень многие факторы. В общем виде перечислить их все не представляется возможным, так как они существенно зависят от назначения программного обеспечения. Одно дело говорить о функциях драйвера некоторого периферийного устройства, другое дело — о работе какой-то утилиты ОС и совершенно иное — о функциях прикладной программы.

Далее перечислены только основные факторы, влияющие на мобильность программного обеспечения:

- состав специализированных аппаратных средств и периферийных устройств, используемых программным обеспечением;
- тип ОС, на которую ориентировано программное обеспечение;
- состав и функции динамически загружаемых библиотек;
- структура и формат хранения ресурсов пользовательского интерфейса, используемых программным обеспечением;
- перечень внешних программ и модулей, с которыми должно взаимодействовать данное программное обеспечение при выполнении своих функций.

Чем меньше зависимость результирующей программы от каждого из этих факторов, тем выше ее мобильность. Если совершенно исключить зависимость от них, то можно добиться полной переносимости результирующей программы. Но, к сожалению, в реальной жизни добиться этого практически невозможно.

Единственный фактор, от которого можно полностью избавиться, — это зависимость от используемых специализированных аппаратных средств и периферийных устройств. Если рассматриваемая программа не является драйвером некоторого периферийного устройства или не ориентирована специально на работу с ним, то от такой зависимости можно и нужно отказаться. Все современные ОС и системы программирования предоставляют широкий диапазон средств для работы с типовыми периферийными устройствами — монитором, клавиатурой, мышью, принтером, а также с файлами и оперативной памятью. Разработчику нет никакой необходимости писать код для прямого взаимодействия с драйверами таких устройств, а уж тем более напрямую с самим аппаратным обеспечением. Это не только снижает мобильность программного обеспечения, но и увеличивает трудоемкость создания исходного кода, повышает вероятность возникновения ошибок. Большинство прикладных программ и значительная часть системных программ создаются вне какой-либо зависимости от аппаратных средств (если, конечно, иного не требует сама функциональность программы). В том случае, когда взаимодействие с нестандартным периферийным устройством все же необходимо, мобильность результирующей программы будет выше, если работа с аппаратными средствами будет идти через функции или API-драйверы этих средств, а если драйвер для них отсутствует, лучше разработать его отдельно от остальной части программы.

Зависимость от других (программных) факторов исключить в результирующей программе практически невозможно, поскольку все они существенно зависят от типа

ОС, на которую ориентирована программа. Тип ОС определяет формат исполняемого файла, формат и функции системных динамических библиотек, способ и форму хранения ресурсов пользовательского интерфейса. Любая современная результирующая программа не может существовать вне связи с этими элементами ОС. В лучшем случае можно добиться мобильности результирующей программы в рамках нескольких ОС одного типа или одного производителя (например, на всех ОС типа Linux или на всех ОС производства Microsoft), но не более того.

Обеспечение переносимости исходного кода программ

В большинстве случаев мобильность резуль тирующей программы обеспечить невозможно, но можно добиться мобильности для исходного кода программы. Программное обеспечение сохраняет способность выполнять свои функции на различных вычислительных системах вне зависимости от их архитектуры в том случае, если оно создано на основе одного и того же исходного кода с помощью некоторой системы программирования и ориентировано на определенную целевую вычислительную систему. В таком варианте говорят о *переносимости* программного обеспечения .

На переносимость программного обеспечения влияют те же факторы, что и на его мобильность, но в меньшей степени[1].

Поэтому для того, чтобы добиться максимально возможной переносимости исходной программы, следует придерживаться следующих основных правил:

- не использовать в исходном коде программы прямые обращения к периферийным устройствам компьютера, к драйверам аппаратных средств (кроме всего прочего, такие обращения могут быть запрещены в ОС для прикладных программ непривилегированных пользователей);
- не включать в исходный код прямые обращения к функциям ОС, а также не использовать статические и динамические библиотеки, ориентированные на определенный тип ОС;
- использовать только широко распространенные динамически загружаемые библиотеки (существующие во всех типах ОС) или динамические библиотеки, для которых имеется исходный код;
- подключать динамически загружаемые библиотеки только средствами системы программирования (первый способ загрузки динамических библиотек), не использовать для этой цели специфические функции ОС;
- использовать только средства системы программирования для создания ресурсов пользовательского интерфейса, не обращаться к системным ресурсам;
- исключить взаимодействие с внешними программами и модулями или ограничить его только программами, доступными во всех типах ОС;
- не использовать для взаимодействия с внешними программами и модулями прямые обращения к средствам ОС.

[1] Термины «мобильность» и «переносимость» близки по смыслу и предполагают разную трактовку в разной литературе (действительно, эти два слова по сути означают одно и то же). Здесь, с точки зрения автора, термин «мобильность» применяется к результирующей программе, в то время как «переносимость» — к исходному тексту программы.

Конечно, добиться соблюдения всех этих правил нелегко, и чем шире функциональность программного обеспечения, чем более развитым и сложным пользовательским интерфейсом оно обладает, тем сложнее обеспечивать его переносимость. Например, обеспечить переносимость программы с текстовым диалоговым интерфейсом, выполняющей математические расчеты, существенно проще, чем обеспечить переносимость графического редактора или компьютерной игры. В первом случае взаимодействие программного обеспечения с ОС минимально, а используемые библиотеки (математические функции) являются более широко распространенными, чем во втором (работа с графическими средствами).

Для реальных исполняемых программ, конечно же, невозможно полностью исключить взаимодействие с ОС. Поэтому для решения проблемы переносимости был предложен подход, основанный на стандартизации функций библиотек, обеспечивающих взаимодействие программного обеспечения с ОС. Для этой цели разработаны несколько стандартов, которые описывают основные функции взаимодействия с ОС, их аргументы, выполняемые ими действия и правила обращения к этим функциям. Если ОС и система программирования поддерживают стандарт, то в их состав обязательно входят библиотеки, обеспечивающие выполнение функций, описанных в стандарте. Тогда взаимодействие программы с ОС происходит так, как показано на рис. 6.2.

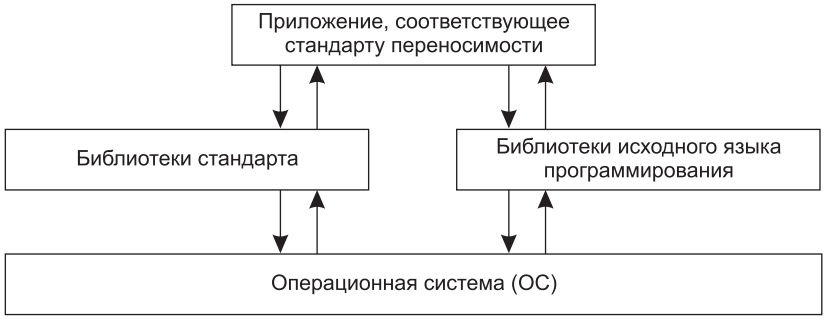

Рис. 6.2. Структура программы, соответствующей стандарту переносимости

Программное обеспечение, построенное строго в соответствии с требованиями стандарта, структура которого изображена на рис. 6.2, является переносимым на все типы ОС, которые поддерживают данный стандарт. Для этого его достаточно только подготовить (откомпилировать и собрать) в системе программирования, ориентированной на данную ОС. Наиболее известным стандартом переносимости программного обеспечения является стандарт POSIX (IEEE Std 1003.1), разработанный IEEE (Institute of Electrical and Electronics Engineers)[14, 15, 62].

Кроме описанных выше требований, на переносимость программного обеспечения влияют возможности системы программирования, в которой оно создано. Во-первых, система программирования должна поддерживать выбранный стандарт переносимости исходного кода, во-вторых, она должна быть ориентирована на все типы ОС, для которых должна обеспечиваться переносимость программного обеспечения. Если программное обеспечение соответствует требованиям стандарта переносимости, то всегда можно выбрать другую систему программирования для перехода на новый

тип ОС. Тогда переносимость исходного кода зависит от распространенности языка программирования, на котором он написан. Чем более распространенным является язык, чем больше для него существует систем программирования, тем выше возможности по переносимости программного обеспечения.

В настоящее время сложилось так, что самыми широкими возможностями по переносимости исходного кода обладают языки C и C++, которые изначально возникли в среде ОС UNIX, ориентированной на соответствие стандарту переносимости POSIX [72]. Для этих языков существует самый широкий выбор систем программирования от различных производителей. Гораздо у́же возможности по переносимости для исходного кода на языке FORTRAN (для этого языка существует широкий набор библиотек, но сам язык существенно устарел). Для других распространенных языков — Visual Basic и Object P ascal — вопрос о переносимости исходного кода остается открытым, поскольку в настоящее время для них доступен очень узкий выбор систем программирования. Фактически программы, созданные на этих языках, переносимостью не обладают.

Однако, даже используя широко распространенные языки программирования и следуя стандартам, практически невозможно добиться абсолютной переносимости исходного кода. Для этой цели существуют другие средства.

Мобильность программного обеспечения на основе интерпретаторов

Другой способ обеспечения мобильности и переносимости программного обеспечения — использование интерпретаторов для выполнения исходного кода программ. При этом для исходного кода обеспечивается полная переносимость (в этом случае понятия «мобильность» и «переносимость» совпадают). При использовании интерпретаторов для обеспечения переносимости исходного кода ограничения на прямое использование функций ОС должны соблюдаться точно так же, как и для компиляторов (зачастую сами интерпретаторы предполагают такого рода ограничения).

Этот способ универсален, но обладает двумя существенными недостатками:

- исходный текст программ может быть выполнен интерпретатором далеко не для всех языков программирования;
- скорость выполнения кода интерпретатором существенно ниже, чем при выполнении откомпилированного объектного кода.

Эти недостатки ограничивают применение интерпретаторов для обычных прикладных, а тем более системных программ. Однако данный способ нашел широкое применение в сети Интернет, где полная переносимость программного обеспечения играет первостепенную роль. Об этом будет сказано далее в соответствующем разделе.

Существует еще один вариант обеспечения мобильности программного обеспечения, ориентированный на использование не только компилятора, но и специального интерпретатора для обеспечения мобильности резуль тирующей программы. В этом случае результатом работы компилятора является не объектный код, а некоторый промежуточный двоичный код, который не может быть непосредственно выполнен, а должен быть обработан специальным интерпретатором. В общем случае выполнение программного обеспечения в этом варианте происходит так, как показано на рис. 6.3.

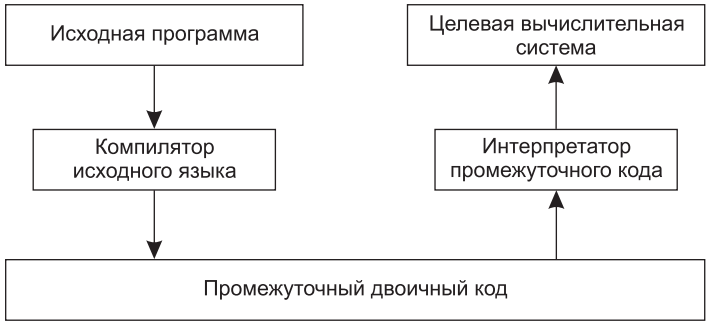

Рис. 6.3. Обеспечение мобильности программного обеспечения на основе промежуточного двоичного кода

Использование промежуточного двоичного кода позволяет снизить влияние основного недостатка интерпретации — низкой скорости выполнения программы. Увеличение скорости выполнения программы в такой схеме происходит , поскольку промежуточный двоичный код представляет собой простейший набор команд, который может быть описан регулярной грамматикой, а потому не требует сложной обработки. Такая схема работы используется в программном обеспечении, написанном на языке Java [8, 15, 16, 73], а также в программах, созданных для платформы .NET компании Microsoft [51, 52]. Эти технологии рассмотрены далее в разделе, посвященном организации распределенных вычислений.

Преимущества и недостатки переносимости программ

В принципе, вопрос о переносимости программного обеспечения должен решаться разработчиком индивидуально в зависимости от функциональности программы и целей, которые перед собой ставят ее создатели.

Переносимость исходного кода обеспечивает создателям программного обеспечения более широкий рынок сбыта, а также снижает их зависимость от изменения архитектуры вычислительных систем в будущем. Кроме того, обеспечивая переносимость программного обеспечения и следуя стандартам, разработчики уменьшают зависимость коммерческого успеха создаваемого программного обеспечения от позиции на рынке производителей систем программирования и операционных систем. Если выбранная ОС или система программирования перестает устраивать разработчиков либо начинает терять свои позиции на рынке, они могут перейти на использование другой ОС или другой системы программирования, удовлетворяющей тому же стандарту.

Таковы преимущества переносимости исходного кода программ. Ноу нее есть и свои недостатки.

Основным недостатком является тот факт, что, стремясь добиться большей переносимости исходного кода, разработчики вынуждены терять в эффективности. Ведь работа с драйверами аппаратных средств, прямые обращения к функциям ОС и библиотек являются самыми эффективными средствами работы с ними. Используя предусмотренный стандартом промежуточный уровень, разработчик вынужден согласиться с потерями в скорости выполнения программы и с увеличением объема

исполняемого кода. И уж тем более теряется эффективность, когда используется интерпретация.

Ориентируясь на указанные преимущества и недостатки, создатель программы сам вправе решать: будет он стремиться к переносимости исходного кода или нет.

Разработка приложений в двухзвенной архитектуре

Принципы работы в двухзвенной архитектуре. Простейшие двухзвенные архитектуры

Принципы организации распределенных вычислений

Распространение динамически загружаемых библиотеки ресурсов пользовательского интерфейса привело к ситуации, когда большинство прикладных программ стало представлять собой не единый программный модуль, а набор сложным образом взаимосвязанных между собой компонентов. Многие из этих компонентов входили в состав ОС, либо же требовалась их поставка и установка от разработчиков, которые очень часто могли быть никак не связаны с разработчиками самой прикладной программы.

Это обстоятельство явилось первым фактором, определившим появление новых программных архитектур, предусматривающих распределенные вычисления.

Вторым фактором в этом направлении стало широкое распространение и бурное развитие локальных вычислительных сетей (ЛВС). Объединение компьютеров в сети дало возможность программам, работающим на отдельных компьютерах в составе сети, взаимодействовать и сообща решать поставленные перед ними задачи. При этом распределение решаемой задачи по нескольким компьютерам позволяет сократить время, необходимое для ее решения, а также снизить требования к вычислительным мощностям отдельных компьютеров.

Наличие двух указанных выше факторов дало возможность разработчикам программ организовывать распределенные вычисления. При этом перед разработчиками встал вопрос о том, как максимально эффективно организовать разбиение программ на части, которые, в свою очередь, можно было бы распределить по компьютерам в составе сети.

Очевидно, что можно предложить множество схем разбиения программы на составные части. При этом для каждой конкретной программы можно разработать свои схемы разбиения. При решении задачи об организации распределенных вычислений были созданы типовые модели разбиения программ на части. Наиболее общая модель разбиения предлагает выделить в каждой программе следующие функциональные составляющие:

- средства и логика пользовательского интерфейса — эта часть отвечает за организацию представления данных на экране, она включает в себя средства и ресурсы пользовательского интерфейса, а также правила и возможные сценарии взаимодействия с пользователем;

- прикладная логика (бизнес-логика) — эта часть выполняет основную обработку данных программой, содержит набор правил для принятия решений, а также все необходимые вычислительные процедуры и операции;
- логика работы с данными — эта часть обеспечивает выполнение простейших операций над данными: выбор данных, поиск, сортировка, филь трация, а также заполнение, модификация и удаление некоторых структур данных;
- файловые операции — эта часть обеспечивает выполнение стандартных операций над файлами и файловой системой, которые, как правило, реализуются с помощью функций ОС.

Конечно, предложенное разбиение достаточно условно и не является единственно возможным, но оно в целом достаточно полно отражает все функциональные составляющие большинства прикладных программ.

Следует отметить также, что в приведенном разбиении первые две составляющие части включают в себя собственно алгоритмы, логику и весь интерфейс, созданные разработчиками прикладной программы. Эти составляющие части предусматривают наличие взаимосвязи с данными, содержащимися в третьей и четвертой составляющих частях, для чего требуются методы доступа к этим данным.

Третья составляющая часть, в свою очередь, должна содержать информацию о структурах данных программы, которые созданы ее разработчиками. Зачастую она может быть организована в виде базы данных (БД). Поэтому для работы с этими структурами может использоваться набор стандартных компонентов, созданных сторонними разработчиками, никак не связанными с созданием данной прикладной программы.

Наконец, четвертая составляющая часть программы может быть целиком и полностью организована средствами ОС, если ОС в своем составе предусматривает наличие средств организации распределенного доступа к файловой системе.

На основе предложенной модели разбиения прикладной программы на части было построено несколько схем распределения частей программы между компьютерами в составе ЛВС. По мере создания этих схем в современных ОС появлялись средства, позволяющие применять их для организации программ, использующих распределенные вычисления. Тенденция развития ОС, системных и прикладных программ была такова, что программы, построенные с использованием распределенных вычислений, стали занимать все больший процент на рынке программного обеспечения.

Эту тенденцию не могли не отметить производители средств разработки, и на рынке стали появляться системы программирования, специально ориентированные на разработку исполняемых программ, использующих распределенные вычисления, — распределенных исполняемых программ, или, как их еще называют, *распределенных приложений*[1]. Причем тенденция развития систем программирования в этом направлении остается актуальной и по сей день: в настоящее время продолжают появляться и совершенствоваться системы программирования для создания распределенных приложений.

Необходимо рассмотреть основные технологии, лежащие в их основе.

[1] Далее термин «приложения» будет использоваться именно для обозначения исполняемых программ.

Структура приложения, построенного в архитектуре «файл—сервер»

Наиболее простая и очевидная схема организации распределенных вычислений заключалась в том, чтобы выделить отдельный компьютер (или несколько компьютеров) для выполнения файловых операций. В этом случае все файлы, обрабатываемые прикладной программой (или несколькими прикладными программами), располагались на выделенном для этой цели компьютере. Программадля получения доступа к файлам и для выполнения любых операций с ними должна была через сеть обратиться к этому компьютеру.

Такой компьютер, выделенный в составе сети для хранения файлов и выполнения операций над ними, получил название «*файл—сервер*». В свою очередь, такая архитектура организации распределенных вычислений получила название «файл—серверная архитектура» или «архитектура файл—сервер».

При такой организации распределенных вычислений доступ к данным файл—сервера осуществлялся на уровне файлов, а разделение доступа к файлам обеспечивалось средствами ОС. Сервер в этой архитектуре выполнял только примитивные процедуры доступа к файлам, хранения файлов, копирования и защиты файлов от несанкционированного доступа.

Использование архитектуры «файл—сервер» не требовало от разработчиков приложений применения каких-либо специальных средств разработки. Для доступа к файлам достаточно было применить стандартные средства, предоставляемые ОС или системой программирования. Необходимо было только, чтобы ОС поддерживала механизмы удаленного доступа к файлам — и такие ОС достаточно быстро появились. Точно так же архитектура «файл—сервер» никак не повлияла и на системы программирования — средства удаленного доступа к файлам, реализованные в ОС, автоматически стали доступны для всех систем программирования, создающих приложения для этих ОС.

Удаленный доступ к файлам в ОС — это отдельная и довольно интересная тема, выходящая за рамки данной книги. С ней лучше ознакомиться в специализированной литературе [14, 45, 61, 62].

С точки зрения развития системного программного обеспечения интересно то, что реализация средств организации распределенных вычислений в ОС положила начало развитию систем управления базами данных (СУБД). Первые СУБД были реализованы именно по технологии «файл—сервер», разграничение доступа к данным в них осуществлялось на уровне файлов. В настоящее время тоже существуют простейшие СУБД, использующие эту технологию, — как правило, это дешевые СУБД, которые часто называют «настольными», или «локальными». Примерами таких СУБД являются СУБД Paradox производства компании Borland и СУБД Microsoft Access производства компании Microsoft.

Приложения, работающие с такими СУБД, могут не использовать средства доступа к файлам, а применять полноценные механизмы работы с БД (которые более подробно рассмотрены далее). При этом СУБД реализует не только файловые операции, но и другую составляющую распределенных вычислений — логику работы с данными. Однако в этом случае логика работы с данными реализуется хотя и с помощью СУБД, но на том же компьютере, где выполняются остальные вычисления, а по сети распределяются только файловые операции. Единственным преимуществом для

разработчика прикладной программы является то, что выполнение распределенных файловых операций организует не он сам, а средства СУБД, к которым он обращается. В остальном логика работы приложения остается такой же, как и при явном использовании распределенного доступа к файлам.

Такие приложения иногда называют приложениями, построенными по принципу «файл—сервер». В общем виде схема их работы представлена на рис. 6.4.

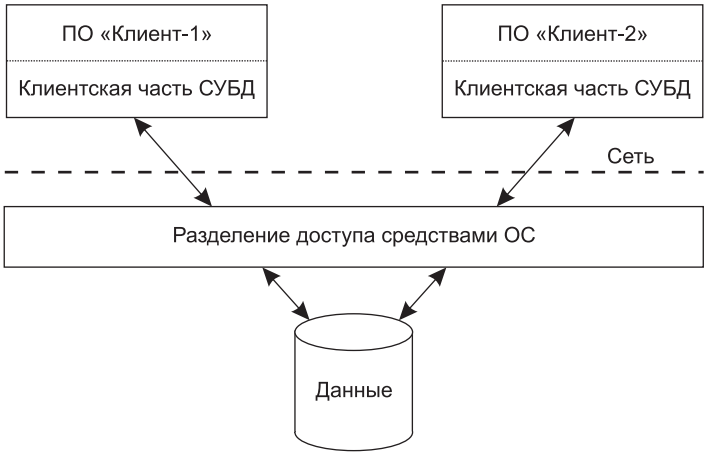

Рис. 6.4. Схема доступа к данным для двух приложений типа «файл—сервер»

Приложения типа «файл—сервер» можно только весьма условно отнести к приложениям, построенным с использованием СУБД. Хотя в них присутствует серверная компонента, ее функции в основном выполняются в ОС.

Недостатки архитектуры «файл—сервер»

Главными недостатками приложений, построенных по принципу «файл—сервер», являются следующие :

- невозможность эффективно разделить доступ к данным при их одновременном использовании несколькими пользователями;
- невозможность организовать защиту данных иначе, как на уровне доступа ОС;
- высокая нагрузка на сеть для передачи файлов, если сервер данных и клиентское приложение работают на разных компьютерах в составе сети.

Последний недостаток является определяющим в ограничении распространения данной архитектуры. Действительно, если программе, работающей в архитектуре «файл—сервер», например, необходимо выполнить изменение нескольких байт в файле, имеющем объем несколько мегабайт, это приведет к значительной нагрузке на сеть. В этом случае программа должна будет получить доступ к файлу — открыть его на запись, после чего переместить внутренний указатель файла к нужному месту выполнить необходимые изменения и закрыть файл. И не важно, будет это сделано с помощью стандартных файловых операций ОС или средствами «настольной» СУБД, — результат будет один и тот же.

Фактически выполнение такой операции в архитектуре «файл—сервер» потребует копирования всех данных файла с файлового сервера на локальный компьютер, где выполняется программа. На локальном компьютере программа выполнит необходимые изменения, после чего данные файла будут скопированы обратно на файловый сервер и записаны в файл. В результате необходимость изменения нескольких байт в файле вызовет выполнение копирования по сети мегабайт информации, причем два раза в разных направлениях: сначала с файл—сервера на локальный компьютер и затем обратно. При этом все время, пока данная программа будет обрабатывать файл на своем компьютере, он будет закрыт на файл—сервере для доступа на запись для всех других программ.

ПРИМЕЧАНИЕ

Если использовать для модификации данных файловые операции ОС, то в доступе будет отказано в явном виде, а при использовании «настольных» СУБД доступ будет получен, но все операции будут задержаны до тех пор, пока программа, уже использующая файл, не освободит его.

Указанные недостатки и ограничения не исключают использования архитектуры «файл—сервер». Напротив, файловые серверы получили очень широкое распространение. Но из-за указанных недостатков они очень редко используются для организации распределенных вычислений. Наиболее типичное их использование — это создание временных или постоянных хранилищ данных, скорость доступа к которым не имеет решающего значения.

Структура приложения, построенного на основе терминального доступа

Еще одним простейшим средством организации приложений, построенных на основе распределенных вычислений, является использование терминального доступа. При использовании терминального доступа пользователь удаленного компьютера (компьютера-терминала) получает доступ к серверу, который в этом случае называется «*терминальный сервер*», и работает на нем таким образом, как будто его устройства ввода и отображения данных подключены непосредственно к этому серверу. В таком варианте пользователь работает на терминальном сервере, как если бы он сидел за терминалом, подключенным к этому серверу , — отсюда происходит и название «терминальный доступ», восходящее к эпохе больших ЭВМ, к которым подключались многочисленные терминалы.

Терминальный доступ может быть использован для организации работы любых приложений. В этом случае компьютер-терминал обеспечивает всю логику пользовательского интерфейса приложения, и фактически на нем происходит взаимодействие приложения с пользователем. А терминальный сервер выполняет все функции прикладной логики приложения, обработку данных и файловые операции. Структура организации распределенных вычислений для простейшего приложения на основе терминального доступа представлена на рис. 6.5.

Так же как использование архитектуры «файл—сервер», использование терминального доступа не требует от разработчиков приложений применения каких-либо специальных средств разработки. Организация терминального сервера и поддержка удаленного доступа к нему целиком и полностью возложены на ОС. Именно

ОС должна обеспечить отображение элементов пользовательского интерфейса с терминального сервера на компьютере-терминале и передачу команд и действий пользователя с компьютера-терминала на терминальный сервер. И если ОС поддерживает терминальный доступ, любое приложение, использующее стандартные функции ОС для организации интерфейса с пользователем, может быть выполнено в режиме терминального доступа. Ограничения по его использованию касаются только тех приложений, которые осуществляют доступ к аппаратуре компьютера, минуя функции ОС, — для таких приложений невозможно организовать распределенные вычисления в режиме терминального доступа.

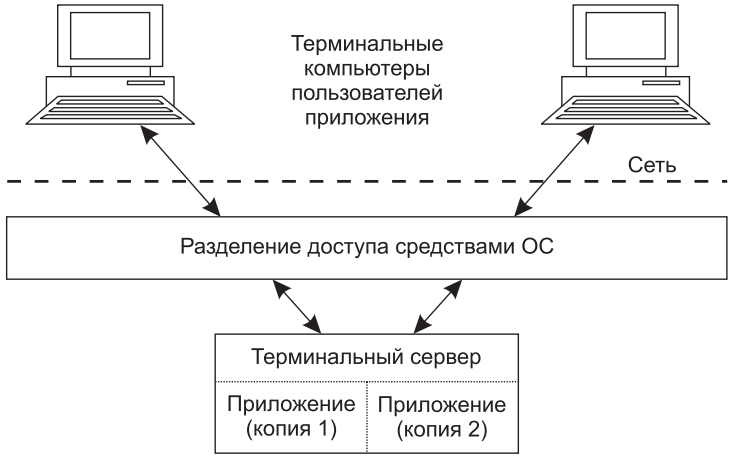

Рис. 6.5. Организация распределенных вычислений с помощью терминального доступа для простейшего приложения

Терминальный доступ прошел долгий путь развития. Основные его идеи были перенесены с терминальных классов больших ЭВМ на компьютерные сети. Более подробно об особенностях его организации можно узнать в специализированной литературе по ОС [14, 45, 61, 62].

Главным недостатком терминального доступа следует признать тот факт , что для одного и того же приложения на терминальном сервере будет выполняться столько его копий, сколько компьютеров-терминалов осуществили запуск этого приложения на выполнение. Причем современные ОС предусматривают, что каждая такая копия приложения будет выполняться в своем адресном пространстве и разделять процессорное время с другими копиями (и другими приложениями) на основе общих принципов ОС. И если приложение не использует никаких средств организации распределенных вычислений, то все запущенные на выполнение его копии будут выполняться независимо друг от друга. Такая организация ведет к неэффективному использованию ресурсов терминального сервера.

Поэтому терминальный доступ нашел применение, в первую очередь, для тех приложений, которые изначально создавались без учета возможности распределенных вычислений. Для более эффективного использования ресурсов сервера были предложены другие решения, которые предусматривают создание приложений, специально ориентированных на распределенные вычисления.

Архитектура приложений «клиент—сервер»

Структура приложения, построенного в архитектуре «клиент—сервер»

Следующий шаг в организации распределенных вычислений после архитектуры «файл—сервер» заключался в том, чтобы вынести на отдельный компьютер выполнение двух составляющих: файловых операций и логики работы с данными. Это стало возможным после появления и широкого распространения различных БД и связанных с ними СУБД.

После появления простейших СУБД, обеспечивающих разделение доступа к данным на уровне файловых операций, дальнейшее развитие СУБД шло в направлении развития средств, обеспечивающих выполнение различных операций работы с данными. В резуль тате на определенном этапе их развития возникла ситуация, когда практически вся логика работы с данными приложений могла быть реализована средствами СУБД. Это предопределило следующий этап в развитии распределенных вычислений — создание приложений, построенных по архитектуре «клиент—сервер».

В архитектуре «клиент—сервер» один компьютер в составе сети выделен для выполнения файловых операций и операций логики работы с данными. Этот компьютер носит название «*сервер данных*», или просто «*сервер*». Остальные компьютеры выполняют обработку данных и операции, связанные с организацией пользовательского интерфейса. Для доступа к данным они должны получать доступ к серверу . Эти компьютеры называют *клиентами*, а выполняющиеся на них программы — *клиентскими приложениями*. Все клиентские приложения являются частью общего приложения, которое выполняет свою задачу в среде распределенных вычислений и состоит из двух частей: клиентской и серверной.

Сервер данных (серверная часть приложения) обеспечивает реализацию всех процедур, связанных с хранением данных, доступом к данным, резервным копированием и защитой данных. Эти операции выполняются с помощью соответствующей СУБД на основании логики работы с данными и реализуются с помощью файловых операций, которые выполняет СУБД. В основе сервера данных лежит база данных — БД.

Клиентская часть приложения (которая может состоять из нескольких однотипных клиентских приложений), получая данные от сервера данных, обеспечивает их обработку и отображение в интерфейсе пользователя на каждом клиенте. При обработке данных клиентское приложение может формировать команды по созданию или изменению структур данных, по добавлению новых данных, обновлению или удалению существующих. Эти команды формируются клиентом и передаются на сервер. По командам клиентской части сервер данных выполняет соответствующие операции: изменение структуры данных, а также их добавление, обновление и удаление. Для выборки необходимых данных клиент также обращается к серверу.

Схема работы такого приложения представлена на рис. 6.6.

Все функции по хранению данных, доступу к ним, защите и резервному копированию данных в такой схеме реализует СУБД на сервере. Т акой подход освобождает клиентские рабочие места от реализации этих функций и снижает требования к ним.

Клиентское приложение не работает непосредственно с файлами и данными — оно обращается к СУБД с запросами на получение (просмотр) данных, их добавление, модификацию и удаление. При работе сервера данных и клиентского приложения на разных компьютерах в составе сети через сеть будут передаваться не все данные из БД, а только запросы клиента и ответы СУБД на них. Таким образом, в архитектуре «клиент—сервер» снижается нагрузка на сеть при обмене данными, но преимущества данной архитектуры этим не ограничиваются.

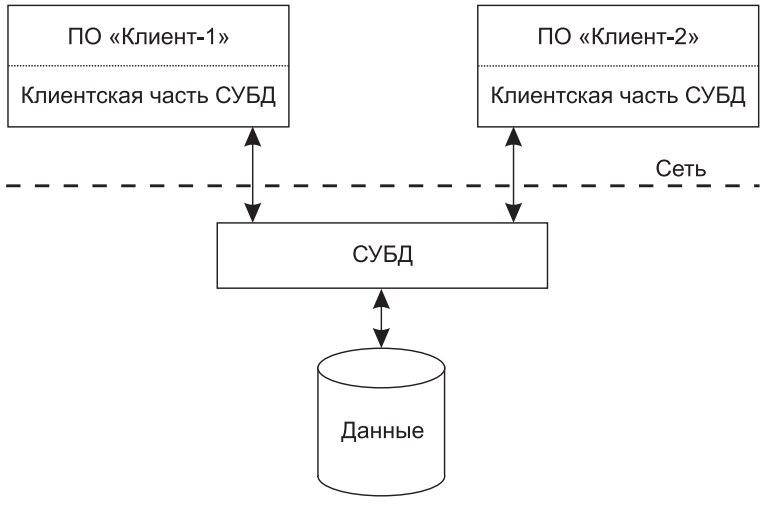

Рис. 6.6. Схема доступа к данным для приложений в архитектуре «клиент—сервер»

Постепенно с развитием архитектуры «клиент—сервер» на рынке стали появляться компании, специализирующиеся на производстве СУБД для серверов данных. В это же время разработчики прикладных программ стали отказываться от создания собственных серверов данных и стали все чаще использовать для этой цели СУБД от известных производителей.

В настоящее время разработчики, создающие программное обеспечение в архитектуре «клиент—сервер», разрабатывают именно клиентскую часть. В качестве сервера данных чаще всего выбирается СУБД одного из известных производителей. Очень редко разрабатываются обе составляющихе — и сервер данных и клиентские приложения. Но и в этом случае с одним сервером данных работают несколько различных клиентских приложений (иначе нет никакого смысла создавать программное обеспечение в данной архитектуре).

Преимущества и ограничения архитектуры «клиент—сервер»

Если вернуться к примеру, связанному с изменением нескольких байт в файле объемом несколько мегабайт, который был рассмотрен выше для архитектуры «файл—сервер», то в архитектуре приложения «клиент—сервер» решение этой задачи будет происходить по-другому. Прежде всего, данные в архитектуре «клиент—сервер» должны быть структурированы и размещены в БД на сервере. Для изменения данных клиент должен сформировать запрос к серверу на изменение этих данных. Этот

запрос передается по сети на сервер, обрабатывается и выполняется там с помощью СУБД, а результат его выполнения сообщается клиенту.

Очевидно, что при таком решении нет необходимости пересылать по сети с сервера на клиент и обратно на сервер весь объемный файл — по сети передается только запрос и результат его выполнения. При этом объем запроса и его результата ограничен именно теми данными, которые необходимо изменить, и не зависят от общего объема данных на сервере (хотя скорость обработки запроса сервером, конечно, будет зависеть от общего объема данных). Кроме того, клиент не обращается непосредственно к данным сервера, а потому не блокирует их для других клиентов на время обработки. За разграничение данных между клиентами отвечает СУБД на сервере, что делает доступ к данным более гибким.

Приложение, работающее на клиенте в архитектуре «клиент—сервер», не должно самостоятельно выполнять операции логики работы с данными и файловые операции. Это снижает нагрузку на клиентский компьютер и уменьшает трудоемкость создания клиентской части приложения. Поэтому требования к вычислительным ресурсам клиентских приложений, работающих в архитектуре «клиент—сервер», могут быть ниже, чем требования аналогичных приложений, не использующих распределенных вычислений.

ПРИМЕЧАНИЕ

Может сложиться впечатление, что клиенты в архитектуре «клиент—сервер» совсем не используют файловых операций и не работают с файлами. На самом деле это не так — клиентские приложения в этой архитектуре могут обращаться к файлам, как и любые другие. Но основные данные клиента должны храниться и обрабатываться на сервере — иначе архитектура «клиент—сервер» теряет смысл.

В результате можно выделить следующие преимущества приложений, созданных в архитектуре «клиент—сервер», по сравнению с приложениями архитектуры «файл—сервер», а также с приложениями, не использующими распределенных вычислений:

- разработчики приложений в архитектуре «клиент—сервер» избавлены от необходимости самостоятельно создавать средства для хранения данных, разделения доступа к ним, защиты и резервного копирования данных — все эти функции берет на себя СУБД;
- СУБД обеспечивает надежные и гибкие механизмы разделения доступа к данным, а также их защиты от несанкционированного доступа, удовлетворяющие общепризнанным стандартам;
- все функции по управлению данными выполняются на сервере данных, что снижает требования к вычислительным ресурсам клиентских компьютеров, на которых выполняются клиентские приложения;
- для обмена данными между клиентским приложением и сервером данных через сеть передаются не все данные, а только запросы клиента и ответы сервера, что снижает нагрузку на сеть;
- при необходимости увеличить количество клиентов в системе достаточно включить в сеть новых клиентов и, если необходимо, увеличить мощность сервера и пропускную способность сети — нет необходимости обновлять программное и аппаратное обеспечение уже существующих клиентов.

Структура приложений, построенных по архитектуре «клиент—сервер», накладывает также определенные ограничения на сами приложения и на обрабатываемые ими данные.

Во-первых, данные для приложений в архитектуре «клиент—сервер» должны быть структурированы таким образом, чтобы их можно было размещать в БД на сервере данных. Поскольку ни одна СУБД не умеет эффективно работать с данными, представленными в виде файлов неопределенной структуры, требование к созданию структуры данных в архитектуре «клиент—сервер» является определяющим. Структура данных в этой архитектуре главным образом представляет собой структуру БД на сервере данных. Если структурировать данные не удается, то все возможности архитектуры «клиент—сервер» теряются — в этом случае лучше воспользоваться технологией «файл—сервер», которая лучше подходит для работы с хранилищами данных произвольной структуры.

Во-вторых, снижение требований к вычислительным ресурсам клиентов влечет за собой перенос части выполняемых ими функций на сервер данных — и, как следствие, увеличение требований к вычислительным ресурсам сервера. Эти ресурсы необходимы для выполнения функций СУБД на сервере, и они тем более значительны, чем больше клиентов обращаются к серверу. Увеличение вычислительных ресурсов сервера и с практической и с экономической точки зрения окупается за счет снижения этих ресурсов на клиентах. Но надо помнить, что существует разумный предел количества клиентов в системе, после достижения которого требования к вычислительным ресурсам сервера возрастают настолько, что его использование становится неоправданным.

Конкретный предел количества клиентов в системе зависит от функций приложений, от распределения между клиентами и сервером, а также от интенсивности работы клиентов. В реальных системах, построенных по архитектуре «клиент—сервер», как правило, количество клиентов ограничено в диапазоне от сотни до нескольких сотен (системы, содержащие более тысячи клиентов, на практике встречаются редко). При достижении этого предела необходимо подумать либо о разбиении системы на части, в каждой из которых может быть свой сервер, либо об использовании других технологий (о некоторых из них сказано далее).

Средства разработки приложений для архитектуры «клиент—сервер»

Ситуация, связанная с распространением на рынке приложений, построенных на основе архитектуры «клиент—сервер», оказала свое влияние и на структуру систем программирования. Многие из них стали предлагать средства, ориентированные на создание приложений в архитектуре «клиент—сервер».

Для создания приложения в архитектуре «клиент—сервер» разработчику необходимо решить два основных вопроса:

- выбрать способ формирования команд обработки данных, которые должны быть понятны серверу данных;
- обеспечить взаимодействие клиентского приложения с сервером данных.

Поскольку в подавляющем большинстве приложений, построенных по архитектуре «клиент—сервер», разработчики не создают сервер данных, а используют в качестве

него одну из доступных на рынке СУБД, способ формирования команд обработки данных не может быть прерогативой разработчиков клиентского приложения — он должен быть согласован с поставщиками серверов данных. В идеале для этого необходимо создать язык манипулирования данными, разработать и утвердить стандарт такого языка. Такой язык появился вместе с выходом на рынок первых СУБД — это *язык структурированных запросов* SQL (Structured Query Language). Данному языку посвящена значительная часть следующего раздела.

Язык SQL в настоящее время является основным стандартом обмена командами между клиентскими приложениями и серверами данных. С его появлением задача формирования команд обработки данных, понятных серверу данных, была практически решена (некоторые тонкости этого вопроса рассмотрены далее). Осталась проблема: как передать команду на сервер — то есть как обеспечить взаимодействие клиентского приложения с сервером данных.

В этой области развитие вычислительных систем тоже не стояло на месте. По мере распространения архитектуры «клиент—сервер» появилось большое количество технологий обеспечения взаимодействия клиентской и серверной части приложения, построенного по этой архитектуре. И хотя при решении этой задачи не сложился такой же общепринятый стандарт, как при решении задачи создания языка манипулирования данными, все-таки количество доступных и широко используемых для этой цели технологий ограничено. Их краткому обзору посвящен один из следующих разделов.

В ходе развития архитектуры «клиент—сервер» появился язык манипулирования данными, а также множество технологий взаимодействия клиентских приложений и сервера данных. Очень скоро эти средства стали входить в состав систем программирования, ориентированных на разработку приложений для архитектуры «клиент—сервер».

Во-первых, системы программирования стали предлагать средства для создания команд и запросов на языке SQL. С точки зрения системы программирования команда SQL — это просто строка, которая никак не обрабатывается на этапах компиляции и построения исполняемой программы, а просто включается в ее состав. Уже во время выполнения программы эта строка при необходимости выполнения соответствующей команды передается клиентом на сервер, где интерпретируется как команда SQL. Такой подход не требует никаких средств от системы программирования, но создает проблемы разработчику, так как если он допустил ошибку в команде SQL, то обнаружена она будет только в момент выполнения программы. Это идет вразрез со сложившимися правилами в самих системах программирования, где все синтаксические и даже часть семантических ошибок входного языка обнаруживаются на этапе компиляции.

Поэтому в состав систем программирования стали входить средства автоматизированного построения команд SQL, которые могут помочь разработчику подготовить текст команды, а после его построения — представить этот текст в наглядном виде, выделяя все ключевые лексемы языка SQL, аналогично тому , как это происходит для текста исходной программы. Многие системы программирования пошли дальше и предлагают более мощные средства: если в тексте исходной программы или в ее пользовательских ресурсах команда SQL выделена не просто как строка, а именно как команда SQL, то система программирования анализирует ее синтаксис и сооб-

щает пользователю об имеющихся ошибках еще на этапе компиляции. Более того, если известна структура БД, на которой должна выполняться эта команда SQL, то система программирования может проверить не только ее синтаксис, но и некоторые возможные семантические ошибки. Такой подход позволяет упростить обнаружение ошибок и сокращает трудозатраты на разработку клиентских приложений.

Во-вторых, все современные системы программирования содержат в своем составе статические и динамические библиотеки, которые позволяют организовать взаимодействие клиентской части приложения с сервером данных. Большинство систем программирования предлагают разработчикам не одну , а несколько таких библиотек, каждая из которых использует свою технологию обмена с сервером данных. Эти средства поставляются в составе системы программирования и поддерживают возможность работы с широким диапазоном известных серверов данных через один или несколько доступных интерфейсов обмена данными. Разработчик прикладной программы выбирает одно из доступных средств плюс возможный тип сервера (или несколько возможных типов), и тогда его задача сводится только к созданию клиентской части приложения, построенной с использованием выбранной библиотеки взаимодействия с сервером данных.

Кроме технических вопросов, связанных с разработкой приложений для архитектуры «клиент—сервер», возникают также вопросы организационно-правового плана. Это связано с тем, что, создав клиентскую часть приложений, разработчик может дальше распространять ее только в комплексе с сервером данных и с теми библиотеками, которые он использовал для организации взаимодействия с сервером. Соответственно, пользователь прикладной программы не сможет использовать ее без наличия сервера и необходимых библиотек.

Что касается сервера данных, то для полномасштабного использования прикладной программы, построенной по архитектуре «клиент—сервер», пользователь, как правило, должен приобрести лицензию у его правообладателя. Т аким образом, он приобретает лицензии на две программы: клиентскую и серверную, причем зачастую у разных поставщиков. Нередко конечный пользователь приложения получает целый комплекс программных продуктов от множества разработчиков. Количество лицензий на клиентские приложения, как правило, должно соответствовать количеству компьютеров, где используются эти приложения. Но и стоимость серверной лицензии часто зависит от количества клиентов либо от мощности компьютера сервера, на котором установлено серверное приложение.

Зачастую пользователь прикладной программы хочет сократить стоимость лицензии на серверную часть, если он собирается использовать ограниченное количество клиентов либо приобретает программу для тестовых или демонстрационных целей. Кроме того, возникает вопрос о лицензии для разработчиков клиентского приложения: ведь для тестирования и отладки своей программы им тоже необходимо использовать сервер.

По мере развития архитектуры «клиент—сервер» в этом направлении также появились различные решения. Во-первых, многие простейшие СУБД, доступные для разработки клиентских программ, либо предлагают дешевые лицензии, либо поставляются бесплатно (или позволяют выполнять бесплатное распространение серверной части вместе с клиентской). Для целей тестирования, отладки и демонстрации клиентского приложения этого часто бывает достаточно. Во-вторых, разработчики промышленных

СУБД, лежащих в основе большинства серверов данных, предлагают широкий спектр лицензий от полных версий, ориентированных на работу десятков и сотен клиентов, до ограниченных версий, позволяющих работать единичным клиентам — в простейшем виде они зачастую поставляются бесплатно. Кроме того, разработчики промышленных СУБД, заинтересованные в распространении своих программных продуктов, предлагают льготные условия лицензирования и партнерские программы для разработчиков клиентских приложений, которые будут ориентированы на их СУБД.

Проблема с распространением библиотек , используемых для взаимодействия клиентской и серверной частей приложения, обычно решается еще проще, чем схожая проблема для сервера данных. Многие библиотеки, необходимые для наиболее известных технологий взаимодействия клиентов с сервером данных, входят в состав ОС. Разработчики клиентского приложения, ориентированного на определенную целевую вычислительную систему, как правило, выбирают для взаимодействия с сервером одну из технологий, доступных именно в этой вычислительной системе. Если же используется какая-то библиотека, входящая в состав системы программирования, то в большинстве случаев разработчики системы программирования позволяют распространять модули этой библиотеки, необходимые для взаимодействия с сервером данных, вместе с клиентским приложением, созданным в данной системе программирования, без каких-либо дополнительных ограничений.

В целом в настоящее время в составе систем программирования присутствует большое количество средств, ориентированных на разработку клиентских приложений для архитектуры «клиент—сервер», а приложения, созданные для этой архитектуры, занимают существенную долю рынка программных средств.

Современные серверы данных. Язык SQL

Принципы построения серверов данных

Как уже было сказано выше, в подавляющем большинстве случаев сервер данных представляет собой СУБД, под управлением которой находится одна или несколько баз данных, размещенных на этом сервере. Архитектура «клиент—сервер» не требует, чтобы сервер данных был построен именно на основе СУБД, и не запрещает создавать серверы, созданные на основе других принципов, но на практике такие системы встречаются исключительно редко. Поэтому развитие архитектуры «клиент—сервер» целиком и полностью связано с развитием СУБД.

В качестве СУБД на сервере данных может быть использована и простейшая («настольная») СУБД с файловым методом доступа типа Microsoft Access или Paradox. В этом случае для доступа к серверу могут использоваться те же команды и технологии, что и для других типов СУБД. Но реально доступ к БД, находящейся под управлением такой СУБД, будет происходить по методу архитектуры «файл—сервер», как показано на рис. 6.4, а не по методу архитектуры «клиент—сервер», как показано на рис. 6.5. В таком варианте из всех преимуществ архитектуры «клиент—сервер» теряются преимущества, связанные со снижением требований к вычислительным ресурсам компьютера клиента и снижением нагрузки на сеть.

Серверы данных, построенные на основе СУБД с файловым доступом («файловых СУБД»), могут использоваться при тестировании и отладке клиентских приложе-

ний, а также в тех случаях, когда количество клиентов невелико (обычно не более десятка). В дальнейшем такие серверы данных здесь не рассматриваются.

ВНИМАНИЕ

Ничто не запрещает построить на основе файловой СУБД сервер данных, обслуживающий большое количество клиентов, но эффективность работы такой системы будет низкой.

Полноценное приложение, созданное в архитектуре «клиент—сервер», имеет все преимущества данной архитектуры, указанные выше, но требует наличия СУБД со специализированным методом доступа и развитых средств взаимодействия с ней. Такие СУБД часто называют «промышленными СУБД», а построенные на их основе серверы данных — «промышленными серверами данных».

По мере развития архитектуры «клиент—сервер» на рынке стали появляться СУБД различных типов. Все СУБД можно разделить на три основных типа: реляционные, сетевые (не путать с распределенными БД) и объектные (объектно-ориентированные). Кроме того, СУБД подразделяются по способам доступа и хранения информации. В настоящее время наибольшее распространение на рынке СУБД получили реляционные БД. В качестве примеров таких СУБД, рассчитанных на применение в качестве промышленных серверов данных, можно назвать СУБД Sybase, Microsoft SQL Server , Oracle, DB2. Принципы, лежащие в основе реляционных СУБД, стали определяющими в выборе методов взаимодействия между клиентской и серверной частью в архитектуре «клиент—сервер». Этих базовых принципов стараются придерживаться все ведущие разработчики СУБД, в том числе и СУБД, построенных на основании иных технологий, чем реляционные БД.

Сейчас на рынке СУБД доступен широкий спектр различных серверов данных от самых простейших настольных СУБД (например уже упоминавшиеся здесь СУБД PARADOX и Microsoft Access) до промышленных СУБД с развитыми средствами управления данными (такие СУБД, как Microsoft SQL Server или ORA CLE). На рынке промышленных СУБД в настоящее время доминируют несколько наиболее известных компаний-производителей, предлагающих стандартизованные методы доступа к создаваемым ими СУБД. На них, в свою очередь, стали ориентироваться и разработчики прикладных программ.

СУБД и все, что связано с ними, — это отдельная очень интересная тема, выходящая за рамки данной книги. Для ознакомления с ними лучше обратиться к специализированной литературе [30, 78, 79]. Здесь же будут рассмотрены только отдельные аспекты, интересные с точки зрения компиляторов и систем программирования.

Язык SQL (Structured Query Language)

Главным фактором, определяющим взаимодействие клиентской части приложения и сервера данных, стал механизм запросов, с которыми клиентское приложение обращается к серверу. Этот механизм должен предоставить клиенту средства, которые давали бы ему возможность определить, какие данные он хочет получить от сервера и какие операции с ними выполнить. Для этой цели был предложен

специальный язык описания запросов — SQL [1] (Structured Query Language, язык структурированных запросов). По мере развития языка SQL он был признан всеми разработчиками серверов данных, и со временем появились международные стандарты языка запросов.

ПРИМЕЧАНИЕ

Аббревиатура «SQL» на английский манер часто читается как «сиквэл» либо как «эскюэль».

Элегантность и независимость от специфики используемых технологий, а также поддержка лидерами промышленности в области технологий СУБД сделали SQL (и, вероятно, в течение обозримого будущего оставят его) основным стандартным языком манипулирования с данными. По этой причине любой разработчик приложений для архитектуры «клиент—сервер» должен знать SQL. Стандарт SQL определяется ANSI (Американским национальным институтом стандартов) и в данное время также принимается ISO (Международной организацией по стандартизации). Наиболее известным и употребительным на данный момент является стандарт SQL 92, который в настоящее время поддерживается всеми серверами данных от известных производителей, но развитие языка SQL и связанных с ним стандартов продолжается.

Язык SQL предназначен для манипулирования данными в реляционных СУБД, определения структуры БД и для управления правами доступа к данным на сервере данных. Поэтому в язык SQL в качестве составных частей входят:

- язык определения данных;
- язык управления данными;
- язык манипулирования данными.

Это не отдельные языки, а различные группы команд одного языка. Деление, приведенное здесь, выполнено с точки зрения различного функционального назначения команд языка, входящих в эти группы.

Язык определения данных используется для создания и изменения структуры БД и ее составных частей — таблиц, индексов, представлений (виртуальных таблиц), а также триггеров и сохраненных процедур.

Язык управления данными используется для управления правами доступа к данным и выполнением процедур в многопользовательской среде. Более точно его можно назвать «язык управления доступом». Он состоит из двух основных команд: дать права (GRANT) и забрать права (REVOKE).

Язык манипулирования данными предоставляет средства для выполнения четырех основных операций с данными на сервере: выборка (`SELECT`), добавление (`INSERT`), обновление (`UPDATE`) и удаление (`DELETE`).

Каждая команда языка SQL описывается соответствующим предложением языка, построенным с учетом его синтаксических и семантических правил. Клиентское приложение, желающее выполнить какое-либо действие с данными, должно сформулировать его в виде предложения языка SQL, направить запрос с этим предложе-

[1] Иногда встречается другая интерпретация аббревиатуры «SQL» — «язык последовательных запросов» (Sequence Query Language).

нием серверу данных и дождаться ответа от него, в котором будет сформулирован результат выполнения операции. Если действие не может быть выполнено с помощью одной команды языка SQL, оно разбивается на несколько команд, которые выполняются последовательно.

С точки зрения клиентского приложения и сервера данных язык SQL — это формализованный язык, принципы использования которого очень похожи на принципы языков программирования.

На стороне клиента клиентское приложение должно построить предложение на языке SQL. То есть клиентская часть выступает как генератор предложений SQL. Чаще всего команды SQL закладываются в клиентское приложение на этапе разработки — то есть их создает разработчик клиентской программы. Но в ряде случаев клиентское приложение формирует предложение SQL динамически, но и тогда разработчик должен позаботиться о том, чтобы правила его формирования соответствовали грамматике SQL.

На стороне сервера каждое предложение SQL должно интерпретироваться в последовательность операций, выполняемых сервером. Получив SQL-запрос от клиента, сервер распознает его на основе грамматики SQL, определяющей синтаксис запроса, а также проверяет семантику , основываясь на структуре БД, которая известна серверу данных. Если запрос содержит синтаксические или семантические ошибки, сервер сообщает об этом клиенту. Если же запрос правильный, на его основе сервер строит схему выполнения запроса, которую затем выполняет и передает результат ее выполнения клиенту. Сервер выступает как распознаватель языка SQL и является интерпретатором данного языка.

ПРИМЕЧАНИЕ

Видно, что схема обработки и выполнения SQL-запросов на сервере данных полностью соответствует принципам, на основе которых функционируют интерпретаторы и компиляторы для языков программирования. С этой точки зрения язык SQL является полноправным компьютерным языком (хотя нельзя сказать, что он является языком программирования).

Видно, что этой схеме присущ недостаток, который связан со всеми интерпретируемыми языками, — каждый раз сервер данных вынужден тратить время на распознавание запроса и проверку его правильности. В том случае, когда клиент направляет серверу часто повторяющиеся типовые запросы, это ведет к неэффективным затратам времени. Эффективность схемы была бы выше, если бы сервер данных мог работать по схеме компилятора — один раз распознавать запрос, а потом выполнять его каждый раз при обращении клиента. Разработчики СУБД обратили на это внимание, и в современных серверах данных присутствуют соответствующие решения в виде запросов с параметрами, предопределенных выборок (`VIEW`) и хранимых процедур сервера. В этих случаях сервер только один раз распознает запрос, находит его синтаксические конструкции и запоминает у себя в виде внутренних команд. При повторном выполнении того же запроса (а также VIEW или хранимой процедуры) сервер находит уже построенную последовательность внутренних команд и выполняет ее. Это не делает текст SQL компилируемым, но повышает эффективность его выполнения.

Более подробно об этом лучше узнать в специализированной литературе, посвященной современным СУБД.

Другим важным механизмом, определяющим взаимодействие клиентской и серверной части, является механизм транзакций. Его суть заключается в том, что клиентское приложение может определить группу взаимосвязанных действий над данными, которые должны быть либо выполнены все вместе в комплексе, либо не выполнены вообще. Взаимосвязь действий определяется логикой клиентской части. Тогда, начав выполнение этой группы действий (открыв транзакцию), клиент последовательно передает серверу данных все команды в виде предложений языка SQL и ожидает результатов. При правильности выполнения всех команд клиентская часть подтверждает это (закрывает транзакцию), и только после этого операция считается выполненной, и все сделанные изменения вносятся в данные на сервере. Если же по каким-то причинам операция не могла быть выполнена, клиент отказывается от нее (отменяет транзакцию), и сервер должен восстановить то состояние данных, которое было до начала транзакции.

Взаимодействие клиентской и серверной части приложения в архитектуре «клиент—сервер» здесь описано только в самом общем виде. Рассмотрение операторов языка SQL, механизма транзакций и других механизмов, связанных с работой СУБД, не входит в рамки данного учебника. Об этом можно узнать из книг , посвященных современным СУБД.

Дополнительные возможности серверов данных

Язык SQL хорошо стандартизован — и этот факт определил его повсеместное использование в приложениях, построенных на основе архитектуры «клиент—сервер». Но этот же факт является причиной главного недостатка языка: как и все, что хорошо стандартизовано, язык SQL развивается достаточно медленно, и скорость его развития зачастую отстает от потребностей клиентских приложений.

Главной движущей силой развития SQL являются разработчики СУБД, так как именно они являются наиболее значительными игроками на рынке программных средств, связанных с архитектурой «клиент—сервер». Разработчики СУБД стараются следить за потребностями разработчиков клиентских приложений, так как они заинтересованы в максимальном количестве приложений, взаимодействующих именно с их СУБД. На основе собранной информации они предлагают различные нововведения в язык SQL, но пока эти нововведения пройдут все необходимые этапы стандартизации, пройдет значительное время. Разработчики СУБД, стремясь удовлетворить разработчиков клиентских приложений, зачастую не ждут завершения этого процесса и вводят новшества в свои СУБД до появления нового стандарта SQL.

И в этом случае перед разработчиками клиентских приложений возникает дилемма: использовать только что появившиеся, но еще не стандартизованные возможности языка, либо ждать завершения процесса их стандартизации (и еще не факт , что эти новшества войдут в стандарт).

В первом случае разработчики клиентского приложения могут выиграть время, опередив конкурентов, но при этом они попадают в зависимость от производителей конкретного типа СУБД, так как нововведения SQL, реализованные вне рамок стандарта в одном типе СУБД, редко могут быть применимы на СУБД другого типа. Кроме того, если эти новшества войдут в новый стандарт , но в другом виде, нежели они были предварительно реализованы разработчиками какой-то СУБД, разработчики клиентского приложения вынуждены будут переделывать свою программу под тре-

бования стандарта (фактически второй раз создавать функции, которые уже были один раз реализованы).

Во втором случае, ожидая выхода стандарта, разработчики клиентской части остаются вне зависимости от производителей СУБД и могут использовать в качестве сервера данных любую СУБД из доступных на рынке. Но, выжидая определенное время, они могут отстать от конкурентов, которые могут реализовать те же функции более эффективными, но нестандартными методами.

Есть еще и третий путь: поскольку на рынке СУБД основную роль играют лишь несколько ведущих производителей СУБД, разработчики клиентских приложений могут выбрать в качестве серверов данных для своих приложений СУБД только от двух-трех основных фирм и не рассматривать другие. Тогда можно ориентироваться только на нововведения, предлагаемые именно этими разработчиками СУБД, и в процессе выполнения клиентского приложения динамически подстраивать SQL-команды под тот тип СУБД, который используется на конкретном сервере данных. Это потребует наличия в составе клиентского приложения гибкого генератора предложений SQL, что не является принципиальной сложностью, так как грамматика языка SQL известна (а нововведения, вносимые в SQL ведущими разработчиками СУБД, по своей сути чаще всего различаются только синтаксисом). Этот путь позволяет совместить преимущества первых двух вариантов, но требует от разработчиков клиентского приложения немного больше трудозатрат и более высокой квалификации[1].

Кроме проблем, связанных со стандартизацией языка SQL, на серверах данных могут существовать дополнительные возможности, которые не охвачены этим языком. Дело в том, что практически все современные СУБД предусматривают возможность создания в БД хранимых процедур, которые выполняются по наступлению определенных событий, по командам клиентов или по расписанию [30]. Язык, на котором создается исходный текст хранимых процедур, лежит вне рамок стандарта языка SQL (фактически язык SQL является подмножеством языка хранимых процедур). И здесь разработчики СУБД реализуют все возможности своих СУБД без всякого согласования друг с другом.

Язык, лежащий в основе текстов хранимых процедур большинства современных СУБД, по своим возможностям практически ни в чем не уступает многим языкам программирования. После создания хранимой процедуры ее текст обрабатывается и анализируется СУБД, которая в этом случае выступает в роли полноценного компилятора. Если исходный текст хранимой процедуры не содержит синтаксических и семантических ошибок, он сохраняется на сервере в виде внутреннего представления этой процедуры. При необходимости выполнить хранимую процедуру СУБД выполняет созданное ранее ее внутреннее представление, выступая при этом в роли интерпретатора.

Если разработчики приложения, функционирующего в архитектуре «клиент—сервер», используют хранимые процедуры, у них появляется возможность вынести на сервер не только функции логики работы с данными, но и значительную часть функций прикладной логики приложения. Поскольку возможности языка хранимых процедур мало в чем уступают возможностям языков программирования, используемых для создания клиентских приложений, в предельном случае на сервер данных может

[1] Автору известна компания-разработчик клиентских приложений для архитектуры «клиент—сервер», где используется именно этот путь.

быть вынесена вся прикладная логика приложения, а на клиентской части останутся только функции, непосредственно связанные с интерфейсом пользователя.

Разработчики современных СУБД предоставляют разработчикам клиентских приложений для создания хранимых процедур в своих СУБД не просто текстовые редакторы, а целые наборы инструментальных средств. В состав этих средств входят интегрированные (чаще всего графические) среды для создания исходного кода хранимых процедур, компиляторы и отладчики этого кода, различные вспомогательные инструменты. Фактически возможности таких инструментариев полностью соответствуют возможностям современных систем программирования с той только разницей, что они ориентированы на создание интерпретируемого кода для единственной целевой вычислительной системы — СУБД соответствующего типа.

Использование хранимых процедур дает разработчикам клиентских приложений в архитектуре «клиент—сервер» широкие возможности, но при этом жестко ограничивает для них выбор сервера данных одним единственным типом СУБД, так как исходный код хранимых процедур не является переносимым.

Технологии доступа к серверам данных

Основные принципы взаимосвязи клиента с сервером данных

Используя стандартизованный язык SQL, разработчик клиентской части приложения, работающего в архитектуре «клиент—сервер» может строить запросы к серверу данных, не ориентируясь на определенный тип СУБД. При этом взаимодействие клиентской части с сервером на уровне запросов не будет зависеть от типа используемого сервера данных (СУБД). Однако остается открытым вопрос о том, как SQL-запросы будут передаваться от клиента серверу и каким образом клиент будет получать ответы.

Наиболее очевидным решением было бы использовать для этой цели динамически загружаемые библиотеки. Клиентское приложение обращается к функции соответствующей библиотеки и передает ей запрос к серверу в виде предложения на языке SQL. В задачу библиотеки входит установить связь с сервером, передать ему запрос, дождаться ответа и вернуть резуль тат клиенту в качестве резуль тата выполнения функции. Однако такое простое решение обладает одним существенным недостатком: для каждого типа сервера должна использоваться своя динамически загружаемая библиотека со своими функциями, что делает клиентское приложение зависимым от типа СУБД.

В настоящее время на рынке СУБД доминируют несколько крупных производителей, среди которых всегда можно выбрать сервер данных, удовлетворяющий тем или иным требованиям, предъявляемым для решения прикладной задачи. Поэтому зачастую разработчики клиентской части программного обеспечения, работающего в архитектуре «клиент—сервер», пренебрегают указанным недостатком, используя описанную выше схему для взаимодействия с сервером данных. Они разрабатывают свои приложения, ориентируясь на определенный тип СУБД. Этому способствует тот факт, что многие фирмы-производители СУБД (такие, например, как Oracle или Microsoft) предлагают не только серверы данных, но и системы программирования, ориентированные на работу с конкретным типом СУБД. Такой путь имеет свои пре-

имущества, так как прямое обращение к динамической библиотеке, взаимодействующей с сервером данных, дает наибольшую эффективность по скорости выполнения запросов клиента.

Однако зависимость от типа СУБД не всегда допустима для клиентской части в архитектуре «клиент—сервер», особенно в том случае, когда разработчики стремятся добиться переносимости своего программного обеспечения. Для этой цели служат универсальные интерфейсы взаимодействия клиентской и серверной части в архитектуре «клиент—сервер». Существует несколько вариантов таких интерфейсов от разных производителей. Одним из наиболее распространенных является интерфейс ODBC.

Технология ODBC

ODBC (Open Database Connectivity , открытая взаимосвязь с БД) является средством, позволяющим унифицировать организацию взаимодействия с различными СУБД. Для этого доступ к базе данных осуществляется при помощи специального ODBC-драйвера, который транслирует запросы к базе данных на язык, поддерживаемый конкретной СУБД.

Для установления соединения с БД технология ODBC использует ODBC-драйверы и источники данных (data sources), которые позволяют настроиться на сеанс конкретного пользователя СУБД. Для этого источник содержит имя пользователя и его пароль, а также, при необходимости, другую информацию, требуемую для присоединения к СУБД. ODBC-драйвер представляет собой динамически загружаемую библиотеку, которая может использоваться приложением для получения доступа к конкретному серверу данных. Каждому типу СУБД соответствует свой ODBC-драйвер. Система ODBC также включает в себя ряд служебных утилит.

Взаимодействие клиентской части программного обеспечения с СУБД осуществляется при помощи набора системных вызовов, которые выполняются по отношению к источнику данных. Источник данных взаимодействует с драйвером ODBC, транслирует ему запросы клиента и получает ответы, а тот , в свою очередь, обращается к СУБД. При необходимости работать с другим типом СУБД достаточно указать другой тип ODBC-драйвера в источнике данных, при этом не требуется изменять саму клиентскую часть программного обеспечения, работающего в архитектуре «клиент—сервер».

ВНИМАНИЕ

Переход клиентской части программного обеспечения в архитектуре «клиент—сервер» с одного типа сервера данных на другой возможен только в том случае, если для взаимодействия с СУБД клиентская часть использует запросы, удовлетворяющие стандарту , поддерживаемому обоими серверами.

Такая двухступенчатая схема взаимодействия клиента с сервером данных обеспечивает независимость клиентской части от типа СУБД на сервере данных, но в целом снижает эффективность работы приложения в архитектуре «клиент—сервер», поскольку запрос, посланный клиентом, дважды обрабатывается различными библиотеками функций прежде, чем попадет на сервер. Как и в случае решения вопроса о переносимости программного обеспечения, решение вопроса о выборе способа

взаимодействия с сервером данных остается за разработчиком клиентской части программного обеспечения и зависит от предъявляемых к нему требований.

В настоящее время ODBC представляет собой развитую и хорошо стандартизированную технологию, доступную для вычислительных систем различных архитектур. Данная технология сейчас широко используется в системах, построенных на основе архитектуры «клиент—сервер», но дальнейшее развитие этой технологии под вопросом, так как основные производители предлагают на рынок новые, более современные технологии.

Технологии OLE DB и ADO

Технологии OLE DB и ADO были созданы компанией Microsoft и получили распространение в среде ОС типа Windows. Основой для их создания, в свою очередь, послужила технология OLE (Object Linking and Embedding — связывание и внедрение объектов), которая первоначально создавалась и развивалась в ОС типа Windows для обеспечения взаимосвязи между прикладными программами (в первую очередь — для связи между собой приложений Microsoft Office). После того как в основу OLE была положена принципиально новая технология COM[1] (Component Object Model — Модель многокомпонентных объектов), стало возможным использовать OLE для связи любых типов программ.

В результате применения технологии OLE к решению задачи организации взаимодействия клиентской и серверной частей приложения была получена новая технология — OLE DB (OLE for DataBases — OLE для баз данных).

С точки зрения Microsoft OLE DB представляет собой следующую ступень развития технологии ODBC. Обе технологии образуют относительно независимый программный слой, использующий однотипные интерфейсы прикладных программ (API)для доступа к разным типам СУБД. Их работа обеспечивается программными модулями, которые учитывают специфические особенности каждой СУБД. В OLE DB полностью реализован принцип универсального доступа к разнотипным СУБД. Наибольшие различия технологий ODBC и OLE DB проявляются в использовании некоторых основных терминов и в окружающем контексте.

Технология ADO родилась в результате объединения OLE DB с еще одной технологией, созданной Microsoft, — ActiveX[2]. Отсюда происходит и название технологии ADO — ActiveX Data Object (объекты данных ActiveX).

С точки зрения разработчика клиентской части приложения в архитектуре «клиент—сервер» ADO представляет собой прикладной объектный интерфейс более высокого уровня, чем OLE DB. Этот интерфейс упрощает доступ к средствам OLE DB разработчикам, использующим языки высокого уровня. В настоящее время разработчик этих технологий — компания Microsoft выпустила уже несколько версий библиотек ADO и продолжает развивать эту технологию.

Однако развитие этих технологий ограничено, прежде всего, тем, что они ориентированы в основном только на вычислительные системы, построенные на базе ОС типа

[1] Особенности технологии COM рассмотрены далее в разделе, посвященном многоуровневой архитектуре.

[2] Технология ActiveX более подробно рассмотрена в разделе, посвященном разработке приложений для сети Интернет.

Windows. И хотя появляются версии ADO для других типов ОС, переносимость приложений, созданных на основе этой технологии, на вычислительные системы, не использующие Windows, остается проблемой.

Способы организации взаимодействия клиентской и серверной частей приложений на основе архитектуры «клиент—сервер» не ограничиваются только перечисленными технологиями. Сложно даже перечислить их все в рамках данной книги, а тем более подробно описать. Более подробно об организации приложений на основе архитектуры «клиент—сервер» можно узнать в [25, 27, 48, 77, 79].

Поддержка технологий доступа к СУБД в системах программирования

Появление и развитие различных технологий доступа клиентской части приложений к серверу данных не осталось без внимания разработчиков систем программирования (особенно если учесть, что такие создатели новых технологий, как компания Microsoft, являются одновременно и разработчиками систем программирования). Поэтому на рынке систем программирования появились системы, поддерживающие основные существующие технологии доступа клиента к серверу данных.

Если разработчик клиентской части приложения останавливает свой выбор на одной из существующих стандартных технологий, то в этом случае он не только избегает зависимости создаваемого приложения от типа СУБД, но и получает возможность выбора среди систем программирования, доступных на рынке. Это обусловлено тем, что практически все современные системы программирования поддерживают такие распространенные технологии, как ODBC или ADO, и предоставляют разработчикам инструменты и библиотеки, снижающие трудоемкость создания клиентов на основе таких технологий. В дальнейшем возможности созданного приложения будут ограничены только возможностями выбранной технологии: например, технология ADO имеет широкие возможности, но ее применение ограничено только вычислительными системами, построенными на базе ОС типа Windows.

Таким образом, выбирая стандартную технологию, разработчик получает выигрыш в выборе СУБД и средств разработки, но проигрывает в производительности, так как универсальные методы доступа к СУБД менее эффективны, чем специализированные.

Если же разработчик клиентской части приложения выберет для взаимосвязи с сервером данных специализированную технологию, ориентированную на определенный тип СУБД, то он, безусловно, получит более высокую скорость обмена данными между сервером и клиентом. Однако в этом случае он будет ограничен и в выборе типа СУБД, и в выборе средств разработки. Первое утверждение очевидно, так как специализированный метод доступа ориентирован на определенный тип СУБД (а зачастую — и на определенную версию СУБД). Второе утверждение основано на том, что специализированные методы доступа к СУБД либо совсем не поддерживаются системами программирования от других производителей, либо имеют ограниченные инструменты поддержки, снижающие эффективность разработки. В этом случае лучше всего использовать систему программирования, созданную тем же производителем, что и СУБД (в настоящее время ведущие производители СУБД предлагают и свои средства разработки).

Выигрыш в эффективности доступа клиента к серверу данных при использовании специализированных методов обусловлен еще одним немаловажным фактором:

все производители СУБД заинтересованы в создании как можно большего количества клиентских приложений, ориентированных именно на их СУБД. Поэтому все полезные для клиентов нововведения появляются, в первую очередь, именно в специализированных средствах доступа и только потом уже, если это возможно, переносятся в средства, обеспечивающие поддержку стандартных технологий.

Каждый из описанных путей создания клиентского приложения в архитектуре «клиент—сервер» не лишен своих преимуществ и недостатков. Выбор одного из них остается за разработчиком и зависит от тех условий, для которых создается то или иное приложение.

Проблемы и недостатки архитектуры «клиент—сервер»

Приложения, созданные на основе архитектуры «клиент—сервер» , получили ряд преимуществ по сравнению с приложениями, не использующими распределенных вычислений, а также приложениями, созданными на основе архитектуры «файл—сервер». Тот факт, что основные производители систем программирования включили в свои системы средства поддержки разработки приложений в этой архитектуре, обусловил широкое распространение таких приложений на рынке программных средств.

В то же время сама по себе архитектура «клиент—сервер» не лишена некоторых недостатков:

- функции управления данными возложены на сервер данных, но обработка данных (прикладная логика или бизнес-логика) по-прежнему выполняется клиентами, что не позволяет существенно снизить требования к ним, если логика приложения предусматривает достаточно сложные манипуляции с данными;
- при необходимости изменить или дополнить логику обработки данных необходимо выполнить обновление клиентских приложений на всех рабочих местах, что может быть достаточно трудоемким;
- если необходимо изменить только внешний вид интерфейса пользователя (отображение данных), но оставить неизменной логику обработки данных, при этом чаще всего требуется заново создать и установить на рабочем месте новый вариант клиентской части системы;
- при использовании мощной промышленной СУБД требуется наличие лицензии на подключение к СУБД для каждого рабочего места, где установлена клиентская часть программного обеспечения.

Наличие хранимых процедур и других мощных средств, расширяющих возможности серверов данных, позволяет перенести на них также и значительную долю функций прикладной логики. Это несколько нивелирует большинство указанных недостатков архитектуры «клиент—сервер», но не позволяет полностью избавиться от них по следующим причинам:

- внутренние языки СУБД, используемые для создания хранимых процедур и других подобных средств, хотя и сравнимы в настоящее время по своим возможностям с языками современных систем программирования, но все же уступают им — поэтому не все функции прикладной логики могут быть реализованы на сервере данных столь же эффективно, как на клиенте;

- хранимые процедуры, триггеры и другие средства СУБД, созданные на ее внутреннем языке, после предварительной компиляции сохраняются во внутреннем коде СУБД (но не в машинных кодах), а во время исполнения интерпретируются СУБД — поэтому они проигрывают в производительности резуль тирующим программам, созданным в машинных кодах.

СОВЕТ

В архитектуре «клиент—сервер» вовсе не следует стремиться все функции прикладной логики возложить на сервер данных. Следует помнить, что СУБД интерпретирует код, а потому серверу всегда потребуется больше ресурсов для выполнения тех же функций, которые на клиенте выполняются в машинных кодах. Разумное распределение функций между клиентской и серверной частью приложения зависит от многих факторов, в том числе от количества клиентов, мощности клиентских и серверного компьютеров и др.

Еще одним средством, которое может расширить возможности приложений, построенных на основе архитектуры «клиент—сервер», является использование терминального доступа в сочетании с возможностями этой архитектуры. В этом случае компьютер-клиент, на котором выполняется клиентская часть приложения, одновременно становится терминальным сервером, к которому могут подключаться компьютеры-терминалы. Тогда вся логика организации пользовательского интерфейса выполняется компьютерами-терминалами, прикладная логика приложения выполняется на терминальном сервере (который одновременно является клиентом), а логика обработки данных и файловые операции выполняются на сервере данных. Поскольку клиентов может быть несколько, то и терминальных серверов в такой структуре может быть несколько. Т огда пользователь компьютера-терминала либо подключается только к заданному терминальному серверу, либо имеет возможность выбрать один из доступных терминальных серверов для подключения.

В общем случае схема организации распределенных вычислений с использованием архитектуры «клиент—сервер» и терминального доступа представлена на рис. 6.7.

Схема, представленная на рис. 6.7, позволяет избежать основных недостатков, присущих приложениям, построенным на основе архитектуры «клиент—сервер». При такой организации распределенных вычислений к вычислительным ресурсам компьютеров-терминалов предъявляются минимальные требования, прикладная логика приложения выполняется на терминальных серверах, которых может быть несколько, но в любом случае значительно меньше, чем терминальных компьютеров, за которыми работают пользователи. При необходимости изменить или дополнить логику обработки данных необходимо выполнить обновление клиентских приложений только на терминальных серверах.

Основной недостаток такой организации распределенных вычислений связан с основным недостатком терминального доступа: для каждого компьютера-терминала на терминальном сервере запускается на выполнение своя копия клиентской части приложения. Все клиентские части приложения, выполняемые на каждом терминальном сервере, функционируют независимо друг от друга и взаимодействуют между собой только через сервер данных. Это ведет к не вполне эффективному использованию вычислительных ресурсов терминальных серверов, так как одни

и те же данные могут несколько раз поступать на терминальный сервер с сервера данных для каждой выполняемой копии клиентского приложения.

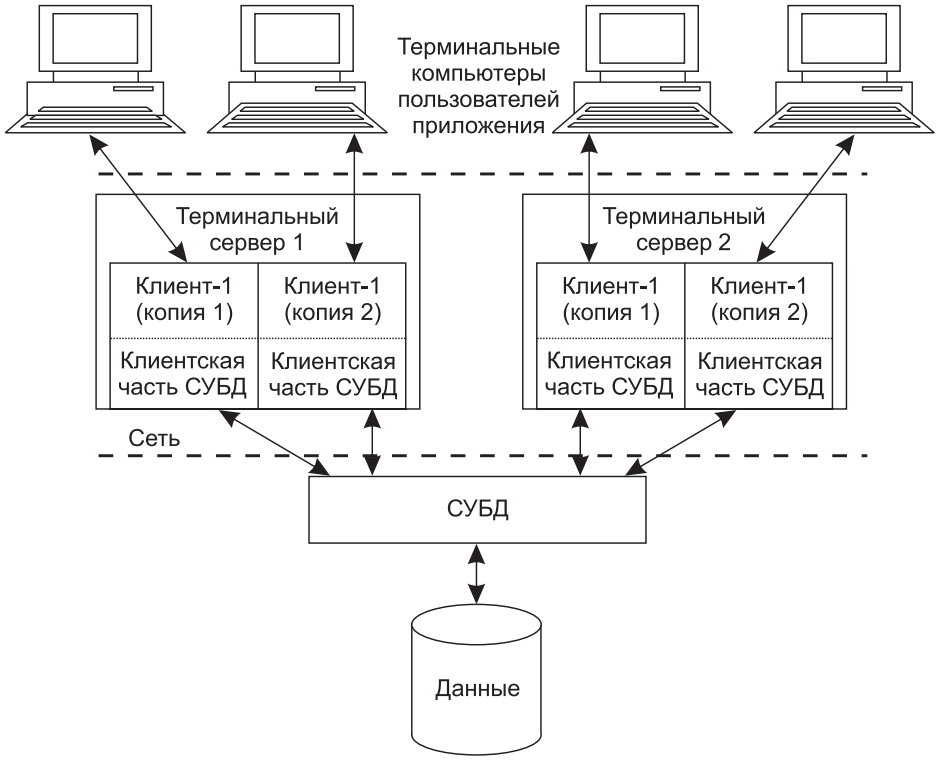

Рис. 6.7. Структура приложения, построенного на основе архитектуры «клиент—сервер» с использованием терминального доступа

Для полного устранения перечисленных выше недостатков была предложена следующая модель разработки приложений — трехуровневая, а в общем случае многоуровневая архитектура.

Разработка программ в многоуровневой архитектуре

Принципы разработки приложений в трехуровневой и многоуровневой архитектуре

Понятие о трехуровневой архитектуре программного обеспечения

Трехуровневая архитектура разработки приложений явилась логическим продолжением идей, заложенных в архитектуре «клиент—сервер».

Главным недостатком архитектуры приложений типа «клиент—сервер» стал тот факт, что в клиентской части приложения совмещались как довольно сложные

функции прикладной логики обработки данных (бизнес-логики), так и более простые функции организации интерфейса пользователя. Для выполнения этих функций к вычислительной системе должны предъявляться разные требования, и в том случае, когда выполняется сложная обработка данных, эти требования со стороны «клиентской» части приложения могут быть непомерно велики. Если же разделить эти функции на разные компьютеры средствами ОС с помощью терминального доступа, то в этом случае параллельное и несогласованное между собой выполнение функций прикладной логики ведет к неэффективному использованию вычислительных ресурсов.

Кроме того, обработка данных (бизнес-логика приложения), как правило, изменяется незначительно по мере прохождения жизненного цикла программного обеспечения, развития приложения и выхода его новых версий. В то же время интерфейсная часть может серьезно видоизменяться и в предельном случае подстраиваться под требования конкретного заказчика.

Еще одним фактором, повлиявшим на дальнейшее развитие архитектуры «клиент—сервер», стало распространение глобальных сетей и развитие Всемирной сети Интернет. Многие приложения стали нуждаться в предоставлении пользователю возможности доступа к данным посредством Глобальной сети. Возможностей архитектуры «клиент—сервер» для этой цели во многих случаях недостаточно, поскольку клиент зачастую мог не иметь никаких вычислительных ресурсов, кроме программы навигации по сети (браузера, browser).

Поэтому дальнейшим развитием архитектуры «клиент—сервер» стало разделение клиентской части еще на две составляющие: сервер приложений (английский термин «application server»), реализующий прикладную логику обработки данных (бизнес-логику приложения), и «тонкий клиент» («thin client»), обеспечивающий интерфейс и доступ к резуль татам обработки (далее «тонкий клиент» будем называть просто «клиентом»). Серверная часть осталась без изменений, но теперь ее стали называть «сервер данных», или «сервер баз данных» («database server»), а не просто «сервер», чтобы не путать с сервером приложений.

Простейшая структура программного обеспечения , построенного на основе трехуровневой архитектуры, представлена на рис. 6.8.

Как уже было сказано, в состав программного обеспечения, построенного на основе трехуровневой архитектуры, входит три основных составляющих:

- сервер данных;
- сервер приложений;
- клиенты (клиентская часть).

Сервер данных, как и в случае архитектуры «клиент—сервер», выполняет все операции, связанные с управлением данными: хранение данных, доступ к данным, защита и резервное копирование данных. Как и в архитектуре «клиент—сервер», в качестве сервера данных в трехуровневой архитектуре чаще всего выступает соответствующая СУБД.

Сервер приложений выполняет все функции, связанные с прикладной логикой обработки данных, а также он формирует все запросы к серверу данных и обеспечивает представление данных для клиентской части в той или иной форме.

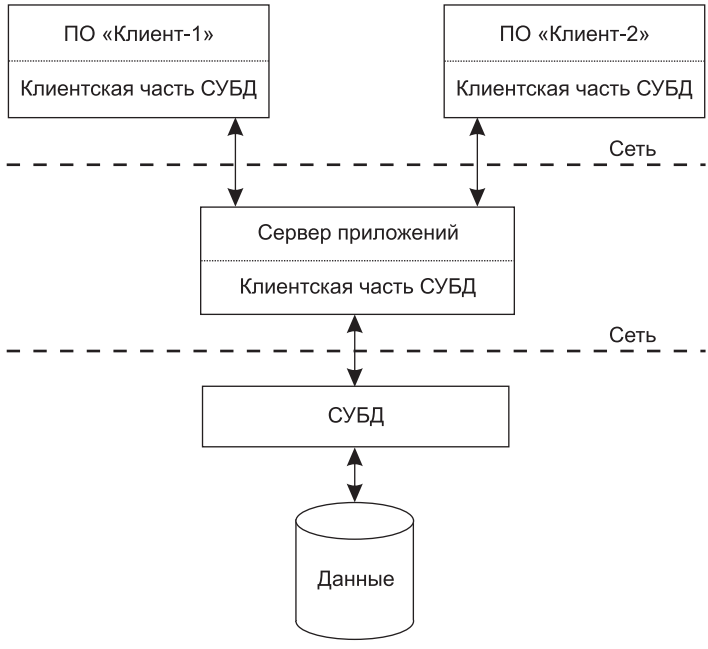

Рис. 6.8. Структура приложения, построенного на основе трехуровневой архитектуры

На клиентскую часть приходятся функции, связанные с отображением данных и организацией интерфейса пользователя.

Особенности разработки программного обеспечения для многоуровневой архитектуры

В общем случае программное обеспечение, построенное на основе трехуровневой архитектуры, может иметь более сложную структуру, чем показано на рис. 6.8. Например, сервер приложений может обмениваться данными не с одним, а с несколькими серверами данных или клиентская часть может взаимодействовать с несколькими серверами приложений. С другой стороны, по своим функциям сам сервер приложений может быть разделен на несколько уровней. В самом простейшем случае первый уровень будет взаимодействовать с сервером данных, второй уровень — выполнять обработку данных, а третий уровень — формировать данные для представления их клиенту. В некоторых случаях такое усложнение системы оправдано. Тогда говорят не о трехуровневой, а о многоуровневой архитектуре программного обеспечения.

Несмотря на рассмотренные выше возможные усложнения трехуровневой архитектуры программного обеспечения, принципы, лежащие в ее основе, остаются неизменными и могут быть рассмотрены в рамках трехуровневой архитектуры. У величение количества серверов данных или серверов приложений в каждом конкретном случае, а также разбиение самого сервера приложений на несколько уровней не вносит принципиальных изменений в организацию механизмов обмена данными. Просто увеличивается количество взаимодействующих приложений, и могут усложняться протоколы обмена данными между ними.

В архитектуре «клиент—сервер» разработчик прикладной программы создавал только клиентскую часть приложения, а в качестве сервера данных использовал, как правило, одну из имеющихся на рынке СУБД. В трехуровневой архитектуре разработчик должен создавать две составляющие: сервер приложений и клиентскую часть, а также выбрать механизм обмена данными между ними и определить протокол обмена данными. В многоуровневой архитектуре разработчик должен создать клиентскую часть и все необходимые серверы приложений. Это является принципиальным отличием разработки программного обеспечения для трехуровневой архитектуры от разработки программного обеспечения для архитектуры «клиент—сервер».

Далее будут рассмотрены механизмы, лежащие в основе организации обмена данными между сервером приложений и клиентской частью.

Серверы приложений. Методы доступа к серверам приложений

Организация серверов приложений

Понятие «сервер приложений» сейчас широко распространено на рынке программных средств. Однако четкое определение данного понятия в настоящее время до сих пор отсутствует. Во многом это связано с тем, что до сих пор не разработаны четкие общепринятые стандарты для многоуровневой архитектуры, как это было сделано в свое время для архитектуры «клиент—сервер». Постараемся восполнить этот пробел.

Сервер приложений — это часть приложения, выполняющая возложенные на него функции по обработке данных в соответствии с функциональной спецификацией, получающая необходимые для обработки данные и обеспечивающая предоставление результатов обработки другим составным частям приложения.

В соответствии с этим определением сервер приложений должен состоять из следующих основных частей (программных модулей):

- модуль получения данных, необходимых для обработки: в трехуровневой архитектуре данные поступают от сервера данных, в многоуровневой архитектуре — от сервера данных или от других серверов приложений;
- модуль обработки данных непосредственно реализует функции прикладной логики (бизнес-логики), возложенные на данный сервер;
- модуль предоставления результатов передает результаты обработки данных на другие части приложения: в трехуровневой архитектуре данные передаются на клиентскую часть, в многоуровневой архитектуре — на клиентскую часть или на другие серверы приложений.

Указанные части сервера приложений не обязательно должны быть реализованы в виде отдельных программных модулей или библиотек, но так или иначе они должны присутствовать в составе программного обеспечения сервера.

При этом основную проблему при создании сервера приложений представляет не разработка модуля обработки данных, непосредственно выполняющего функции этого сервера. Основную проблему представляет разработка модулей взаимодействия с другими частями приложения для получения исходных данных и передачи результатов их обработки. И если для получения данных от сервера данных в трех-

уровневой архитектуре можно использовать технологии, рассмотренные выше для архитектуры «клиент—сервер», то для взаимодействия с клиентами в трехуровневой архитектуре, а также с другими серверами приложений в многоуровневой архитектуре требуются принципиально иные технологии.

Используя эти технологии и стандарты (которые частично описаны далее), разработчик может сам создать сервер приложений, а затем и клиентскую часть для своего программного продукта, разработать протоколы обмена данными между ними и создать соответствующие функции. Однако создание сервера приложений и разработка протоколов взаимодействия с ним — это достаточно сложный и трудоемкий процесс, содержащий много факторов, которые потенциально могут привести к появлению сложно обнаруживаемых ошибок. С другой стороны, в составе сервера приложений только связанные с логикой работы программной системы функции (обработка данных, их получение от сервера и представление клиенту) представляют собою «значимую» часть, которая требует интеллектуального труда разработчика. А весь программный код, связанный с обеспечением обмена данных сервера приложений и клиентской части, как правило, требует кропотливого рутинного труда, который может быть автоматизирован.

На этот факт обратили внимание производители систем программирования , и сейчас на рынке средств разработки программного обеспечения появились программные продукты, часто носящие название «сервер приложений». Эти продукты зачастую входят в состав соответствующих систем программирования, но могут поставляться и отдельно от них. Такой сервер приложений представляет собой своеобразный контейнер, обеспечивающий разработчику взаимодействие с клиентской частью и с сервером данных по определенному стандарту , а также облегчающий написание программного кода, реализующего логику обработки данных. От разработчика требуется только создать код, отвечающий за саму обработку, — все остальное большей частью обеспечит сервер приложений. Единственным ограничением при таком способе создания программного обеспечения для трехуровневой архитектуры является тот факт, что созданный таким образом полномасштабный сервер приложений должен поставляться заказчику в комплекте с программным кодом сервера приложений из состава системы программирования, а это чаще всего требует приобретения дополнительных лицензий от производителя системы программирования. Однако при таком пути достигается существенный выигрыш в объеме трудозатрат на разработку сервера приложений [70, 82].

Программное обеспечение, обеспечивающее взаимодействие между частями приложений, функционирующими на разных уровнях, получило название «промежуточное программное обеспечение» (от английского термина «middleware»).

В принципе задача организации взаимодействия параллельно функционирующих приложений возникла задолго до появления трехуровневой архитектуры (просто с развитием распределенных вычислений она стала более актуальной). Поэтому основы некоторых технологий, лежащих в основе взаимодействия частей приложения, построенного по многоуровневой архитектуре, возникли задолго до ее появления. Одной из таких технологий является технология RPC.

Технология RPC (Remote Procedure Call)

Технология RPC (Remote Procedure Call, вызов удаленных процедур) возникла для обеспечения взаимодействия двух параллельно выполняющихся процессов в ОС типа

UNIX задолго до появления многоуровневых архитектур программного обеспечения. Ее основы были заложены в саму архитектуру ОС типа UNIX [31, 45, 53, 61, 62, 72].

Вызов удаленной процедуры заключается в том, что один из процессов («процесс-клиент») запрашивает у другого процесса («процесса-сервера») некоторую услугу (сервис) и не продолжает свое выполнение до тех пор, пока не получит соответствующие результаты. Видно, что по смыслу такой механизм взаимодействия процессов напоминает вызов процедуры или функции в прикладной программе (отсюда и происходит название). Разница заключается в том, что выполнение вызова может происходить не только в рамках разных задач, но и на разных компьютерах.

Кроме того, видно, что механизм вызова RPC очень похож на обмен данными между клиентской частью и сервером данных в приложении, построенном на основе архитектуры «клиент—сервер» (не случайно процессы и приложения имеют схожие названия). Это неудивительно, поскольку в большинстве случаев программное обеспечение в архитектуре «клиент—сервер» также использует механизмы RPC, просто они остаются закрытыми для разработчика прикладной программы.

Реализация технологии RPC достаточно сложна, поскольку этот механизм должен обеспечить работу взаимодействующих приложений, возможно, находящихся на разных компьютерах. При обращении к процедуре или функции в рамках одного приложения на одном компьютере передача данных выполняется через стек (см. раздел об организации дисплея памяти процедуры или функции в главе 5) или через общие области памяти (глобальные переменные). В случае удаленного вызова передача параметров процедуре превращается в задачу синхронизации двухприложений и осуществления путем передачи запроса по сети. Соответственно, получение результата также должно использовать сетевые механизмы для взаимодействия двух приложений.

Также надо отметить, что одни и те же данные могут иметь разное представление в компьютерах с разной архитектурой. Поэтому еще одной задачей RPC является автоматическое обеспечение преобразования форматов данных при взаимодействии приложений, выполняющихся на разнородных компьютерах.

Удаленный вызов процедур включает в себя следующие шаги [45, 61]:

- клиент осуществляет локальный вызов процедуры, называемой «заглушкой» (stub);
- процедура-заглушка принимает аргументы, преобразует их в стандартную форму и формирует сетевой запрос к серверу, упаковка аргументов и создание сетевого запроса называется «сборкой» (marshalling);
- сетевой запрос пересылается на удаленный компьютер, где соответствующий модуль RPC должен ожидать такой запрос;
- модуль RPC (который, как правило, входит в состав ОС) при получении запроса извлекает параметры вызова (unmarshalling), определяет тип сервера, которому предназначается вызов;
- модуль RPC на удаленном компьютере обращается к серверу, передает ему параметры вызова и ожидает получения результата;
- полученный результат преобразуется в стандартную форму , формируется сетевой запрос (marshalling) и передается процедуре-заглушке;

- процедура-заглушка извлекает результат вызова из полученного сетевого запроса (unmarshalling) и возвращает его клиенту, выполнение вызова RPC закончено.

В общем виде схема выполнения вызова RPC представлена на рис. 6.9.

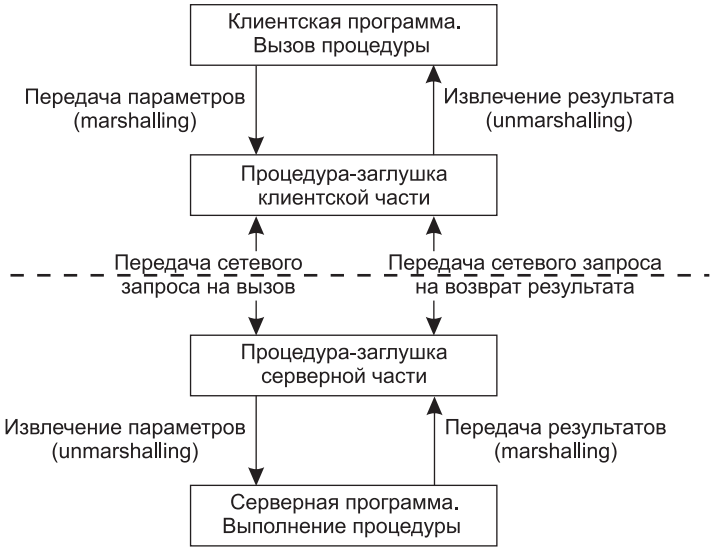

Рис. 6.9. Выполнение вызова удаленной процедуры по технологии RPC

Столь сложный механизм взаимодействия определяет тот факт , что реализация механизма обмена данными между клиентской частью программного обеспечения и сервером приложений не может быть полностью возложена на разработчика прикладной программы. Для этой цели должны использоваться функции ОС, а для организации обмена данными между приложениями, выполняющимися на компьютерах с различной архитектурой, должны быть разработаны соответствующие стандарты.

Использование RPC накладывает определенные ограничения на тип связи между приложениями. Дело в том, что в RPC применяется синхронный механизм взаимодействия: запрашивающее приложение выдает запрос и ждет ответа. На время ожидания приложение оказывается заблокированным. В связи с этим развертывать основанные на RPC приложения представляется целесообразным в локальных сетях, где время ответа обычно не очень велико. Кроме того, RPC является механизмом взаимодействия, ориентированным на соединение. В связи с этим RPC не может быть применена в глобальных сетях, так как вероятность недоступности приложения в этом случае достаточно велика.

Технологии MOM (Message Oriented Middleware) и ORB (Object Request Broker)

С развитием программного обеспечения, построенного на основе трехуровневой архитектуры, на рынке стали появляться новые технологии организации взаимодействия клиентской части с сервером приложений, отличные от RPC. Следующим шагом в разработке промежуточного программного обеспечения для взаимодействия

между активными приложениями стали системы передачи сообщений. Принцип их построения достаточно прост, но тем не менее они предоставляют огромные возможности по связыванию программ.

В основе систем передачи сообщений лежит технология очередей сообщений: приложения обмениваются информацией не непосредственно друг с другом, а используя специальные буферы (очереди). В случае необходимости обмена данными программа пересылает их в принадлежащую ей очередь и продолжает функционирование. Доставку сообщения по назначению и его хранение обеспечивает специальное программное обеспечение — MOM (Message Oriented Middleware, ориентированное на сообщения промежуточное программное обеспечение). При этом MOM, как правило, может работать на разных программно-аппаратных платформах и с использованием различных сетевых протоколов. Муль типлатформность достигается за счет минимизации функций, выполняемых клиентской частью, а поддержка различных протоколов — за счет использования внутреннего протокола обмена информацией. Разработчику же предоставляется несложный и высокоуровневый API для работы с очередями сообщений.

Наличие очередей сообщений гарантирует доставку информации: в случае сбоев или отказов в сети, а также при отказе серверов будет обеспечено либо сохранение сообщения до восстановления соединения, либо его повторная передача или же будет произведен поиск нового пути в обход отказавшего участка.

Системы, построенные на базе MOM, похожи на системы электронной почты, но отличаются от последних тем, что обеспечивают взаимодействие между приложениями, а не между людьми. Соответственно, MOM предназначена для передачи структурированной информации преимущественно технологического типа (то есть порождаемой без участия пользователя), в то время как системы электронной почты осуществляют в большинстве случаев передачу неструктурированной информации.

Технология брокеров запросов к объектам (ORB, Object Request Broker) является наряду с MOM наиболее бурно развивающимся типом программного обеспечения для серверов приложений. ORB управляют обменом сообщениями в сети. Подобно «человеческому» брокеру, ORB выполняет функции интеллектуального посредника, то есть принимает запросы от клиента (клиентского приложения), осуществляет поиск и активизацию удаленных объектов (серверов приложений), которые принципиально могут ответить на запрос, и передает ответ объектам запрашивающего приложения.

ORB — технология объектно-ориентированного подхода, базирующаяся на спецификациях CORBA консорциума OMG (о спецификациях CORBA более подробно сказано в следующем разделе). CORBA включает 13 пунктов (служб). Основные службы CORBA:

- служба именования, присваивает объектам уникальные имена, в результате чего пользователь может искать объект в сети;
- служба обработки транзакций, осуществляет управление транзакциями из приложений или из ОС;
- служба событий, обеспечивает асинхронное распространение и обработку сообщений о событиях;
- служба обеспечения безопасности и поддержки целостности данных.

При применении ORB (в отличие от RPC) клиенту нет необходимости хранить сведения о расположении серверных объектов, ему достаточно знать расположение в сети программы-посредника ORB. Поэтому доступ пользователя к различным объектам (программам, данным, принтерам и т. п.) существенно упрощен. Посредник должен определять, в каком месте сети находится запрашиваемый ресурс, направлять запрос пользователя в соответствующее место, а после выполнения запроса возвращать результаты пользователю.

Применение ORB может несколько увеличить нагрузку на сеть, однако имеет и ряд преимуществ: обеспечивается взаимодействие разных платформ, не требуется дублирования прикладных программ во многих узлах, упрощается программирование сетевых приложений и поддержка мультимедиа. В CORBA создан протокол IIOP (Internet Inter-ORB Protocol), который обеспечивает взаимодействие между брокерами разных производителей.

Взаимодействие клиента с сервером в ORB, как и в RPC происходит через создаваемые для такого взаимодействия специальные «заглушки» (stub), как это показано выше на рис. 6.9. В случае ORB из клиентской процедуры-заглушки происходит обращение к ORB, который, в свою очередь, создает соединение.

ORB, как и RPC и MOM, скрывает от пользователя процесс доступа к удаленным объектам. Запрашивающий клиент должен знать имя активизируемого объекта (либо посредника, либо сервера приложений) и передать ему некоторые параметры. Как правило, это информация об интерфейсе вызываемого объекта — своего рода API для ORB. Интерес к ORB подогревается тем, что это программное обеспечение для серверов приложений поддерживает объектную модель, ставшую де-факто стандартом при разработке больших информационных систем.

Организация взаимодействия приложений в многоуровневой архитектуре

Необходимая поддержка многоуровневой архитектуры

Сложная организация взаимодействия клиентской части с сервером приложений, с одной стороны, и большой интерес к разработке приложений длятрехуровневой архитектуры, с другой стороны, привели к появлению и развитию средств разработки, ориентированных на создание приложений для трехуровневой или многоуровневой архитектуры. Основным отличием разработки приложений для многоуровневой архитектуры от разработки приложений для архитектуры «клиент—сервер» является тот факт, что для многоуровневой архитектуры разработчик должен создать несколько частей приложения (как минимум две: клиентскую часть и сервер приложений), в то время как для архитектуры «клиент—сервер» создается, как правило, только одна клиентская часть. При этом должно обеспечиваться взаимодействие создаваемых частей приложений на основе выбранной технологии.

Поэтому разработка приложений для многоуровневой архитектуры является более сложной задачей, чем разработка приложений для архитектуры «клиент—сервер». Для эффективного создания таких приложений должны выполняться следующие условия:

- целевая вычислительная система должна обеспечивать поддержку выбранной технологии взаимодействия частей приложения, а если для разных частей приложения будут использоваться различные целевые вычислительные системы, то все они должны поддерживать выбранную технологию;
- исходный язык программирования и система программирования должны предоставлять инструменты, ориентированные на создание приложений в трехуровневой архитектуре.

Первое условие является обязательным, второе — настоятельно рекомендуемым. Конечно, зная функции целевой ОС, обеспечивающие взаимодействие различных частей приложения, можно реализовать сервер приложений и клиентскую часть на основе системы программирования, не предоставляющей средств поддержки многоуровневой архитектуры. Однако трудоемкость создания приложения в этом случае будет значительно выше, а переносимость его частей — существенно хуже.

Развитие многоуровневой архитектуры привело к появлению в составе современных ОС соответствующих средств, которые обеспечивают выполнение обращений клиентской части к серверу приложений. Это обеспечило выполнение необходимых условий для создания приложений в многоуровневой архитектуре и определило направления развития средств их разработки и поддержки в системах программирования.

Современные системы программирования обеспечивают поддержку разработки приложений в трехуровневой и многоуровневой архитектуре. Средства поддержки тех или иных стандартов, необходимых для работы клиентской части и сервера приложений, сейчас присутствуют в составе многих систем программирования [16, 45, 61, 81, 82]. Принципы их использования и распространения аналогичны принципам, применяемым для средств поддержки архитектуры типа «клиент—сервер». Следует заметить, что существуют системы программирования, ориентированные на разработку либо клиентской части, либо сервера приложений; и в то же время многие системы программирования стремятся предоставить разработчикам средства для создания обеих частей приложения. Сервер баз данных в трехуровневой архитектуре по-прежнему чаще всего является продуктом стороннего разработчика[1] (как правило, производителя СУБД).

В настоящее время на рынке конкурируют стандарт CORBA и технология COM/DCOM корпорации Microsoft. Семейство стандартов COM/DCOM создано фирмой Microsoft для ОС семейства Microsoft Windows [45, 61, 71, 76, 80, 81], а семейство стандартов CORBA (Common Object Request Broker Architecture) поддерживается специальным консорциумом, состоящим из широкого круга производителей ОС и систем программирования для большого спектра различных вычислительных архитектур [71]. Этот последний стандарт (CORBA) претендует на то, чтобы обеспечивать переносимость серверов приложений и клиентских частей, соответствующих ему.

Детальное рассмотрение такого рода стандартов и принципов их организации не входит в рамки данного учебника. Но далее автор постарается дать обзор тех новых средств в языках программирования и системах программирования, которые появи-

[1] Конечно, сервер БД — это программа, и как всякая программа она создается с использованием соответствующего компилятора; но в этом случае речь уже не идет о создании части трехуровневого приложения — здесь сервер БД выступает как отдельное законченное приложение.

лись благодаря созданию и развитию двух основных технологий разработки приложений для многоуровневой архитектуры.

Модель и технологии COM/DCOM

Аббревиатура COM расшифровывается как «Component Object Model» (модель компонентных объектов). Эта объектная модель была создана компанией Microsoft в рамках ОС типа Windows как дальнейшее развитие средств взаимодействия приложений в этих ОС. При этом были объявлены следующие основные цели COM:

- поддержка средств определения спецификации для создания набора стандартных объектов, не ориентированных на специфический язык программирования;
- поддержка средств, разрешающих реализовывать объекты, которые могут вызываться различными процессами, даже если эти процессы выполняются на разных компьютерах.

Вторая цель привела к появлению модели DCOM(Distributed COM — распределенная COM), которая является неразрывной составляющей COM.

В модели COM все приложения, которые поддерживают данную модель и на которых выполняются распределенные вычисления, являются серверами (COM-серверами). COM-серверы могут быть внутренними (реализованными в рамках одного процесса), локальными (реализованными на том же компьютере) и удаленными (реализованными на другом компьютере в сети). Физически они могут быть реализованы в виде исполняемых файлов или динамически загружаемых библиотек. Местоположение и реализация сервера являются прозрачными для клиента. В принципе, клиент при соединении может указать, какой сервер (внутренний, локальный или удаленный) ему нужен, но обычно эта возможность не используется. Дальнейшая работа с сервером не зависит от его типа, клиент может даже не знать, с сервером какого типа он работает.

COM/DCOM — это объектная модель, вся работа строится на объектах. Каждый COM-объект имеет тип, называемый коклассом (английский термин «CoClass», префикс «Co» указывает на принадлежность к технологии COM). COM-сервер может создавать свой COM-объект для обработки вызовов от каждого клиента или использовать один COM-объект для обработки вызовов всех клиентов. Кроме того, один сервер может содержать несколько разных коклассов. Именно кокласс, а не сервер, является центральным объектом для COM-клиента. Если клиент работает с разными коклассами, то он не может узнать, реализуются ли они одним сервером или разными. Каждый кокласс имеет свое имя и глобальный идентификатор (GUID — Globally Unique Identifier, глобально уникальный идентификатор).

ПРИМЕЧАНИЕ

Не следует путать понятие кокласса в COM с понятиями класса или объекта в объектно-ориентированном программировании. В объектно-ориентированных языках программирования кокласс действительно удобнее всего реализовывать как класс (объект). Но COM-сервер может быть написан и на таком не объектно-ориентированном языке, как, например, C или Visual Basic [68].

Другим не менее важным понятием технологии COM/DCOM является интерфейс. Интерфейс — это список функций. С точки зрения двоичного кода интер-

фейс представляет собой таблицу , содержащую указатели на функции. Количество элементов в этой таблице соответствует количеству функций в интерфейсе. По своей реализации такая таблица функций подобна таблице RTTI, используемой при организации виртуальных функций в объектно-ориентированных языках программирования (подробнее о таблицах RTTI сказано в предыдущей главе). COM-сервер реализует функции, создает таблицы с указателями на них, а указатель на каждую таблицу функций интерфейса может быть получен клиентом. Именно через эти указатели клиент получает доступ к функциям COM-сервера. Количество интерфейсов, которые может экспортировать один кокласс, не ограничивается.

Каждый интерфейс, как и кокласс, имеет имя и уникальный GUID. Имя интерфейса должно быть идентификатором, корректным с точки зрения языка, на котором разрабатывается клиент или сервер. Т ак как различные языки программирования имеют разные требования к идентификаторам, в имени интерфейса рекомендуется использовать только английские буквы и символ подчеркивания («_»). Кроме того, многие языки программирования нечувствительны к регистру символов, поэтому нежелательно давать разным интерфейсам имена, отличающиеся лишь регистром символов. И наконец, названия интерфейсов принято начинать с буквы «I».

Однажды созданный и опубликованный интерфейс не должен меняться, потому что иначе клиент, получивший указатель на данный интерфейс, может попытаться вызвать функцию, которой он не содержит , или передать ей не те параметры. Если интерфейс устарел, то вместо его модификации нужно создать новый интерфейс, оставив старый без изменения. При создании интерфейсов их можно наследовать от уже имеющихся. Интерфейс-наследник может добавлять новые функции, но перекрытие старых функций не имеет смысла, потому что интерфейс не содержит реализации функций.

Имена интерфейсов удобны при использовании интерфейса для разработчиков, однако они не годятся, когда клиент должен точно указать, какой именно интерфейс объекта ему нужен. Для уникальной идентификации клиентом каждого интерфейса COM-сервера используется GUID. GUID — это 16-байтовая (128-битная) величина, обычно автоматически генерируемая соответствующей утилитой. Каждый может запустить такую утилиту на любом компьютере и гарантированно (со всех практических точек зрения) получит GUID, который будет отличаться от всех других GUID, которые когда-либо были созданы или будут созданы. У никальность GUID обеспечивается его привязкой к моменту времени и к уникальному идентификатору компьютера в сети (при отсутствии сети используются дополнительные методы уникальной идентификации компьютера).

В модели COM/DCOM сервер приложений должен представлять собой множество коклассов и связанных с ними интерфейсов, а клиентская часть приложения для выполнения необходимых ей функций на сервере приложений должна обращаться к соответствующим коклассам и интерфейсам.

В настоящее время модель COM/DCOM и основанные на ее технологиях многоуровневые приложения получили широкое распространение, модель постоянно развивается и совершенствуется производителем (компанией Microsoft), находит все более мощную поддержку в новых ОС типа Windows. Главным недостатком данной

модели является ее ориентация на одного производителя и один тип ОС (Windows). Более подробно о модели COM/DCOM и связанных с нею технологиях можно узнать в соответствующей литературе [25, 26, 27, 28, 37, 48, 51, 62, 76, 77, 78, 81].

Модель и семейство стандартов CORBA

Семейство стандартов CORBA (Common Object Request Broker Architecture — общая архитектура брокера объектных запросов) было разработано международным консорциумом OMG (Object Management Group), который включает более 500 компаний, связанных с производством компьютерной аппаратуры и программного обеспечения (в частности, членами OMG являются компании IBM, Hewlett Packard, Intel, Microsoft, Borland International, Oracle и др.).

Идея CORBA заключается в следующем: во-первых, в каждый объект, который должен быть включен в интегрированную систему, добавляется специальный программный код, обеспечивающий принципиальную возможность взаимодействия объектов. Этот код генерируется автоматически за счет использования определенного OMG языка определения интерфейсов IDL (Interface Definition Language). В исходный текст программы включаются спецификации интерфейса на языке IDL. Затем этот текст должен быть обработан специальным предварительным компилятором (а точнее, транслятором), который генерирует дополнительный программный код. Во-вторых, для реального взаимодействия должным образом настроенных объектов предполагается наличие специального программного обеспечения, называемого в документах OMG брокером объектных заявок (ORB — Object Request Broker). Брокер объектных заявок должен существовать и на стороне вызывающего объекта, и на стороне вызываемого объекта.

При обработке заявки ORB ищет соответствующий код, пересылает параметры обращения и передает управление реализации вызываемого объекта. Вызываемый объект принимает заявку через генерируемый компилятором IDL промежуточный программный код — скелетон, а также может обращаться к объектному адаптеру и ORB. После завершения обработки заявки результаты обращения возвращаются клиенту, который продолжает свое выполнение.

Брокеры объектных заявок могут быть по-разному реализованы и могут поддерживать различные объектные механизмы. В структуре ORB выделяется ядро, обеспечивающее внутреннее представление объектов и передачу заявок, и набор настраиваемых компонентов, интерфейсы которых маскируют различия в реализации ORB. Объекты-клиенты имеют возможность без изменения своего кода работать в среде любого ORB, который поддерживает отображение IDL в соответствующий язык программирования. Реализации вызываемых объектов также могут работать в среде разных ORB, если брокер поддерживает требуемое языковое отображение и снабжен требуемым объектным адаптером.

Языковое отображение включает определение характерных для языка программирования типов данных и интерфейсов доступа к объектам при помощи ORB. В отображении определяется структура интерфейса процедуры-заглушки (stub) клиента, интерфейса динамического вызова, скелетона реализации объекта, объектных адаптеров, а также интерфейсы прямого взаимодействия с ORB. В настоящий момент стандартизовано отображение языка IDL на основные языки программирования, для которых существуют общепринятые стандарты, — Ada, C, C++, Cobol, Java

и Smalltalk. Существуют также отображения на Object Pascal, Perl, Python и некоторые другие языки программирования.

Предоставляемые объектными службами операции доступны через интерфейсы, определенные на языке IDL или его расширении, если оно совместимо с базовой объектной моделью OMG. Примерами специфицированных объектных служб являются служба именования объектов, служба долговременного хранения объектов, служба внешнего представления объектов и т. д.

Понятие «объекта» в CORBA принципиально отличается от аналогичного понятия в COM. Объект CORBA не является переменной языка программирования и в общем случае время его существования не связано со временем работы серверных или клиентских приложений. Объект CORBA не занимает никаких ресурсов компьютера — оперативной памяти, сетевых ресурсов и т .п. Эти ресурсы занимает только так называемый «сервант» (servant), который является «инкарнацией» одного или нескольких объектов CORBA. Именно сервант является переменной языка программирования. Пока не существует сервант, сопоставленный с конкретным объектом CORBA, этот объект не может обслуживать вызовы клиентов, но тем не менее он существует . Результатом создания объекта является так называемая «объектная ссылка» CORBA. Объектная ссылка сопоставлена с этим, и только с этим объектом, и это сопоставление остается корректным в течение всего срока существования объекта CORBA. Объектная ссылка CORBA правильно интерпретируется ORB от любого производителя программного обеспечения. После уничтожения объекта CORBA все объектные ссылки на него навсегда теряют смысл. С помощью объектной ссылки клиент вызывает методы объекта, при этом «инкарнациями» этого объекта могут быть различные серванты (но не более одного одновременно), которые физически могут находиться даже на различных компьютерах.

Семейство стандартов CORBA в настоящее время по распространенности на рынке соперничает с технологиями COM/DCOM, но ему практически нет альтернативы в том случае, когда требуется обеспечить переносимость приложений на вычислительные системы под управлением ОС, отличной от W indows. Также следует отметить, что CORBA, в отличие от COM, является концепцией, а не ее реализацией. T ехнологии COM/DCOM подразумевают набор конкретных средств — элементов ОС, библиотек, утилит и т . п., являющихся составной частью ОС типа Microsoft Windows. Под термином «CORBA» понимается именно концепция, сформулированная на уровне специального языка описаний — IDL. Реализации же этой концепции могут сильно отличаться друг от друга по различным критериям, наиболее важным в том или другом случае. Например, Inprise/Corel VisiBroker и Application Server , BEA W ebLogic, Oracle Application Server и «картриджи» Oracle, IBM BOSS — все эти продукты используют те или иные возможности CORBA.

Язык IDL. Интерфейсы и библиотеки классов

Описанные выше технологии и стандарты имеют существенные различия, но при этом между ними есть и те общие черты, которые не могли не найти отражение в системах программирования, ориентированных на разработку приложений для трехуровневой и многоуровневой архитектуры.

В качестве таких общих черт можно выделить следующие особенности:

- наличие специального языка описания интерфейсов объектов (языка IDL), предложения которого должны транслироваться на предложения входного языка системы программирования;
- появление нового понятия «интерфейс», которое соответствует описанию объекта, реализация которого может находиться за пределами данной результирующей программы.

ВНИМАНИЕ

На самом деле язык IDL не является универсальным для всех технологий описания интерфейсов. В настоящее время существует три различных языка описаний с одним и тем же названием — OSF IDL (исходный прообраз языка описания интерфейсов), Microsoft IDL (используемый в технологии COM/DCOM) и OMG IDL (используемый в CORBA). Эти языки имеют много общих черт (все они происходят от OSF IDL), но и ряд различий.

Сервер приложений и клиент должны иметь заранее согласованный способ описания взаимодействия, то есть способ определения методов, которые могут быть вызваны клиентом на сервере, а также параметров этих методов. Таким способом является описание на языке IDL. Язык IDL является единым средством описания спецификаций взаимодействия между сервером и клиентом.

Использование языка IDL в системах программирования предполагает наличие в их составе трансляторов (речь идет именно о трансляторах, не о компиляторах), которые могли бы осуществлять перевод предложений с языка IDL на входной язык системы программирования и наоборот — переводить описания объектов с входного языка на язык IDL. Выполнив подобным образом перевод описания с языка IDL на входной язык программирования, разработчик сразу получает исходный текст для реализации соответствующего объекта. Причем на основе одного и того же текста IDL можно получить описание как для вызываемого объекта (серверной части), так и для вызывающего объекта (клиентской части) в зависимости от нужд разработчика. Поскольку оба описания создаются на основе одного и того же исходного текста, они гарантированно будут соответствовать друг другу. После этого разработчику остается только «наполнить» описания содержательным текстом на входном языке, который будет обеспечивать реализацию функций объекта.

С другой стороны, создав новый объект, доступный для других частей приложения, на исходном языке программирования, разработчик должен иметь возможность сформировать его описание, не зависящее от используемого языка. В дальнейшем это описание может быть использовано при разработке других частей приложения, взаимодействующих с созданным объектом.

Современные системы программирования, ориентированные на разработку приложений для многоуровневой архитектуры, предоставляют разработчикам средства трансляции текстов языка IDL. В зависимости от того, какую технологию взаимодействия частей приложения поддерживает система программирования, транслятор IDL в ее составе должен обрабатывать соответствующее подмножество этого языка (или несколько его подмножеств, если поддерживаются разные технологии). При этом тексты на языке IDL являются частью исходной программы, так же как и тек-

сты на входном языке. Поскольку трансляция с разных подмножеств языка IDL возможна не на все языки программирования, разработка приложения в трехуровневой архитектуре накладывает ограничения не только на выбор системы программирования, но и на выбор входного языка программирования.

Не меньший интерес с точки зрения развития систем программирования и языков программирования представляет понятие «интерфейс», которое появилось в технологиях, реализующих взаимодействие частей многоуровневого приложения. И хотя у двух представленных выше технологий для этого понятия имеются существенные различия, тем не менее оно имеет и ряд существенных общих черт.

С точки зрения входного языка «интерфейс» — это определение определенного объекта, его свойств и функций (методов). При этом, в отличие от класса (или объекта) в объектно-ориентированных языках программирования, интерфейс никак не связан с реализацией объекта. Его взаимосвязь происходит с объектом во время выполнения результирующей программы, а способ взаимосвязи зависит от выбранной технологии, причем в обеспечении этой взаимосвязи может принимать участие несколько выполняющихся программ, которые могут находиться на разных компьютерах. Один и тот же интерфейс могут реализовывать разные объекты, и то, какой конкретно объект будет связан с данным интерфейсом, определяется только во время выполнения приложения. Поэтому интерфейс не может иметь полей, конструкторов и деструкторов.

С точки зрения системы программирования интерфейс представляет собой универсальное средство обращения к объектам и классам, которые могут находиться в пределах адресного пространства результирующей программы или вне его. Вызов методов интерфейса, по сути, схож с вызовом виртуальных функций через информацию, хранимую в таблицах RTTI (назначение таблиц RTTI и методы их организации рассмотрены в предыдущей главе учебника). Но, в отличие от таблиц RTTI, реализация которых никак не стандартизована и зависит от используемой системы программирования, принципы использования интерфейсов регламентируются соответствующей технологией. Поэтому объекты, реализующие интерфейсы и созданные с помощью некоторой системы программирования, могут быть вызваны через те же интерфейсы результирующей программой, построенной в совершенно другой системе программирования (что невозможно осуществить при использовании обычных библиотек классов, основанных на таблицах RTTI). Важно только, чтобы обе системы программирования поддерживали разработку приложений многоуровневой архитектуры на основе используемой технологии.

Интерфейс — это новое понятие, появившееся в связи с развитием трехуровневой и многоуровневой архитектуры. В исходном языке программирования он, в принципе, может быть реализован и без введения новых понятий и типов в синтаксис и семантику языка. Но подход взаимодействия с внешними объектами, основанный на понятии интерфейса, можно признать настолько удачным, что во многих современных языках программирования появились синтаксические конструкции, специально предназначенные для реализации интерфейсов (в первую очередь это касается технологии COM/DCOM, которая подразумевает вполне не только идею, но и ее реализацию).

Технологии и средства программирования (включающие в себя и языки, и системы программирования), связанные с созданием приложений для трехуровневой и мно-

гоуровневой архитектуры, продолжают развиваться и совершенствоваться. Поэтому в этом направлении можно ожидать появления новых решений.

Возможности трехуровневой и многоуровневой архитектуры

По сравнению с архитектурой «клиент—сервер» трехуровневая архитектура предоставляет разработчику программного обеспечения следующие преимущества:

- функции обработки данных возложены на сервер приложений, а на клиентскую часть приходятся только функции отображения данных и организации интерфейса пользователя, что существенно снижает требования к ресурсам вычислительной системы клиентской части программного обеспечения;
- при необходимости изменить или дополнить логику обработки данных достаточно модифицировать только программное обеспечение на сервере приложений, а обновление клиентских приложений на всех рабочих местах необязательно;
- если необходимо изменить только внешний вид интерфейса пользователя, то достаточно только изменить простую по своей структуре клиентскую часть на тех рабочих местах, где это необходимо;
- для подключения к серверу данных (который, как правило, поставляется сторонним разработчиком) можно ограничиться одной лицензией[1].

Главным недостатком трехуровневой архитектуры следует признать довольно сложный и трудоемкий процесс организации взаимодействия между клиентами и сервером приложений. Именно по этой причине далеко не все современные программные системы, построенные на основе архитектуры «клиент—сервер» быстро переходят на трехуровневую архитектуру, поскольку выделить в составе программного комплекса сервер приложений, описать его функции и организовать взаимодействие клиентских частей с ним — это сложный процесс, требующий усилий от разработчиков.

Очень часто, если функциональность программного обеспечения, построенного на основе архитектуры «клиент—сервер», достаточно стабильна, а обработка данных не содержит сложных ресурсоемких операций, переход на трехуровневую архитектуру не всегда оправдан. В ряде случаев сочетание архитектуры «клиент—сервер», с возможностями терминального доступа (как это было показано выше на рис. 6.7) позволяет приложению достичь того же эффекта, что и использование многоуровневой архитектуры. И хотя в этом случае, как было сказано ранее, неэффективно используются вычислительные ресурсы терминальных серверов, но зачастую это бывает оправдано, так как позволяет избежать трудоемкого процесса перевода приложения в трехуровневую архитектуру.

Этому способствует также современная ситуация с развитием программно-аппаратных средств: замена программно-аппаратного обеспечения клиентских компьютеров зачастую обходится намного дешевле, чем переработка программного

[1] Это не всегда справедливо, так как многие фирмы-производители СУБД учли это обстоятельство и требуют специальных лицензий при использовании их СУБД в трехуровневой архитектуре либо же рассчитывают стоимость лицензии, исходя не из количества подключений, а из параметров архитектуры сервера данных.

кода приложения. И при этом средний по характеристикам компьютер, доступный на рынке, вполне способен выполнять функции терминального сервера.

Другим фактором, сдерживающим развитие и распространение приложений, построенных на основе трехуровневой и многоуровневой архитектуры, является отсутствие единого для всех стандарта, каким в свое время стал стандарт языка SQL для архитектуры «клиент—сервер». Технологии COM/DCOM активно развиваются и продвигаются компанией Microsoft, но они ориентированы исключительно на ОС производства этой компании. В то же время концепция CORBA является стандартом, совместимым со всеми современными ОС, но по реализации она зачастую проигрывает ОС типа Windows, которые занимают существенную долю рынка вычислительных систем[1].

СОВЕТ

Автор рекомендует хорошо подумать, прежде чем приступать к переводу на многоуровневую архитектуру уже существующего приложения в архитектуре «клиент—сервер». Но если уже принято такое решение, то надо второй раз подумать о выборе соответствующей технологии. Оба этих вопроса не имеют однозначного ответа, и для принятия решений требуется учесть все факторы, связанные с разрабатываемым приложением: тип целевой вычислительной системы, необходимость переносимости приложения и его составных частей, требования к масштабируемости приложения и др.

Более подробно о разработке программного обеспечения на основе трехуровневой и многоуровневой архитектуры можно узнать в [25, 27, 28, 45, 51, 61, 70, 71, 82].

Разработка программного обеспечения для сети Интернет

Особенности разработки программного обеспечения для сети Интернет

Интернет — это Всемирная «сеть сетей». Он объединяет в себе вычислительные системы и локальные сети, построенные на базе различных аппаратно-программных архитектур. При осуществлении взаимодействия по сети двух компьютеров один из них выступает в качестве источника данных (интернет-сервера), а другой — приемника данных (интернет-клиента). Сервер подготавливает данные, а клиент принимает их и каким-то образом обрабатывает. Нередко в качестве данных выступают тексты программ, которые подготавливает сервер, а исполнять должен клиент.

В таких условиях определяющим становится требование унифицированного исполнения кода программы вне зависимости от архитектуры вычислительной системы. Компиляция и создание объектного кода в условиях Всемирной сети становятся бессмысленными, потому что заранее не известно, на какой целевой вычислительной системе потребуется исполнять полученный в результате компиляции код.

[1] Такая ситуация была на рынке технологий разработки приложений в трехуровневой архитектуре при подготовке первого издания данного учебника, и пока что она осталась без изменений на момент подготовки его второго издания.

Поэтому основной особенностью программирования в сети Интернет является использование в качестве основного средства программирования интерпретируемых языков. При интерпретации исполняется не объектный код, а сам исходный код программы, и уже непосредственно интерпретатор на стороне клиента отвечает за то, чтобы этот исходный код был исполнен всегда одним и тем же образом вне зависимости от архитектуры вычислительной системы. Тогда сервер готовит код программы всегда одним и тем же образом вне зависимости от типа клиента.

В такой ситуации нагрузка на интернет-клиента может возрасти. Но задачу можно несколько упростить: интернет-сервер может готовить не высокоуровневый код исходной программы, а некий унифицированный промежуточный код низкого уровня, предназначенный для исполнения на стороне клиента. Т огда в обмене данными участвуют еще две дополнительные программы: компилятор (точнее — транслятор) на стороне сервера, транслирующий исходный код программы на некотором языке высокого уровня в промежуточный низкоуровневый код, и интерпретатор на стороне клиента, отвечающий за исполнение промежуточного кода вне зависимости от архитектуры вычислительной системы клиента.

Для реализации такого рода схем существует много технических и языковых средств. Существует также значительное количество языков программирования, ориентированных на использование в Глобальной сети [51, 67, 73, 74, 75]. Интернет-программирование — это отдельная и довольно интересная область разработки программ, но в целом вопросы, затрагиваемые в ней, лежат за пределами темы данного учебника. Далее дается только очень краткий обзор принципов, положенных в основу тех или иных языков программирования с Глобальной сети.

Разработка статических веб-страниц

Язык HTML. Программирование статических веб-страниц

Язык HTML (HyperText Markup Language, язык разметки гипертекста) во многом определил развитие и широкое распространение сети Интернет по всему миру . Сам по себе язык достаточно прост, а для овладения им нужны только самые примитивные знания в области программирования.

Язык позволяет описывать структурированный текст (гипертекст), содержащий ссылки и взаимосвязи фрагментов; графические элементы (изображения), которые могут быть связаны как с текстовой информацией, так и между собой; а также простейшие элементы графического интерфейса пользователя (кнопки, списки, поля редактирования). На основе описания, построенного в текстовом виде на HTML, эти элементы могут располагаться на экране, им могут присваиваться различные атрибуты, определяющие используемые ресурсы интерфейса пользователя (такие как цвет, шрифты, размер и т. п.). В результате получается графический образ — веб-страница (от «web» — «паутина» — слова, входящего в состав аббревиатуры WWW — World Wide Web — Всемирная паутина). Она в принципе может содержать различные мультимедийные элементы, включая графику, видео и анимацию.

Широкому распространению HTML послужил принцип, на основе которого этот язык стал использоваться в Глобальной сети. Суть его достаточно проста: интернет-сервер создает текст на языке HTML и передает его в виде текстового файла

на клиентскую сторону сети по специальному протоколу обмена данными HTTP (HyperText Transfer Protocol, протокол передачи гипертекста). Клиент, получая исходный текст на языке HTML, интерпретирует его и в соответствии с резуль татом интерпретации строит соответствующие интерфейсные формы и изображения на экране клиентской ЭВМ.

Тогда в общем виде работу сервера и клиента с простейшей веб-страницей можно представить так, как показано на рис. 6.10. Клиент обращается к серверу , передавая ему запрос, содержащий указание на веб-страницу, которую он хочет получить. В ответ на это сервер передает клиенту текст страницы на языке HTML. Клиент интерпретирует текст и отображает страницу на экране компьютера. Для однозначной идентификации всех веб-страниц, которые могут существовать в сети, была придумана система доменных имен и универсальный указатель ресурса — URL (Universal Resource Locator) [51, 67]. Тогда клиент, запрашивая у сервера веб-страницу, должен передать ему URL этой страницы.

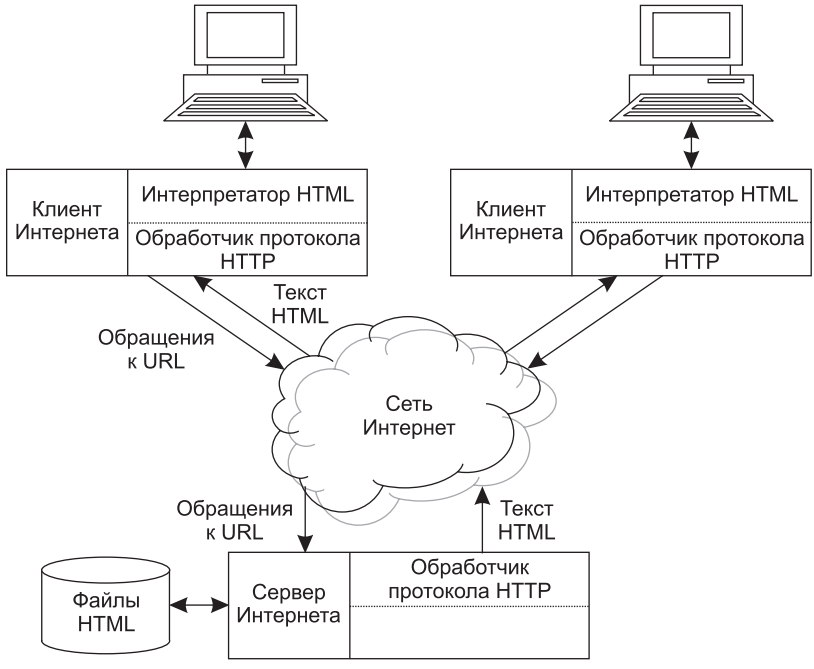

Рис. 6.10. Взаимодействие сервера и клиента в сети Интернет для отображения простейшей веб-страницы

Грамматика языка HTML проста, а потому не составляет сложности построить соответствующий интерпретатор. Т акими интерпретаторами явились программы-навигаторы в сети Интернет (браузеры, browser), которые, по сути, должны были содержать как минимум две составляющие: клиентскую часть для обмена данными по протоколу HTTP и интерпретатор языка HTML. Программы-навигаторы доступны практически всем, так как либо являются свободно распространяемыми программами, либо входят в состав ОС (в этом смысле показателен пример компании Microsoft, которая первой оценила преимущества включения браузера в состав ОС).

Некоторое время на рынке существовало огромное количество такого рода программ, а несколько компаний—производителей таких программ боролись за свою долю рынка («война браузеров»). В настоящее время на вычислительных системах, построенных на основе ОС типа Windows, преобладает браузер производства Microsoft, входящий в состав этой ОС, — Internet Explorer . Для других типов ОС существует несколько распространенных типов браузеров (например, Mozilla, Opera и др.). А вот браузер Netscape Navigator (производства компании Netscape), который одно время завоевал значительную долю рынка, сейчас почти не используется.

Но все существующие браузеры должны соответствовать стандартам протокола HTTP и языка HTML (к сожалению, по мере развития Интернета было создано несколько версий этих стандартов, что иногда затрудняет работу в сети, особенно для браузеров устаревших версий). Описание языка HTML можно найти на многих сайтах в Глобальной сети, а также в многочисленной литературе по Интернет-программированию (например, в [51, 52, 67]).

Гораздо разнообразнее программное обеспечение серверной части. Это вызвано тем, что в протоколе HTTP нигде строго не специфицирован источник HTML-текста. Самым простым источником данных на сервере может служить множество файлов, содержащих текст на языке HTML. В начале появления и развития сети WWW большинство веб-страниц было организовано именно таким образом. В этом случае пользователи, присоединявшиеся к этим серверам, видели на экране своих компьютеров неизменную (статическую) картинку — отсюда название «статические веб-страницы».

Недостатки статических веб-страниц

Язык HTML прост и поэтому удобен. Однако отсюда же проистекают и основные его недостатки.

Во-первых, он не предоставляет средств динамического изменения содержимого интерфейсных форм и изображений, поэтому основной метод — динамическое изменение самого текста HTML. Во-вторых, данный язык не предоставляет никаких методов поддержки современных архитектур типа «клиент—сервер» или трехуровневой архитектуры. Он не позволяет обмениваться данными ни с серверами БД, ни с серверами приложений как на стороне сервера, где готовятся тексты HTML, так и на стороне клиента, где эти тексты интерпретируются. Наконец, этот язык имеет очень ограниченные средства для реакции на действия пользователя в интерфейсных формах, созданных с его помощью.

В том случае, когда источником данных на сервере служит файл, содержащий текст на языке HTML, пользователь на клиентском компьютере получает у себя на экране картинку в соответствии с содержимым этого файла. Изображение на экране может меняться в соответствии с достаточно скромными возможностями языка HTML, пользователь может перейти по ссылкам, содержащим URL других веб-страниц, к другим страницам этого или другого сервера. Но в этом простейшем случае он не имеет возможности как-то повлиять на содержимое страницы, на ту информацию, которую предоставляет ему сервер.

Простейшая реализация веб-страниц ограничивала их возможности возможностями языка HTML. Для устранения указанных недостатков были предложены средства,

выходящие за рамки HTML и позволяющие расширить возможности как сервера, так и клиента. Некоторые из них рассмотрены ниже.

Самая простая идея расширения возможностей веб-страниц заключалась в том, чтобы отказаться от статического источника текста HTML в виде файла. Сервер может порождать новый HTML-текст всякий раз, когда клиент устанавливает с ним соединение, или даже менять текст по мере работы клиента с сервером. Причем клиент может передавать серверу в тексте URL дополнительную информацию о том, какие данные и в каком виде он хотел бы получить. Тогда и изображение на стороне клиента, зависящее от интерпретируемого текста HTML, будет динамически изменяться по мере изменения текста HTML, порождаемого сервером. Вот в этом направлении и шло развитие основных средств интернет-программирования.

Разработка динамических веб-страниц

Динамические веб-страницы на основе CGI и интерпретаторов

Основная идея динамической генерации веб-страниц заключается в том, что вся HTML-страница или ее часть не хранится в файле на сервере, а порождается непосредственно каждый раз при обращении клиента к серверу. Тогда сервер формирует страницу и сразу же по готовности передает ее клиенту. Таким образом, всякий раз при новом обращении клиент получает новый текст HTML, и не исключено, что тексты могут значительно различаться между собой даже при обращении к одному и тому же серверу.

Вопрос только в том, как обеспечить динамическую генерацию HTML-кода на стороне сервера.

Самое очевидное решение заключается в том, чтобы разработать некоторый исполняемый файл (приложение), который будет динамически строить новый HTML-код. Тогда на вход такого исполняемого файла поступают некоторые параметры (например, данные, которые пользователь ввел вформе или командной строке), а на выходе он должен порождать HTML-код в виде текста. Для этой цели служат специальные приложения, называемые CGI-приложениями.

CGI (Common Gateway Interface, общедоступный шлюзовой интерфейс) — это интерфейс для запуска внешних программ на сервере в ответ на действия клиента, установившего соединение с ним через Глобальную сеть. Пользуясь этим интерфейсом, приложения могут получать информацию от удаленного пользователя, анализировать ее, формировать HTML-код и отсылать его клиенту. CGI-приложения могут получать данные из заполненной формы, построенной с помощью HTML, либо из командной строки описания URL. Строка URL вводится в программе-навигаторе, осуществляющей доступ к серверу со стороны клиента. То, какие CGI-приложения по каким действиям пользователя должны выполняться на сервере, указывается непосредственно в коде HTML-страницы, которую сервер передает клиенту.

В этом случае работу сервера и клиента для отображения динамической веб-страницы можно представить так, как показано на рис. 6.11.

Кроме интерфейса CGI, существуют и другие варианты интерфейсов, позволяющие динамически создавать HTML-код путем запуска на сервере приложений в ответ на действия клиента. Например, можно выделить интерфейс ISAPI (Internet Server

Application Programming Interface, интерфейс прикладных программ интернет-сервера). Отличие ISAPI от CGI заключается в том, что для поддержки CGI создаются отдельные приложения, выполняющиеся в виде самостоятельных программ, а ISAPI поддерживается с помощью библиотек, динамически подключаемых к серверу. ISAPI-библиотеки исполняются непосредственно в адресном пространстве сервера, имеют большие возможности и обеспечивают более высокую производительность сервера, в то время как CGI-приложения исполняются в ОС сервера как отдельные процессы и вынуждены определенным образом организовывать обмен данными с самим сервером (что снижает производительность). Но, с другой стороны, ошибка в библиотеке ISAPI может привести к выходу всего сервера из строя и его длительной неработоспособности. В то же время ошибка в коде CGI-приложения может в худшем случае привести только к аварийному завершению выполнения этого приложения, а сам сервер при этом сохранит работоспособность. Т огда в результате ошибки будет неверно отображена только какая-то одна HTML-страница либо часть страницы, а все остальные части сервера будут продолжать исправно работать.

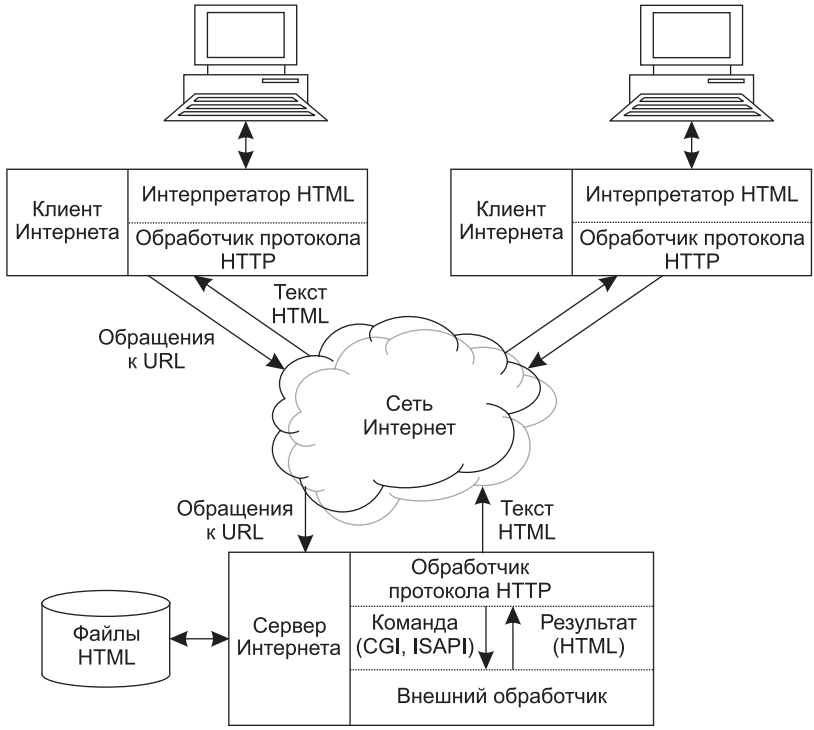

Рис. 6.11. Взаимодействие сервера и клиента в сети Интернет для отображения динамической веб-страницы

Современные системы программирования позволяют создавать приложения и библиотеки, рассчитанные на работу в Глобальной сети, в соответствии со стандартами CGI или ISAPI. При этом создание исходного кода приложения практически ничем не отличается от создания обычной исполняемой программы: компилятор по-прежнему порождает объектные файлы, но компоновщик собирает исполняемый файл или библио-

теку с учетом того, что они будут исполняться в архитектуре сервера Глобальной сети. Функции загрузчика выполняет ОС по команде сервера либо сам интернет-сервер.

Этот метод удобен, но имеет один серьезный недостаток: при изменении содержимого динамической HTML-страницы или же при изменении логики ее реакции на действия интернет-клиента требуется создать новые код CGI или ISAPI-приложения. А для этого нужно выполнить полностью весь цикл построения результирующей программы, начиная от изменения исходного кода, включая компиляцию и компоновку. Поскольку содержимое веб-страниц меняется довольно часто (в отличие от обычных программ), такой подход нельзя признать очень эффективным. Кроме того, может потребоваться перенос интернет-сервера с одной архитектуры вычислительной системы на другую, а это также потребует перестройки всех используемых CGI-приложений и ISAPI-библиотек, кроме того, они в этом случае должны удовлетворять всем требованиям переносимости программного обеспечения.

Лучших результатов можно добиться, если не выполнять на сервере уже скомпилированный и готовый объектный код, а интерпретировать код программы, написанной на некотором языке. При интерпретации исходного кода сервер, конечно, будет иметь производительность ниже, чем при исполнении готового объектного кода, но этим можно пренебречь, поскольку производительность серверов Интернета ограничивает чаще всего не мощность вычислительной системы, а пропускная способность канала обмена данными. Тогда зависимость кода сервера от архитектуры вычислительной системы будет минимальной, а изменить содержимое порождаемой HTML-страницы можно будет сразу же, как только будет изменен порождающий ее исходный код (без дополнительной перекомпиляции).

Существует несколько языков и соответствующих им интерпретаторов, которые нашли применение в этой области и успешно служат цели порождения HTML-страниц. Среди них можно назвать язык Perl; язык, лежащий в основе различных версий веб-технологии PHP (Personal Home Pages); а также язык сценариев, на котором основана веб-технология ASP (Active Server P ages) — последний предложен и поддерживается фирмой Microsoft.

Текст на интерпретируемых языках, которые поддерживаются такими веб-технологиями, как ASP или PHP, представляет собой часть текста обычных HTML-страниц со встроенными в них сценариями (script). Эти сценарии можно писать на любом языке, поддерживаемом сервером. Интернет-сервер обрабатывает их при поступлении запроса об URL-адресе соответствующего файла. Он разбирает текст HTML-страницы, находит в нем тексты сценариев, вырезает их и интерпретирует в соответствии с синтаксисом и семантикой данного языка. В результате интерпретации получается выходной текст на языке HTML, который сервер вставляет непосредственно в то место исходной страницы, где встретился сценарий. Так обрабатывается динамическая веб-страница на любом интерпретируемом языке, ориентированном на работу в Глобальной сети. Естественно, для работы со страницей сервер должен иметь в своем составе интерпретатор соответствующего языка.

Все эти языки сценариев обладают присущими им характерными особенностями. Во-первых, они имеют мощные встроенные функции и средства для работы со строками, поскольку основной задачей программ, написанных с помощью таких языков, является обработка входных параметров (строковых) и порождение HTML-кода (который также является текстом). Во-вторых, все они имеют средства для работы

в архитектуре «клиент—сервер» для обмена информацией с серверами БД, а многие современные версии таких языков (например, язык, поддерживаемый технологией ASP) — средства для функционирования в трехуровневой архитектуре для обмена данными с серверами приложений.

В том случае, если CGI-приложение или интерпретируемый код на сервере используют БД для создания веб-страниц (а такая практика используется на многих серверах Интернета), схема, представленная на рис. 6.11, усложняется. Т еперь уже программный код, выполняемый на сервере, представляет собой приложение, построенное по архитектуре «клиент—сервер», к которому, в свою очередь, обращаются клиенты Интернета. В таком варианте работу сервера и клиента для отображения динамической веб-страницы можно представить так, как показано на рис. 6.12.

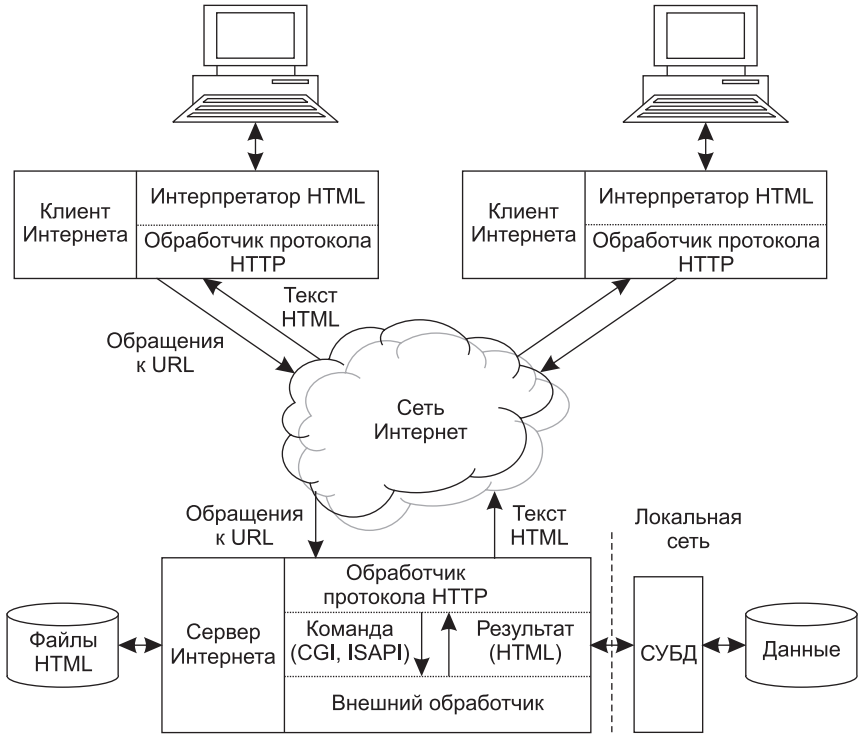

Рис. 6.12. Взаимодействие сервера и клиента в сети Интернет для отображения динамической веб-страницы с использованием БД

В схеме взаимодействия приложений, представленной на рис. 6.12, выделяются три основных компоненты:

- клиент Интернета, отображающий веб-страницу для пользователя;
- сервер Интернета, подготавливающий веб-страницу в соответствии с логикой приложения, заложенной на этом сервере, и одновременно являющийся клиентом по отношению СУБД;
- СУБД, осуществляющая операции с данными и файловые операции.

ПРИМЕЧАНИЕ

Получается, что по распределению функций схема, представленная на рис. 6.12, аналогична приложению, построенному по трехуровневой архитектуре. Наверное, приложения, работающие подобным образом, можно считать построенными по трехуровневой архитектуре (и некоторые производители в маркетинговых целях именно так и поступают).

Но здесь следует учитывать одно важное допущение: в такой схеме взаимодействие тонкого клиента с сервером возможно только по протоколу HTTP , который накладывает существенные ограничения на самого тонкого клиента и его возможности, в то время как полноценное приложение, построенное по трехуровневой архитектуре, лишено такого рода ограничений.

Первый недостаток всех перечисленных методов организации динамических веб-страниц заключается в том, что сначала сервер вынужден строить HTML-описание, собирая его некоторым образом из каких-то своих данных, а затем клиент Интернета на своей стороне разбирает (интерпретирует) полученное описание HTML-страницы. Таким образом, выполняется как бы двойная работа по генерации, а затем интерпретации текстов языка HTML. Кроме того, между клиентом и сервером по сети передаются довольно громоздкие описания HTML-страниц, что может значительно увеличивать трафик сети.

Еще один недостаток заключается в том, что на любые действия пользователя веб-страница в такой схеме может реагировать только с некоторым опозданием: сначала клиент должен передать на сервер информацию о действиях пользователя, потом сервер должен обработать полученные данные и передать клиенту новый текст HTML для веб-страницы, после чего клиент должен обработать полученные данные и отобразить изменения на экране. Все эти операции, а особенно передача данных по Глобальной сети, занимают заметное время, поэтому интерактивные возможности таких веб-страниц ограничены.

Тем не менее, несмотря на недостатки, данные методы довольно распространены в Глобальной сети, поскольку они очень просты, а кроме того, не требуют от клиентского компьютера ничего, кроме способности интерпретировать тексты HTML. Эта особенность довольно существенна.

Интерактивные веб-страницы

Следующим шагом в развитии веб-страниц стало добавление в них интерактивных возможностей. Наличие таких возможностей подразумевает , что некоторая часть веб-страницы может, реагируя на действия клиента, менять свой вид и содержимое, не передавая при этом на сервер Интернета команды для изменения текста страницы. Иными словами, какая-то часть веб-страницы должна выполняться на компьютере клиента Интернета.

Поскольку вычислительная система клиента Интернета может быть любой, возможны только следующие способы организации выполнения кода веб-страницы на клиенте:

- использование интерпретируемых языков сценариев (script), выполняемых на стороне клиента;
- использование языков и технологий, не зависящих от архитектуры целевой вычислительной системы.

Интерпретируемые языки уже были рассмотрены выше. В простейшем случае (например, для создания интерактивных элементов вроде «всплывающего» меню) исполнение языков сценариев (script) на клиенте Интернета является приемлемым решением. Но для реализации сложных интерактивных функций, связанных с обработкой информации и БД, использование языков сценариев будет связано с серьезными проблемами. Ведь в этом случае на клиентскую часть через Интернет должен быть передан значительный объем текста на языке сценария, который затем должен быть обработан и интерпретирован на клиенте.

Во втором варианте на клиентскую часть передается не код, а некоторый компонент (апплет, applet), который выполняется клиентом Интернета. Апплет может быть написан на полноценном языке программирования, но должен быть построен таким образом, чтобы не зависеть от типа целевой вычислительной системы. Если апплет выполняет обращение к данным сервера, то схему работы интерактивной веб-страницы можно представить, как показано на рис. 6.13.

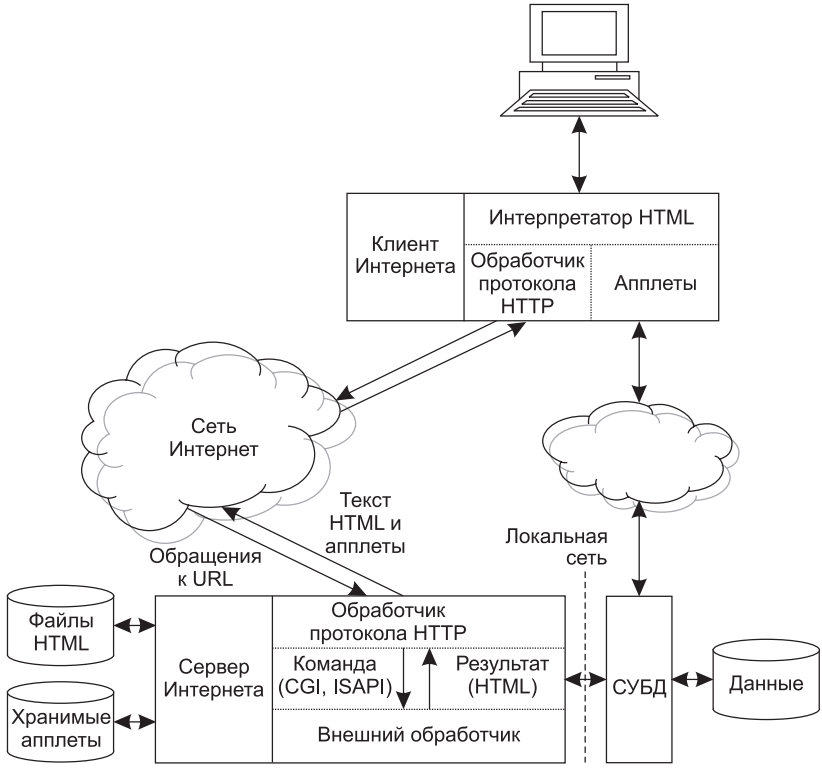

Рис. 6.13. Взаимодействие сервера и клиента в сети Интернет для отображения интерактивной веб-страницы с использованием апплетов

Далее даны только основы технологий построения интерактивных динамических веб-страниц. Рассмотренные далее методы хотя и снижают нагрузку на сеть, но предъявляют определенные требования к клиентской части, что не всегда приемлемо. Более подробную информацию по кратко описанным далее языкам и технологи-

ям, поддерживающим такие методы, можно найти в соответствующей технической литературе (она частично имеется в [13, 51, 67]), но лучше обратиться на соответствующие сайты во Всемирной сети [73, 74, 75].

Технология ActiveX. Платформа .NET. Веб-службы

Технология ActiveX была разработана корпорацией Microsoft для предоставления разработчикам веб-страниц возможностей использования функций других приложений. Основу технологии ActiveX составила технология взаимодействия приложений OLE (Object Linking and Embedding — связывание и внедрение объектов)[1]. Фактически сейчас под термином «ActiveX» объединены все современные технологии Microsoft, построенные на основе OLE и тесно связанные с технологиями COM/DCOM .

ActiveX — это архитектура, которая позволяет программному компоненту (управляющему элементу ActiveX) взаимодействовать с другими компонентами через сеть (в том числе Интернет). В основе архитектуры ActiveX лежат разработанные Microsoft стандарты COM/DCOM, которые ориентированы на обеспечение взаимодействия любых приложений. Поскольку OLE уже далеко не новая технология, разработчики создали большое количество объектов OLE, которые сейчас могут использоваться в качестве управляющих элементов ActiveX. Ряд компаний-разработчиков продают библиотеки готовых объектов OLE, с которыми можно работать при написании своих программ и веб-страниц.

ActiveX позволяет применять не только библиотеки управляющих элементов, созданные сторонними компаниями, но и однократно использовать управляющие элементы собственной разработки. Использование элементов ActiveX позволяет привнести в веб-страницы возможности приложений, построенных по архитектуре «клиент—сервер». У правляющие элементы ActiveX позволяют пользователям веб-страниц выполнять сложные операции, получать нужную информацию из БД и даже от приложений, работающих на других серверах, как показано на рис. 6.13. Например, управляющий элемент ActiveX может хранить свои данные на странице где-то в сети WWW либо может быть выкачан с сервера WWW и затем запущен на машине клиента.

Управляющий элемент ActiveX — независимый программный компонент , выполняющий специфические задачи стандартным способом. Разработчики могут задействовать один или несколько таких элементов в приложении или на веб-странице. В результате веб-страница получается в виде текста HTML, включающего в себя компоненты ActiveX в качестве готовых составных частей.

Управляющие элементы ActiveX — не отдельные приложения. Напротив, они являются серверами, которые подключаются к контейнеру элементов. В случае с веб-страницей контейнером является программа-браузер, выполняющаяся на стороне клиента. Взаимодействие между управляющим элементом и его контейнером определяется различными интерфейсами, поддерживаемыми COM-объектами. Технология управляющих элементов ActiveX была расширена, чтобы код и данные управляющих элементов могли при необходимости загружаться с сервера WWW и исполняться внутри средства просмотра. Сценарии ActiveX (ActiveX Scripting) —

[1] Эта технология уже упоминалась ранее при описании технологий ADO и COM/DCOM.

универсальный способ исполнения клиентами сценариев, написанных на любом языке, тогда как технология гиперсвязей ActiveX (ActiveX Hyper-links) обеспечивает создание гиперсвязей в стиле WWW не только между страницами HTML, но и между любыми типами документов.

Дальнейшим развитием средств программирования веб-страниц от корпорации Microsoft стало появление платформы .NET и технологии веб-служб [51, 52].

Платформа .NET (обозначение читается как «дот-нет», без расшифровки) стала средством реализации мобильности приложений на основе промежуточного двоичного кода, созданным компанией Microsoft. Общая схема такой реализации мобильности приложений была рассмотрена выше (см. рис. 6.3). При ее использовании все приложения выполняются единообразно на всех типах целевых вычислительных систем вне зависимости от исходного языка и использованной системы программирования. Компания Microsoft сначала реализовала такую схему с помощью дополнительных компонент в ОС типа Windows, а затем сделала ее неотъемлемой частью ОС (последняя версия ОС типа Windows — Windows Vista — полностью построена на основе платформы .NET).

Платформа .NET содержит общеязыковую среду выполнения (Common Language Runtime — CLR). Общеязыковая среда выполнения CLR поддерживает управляемое выполнение, которое основано на интерпретации средствами CLR промежуточного двоичного кода приложений. Системы программирования, ориентированные на создание приложений для .NET , должны порождать не машинный код, а промежуточный код CLR. Тогда выполнение приложений в среде .NET идет независимо от целевой вычислительной системы.

Совместно с общей системой типов (Common T ype System — CTS) общеязыковая среда выполнения CLR поддерживает возможность взаимодействия приложений, построенных на основе любых исходных языков и с помощью любых систем программирования, поддерживающих платформу .NET. Кроме того, платформа .NET предоставляет в распоряжение разработчиков большую полнофункциональную библиотеку классов .NET Framework. Чтобы решить все проблемы, связанные с разработкой приложений, платформа .NET обладает базовым набором служб, которые в доступны в любой системе программирования. Службы и библиотеки .NET доступны также и для разработки веб-страниц.

Создание и развитие платформы .NET послужило основой для появления технологии веб-служб. Другим краеугольным камнем этой технологии стал язык разметки XML (eXtensible Markup Language — расширяемый язык разметки).

Язык XML, как и HTML, является языком текстовой разметки. Файл XML — это обычный текстовый файл, содержимое которого структурировано в соответствии с синтаксисом языка XML. Распознаватель текста XML оперирует синтаксическими конструкциями языка, главной из которых является тег (от английского «tag»). Более строгие по сравнению с HTML правила и четкая стандартизация языка определили широкое распространение XML в качестве языка обмена данными. Удобство его использование в этом качестве определяет тот факт, что представление любых структурированных данных на XML не зависит от вычислительной системы и определяется только синтаксисом этого языка. При этом семантика (смысл данных, представленных на XML) регламентируется разработчиком, который определяет смысл каждого тэга языка и его атрибутов. В резуль тате широ-

кого распространения языка XML библиотеки, содержащие распознаватели этого языка, стали доступны практически для всех существующих вычислительных систем.

Веб-служба (английский термин Web Service, иногда и по-русски это средство называют «веб-сервис») — это приложение или блок находящегося на сервере выполняемого кода, функционирование которого основано на применении протоколов обмена данными на основе языка XML. Поиск этого кода, его извлечение и получение посредством него требуемого результата выполняется в среде .NET Framework. В этой среде веб-служба вызывается так же просто, как и локальная функция.

Веб-служба — это не объект в его традиционном представлении в объектно-ориентированных языках. Любой веб-метод, используемый веб-службой, является неделимым и не имеющим постоянного местонахождения. В некоторой степени веб-служба подобна библиотеки динамических функций. Это упрощение в значительной мере обеспечивает преимущества веб-служб и возможность их использования для создания веб-страниц. Веб-службы не ограничены конкретной технологией, и потому они могут быть использованы в любом сценарии, что существенным образом отличает их от таких технологий как COM/DCOM и CORBA .

В кратком обзоре сложно дать даже общие представления о платформе .NET и связанных с нею технологиях. Автор рекомендует обратиться к специализированной литературе (в частности [19, 45, 51, 52, 62]) или к ресурсам сети Интернет [76, 77, 81]. Эта область технологий сейчас продолжает развиваться и совершенствоваться, так как в настоящее время является перспективным направлением корпорации Microsoft. Пожалуй, главным недостатком этих технологий следует признать их узкую ориентированность на одного основного производителя (пусть и очень влиятельного на рынке) и, как следствие, ограниченность рамками ОС типа Windows.

Языки программирования Java и C#

Язык Java был разработан компанией Sun в качестве средства веб-программирования. Этот язык, в отличие от языка описания гипертекста HTML, является полноценным языком программирования. Он содержит в себе все основные операторы, конструкции и структуры данных, присущие языкам программирования. Синтаксические конструкции и семантика языка Java большей частью были заимствованы из языков программирования C и C++.

Основная идея, отличающая язык Java от многих других языков программирования, не ориентированных на применение в Глобальной сети, заключается в том, что Java не является полностью компилируемым языком. Исходная программа, созданная на языке Java, не преобразуется в машинные коды. Компилятор языка порождает некую промежуточную результирующую программу на специальном низкоуровневом двоичном коде (эта результирующая программа называется Java-апплет, а код, на котором она строится — Java байт-код). Именно этот код интерпретируется при исполнении результирующей Java-программы. Такой метод исполнения кода, основанный на интерпретации, делает его практически независимым от архитектуры целевой вычислительной системы (общая схема выполнения результирующей программы на основе байт-кода была представлена выше на рис. 6.3).

Таким образом, для исполнения Java-программы необходимы две составляющих: компилятор промежуточного двоичного кода, порождающий его из исходного текста Java-программы, и интерпретатор, исполняющий этот промежуточный двоичный код. Такой интерпретатор получил название виртуальной Java-машины. Описание основных особенностей реализации виртуальной Java-машины с точки зрения системы программирования описаны в [8, 16, 73].

Одной из отличительных особенностей данного языка является использование специального механизма распределения памяти (менеджера памяти). В языке Java не могут быть использованы функции динамического распределения памяти по запросам пользователя и связанные с ними операции над адресами и указателями, поскольку они зависят от архитектуры вычислительной системы. Динамическая память в языке может выделяться только системой программирования под классы и объекты самого языка. Для этого менеджер памяти должен сам организовывать своевременное выделение областей памяти при создании новых классов и объектов, а затем освобождать области памяти, которые уже больше не используются. В последнем случае должна решаться непростая задача «сборки мусора» — поиска неиспользуемых фрагментов в памяти и их освобождение. Причем, поскольку в языке Java за распределение памяти отвечает не пользователь, а интерпретатор кода, эти задачи должны решаться независимо от хода выполнения самой Java-программы.

Менеджер памяти входит в состав виртуальной Java-машины и, безусловно, зависит от архитектуры вычислительной системы, где функционирует эта машина, но интерпретируемые ею программы при этом остаются независимыми от архитектуры. Хороший менеджер памяти — важная составляющая виртуальной Java-машины наряду с быстродействующим интерпретатором промежуточного низкоуровневого кода.

При выполнении Java-программы в Г лобальной сети компилятор, порождающий промежуточный низкоуровневый код, находится на стороне сервера, а интерпретатор, выполняющий этот код — на стороне клиента. По сети от сервера к клиенту передается только скомпилированный код. С этой точки зрения использование языка Java для исполнения программ в сети дает преимущества по сравнению с использованием других языков, выполняемых на стороне сервера, как описано выше. Преимущество заключается в том, что по сети не надо передавать громоздкие HTML-описания страниц или тексты команд и программ сценариев (script), что значительно снижает трафик в сети, а на стороне клиента не надо распознавать эти команды и программы.

Однако отсюда проистекают и основные недостатки, присущие языку Java. Главный из них заключается в том, что на клиентской стороне должна присутствовать виртуальная Java-машина для интерпретации поступающего из сети кода. Это значит , что так или иначе интерпретатор языка Java должен входить в состав архитектуры целевой вычислительной системы, а без его наличия функционирование такой схемы становится невозможным. Кроме того, промежуточный код языка исполняется на стороне клиента, а значит, скорость его выполнения и возможности Java-программы во многом зависят от производительности клиентского компьютера, которая может оказаться недостаточно высокой у машины, ориентированной только на подключение к Г лобальной сети. Речь идет не только о скорости интерпретации команд, но и о том, что клиентская система должна быть способна работать со всеми базовыми классами, присущими языку Java.

Еще одна особенность связана с необходимостью обеспечения безопасности при исполнении Java-программ. Поскольку код исполняемой программы поступает из Гло-

бальной сети, то нет никакой гарантии, что этот код не содержит фатальную ошибку Кроме того, такой код может быть преднамеренно создан со злым умыслом. Имея все возможности языка программирования (в отличие от языка HTML) и исполняясь в среде вычислительной системы клиента, он может привести к выходу из строя этой системы или к повреждению хранящихся в ней данных. Потому при построении интерпретатора виртуальной Java-машины нужно целенаправленно выделять и отслеживать потенциально опасные команды (такие как выполнение программ или обращение к файлам на клиентской машине). Несмотря на то что большинство производителей виртуальных Java-машин считаются с этим правилом, проблема безопасности остается актуальной при использовании языка Java.

Язык Java быстро завоевал популярность и занял значительное место на рынке языков программирования и связанных с ними средств разработки. По сути, этот язык явился первой удачной реализацией двухэтапной модели выполнения программ, построенной на использовании компилятора промежуточного кода с последующей интерпретацией полученного кода. Независимость выполнения Java-программ от архитектуры целевой вычислительной системы способствовала росту популярности языка.

Росту популярности языка программирования Java и построенных на его основе систем программирования способствовал также тот факт , что в его состав входят встроенные средства поддержки интерфейса с серверами БД, ориентированные на разработку приложений, выполняющихся в архитектуре «клиент—сервер» . Системы программирования языка Java совместимы также с известной группой стандартов, ориентированной на разработку резуль тирующих программ, выполняющихся в трехуровневой архитектуре — CORBA. В настоящее время компании, поддерживающие системы программирования на основе языка Java, образуют специальное сообщество для поддержки и распространения средств разработки на основе данного языка. Всю необходимую информацию можно получить в литературе [8, 16], на сайте [73] и во многих других местах Всемирной сети Интернет[1].

Основной проблемой языка Java остается его производительность. При прочих равных условиях интерпретируемая программа, построенная на основе системы программирования для Java, будет уступать в скорости откомпилированной программе, созданной в любой другой современной системе программирования. Поэтому язык Java практически не применяется в программах реального времени, а также в программах, требующих сложных расчетов. Но во многих прикладных программах, где производительность не играет столь принципиального значения, он вполне может быть использован, так как дает преимущества в независимости результирующей программы от архитектуры целевой вычислительной системы. Большинство современных ОС допускают наличие в своем составе виртуальных Java-машин для интерпретации результирующего кода Java-программ.

На том же технологическом решении, на котором основан язык программирования Java, построен и язык программирования C# (читается как «Си-шарп») [19, 58], соз-

[1] Известная компания — производитель ОС и систем программирования Microsoft некоторое время тоже способствовала распространению языка Java и занималась его поддержкой. Однако затем, к сожалению, по причинам маркетинговых и других разногласий с компанией Sun отказалась от использования данного средства разработки. Многие идеи, первоначально заложенные в Java, нашли свою реализацию в языке C# (Си-шарп), предложенном этой компанией и продвигаемым ею в настоящее время на рынок средств разработки.

данный компанией Microsoft. Язык C# появился позже языка Java, поскольку первоначально компания Microsoft поддерживала развитие Java, но некоторые ограничения стандарта Java, не устраивавшие Microsoft, побудили ее в дальнейшем отказаться от развития языка Java и предложить свой язык программирования, ориентированный на мобильность программного обеспечения на базе промежуточного двоичного кода.

Синтаксис языка C#, как и синтаксис языка Java, построен на основе синтаксических конструкций языков C и C++ (название языка «C#» было выбрано как следующий шаг в цепочке C — C++). Но в то же время язык C# не является точной копией Java и имеет ряд синтаксических и семантических отличий. Кроме различий в синтаксисе, C# имеет и ряд принципиальных отличий от Java.

Во-первых, код, порождаемый системой программирования, которая обрабатывает исходный текст на языке C#, должен быть ориентирован на общеязыковую среду выполнения CLR платформы .NET . Результатом компиляции исходной программы на C# является результирующая программа на промежуточном языке двоичного кода IL (обозначение «IL» именно так и расшифровывается: Intermediate Language — промежуточный язык). Соответственно, разработчики на языке C# должны использовать полнофункциональную библиотеку классов .NET Framework и службы .NET . И хотя с точки зрения идеологии построения платформа .NET никак не привязана к какой-либо вычислительной системе, по факту полномасштабная ее реализация в настоящее время существует пока только для ОС семейства W indows (в то время как Java-машины доступны практически для всех типов ОС). Эта ситуация существенно ограничивает мобильность промежуточного кода, полученного с помощью C#.

Во-вторых, существенное отличие заключается в том, что промежуточный код на языке IL не интерпретируется на целевой вычислительной системе, а компилируется в машинный код в момент запуска программы на выполнение. Функция компиляции языка IL на платформе .NET возложена на настраивающий загрузчик . Это позволяет достичь большего быстродействия программ. При этом язык IL не привязан к исходному языку C# — резуль тирующая программа на IL может быть построена и на основе других языков программирования, отличных от C#.

В-третьих, хотя язык C#, как и язык Java, является строго типизированным объектно-ориентированным языком, не допускающим произвольных манипуляций исходного кода с оперативной памятью (как это возможно на C++), в нем для разработчиков были сделаны некоторые послабления. Эти дополнительные возможности языка позволяют использовать так называемые «небезопасные» участки кода (если они соответствующим образом обозначены в тексте исходной программы) и объекты языка, не являющиеся классами, а также пространства имен (чего нет в языке Java). Видимо, тут дело в том, что C# появился позже языка Java и при его создании компания Microsoft учла некоторые замечания, зачастую высказываемые разработчиками по отношению к строгостям исходных программ на языке Java.

В настоящее время компания Microsoft продолжает развивать язык C# в рамках системы программирования Microsoft Visual Studio . Эта система программирования объединяет в едином техническом решении компиляторы с трех исходных языков: Visual Basic, C++ и C# (ранее для каждого из этих языков компания Microsoft предлагала свою отдельную систему программирования). На основании исходного кода Visual Basic и C++ эта система программирования позволяет порождать либо обычный исполняемый код (машинный код), либо управляемый код на языке IL,

ориентированный на CLR в .NET . В то же время на основании исходного кода C# машинный код построен быть не может, результирующая программа для этого языка может быть построена только на управляемом коде языка IL.

Оба описанных языка (Java и C#) являются по своей сути двумя различными подходами к реализации идеи мобильности приложений на основе промежуточного двоичного кода. Оба они и построенные на их основе системы программирования сейчас получили широкое распространение на рынке средств разработки. Скорее всего, эти языки и связанные с ними технологии и дальше будут развиваться параллельно.

Контрольные вопросы и задачи

Вопросы

1. Укажите, что из перечисленного является составной частью системы программирования:
   - лексический анализатор;
   - компоновщик;
   - редактор ресурсов пользовательского интерфейса;
   - исходная программа;
   - компилятор с исходного языка;
   - компилятор ресурсов;
   - целевая вычислительная система;
   - библиотеки исходного языка;
   - библиотеки операционной системы;
   - отладчик;
   - файл подсказки?
2. Объясните, чем является обработчик языка Makefile: компилятором, транслятором или интерпретатором?
3. Что такое Application Program Interface (API)?
4. Относятся ли к понятию «ресурсы прикладной программы» следующие данные:
   - текст сообщения, выдаваемого программой при возникновении ошибки;
   - динамические библиотеки, используемые программой;
   - тексты пунктов главного меню программы;
   - статические библиотеки, используемые программой;
   - файлы, обрабатываемые прикладной программой;
   - цвет фона главного окна программы;
   - структуры данных, используемые программой в процессе работы;
   - файл подсказки, выдаваемой программой по запросу пользователя?
5. Какие из перечисленных функций выполняет текстовый редактор в современных системах программирования:
   - выдача контекстной подсказки по функциям библиотек системы программирования;

- анализ и выделение (цветом или шрифтом) лексем исходной программы в процессе ее подготовки;
- выдача контекстной подсказки по функциям исходной программы в процессе ее подготовки;
- позиционирование на место ошибки, обнаруженной в процессе компиляции;
- позиционирование на место предупреждения, обнаруженного в процессе компиляции;
- исправление по команде разработчика текста исходной программы в соответствии с типом предупреждения, обнаруженного в процессе компиляции;
- выдача контекстной подсказки по операторам входного языка?

6. Если в процессе создания современной системы программирования встанет вопрос о разработке более эффективного компилятора либо о совершенствовании сервисных средств и развитого интерфейса пользователя, то какому направлению будет отдано предпочтение?
7. В чем особенности функционирования компилятора в составе системы программирования по сравнению с его функционированием в виде отдельного программного модуля?
8. Какие адреса обрабатывает компоновщик:
   - относительные;
   - логические;
   - физические.
9. С какими адресами работает загрузчик:
   - относительными;
   - логическими;
   - физическими.
10. Если система программирования выполняет отладку результирующей программы путем моделирования архитектуры целевой вычислительной системы, то чем отличается такое выполнение программы от работы интерпретатора?
11. В чем преимущества и недостатки динамически загружаемых библиотек по сравнению со статически подключаемыми библиотеками?
12. Если подключение к резуль тирующей программе статической библиотеки выполняет компоновщик, то какая часть системы программирования или ОС ответственна за подключение динамически загружаемой библиотеки?
13. Должна ли статически подключаемая библиотека быть доступна системе программирования в момент компиляции результирующей программы? А в момент компоновки результирующей программы?
14. Должна ли динамически загружаемая библиотека быть доступна системе программирования в момент компиляции результирующей программы? А в момент компоновки результирующей программы?
15. Какие дополнительные возможности предоставляет пользователю лексический анализ исходного текста исходной программы «на лету»? Как влияет реализация

такого лексического анализа в системе программирования на скорость работы компилятора, на количество проходов, выполняемых компилятором?

16. Какие преимущества имеет приложение, функционирующее в составе архитектуры «клиент—сервер», по сравнению с приложением, функционирующим в составе архитектуры «файл—сервер»?
17. Клиентскую часть приложения, функционирующего в составе архитектуры «клиент—сервер», часто называют «толстым клиентом». В чем отличие «толстого клиента» в архитектуре «клиент—сервер» от «тонкого клиента» в трехуровневой архитектуре?
18. Может ли сервер приложений функционировать без сервера данных?
19. Почему в сети Интернет часто используются интерпретируемые языки?

Задачи

1. Функция `F1` содержится в библиотеке, построенной с помощью компилятора в некоторой системе программирования. При попытке использовать эту функцию при построении исполняемого файла в другой системе программирования компоновщик выдает сообщение об ошибке вида «`функция F1 не найдена`». Какова может быть причина появления такого рода ошибки? Что следует предпринять разработчику программы?
2. Функция `F1` содержится в библиотеке, построенной с помощью компилятора в некоторой системе программирования. Функция `F1` отлажена и проверена, однако при использовании ее в исполняемом файле, созданном в другой системе программирования, возникает ошибка, связанная с защитой памяти. Результирующая программа не может выполняться корректно. Какова может быть причина появления такого рода ошибки? Что следует предпринять разработчику программы?
3. Функция `F1` является виртуальной функцией некоторого класса `C1`. На основе одного и того же исходного кода в одной системе программирования построены динамически загружаемая библиотека и исполняемый файл. Исполняемый файл создает экземпляр класса `C1` и передает его в качестве параметра в динамически загружаемую библиотеку, где вызывается функция `F1`. Приведет ли такая схема к возникновению ошибок во время выполнения программы? Если ошибки возможны, то что следует предпринять разработчику программы? Как изменится ситуация, если предположить, что для создания исполняемого модуля и динамически загружаемой библиотеки использовался один и тот же исходный код, но разные системы программирования?
4. Имеется класс `C1`. На основе одного и того же исходного кода в одной системе программирования построены динамически загружаемая библиотека и исполняемый файл. Исполняемый файл создает экземпляр класса `C1` и передает его в качестве параметра в динамически загружаемую библиотеку Динамически загружаемая библиотека проверяет экземпляр класса на принадлежность к типу `C1`. Приведет ли такая схема к возникновению ошибок во время выполнения программы? Если ошибки возможны, то что следует предпринять разработчику программы?

Ответы на вопросы и задачи

Глава 1

Ответы на вопросы

1. Над цепочками символов определены следующие операции: конкатенация (или сложение) двух цепочек, итерация (или повторение) цепочки, обращение цепочки, а также замена (или подстановка) подцепочки.
2. $|\alpha\beta| = |\alpha| + |\beta| = |\beta\alpha|$ — верно;
   $\alpha\beta = \beta\alpha$ — неверно;
   $|\alpha^R| = |\alpha|$ — верно;
   $(\alpha^2\beta^2)^R = (\beta^R\alpha^R)^2$ — неверно;
   $(\alpha^2\beta^2)^R = (\beta^R)^2(\alpha^R)^2$ — верно.
3. Существует три метода определения языков: перечисление, генерация и распознавание. Перечисление не находит практического применения, так как большинство реальных языков содержит бесконечное множество цепочек.
4. Для задания языка программирования необходимо решить три вопроса: определить множество допустимых символов языка, определить множество правильных программ языка, задать смысл для каждой правильной программы. Первый вопрос решается в рамках теории формальных языков, второй решается в рамках этой теории частично, а третий выходит за рамки теории.
5. Генератором цепочек для языка программирования обычно выступает человек—разработчик программы. Распознавателем цепочек языка программирования обычно является компилятор.
6. Задача разбора заключается в построении распознавателя, эквивалентного заданной грамматике языка. Она разрешима не для всех типов языков.
7. Грамматика — это описание способа построений предложений некоторого языка.
8. Грамматика в форме Бэкуса—Наура выглядит так: G(**VT**,**VN**,**P**,S), где **VT** — множество терминальных символов или алфавит терминальных символов; **VN** — множество нетерминальных символов или алфавит нетерминальных символов; **P** — множество правил (продукций) грамматики вида $\alpha \to \beta$, где $\alpha \in (\mathbf{VN} \cup \mathbf{VT})^+$, $\beta \in (\mathbf{VN} \cup \mathbf{VT})^*$; S — целевой (начальный) символ грамматики $S \in \mathbf{VN}$. Существует бесконечное множество способов определения грамматик, в данной книге были рассмотрены два из них: форма с метасимволами и графическая форма.

9. В форме Бэкуса—Наура невозможно построить грамматику для реального языка так, чтобы она не содержала рекурсивных правил, поскольку все реальные языки содержат бесконечное множество цепочек. А рекурсия в форме Бэкуса—Наура является тем средством, с помощью которого на основе конечного множества правил строится бесконечное множество цепочек языка.
10. Распознаватель — это специальный автомат, который позволяет определить принадлежность цепочки символов некоторому языку.
11. Распознаватели классифицируются по вариантам трех основных своих составляющих: считывающей головки, устройства управления и внешней памяти. Каждому типу языков соответствует свой тип распознавателей.
12. Грамматики в классификации Хомского классифицируются на основе структуры их правил.
13. По классификации Хомского выделяют четыре типа грамматик: грамматики с фразовой структурой, контекстно-зависимые (или неукорачивающие), контекстно-свободные и регулярные. Они соотносятся между собой нетривиальным образом. Но при этом регулярные являются самыми простыми, контекстно-свободные — более сложными, контекстно-зависимые — еще более сложными и грамматики с фразовой структурой — самыми сложными.
14. По классификации Хомского выделяют четыре типа языков: языки с фразовой структурой, контекстно-зависимые, контекстно-свободные и регулярные. Тип языка определяется по типу самой простой грамматики, задающей этот язык.
15. Сентенциальной формой грамматики называется любая цепочка символов, которая выводима из целевого символа грамматики.
16. Вывод называется левосторонним , если на каждом шаге вывода правило грамматики применяется к крайнему левому нетерминальному символу в цепочке. Вывод называется правосторонним, если на каждом шаге вывода правило грамматики применяется к крайнему правому нетерминальному символу в цепочке. Можно построить и другие варианты выводов (в общем случае бесконечно много вариантов), но на практике они не используются.

Решения и ответы для задач

1. Типы приведенных грамматик:

   $\mathbf{G}_1$ — грамматика с фразовой структурой (не может быть КЗ-грамматикой или неукорачивающей из-за наличия пустых цепочек в правой части правил и не может быть КС-грамматикой, так как содержит более одного символа в левой части правила: <цифра><цифра>→<цифра><цифра><цифра>);

   $\mathbf{G}_2$ — КЗ или неукорачивающая грамматика (может быть отнесена к обоим классам);

   $\mathbf{G}_3$ — КС-грамматика (обратите внимание: не может быть регулярной, так как в разных правилах терминальные символы стоят с разных сторон от нетерминальных);

   $\mathbf{G}_4$ — регулярная (леволинейная);

   $\mathbf{G}_5$ — регулярная (леволинейная, также является автоматной — см. главу 3).

Язык десятичных чисел с фиксированной точкой относится к регулярным языкам, так как существует регулярная грамматика, задающая этот язык.

2. Построим цепочки вывода на основе грамматики $\mathbf{G}_1$:

<число> ⇒ <знак><осн> ⇒ <знак><цел> ⇒ <знак><цифра><цифра> ⇒ <знак><цифра>3 ⇒ <знак>83 ⇒ -83

<число> ⇒ <знак><осн> ⇒ <знак><цел> ⇒ <знак><цифра><цифра> ⇒ <знак><цифра><цифра><цифра> ⇒ <знак><цифра><цифра>9 ⇒ <знак><цифра>39 ⇒ <знак>239 ⇒ 239

<число> ⇒ <знак><осн> ⇒ <знак><цел>.<дроб> ⇒ <знак><цел>.<цел> ⇒ <знак><цел>.<цифра> ⇒ <знак><цел>.4 ⇒ <знак><цифра><цифра>.4 ⇒ <знак><цифра>0.4 ⇒ <знак>10.4 ⇒ 10.4

Построим цепочки вывода на основе грамматики $\mathbf{G}_2$:

<число> ⇒ -<осн> ⇒ -<часть> ⇒ -<цифра><цифра> ⇒ -<цифра>3 ⇒ -83

<число> ⇒ <осн> ⇒ <часть> ⇒ <цифра><цифра> ⇒ <цифра><цифра><цифра> ⇒ <цифра><цифра>9 ⇒ <цифра>39 ⇒ 239

<число> ⇒ <осн> ⇒ <часть>.<часть> ⇒ <часть>.<цифра> ⇒ <часть>.4 ⇒ <цифра><цифра>.4 ⇒ <цифра>0.4 ⇒ 10.4

Построим цепочки вывода на основе грамматики $\mathbf{G}_3$:

<число> ⇒ -<осн> ⇒ -<часть> ⇒ -<часть>3 ⇒ -83

<число> ⇒ <осн> ⇒ <часть> ⇒ <часть>9 ⇒ <часть>39 ⇒ 239

<число> ⇒ <осн> ⇒ <осн>4 ⇒ <часть>.4 ⇒ <часть>0.4 ⇒ 10.4

Построим цепочки вывода на основе грамматики $\mathbf{G}_4$:

<число> ⇒ <часть> ⇒ <часть>3 ⇒ <знак>83 ⇒ -83

<число> ⇒ <часть> ⇒ <часть>9 ⇒ <часть>39 ⇒ <знак>239 ⇒ 239

<число> ⇒ <число>4 ⇒ <часть>.4 ⇒ <часть>0.4 ⇒ <знак>10.4 ⇒ 10.4

Построим цепочки вывода на основе грамматики $\mathbf{G}_5$:

<число> ⇒ <часть>3 ⇒ <знак>83 ⇒ -83

<число> ⇒ <часть>9 ⇒ <часть>39 ⇒ 239

<число> ⇒ <осн>4 ⇒ <часть>.4 ⇒ <часть>0.4 ⇒ 10.4

Все представленные выводы являются правосторонними.

3. Для решения поставленной задачи возьмем грамматику $\mathbf{G}_5$. В нее достаточно добавить одно правило: <осн>→. и получим грамматику $\mathbf{G}_5'$:

G_5'({".",+,-,0,1,2,3,4,5,6,7,8,9},{<знак>,<число>,<часть>,<осн>},$\mathbf{P}_5$,<число>):

$\mathbf{P}_5$:

<число> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | <часть>. | <осн>0 | <осн>1 | <осн>2 | <осн>3 | <осн>4 | <осн>5 | <осн>6 | <осн>7 | <осн>8 | <осн>9 | <часть>0 | <часть>1 | <часть>2 | <часть>3 | <часть>4 | <часть>5 | <часть>6 | <часть>7 | <часть>8 | <часть>9

<осн> → . | <часть>. | <осн>0 | <осн>1 | <осн>2 | <осн>3 | <осн>4 | <осн>5 | <осн>6 | <осн>7 | <осн>8 | <осн>9

<часть> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | <знак>0 | <знак>1 | <знак>2 | <знак>3 | <знак>4 | <знак>5 | <знак>6 | <знак>7 | <знак>8 | <знак>9 | <часть>0 | <часть>1 | <часть>2 | <часть>3 | <часть>4 | <часть>5 | <часть>6 | <часть>7 | <часть>8 | <часть>9

<знак> → + | -

Построим цепочки вывода для заданных цепочек:

<число> ⇒ <осн>9 ⇒ <осн>29 ⇒ .29

<число> ⇒ <осн>4 ⇒ <осн>04 ⇒ <осн>104 ⇒ .104

4. Ответы:
   - язык регулярный, ответ окончательный (более простого типа языков не существует);
   - язык контекстно-свободный, но из имеющихся условий ответ не окончательный — необходимо проверить его принадлежность к регулярным языкам;
   - язык регулярный, ответ окончательный (более простого типа языков не существует).
5. Грамматика для определения «поезда» $\mathbf{G_п}$:

$G_п$({локомотив, вагон}, {<поезд>}, $\mathbf{P_п}$, <поезд>):

$\mathbf{P_п}$: <поезд> → локомотив | <поезд>локомотив | <поезд>вагон

   - все локомотивы должны быть сосредоточены в начале поезда:

$G_{п2}$({локомотив, вагон}, {<поезд>, <начало>}, $\mathbf{P_{п2}}$, <поезд>):

$\mathbf{P_{п2}}$:

<поезд> → <начало> | <поезд>вагон

<начало> → локомотив | <начало>локомотив

   - поезд начинается с локомотива и заканчивается локомотивом:

$G_{п3}$({локомотив, вагон}, {<поезд>, <середина>}, $\mathbf{P_{п3}}$, <поезд>):

$\mathbf{P_{п3}}$:

<поезд> → <середина>локомотив

<середина> → локомотив | <середина>вагон

   - поезд не должен содержать два локомотива либо два вагона подряд:

$G_{п4}$({локомотив, вагон}, {<поезд>, <п1>, <п2>}, $\mathbf{P_{п4}}$, <поезд>):

$\mathbf{P_{п4}}$:

<поезд> → <п1>локомотив | <п2>вагон | локомотив

<п1> → <п2>вагон

<п2> → локомотив | <п1>локомотив

6. Чтобы построить в форме Бэкуса—Наура определение поезда, содержащего не более 60 вагонов, потребуется как минимум 60 правил. В форме с метасимволами это определение будет выглядеть гораздо проще:

$G_{п5}$({локомотив, вагон}, {<поезд>}, $\mathbf{P_{п5}}$, <поезд>):

$\mathbf{P_{п5}}$: <поезд> → локомотив{(локомотив вагон)}[59]

   - поезд начинается с локомотива и заканчивается локомотивом:

$G_{пб}$({локомотив,вагон},{<поезд>},$\mathbf{P}_{пб}$,<поезд>):

$\mathbf{P}_{пб}$: <поезд> → локомотив{(вагон)}[58] локомотив

7. Цепочка вывода:

S ⇒ AE ⇒ aABCE ⇒ aaABCBCE ⇒ aaaABCBCBCE ⇒ aaaaBCBCBCBCE ⇒
aaaaBCDCBCBCE ⇒ aaaaBBDCBCBCE ⇒ aaaaBBCCBCBCE ⇒ aaaaBBCCDCBCE ⇒
aaaaBBCBDCBCE ⇒ aaaaBBCBCCBCE ⇒ aaaaBBCDCCBCE ⇒ aaaaBBBDCCBCE ⇒
aaaaBBBCCCBCE ⇒ aaaaBBBCCCDCE ⇒ aaaaBBBCCBDCE ⇒ aaaaBBBCCBCCE ⇒
aaaaBBBCCDCCE ⇒ aaaaBBBCBDCCE ⇒ aaaaBBBCBCCCE ⇒ aaaaBBBCDCCCE ⇒
aaaaBBBBDCCCE ⇒ aaaaBBBBCCCCE ⇒ aaaaBBBBCCCCE ⇒ aaaaBBBBCCCFE ⇒
aaaaBBBBCCCFc ⇒ aaaaBBBBCCCEc ⇒ aaaaBBBBCCFEc ⇒ aaaaBBBBCCFcc ⇒
aaaaBBBBCCEcc ⇒ aaaaBBBBCFEcc ⇒ aaaaBBBBCFccc ⇒ aaaaBBBBCEccc ⇒
aaaabBBBCEccc ⇒ aaaabbBBCEccc ⇒ aaaabbbBCEccc ⇒ aaaabbbbCEccc ⇒
aaaabbbbCcccc ⇒ aaaabbbbcccc

8. Грамматика является заведомо неоднозначной, так как содержит правила вида $A \to \alpha A \mid \alpha A \beta A \mid \gamma$. Построим в ней цепочку левостороннего вывода для цепочки: if b then if b then a else a

E ⇒ if b then E ⇒ if b then if b then E else E ⇒ if b then if b then a else E ⇒ if b then if b then a else a

Но для той же цепочки возможен другой левосторонний вывод:

E ⇒ if b then E else E ⇒ if b then if b then E else E ⇒ if b then if b then a else E ⇒ if b then if b then a else a

Поскольку существует два различных левосторонних вывода для одной и той же цепочки, следовательно, грамматика является неоднозначной. Если принять во внимание, что грамматика описывает условный оператор, то неоднозначность связана с тем, к какому оператору относить условие else при наличии нескольких вложенных операторов.

Для того же языка можно построить однозначную грамматику:

G'({if,then,else,b,a},{E,F},**P**,E):

P:

E → if b then F else E | if b then E | a

F → if b then F else F | a

Построим вывод для рассмотренной выше цепочки:

E ⇒ if b then E ⇒ if b then if b then F else E ⇒ if b then if b then a else E ⇒ if b then if b then a else a

Глава 2

Ответы на вопросы

1. Основные этапы компиляции: анализ и синтез. Основные фазы компиляции (в порядке их следования): лексический анализ, синтаксический анализ, семантический анализ, подготовка к генерации кода, генерация и оптимизация кода.

2. Любой компилятор является транслятором — это верно. Не любой транслятор является компилятором, но может существовать такой транслятор, который является компилятором. Не может существовать компилятор, который является интерпретатором.
3. Теоретически можно построить компилятор, который не содержит лексический анализатор. Существует два основных способа взаимосвязи лексического и синтаксического анализов: последовательное и параллельное взаимодействие.
4. У любого интерпретатора будет отсутствовать фаза генерации и оптимизации кода, так как интерпретатор не порождает результирующую программу.
5. Язык ассемблера характеризуется простым синтаксисом и простой семантикой, а также отсутствием дополнительных сервисных функций, которые в языках высокого уровня возложены на компилятор и выполняются при подготовке к генерации кода. У компилятора с языка ассемблера отсутствует фаза подготовки к генерации кода, а также отсутствует оптимизация на последней фазе компиляции (генерация кода).
6. Количество проходов, необходимых компилятору для обработки исходной программы, может увеличиться при усложнении синтаксиса или семантики исходного языка. От архитектуры целевой вычислительной системы количество проходов будет зависеть только в том случае, если используются машинно-зависимые методы оптимизации, ориентированные на особенности именно этой архитектуры.
7. В Глобальной сети Интернет используются в основном языки программирования, предусматривающие интерпретацию исходного кода, потому что часто неизвестно, на какой архитектуре целевой вычислительной системы будет выполняться результирующая программа.
8. Наличие переходов и обращений вперед по тексту исходной программы мешает сократить количество проходов компилятора с языка ассемблера до одного.
9. В макроопределении `#define f2(a) ((a)*(a))` параметр `a` взят в скобки, поскольку текстовая подстановка, выполняемая макрокомандой, никак не учитывает приоритет операций. Без использования скобок макрокоманда типа `f2(b+c)` привела бы к подстановке `b+c*b+c`, что дало бы неверный результат.
10. В таблице идентификаторов хранится информация, определяющая характеристики лексических единиц исходной программы.
11. Методы организации таблиц идентификаторов оцениваются исходя из характеристик времени, затрачиваемого на добавление нового элемента в таблицу и на поиск элемента в таблице, при этом характеристика времени поиска элемента в таблице является определяющей.
12. Основные способы организации таблиц идентификаторов: линейный список элементов, упорядоченный список элементов, бинарное дерево, хеш-адресация с рехешированием, хеш-адресация с применением метода цепочек и хеш-адресация в комбинации с одним из простейших методов (линейный список или бинарное дерево).
13. Коллизия — это ситуация, когда двум различным элементам из области определения хеш-функции соответствует одно и то же значение этой функции. Коллизия

происходит при использовании хеш-функций для организации таблиц идентификаторов, так как в этом случае хеш-функция отображает бесконечное множество всех возможных идентификаторов на конечное множество доступных адресов из адресного пространства компьютера.

14. Метод цепочек по сравнению с методом рехеширования позволяет избежать дополнительных коллизий и сократить накладные расходы оперативной памяти, но он более сложен в организации и требует работы с динамической памятью.
15. Метод логарифмического поиска имеет преимущество по сравнению с простым линейным списком, несмотря на то что он увеличивает время, необходимое на добавление нового элемента в таблицу , так как он существенно сокращает время поиска элемента. А поиск в таблицах идентификаторов выполняется намного чаще, чем добавление новых элементов в них.
16. Увеличение адресного пространства за счет использования виртуальной памяти не используется при организации таблиц идентификаторов, так как время обращения к такой памяти будет велико и сведет на нет выигрыш от улучшения качества хеш-функции.

Ответы и решения для задач

1. Результаты выполнения:

```
   k=Inc1(i,1); и k=Inc2(i,1); — результаты совпадают, логических
противоречий нет.

   k= Inc1(i,j); и k= Inc2(i,j); — результаты совпадают, логических
противоречий нет.

   k= Inc1(i,j+1); и k= Inc2(i,j+1); — результаты совпадают,
логических противоречий нет (но при выполнении макрокоманды будет
выполнено на 1 операцию сложения больше).

   k= Inc1(i,j++); и k= Inc2(i,j++); — выполнение макрокоманды даст
результат на 1 больший, чем выполнение функции, при этом возникнет
противоречие, связанное с тем, что значение переменной j будет
увеличено на 2 вместо ожидаемого увеличения на 1.

   k= Inc1(i++,Inc2(j,1)); и k= Inc2(i++,Inc1(j,1)); — результаты
совпадают, логических противоречий нет.
```

2. Результат изменится для макрокоманды k= Inc1(i,j+1), при этом исчезнет и возникшее для данного случая логическое противоречие.
3. Определение макрокоманды #defi ne Inc1(a,b) ((a)+(2*b)) неверно, так как при вызове макрокоманды, например Inc1(x,y+z), оно даст макроподстановку ((x)+(2*y+z)), что приведет к неверному результату.

Глава 3

Ответы на вопросы

1. К регулярным грамматикам относятся грамматики, имеющие правила вида $A \to B\gamma$ или $A \to \gamma$ либо вида $A \to \gamma B$ или $A \to \gamma$, где $A, B \in \mathbf{VN}$, $\gamma \in \mathbf{VT}^*$. Соответственно, существует два класса регулярных грамматик: леволинейные и праволинейные.

2. Разница между автоматными грамматиками и регулярными грамматиками заключается в следующем: там, где в правилах регулярных грамматик может присутствовать цепочка терминальных символов, в автоматных грамматиках может присутствовать только один терминальный символ. Любая автоматная грамматика является регулярной, но не наоборот — не всякая регулярная грамматика является автоматной. Всякая регулярная грамматика может быть преобразована к автоматному виду.
3. Язык, заданный регулярной грамматикой и содержащий пустую цепочку, может быть задан автоматной грамматикой, так как в автоматных грамматиках разрешается дополнительное правило вида $S \to \lambda$, где S — целевой символ грамматики. При этом символ S не должен встречаться в правых частях других правил грамматики.
4. Граф переходов конечного автомата можно использовать для однозначного определения данного автомата.
5. Недетерминированный КА всегда может быть преобразован к детерминированному виду.
6. После преобразования из недетерминированного КА в ДКА автомат может содержать максимум $2^n - 1$ состояний (где n — количество состояний исходного КА). Это количество состояний зачастую можно сократить, но не всегда.
7. Различия таблицы лексем и таблицы идентификаторов.
   - Таблица лексем содержит лексемы всех типов, а таблица идентификаторов — только определенные типы лексем.
   - Таблица лексем содержит лексемы в определенном порядке, а для таблицы идентификаторов порядок не имеет значения.
   - Каждая лексема встречается в таблице лексем столько же раз, сколько в исходной программе, а в таблице идентификаторов — один раз или ни разу.

   Лексический анализатор не должен помещать ключевые слова, разделители и знаки операций в таблицу идентификаторов.
8. Являются ли регулярными множествами следующие множества:
   - множество целых натуральных чисел — является регулярным;
   - множество вещественных чисел — не является регулярным (так как в цифровом представлении могут быть вещественные числа бесконечной длины);
   - множество всех слов русского языка — является регулярным;
   - множество всех возможных строковых констант в языке P ascal — является регулярным;
   - множество иррациональных чисел — не является регулярным;
   - множество всех возможных вещественных констант языка C — является регулярным.
9. Доказательства:

   $(\alpha+\beta)(\delta+\gamma) = (\alpha+\beta)\delta+(\alpha+\beta)\gamma = \alpha\delta+\alpha\gamma+\beta\delta+\beta\gamma;$

   $\delta(\alpha+\beta)\gamma = \delta((\alpha+\beta)\gamma) = \delta(\alpha\gamma+\beta\gamma) = \delta\alpha\gamma+\delta\beta\gamma;$

   $\beta+\beta\alpha+\beta\alpha^* = \beta(\lambda+\alpha+\alpha^*) = \beta(\lambda+(\alpha+\alpha^*)) = \beta(\lambda+\alpha^*) = \beta\alpha^*;$

$\beta+\beta\alpha\alpha^*+\beta\alpha\alpha\alpha^* = \beta(\lambda+\alpha\alpha^*+\alpha\alpha\alpha^*) = \beta(\lambda+\alpha(\alpha^*+\alpha\alpha^*)) = \beta(\lambda+\alpha\alpha^*) = \beta\alpha^*$;

$\delta\alpha\gamma 0^*+\delta\alpha^*\gamma+\delta\gamma = \delta\alpha\gamma\lambda+\delta\alpha^*\gamma+\delta\gamma = \delta\alpha\gamma+\delta\alpha^*\gamma+\delta\gamma = \delta(\alpha\gamma+\alpha^*\gamma+\gamma) = = \delta((\alpha+\alpha^*+\lambda)\times \times\gamma) = \delta((\alpha^*+\lambda)\gamma) = \delta((\alpha^*)\gamma) = \delta\alpha^*\gamma$;

$(0^*+\alpha\alpha^*+\alpha\alpha\alpha^*)^* = (\lambda+\alpha\alpha^*+\alpha\alpha\alpha^*)^* = (\lambda+\alpha(\alpha^*+\alpha\alpha^*))^* = (\lambda+\alpha\alpha^*)^* = (\alpha^*)^* = =\alpha^*$.

10. Истинность тождеств:

    $(\alpha+\beta)^* = (\alpha^*\beta^*)^*$ — верно;

    $(\alpha\beta)^* = \alpha^*\beta^*$ — неверно;

    $(\alpha+\lambda)^* = \alpha^*$ — верно;

    $\alpha^*\beta^*+\beta^*\alpha^* = \alpha^*\beta^*\alpha^*$ — неверно;

    $\alpha^*\alpha^* = \alpha^*$ — верно.

11. Две формы записи уравнений с регулярными коэффициентами существуют, так как операция конкатенации не обладает свойством коммутативности, а потому запись коэффициента слева отличается от его записи справа. Уравнения с коэффициентами слева соответствуют праволинейным грамматикам, а уравнения с коэффициентами справа — леволинейным.

12. Для языка, заданного леволинейной грамматикой, всегда можно построить праволинейную грамматику, задающую эквивалентный язык.

13. Не всякая регулярная грамматика является однозначной. Вот пример неоднозначной регулярной грамматики:

```
G({0,1},{A,B,S},  P,S):
P  1: S → A | B
A   → A0 | A1 | 0 | 1
B   → B0 | B1 | 0 | 1
```

14. Какие из следующих утверждений являются истинными:
    - если язык задан регулярным множеством, то он может быть задан праволинейной грамматикой — верно (свойство регулярных языков);
    - если язык задан КА, то он может быть задан КС-грамматикой — верно (по свойствам регулярных языков он может быть задан регулярной грамматикой, а любая регулярная грамматика является и КС-граматикой);
    - если язык задан КС-грамматикой, то для него можно построить регулярное выражение — неверно (не любая КС-грамматка является регулярной);
    - если язык задан КА, то для него можно построить регулярное выражение —верно (свойство регулярных языков).

15. Для двух леволинейных грамматик можно доказать, что они задают один и тот же язык. Такое же доказательство можно выполнить для леволинейной и праволинейной грамматик.

Ответы и решения для задач

1. Определите, какие из перечисленных ниже грамматик являются регулярными, леволинейными, праволинейными, автоматными:

   $\mathbf{G_1}$ — не является регулярной (это КС-грамматика);

$\mathbf{G}_2$ — не является регулярной (это КС-грамматика);

$\mathbf{G}_3$ — регулярная, праволинейная, но не автоматная;

$\mathbf{G}_4$ — регулярная, леволинейная, но не автоматная;

$\mathbf{G}_5$ — регулярная, леволинейная, автоматная.

2. Преобразуйте к автоматному виду грамматики $\mathbf{G}_3$ и $\mathbf{G}_4$ из задачи 1:

G $_3$'({".",+,−,0,1}, {<число>,<часть>,<осн>,<осн>$_1$}, $\mathbf{P}_3$', <число>):

P $_3$': <число> → +<осн> | −<осн> | <осн>

<осн> → 0 | 1 | 0<осн>$_1$ | 1<осн>$_1$ | 0<осн> | 1<осн>

<осн>$_1$ → .<часть>

<часть> → 0 | 1 | 0<часть> | 1<часть>

G $_4$'({".",+,−,0,1}, {<знак>,<число>,<часть>}, $\mathbf{P}_4$', <число>):

P $_4$': <число> → 0 | 1 | <знак>0 | <знак>1 | <часть>. | <число>0 | <число>1

<часть> → 0 | 1 | <знак>0 | <знак>1 | <часть>0 | <часть>1

<знак> → + | −

3. Постройте КА на основании грамматик $\mathbf{G}_4$ или $\mathbf{G}_5$ из задачи 1. Построим КА $\mathbf{M}_5$ для грамматики $\mathbf{G}_5$. Для удобства возьмем другие обозначения нетерминальных символов грамматики $\mathbf{G}_5$:

G $_5$({".",+,−,0,1}, {Z,S,N,B}, $\mathbf{P}_5$, S):

P $_5$: S → N. | B0 | B1 | N0 | N1

B → N. | B0 | B1

N → 0 | 1 | Z0 | Z1 | N0 | N1

Z → + | −

Получим автомат $\mathbf{M}_5$: M_5({H,Z,S,N,B}, {".",+,−,0,1}, δ, H, {S}), δ(H,+)={Z}, δ(H,-)={Z}, δ(H,0)={N}, δ(H,1)={N}, δ(Z,0)={N}, δ(Z,1)={N}, δ(N,0)={N,S}, δ(N,1)={N,S}, δ(N,.)={B,S}, δ(B,0)={B,S}, δ(B,1)={B,S}

Преобразовав этот КА к детерминированному виду , получим ДКА $\mathbf{M}_5$': M_5'({H,Z,S,N,B}, {".",+,−,0,1}, δ', H, {B,S}), δ'(H,+)=Z, δ'(H,-)=Z, δ'(H,0)=N, δ'(H,1)=N, δ'(Z,0)=N, δ'(Z,1)=N, δ'(N,0)=S, δ'(N,1)=S, δ'(N,.)=B, δ'(B,0)=B, δ'(B,1)=B

4. Задан КА:M({H,I,S,E,F,G}, {b,d}, δ, H, {S}),δ(H,b)={I,S}, δ(H,d)={E}, δ(I,b)={I,S}, δ(I,d)={I,S}, δ(E,b)={E,F}, δ(E,d)={E,F}, δ(F,b)={E}, δ(F,d)={E}, δ(G,b)={F,S}, δ(G,d)={S}.

После устранения недостижимых состояний, преобразования к детерминированному виду и минимизации получим ДКА:

M'({H,S,E}, {b,d}, δ', H, {S}), δ'(H,b)=S, δ'(H,d)=E, δ'(S,b)=S, δ'(S,d)=S, δ'(E,b)=E, δ'(E,d)=E.

5. Задан КА: M({S,R,Z}, {a,b}, δ, S, {Z}), δ(S,a)={S,R}, δ(R,b)={R}, δ(R,a)={Z}.

После преобразования к детерминированному виду и минимизации получим ДКА: M'{S,R,SR,Z,SRZ}, {a,b}, δ', S, {Z,SRZ}), δ'(S,a)=SR, δ'(SR,a)=SRZ, δ'(SR,b)=R, δ'(SRZ,a)=SRZ, δ'(SRZ,b)=R, δ'(R,b)=R, δ'(R,a)=Z.

6. После замены нетерминальных символов в грамматике $\mathbf{G}_4$ получим грамматику

$G_4'(\{".",+,-,0,1\}, \{X_1,X_2,X_3\}, P_4', X4)$

$P_4': X_3 \to X_1 0 \mid X_1 1 \mid X_2 . \mid X_3 0 \mid X_3 1$

$X_2 \to X_1 0 \mid X_1 1 \mid X_2 0 \mid X_2 1$

$X_1 \to \lambda \mid + \mid -$

Для этой грамматики построим систему уравнений с регулярными коэффициентами:

$X_3 = X_1(0+1) + X_2"." + X_3(0+1);$

$X_2 = X_1(0+1) + X_2(0+1);$

$X_1 = (\lambda+"+"+"-").$

Эта система уравнений имеет решение:

$X_3 = (\lambda+"+"+"-")(0+1)(\lambda + (0+1)*".")(0+1)^*;$

$X_2 = (\lambda+"+"+"-")(0+1)(0+1)^*;$

$X_1 = (\lambda+"+"+"-").$

Если посмотреть на решение, найденное для целевого символа X_3, то видно, что эта грамматика задает язык двоичных чисел с фиксированной точкой со знаком.

7. Для доказательства подставим решение $X = \alpha^*(\beta+\gamma)$ в правую часть уравнения $X=\alpha X+\beta$: $\alpha X+\beta = \alpha(\alpha^*(\beta+\gamma))+\beta = \alpha\alpha^*(\beta+\gamma)+\beta = (\delta+\lambda)(\delta+\lambda)^*(\beta+\gamma)+\beta = = (\delta+\lambda)\delta^*(\beta+\gamma)+\beta=\delta\delta^*(\beta+\gamma)+\delta^*(\beta+\gamma)+\beta=\delta\delta^*\beta+\delta\delta^*\gamma+\delta^*\beta+\delta^*\gamma+\beta==\delta\delta^*\beta+\delta^*\beta+\beta+\delta\delta^*\gamma+\delta^*\gamma=(\delta\delta^*+\delta^*+\lambda)\beta+(\delta\delta^*+\delta^*)\gamma=(\delta\delta^*+\delta^*+\lambda)\beta+(\delta\delta^*+\delta^*+\lambda)\gamma==((\delta+\lambda)\delta^*+\lambda)\beta+((\delta+\lambda)\delta^*+\lambda)\gamma=((\delta+\lambda)\alpha^*+\lambda)\beta+((\delta+\lambda)\alpha^*+\lambda)\gamma==(\alpha\alpha^*+\lambda)\beta+(\alpha\alpha^*+\lambda)\gamma=\alpha^*\beta+\alpha^*\gamma=\alpha^*(\beta+\gamma)=X.$

8. Решение системы уравнений:

$X_1 = (01^*01^*0+1)^*;$

$X_2 = 1^*01^*0(01^*01^*0+1)^*;$

$X_3 = 1^*0(01^*01^*0+1)^*.$

9. Если обозначить символом «v» знак конца строки, а символом «a» все алфавитно-цифровые символы, кроме символов / и *, то получим следующую грамматику **G** для языка комментариев C++:

$G(\{/,*,a,v\}, \{K,C,S\}, P, S):$

$P: S \to Kv \mid C*/$

$K \to // \mid K/ \mid K* \mid Ka$

$C \to /* \mid C/ \mid C* \mid Ca \mid Cv$

Преобразовав эту грамматику к автоматному виду, получим грамматику

$G'(\{/,*,a,v\}, \{K,K_1,C,C_1,S_1,S\}, P, S):$

$P: S \to Kv \mid S_1/$

$S_1 \to C*$

$K \to K_1/ \mid K/ \mid K* \mid Ka$

$K_1 \to /$

$C \to C_1* \mid C/ \mid C* \mid Ca \mid Cv$

$C_1 \to /$

Глава 4

Ответы на вопросы

1. Какие из следующих утверждений справедливы:
   - если язык задан КС-грамматикой, то он может быть задан с помощью МП-автомата — верно;
   - если язык задан КС-грамматикой, то он может быть задан с помощью ДМП-автомата — неверно (не для любого КС-языка существует ДМП-автомат);
   - если язык задан КА, то он может быть задан КС-грамматикой — верно (может быть задан регулярной грамматикой, а следовательно, и КС-грамматикой);
   - если язык задан ДМП-автоматом, то он может быть задан КС-грамматикой — верно;
   - если язык задан однозначной КС-грамматикой, то для него можно построить КА — неверно (если язык задан однозначной КС-грамматикой, он не обязательно является регулярным языком);
   - если язык задан однозначной КС-грамматикой, то он может быть задан с помощью ДМП-автомата — верно;
   - если язык задан расширенным МП-автоматом, то он может быть задан КС-грамматикой — верно.
2. Детерминированными являются МП-автоматы: $\mathbf{R}_1$ и $\mathbf{R}_4$; не являются детерминированными: $\mathbf{R}_2$ (так как $\delta_2(q,a,A)=\{(q,AA)\}$ и $\delta_2(q,\lambda,A)=\{(q,A)\}$), $\mathbf{R}_3$ (так как $\delta_3(q,a,A)=\{(q,AA),(q,a),(q,\lambda)\}$).
3. Синтаксические конструкции языков программирования могут быть распознаны с помощью ДМП-автоматов, так как они всегда должны быть заданы однозначной КС-грамматикой, а для такой грамматики всегда существует ДМП-автомат, задающий тот же язык.
4. Структуру входной программы с помощью ДМП-автоматов для большинства языков программирования полностью проверить нельзя, так как многие языки программирования являются КЗ-языками. По этой же причине нельзя решить эту же задачу, используя МП-автоматы или расширенные МП-автоматы.
5. Преобразование правил КС-грамматик не всегда ведет к упрощению правил, наоборот, после преобразования правила грамматики зачастую усложняются, а их количество увеличивается, если исходная грамматика является приведенной. Целью преобразования является построение распознавателя, а не упрощение правил.
6. При преобразовании КС-грамматики к приведенному виду сначала необходимо удалить бесплодные символы, а потом — недостижимые символы, так как при удалении бесплодных символов в грамматике могут появиться новые недостижимые символы.
7. Для языка, заданного КС-грамматикой, содержащей λ-правила, существуют трудности с моделированием работы расширенного МП-автомата, выполняющего восходящий разбор с правосторонним выводом, поскольку в этом случае наличие функции перехода вида $\delta(q,\lambda,\lambda)$ ведет к зацикливанию программы, моделирующей работу МП-автомата.

8. Необходимо устранить именно левую рекурсию из правил грамматики, поскольку все реально существующие распознаватели выполняют разбор исходной программы слева направо, а не справа налево. Левая рекурсия представляет сложности для моделирования работы нисходящего распознавателя.
9. Полностью устранить рекурсию из правил грамматики, записанных в форме Бэкуса—Наура, невозможно, так как рекурсия — основной метод построения бесконечного множества цепочек языка на основе конечного множества правил грамматики в форме Бэкуса—Наура.
10. Метод рекурсивного спуска основан на алгоритме работы нисходящего распознавателя с подбором альтернатив.
11. Распознаватель по методу расширенного рекурсивного спуска для грамматики, содержащей левую рекурсию, реализовать невозможно.
12. Класс LL(1)-грамматик является более широким, чем класс КС-грамматик, для которых можно построить распознаватель по методу рекурсивного спуска, так как LL(1)-грамматики иногда допускают λ-правила, а также правила, имеющие одинаковую левую часть и начинающиеся с различных нетерминальных символов.
13. Работа распознавателя для LL(k)-грамматики основана на алгоритме нисходящего распознавателя с подбором альтернатив.
14. Работа распознавателя для LR(k)-грамматики основана на алгоритме «сдвиг—свертка».
15. В грамматиках простого и операторного предшествования не могут присутствовать λ-правила, так как работа распознавателей для этих грамматик основана на алгоритме «сдвиг—свертка», который не допускает таких правил.
16. Несмотря на то что класс языков, заданных LR(1)-грамматиками, совпадает с классом ДКС-языков, используются и другие классы грамматик, так как проблема преобразования для КС-грамматик в общем случае неразрешима, и это может затруднять построение LR(1)-грамматики для заданного языка.

Ответы и решения для задач

1. Обычный МП-автомат для заданной грамматики:

   R_1({q}, {".",+,-,0,1}, {".",+,-,0,1, <число>, <часть>, <цифра>, <осн>}, δ_1, q, <число>, {q}),

   δ_1(q,.,.)={(q,λ)}, δ_1(q,+,+)={(q,λ)}, δ_1(q,-,-)={(q,λ)}, δ_1(q,0,0)={(q,λ)}, δ_1(q,1,1)={(q,λ)}, δ_1(q,λ,<число>)={(q,+<осн>),(q,-<осн>),(q,<осн>)}, δ_1(q,λ,<осн>)={(q,<часть>.<часть>),(q,<часть>.),(q,<часть>)}, δ_1(q,λ,<часть>)={(q,<цифра>),(q,<часть><цифра>)}, δ_1(q,λ,<цифра>)={(q,0),(q,1)}

   Расширенный МП-автомат для заданной грамматики:

   R_2({q}, {".",+,-,0,1}, {".",+,-,0,1, <число>, <часть>, <цифра>, <осн>}, δ_2, q, λ, {q}),

   δ_2(q,.,λ)={(q,.)}, δ_2(q,+,λ)={(q,+)}, δ_2(q,-,λ)={(q,-)}, δ_2(q,0,λ)={(q,0)}, δ_2(q,1,λ)={(q,1)}, δ_2(q,λ,+<осн>)={(q,<число>)},

δ_2(q,λ,-<осн>)={(q,<число>)}, δ_2(q,λ,<осн>)={(q,<число>)}, δ_2(q,λ,<часть>.<часть>)={(q,<осн>)}, δ_2(q,λ,<часть>.)={(q,<осн>)}, δ_2(q,λ,<часть>)={(q,<осн>)}, δ_2(q,λ,<цифра>)={(q,<часть>)}, δ_2(q,λ,<часть><цифра>)={(q,<часть>)}, δ_2(q,λ,0)={(q,<цифра>)}, δ_2(q,λ,1)={(q,<цифра>)}

Очевидно, что эти два автомата являются недетерминированными.

2. Результирующая приведенная грамматика:

   G'({a,b,c},{A,B,C,S},**P'**,S):

   P': S → aAbB

   A → BCa | a | λ

   B → ACb | b | λ

   C → A | B | bA | aB | cC

3. Грамматика из задачи 2 после устранения λ-правил:

   G"({a,b,c},{A,B,C,S},**P»**,S):

   P": S → aAbB | aAb | abB | ab

   A → BCa | a | Ca | Ba

   B → ACb | b | Cb | Ab

   C → A | B | bA | aB | cC | b | a | c

4. Исходная грамматика без цепных правил:

   G'({" ",(,),o,r,a,n,d,t,b},{S,T,E,F},**P'**,S):

   P': S → S or T | T and E | not E | (S) | b

   T → T and E | not E | (S) | b

   E → not E | (S) | b

   F → (S) | b

   Исходная грамматика без левой рекурсии:

   G"({" ",(,),o,r,a,n,d,t,b},{S,S',T,T',E,F},**P»**,S):

   P": S → T | TS'

   S' → or T | or TS'

   T → E | ET'

   T' → and E | and ET'

   E → not E | F

   F → (S) | b

5. Разбор цепочки символов `if b then if b then if b then a else a` для заданной грамматики:

   S ⇒ if b then S ⇒ if b then if b then S ⇒ if b then if b then if b then T else S⇒ if b then if b then if b then a else S⇒ if b then if b then if b then a else a

   Это левосторонний разбор, из него видно, что `else` относится к последнему `if` в цепочке.

Глава 5

Ответы на вопросы

1. Справедливы ли следующие утверждения:
   - любой язык программирования является КС-языком — неверно;
   - синтаксис любого языка программирования может быть описан с помощью КС-грамматики — верно;
   - любой язык программирования является КЗ-языком — верно;
   - семантика любого языка программирования может быть описана с помощью КС-грамматики — неверно.
2. Компилятор можно построить без семантического анализатора, если исходный язык может быть задан КС-грамматикой и не предусматривает преобразований типов.
3. Несмотря на то что цепочка символов, принадлежащая любому языку программирования, может быть распознана с помощью распознавателя для КЗ-языков, на практике в компиляторах такие распознаватели не используют из-за экспоненциальной зависимости времени разбора от длины входной цепочки.
4. Совпадают ли значения:
   - `i = x / y;` и `i = (int)(x / y);` — совпадают;
   - `i = x / y;` и `i = ((int)x) / ((int)y);` — не совпадают;
   - `i = j / k;` и `i = ((double)j) / (double)k);` — не совпадают;
   - `z = x / y;` и `z=((int)x) / ((int)y);` — не совпадают;
   - `z = j / k;` и `z = (double)(j / k);` — совпадают;
   - `z = j / k;` и `z = ((double)j) / ( (double)k)` — не совпадают.

   Функции неявного преобразования типов:
   - `i = x / y;` — преобразование вещественного результата (`double`) к целому (`int`);
   - `i = x / y;` — преобразование вещественного результата (`double`) к целому (`int`);
   - `i = ((double)j) / (double)k);` — преобразование вещественного результата (`double`) к целому (`int`);
   - `z = ((int)x) / ((int)y);` — преобразование целого результата (`int`) к вещественному (`double`);
   - `z = j / k;` — преобразование целого результата (`int`) к вещественному (`double`);
   - `z = j / k;` — преобразование целого результата (`int`) к вещественному (`double`).
5. Даже если разработчик пользуется одним и тем же компилятором, ему лучше не использовать внутреннее имя функции, так как нет гарантии, что оно сохранится в последующих версиях даже того же самого компилятора. Разработчик должен описать функцию как экспортную (на языке C++ это делается как `export void ftest(int i)`).

6. Для организации вложенных процедур и функций в языке Pascal с помощью стекового дисплея памяти во вложенную процедуру в качестве параметра должно передаваться значение базового регистра объемлющей процедуры. Тогда адресация локальных данных объемлющей процедуры во вложенной процедуре происходит на основе этого значения.
7. Состав информации, хранящейся в таблице RTTI, жестко не регламентируется ни синтаксисом, ни семантикой исходного языка и зачастую остается на усмотрение разработчиков компилятора. Кроме того, информация в таблице RTTI может зависеть от архитектуры целевой вычислительной системы.
8. Можно построить компилятор, исключив фазу оптимизации кода. Такой компилятор будет проигрывать только в эффективности результирующего кода.
9. Эффективность объектного кода, построенного компилятором, зависит от качества используемых схем СУ-перевода и качества оптимизации кода в компиляторе. Синтаксис и семантика исходного языка программирования обычно не влияют на эффективность результирующего кода.
10. СУ-перевод не является наиболее эффективным методом порождения результирующего кода, его использование в компиляторах — это вынужденное упрощение.
11. Компилятор обязательно должен уметь обрабатывать машинные коды, а также коды ассемблера, если он допускает порождение результирующей программы на языке ассемблера.
12. Трудности с оптимизацией выражений, представленных в форме обратной польской записи, связаны с тем, что промежуточные результаты их вычислений хранятся в стеке (при этом доступ возможен лишь к верхушке стека).
13. Более эффективный результирующий код может порождать машинно-зависимая оптимизация, так как она ориентирована на конкретную целевую вычислительную систему. Обычно в компиляторе используются оба метода оптимизации: и машинно-зависимый, и машинно-независимый.
14. Оператор: `if (i<10) and (A[i]=0) then …` будет неправильным, так как не учитывает индексы, меньшие 0. Оператор: `if (i<=0) or (i>=10) or (A[i]=0) then …` с этой точки зрения выполняется правильно. Если компилятор не будет выполнять оптимизацию логических выражений, то и второй оператор будет выполняться неверно.
15. Значения `i` после выполнения оператора языка C:
    - 9 → 12;
    - 11 → 11;
    - 21 → 21;
    - 0 → 3;
    - –3 → –1.

    Значения той же переменной при условии, что компилятор не выполняет оптимизацию логических выражений:
    - 9 → 13;
    - 11 → 14;
    - 21 → 24;

- 0 → 4:
- –3 → 1.

Ответы и решения для задач

1. Содержимое стека представлено в виде таблиц

Таблица после одного вызова функций B и A

Содержимое стека
адрес возврата процедуры B
базовый регистр процедуры A
локальная переменная c процедуры B
локальная переменная b процедуры B
локальная переменная a процедуры B
параметр j процедуры A
параметр i процедуры A
адрес возврата процедуры A
базовый регистр процедуры B
локальная переменная t процедуры A

Таблица после двух вызовов функций B и A

Содержимое стека
адрес возврата процедуры B
базовый регистр процедуры A
локальная переменная c процедуры B
локальная переменная b процедуры B
локальная переменная a процедуры B
параметр j процедуры A
параметр i процедуры A
адрес возврата процедуры A
базовый регистр процедуры B
локальная переменная t процедуры A
адрес возврата процедуры B
базовый регистр процедуры A
локальная переменная c процедуры B
локальная переменная b процедуры B
локальная переменная a процедуры B
параметр j процедуры A
параметр i процедуры A
адрес возврата процедуры A
базовый регистр процедуры B
локальная переменная t процедуры A

2. Для указанных переменных потребуется 61 байт памяти. При выравнивании данных по границе 2 байта потребуется 66 байт , если данные не выравниваются внутри структур, и 68 байт , если данные выравниваются внутри структур. При

выравнивании данных по границе 4 байта потребуется 76 байт , если данные не выравниваются внутри структур, и 136 байт, если данные выравниваются внутри структур.

3. Результаты вычисления выражений, записанных в форме обратной польской записи:
   - `25;`
   - `63;`
   - `10;`
   - `0.`

   Выражения в форме инфиксной (обычной) записи:
   - `1*2+(3+5*4);`
   - `1+(2+3*(5*4));`
   - `(1+2)*3+5-4;`
   - `(1+2)*3-(5+4).`
4. Последовательность операций в форме триад:
   1. `:=(A,2);`
   2. `*(A,3);`
   3. `:=(B,^2);`
   4. `*(4,3);`
   5. `+(^4,1);`
   6. `:=(C,^5);`
   7. `:=(A,D);`
   8. `+(B,C);`
   9. `:=(D,^8);`
   10. `:=(C,A);`
   11. `*(4,C);`
   12. `+(A,^11);`
   13. `+(D,12);`
   14. `*(3,^13);`
   15. `+(^12,^14);`
   16. `:=(B,^15).`

   Последовательность операций в форме триад после выполнения свертки операций:
   1. `:=(A,2);`
   2. `:=(B,6);`
   3. `:=(C,13);`
   4. `:=(A,D);`
   5. `:=(D,19);`
   6. `:=(C,A);`
   7. `*(4,C);`
   8. `+(A,^7);`
   9. `+(^8,93);`
   10. `:=(B,^9).`

5. Последовательность операций в форме триад:

```
1.  *(B,C);
2.  *(D,E);
3.  +(^1,^2);
4.  :=(A,^3);
5.  *(B,C);
6.  *(D,E);
7.  +(^5,^6);
8.  :=(B,^7);
9.  *(B,C);
10. *(D,E);
11. +(^9,^10);
12. :=(C,^11);
13. *(B,C);
14. *(D,E);
15. +(^14,2);
16. –(^13,^15);
17. :=(D,^16);
18. *(B,C);
19. *(D,E);
20. +(^19,3);
21. –(^18,^20);
22. :=(E,^21);
23. *(A,E);
24. *(B,E);
25. +(^23,^24);
26. +(^25,1);
27. :=(A,^26);
28. *(A,E);
29. *(B,E);
30. +(^28,^29);
31. +(^30,1);
32. :=(B,^31).
```

Последовательность операций в форме триад после исключения лишних операций:

```
1.  *(B,C);
2.  *(D,E);
3.  +(^1,^2);
4.  :=(A,^3);
5.  :=(B,^3);
6.  *(B,C);
7.  +(^6,^2);
```

```
8.  :=(C,^7);
9.  *(B,C);
10. +(^2,2);
11. –(^9,^10);
12. :=(D,^11);
13. *(D,E);
14. +(^13,3);
15. –(^9,^14);
16. :=(E,^15);
17. *(A,E);
18. *(B,E);
19. +(^17,^18);
20. +(^19,1);
21. :=(A,^20);
22. *(A,E);
23. +(^22,^18);
24. +(^23,1);
25. :=(B,^24).
```

Глава 6

Ответы на вопросы

1. Укажите, что из перечисленного является составной частью системы программирования:
   - лексический анализатор — нет (это часть компилятора);
   - компоновщик — да;
   - редактор ресурсов пользовательского интерфейса — да;
   - исходная программа — нет;
   - компилятор с исходного языка — да;
   - компилятор ресурсов — да;
   - целевая вычислительная система — нет;
   - библиотеки исходного языка — да;
   - библиотеки операционной системы — нет;
   - отладчик — да;
   - файл подсказки — да.
2. Обработчик языка Makefile является интерпретатором.
3. Application Program Interface (API) — прикладной программный интерфейс (или интерфейс прикладного программирования). Предназначен для использования прикладными программами системных ресурсов ОС и реализуемых ею функций. API описывает совокупность функций и процедур, принадлежащих ядру или надстройкам ОС.

4. Относятся ли к понятию «ресурсы прикладной программы» следующие данные:
   - текст сообщения, выдаваемого программой при возникновении ошибки, — да;
   - динамические библиотеки, используемые программой, — нет;
   - тексты пунктов главного меню программы — да;
   - статические библиотеки, используемые программой — нет;
   - файлы, обрабатываемые прикладной программой — нет;
   - цвет фона главного окна программы — да;
   - структуры данных, используемые программой в процессе работы, — нет;
   - файл подсказки, выдаваемой программой по запросу пользователя, — да.
5. Какие из перечисленных функций выполняет текстовый редактор в современных системах программирования:
   - выдача контекстной подсказки по функциям библиотек системы программирования — да;
   - анализ и выделение (цветом или шрифтом) лексем исходной программы в процессе ее подготовки — да;
   - выдача контекстной подсказки по функциям исходной программы в процессе ее подготовки — нет;
   - позиционирование на место ошибки, обнаруженной в процессе компиляции, — да;
   - позиционирование на место предупреждения, обнаруженного в процессе компиляции, — да;
   - исправление по команде разработчика текста исходной программы в соответствии с типом предупреждения, обнаруженного в процессе компиляции, — нет;
   - выдача контекстной подсказки по операторам входного языка — да.
6. Предпочтение будет отдано совершенствованию сервисных средств и развитого интерфейса пользователя, так как в настоящее время это направление является определяющим на рынке современных систем программирования.
7. Особенности функционирования компилятора в составе системы программирования по сравнению с его функционированием в виде отдельного программного модуля связаны с тем, что компилятор должен взаимодействовать с другими модулями, входящими в состав системы программирования.
8. Компоновщик обрабатывает только относительные адреса.
9. Загрузчик работает с относительными и логическими адресами: он преобразует относительные адреса в логические. А преобразование логических адресов в физические выполняет уже ОС в ходе выполнения результирующей программы.
10. Выполнение отладки результирующей программы путем моделирования архитектуры целевой вычислительной системы отличается от интерпретации тем, что и в этом случае компилятор обрабатывает всю исходную программу целиком, порождает результирующую программу, которая потом идет в отладку. В то вре-

мя как интерпретатор выполняет исходную программы не всю сразу , а по мере поступления ее на вход и не порождает результирующей программы.

11. Динамически загружаемые библиотеки позволяют более эффективно использовать оперативную память по сравнению со статически подключаемыми библиотеками. Но использование динамических библиотек снижает переносимость результирующей программы и может привести к конфликтам.
12. За подключение динамически загружаемой библиотеки к результирующей программе отвечает динамический загрузчик, который в современных системах является частью ОС.
13. Статически подключаемая библиотека должна быть доступна системе программирования в момент компоновки результирующей программы, а в момент компиляции в этом нет необходимости.
14. Нет необходимости в доступе к динамически подключаемой библиотеке ни в момент компиляции, ни в момент компоновки результирующей программы. Такая библиотека должна быть доступна только во время выполнения программы.
15. Лексический анализ исходного текста программы во время ее подготовки («на лету») позволяет разработчику этой программы непосредственно увидеть все лексемы в ходе подготовки программы, а также получать подсказку по типам переменных и функций. Реализация такого анализа повышает скорость работы компилятора, так как при начале компиляции исходной программы ее лексический анализ уже выполнен. Это может вести к сокращению количества проходов, выполняемых компилятором.
16. Преимущества приложений, созданных в архитектуре «клиент—сервер», по сравнению с приложениями архитектуры «файл—сервер»:
    - СУБД обеспечивает надежные и гибкие механизмы разделения доступа к данным, а также их защиты от несанкционированного доступа, удовлетворяющие общепризнанным стандартам;
    - все функции по управлению данными выполняются на сервере данных, что снижает требования к вычислительным ресурсам клиентских компьютеров, на которых выполняются клиентские приложения;
    - для обмена данными между клиентским приложением и сервером данных через сеть передаются не все данные, а только запросы клиента и ответы сервера, что снижает нагрузку на сеть;
    - при необходимости увеличить количество клиентов в системе достаточно включить в сеть новых клиентов и, если необходимо, увеличить мощность сервера и пропускную способность сети — нет необходимости обновлять программное и аппаратное обеспечение уже существующих клиентов.
17. Отличие «толстого клиента», функционирующего в составе архитектуры «клиент—сервер», от «тонкого клиента», функционирующего в составе многозвенной архитектуры, заключается в том, что «толстый клиент» выполняет операции, связанные с обработкой данных, в то время как «тонкий клиент» выполняет только функции, связанные с интерфейсом пользователя. Поэтому требования к вычислительным ресурсам «толстого клиента» выше, чем к вычислительным ресурсам «тонкого клиента».

18. Сервер приложений функционировать без сервера данных не может . В принципе возможно, что оба сервера будут размещены на одном компьютере в составе сети, но и в этом случае сервер данных необходим для функционирования сервера приложений.

19. В сети Интернет часто используются интерпретируемые языки, поскольку в Глобальной сети часто неизвестно, на какой целевой вычислительной системе будет выполняться программа.

Ответы и решения для задач

1. Причина ошибки может быть либо в том, что при создании библиотеки функция `F1` не была объявлена как экспортная, либо в том, что при создании исходного текста новой программы эта функция `F1` не была описана как внешняя. В первом случае разработчику надо изменить исходный текст библиотеки, заново откомпилировать и собрать ее (если библиотека внешняя — обратиться к ее поставщику). Во втором случае разработчику надо в тексте исходной программы описать функцию `F1` как внешнюю библиотечную функцию и заново откомпилировать исходную программу.

2. Причина ошибки заключается в том, что соглашение о вызове функции `F1`, указанное в библиотеке, не совпадает с соглашением о вызове, используемом в результирующей программе. Разработчику необходимо узнать, какое соглашение о вызове требуется для функции `F1`, в текст исходной программы добавить описание соглашения о вызове, заново откомпилировать и собрать результирующую программу.

3. Предложенная схема не приведет к возникновению ошибок, если в динамически загружаемой библиотеке не используются в явном виде функции проверки соответствия типов для `C1` (см. решение следующей задачи). При использовании разных систем программирования ситуация кардинально изменится, и возможны любые ошибки, так как нет гарантии, что в разных системах программирования будут совпадать форматы таблиц RTTI. В этом случае разработчику нельзя напрямую передавать и использовать в динамической библиотеке экземпляр класса `C1`, а необходимо использовать какой-то интерфейс взаимодействия с внешними объектами и классами.

4. Указанная схема приведет к тому , что динамически загружаемая библиотека не будет правильно опознавать принадлежность экземпляра объекта к классу `C1` из-за разных адресов RTTI у результирующей программы и библиотеки (к ошибкам выполнения программы это может и не привести, но вот логика ее работы нарушится). Разработчику программы следует либо использовать другой способ проверки принадлежности экземпляра объекта к классу (например, по наименованию класса), либо перейти от прямой передачи экземпляра объекта в библиотеку к использованию какого-то интерфейса взаимодействия с внешними объектами и классами.

Указатель литературы

1. *Абрамова Н. А.* Новый математический аппарат для анализа внешнего поведения и верификации программ / Н. А. Абрамова [и др.]. — М.: Институт проблем управления РАН, 1998. — 109 с.
2. *Архангельский А. Я.* Русская справка (HELP) по Delphi 5 и Object P ascal / А. Я. Архангельский [и др.]. — М.: БИНОМ, 2000. — 32 с.
3. *Афанасьев А. Н.* Формальные языки и грамматики: учеб. пособие. — У льяновск: УлГТУ, 1997. — 84 с.
4. *Ахо А., Ульман Дж.* Теория синтаксического анализа, перевода и компиляции. — М.: Мир, 1978. — т.1, 612 с. — т.2, 487 с.
5. *Ахо А.* Компиляторы: принципы, технологии и инструментарий / А. Ахо [и др.]. — М.: Издательский дом «Вильямс», 2008. — 1184 с.
6. *Бартеньев О. В.* Фортран для студентов. — М.: Диалог-МИФИ, 1999. — 342 с.
7. *Березин Б. И., Березин С. Б.* Начальный курс С и С++. — М.: Диалог-МИФИ, 1996. — 288 с.
8. *Бранденбау Дж.* JavaScript: Сборник рецептов для профессионалов. — СПб.: Питер, 2000. — 416 с.
9. *Браун С.* Операционная система UNIX. — М.: Мир, 1986. — 463 с.
10. *Вендров А. М.* CASE-технологии. Современные методы и средства проектирования информационных систем. — М.: Финансы и статистика, 1998 — 176 с.
11. *Вирт Н.* Алгоритмы и структуры данных. — М.: Мир, 1989. — 360 с.
12. *Волкова И. А., Руденко Т. В.* Формальные языки и грамматики. Элементы теории трансляции. — М.: Диалог-МГУ, 1999. — 62 с.
13. *Гончаров А.* Самоучитель HTML. — СПб.: Питер, 2000. — 240 с.
14. *Гордеев А. В.* Операционные системы: учебник для вузов. — СПб.: Питер, 2004. — 416 с.
15. *Гордеев А. В., Молчанов А. Ю.* Системное программное обеспечение. — СПб.: Питер, 2001. — 734 с.
16. *Гослинг Д Арнольд К.* Язык программирования Java. — СПб.: Питер, 1997. — 565 с.
17. *Гордеев А. В.* Дисковая операционная система реального времени /А. В. Гордеев, Н. Н. Решетникова, А. П. Соловьев. — СПб.: ГААП, 1994. — 44 с.

18. *Грис Д.* Конструирование компиляторов для цифровых вычислительных машин. — М.: Мир, 1975. — 544 с.
19. *Гуннерсон Э.* Введение в C#. Библиотека программиста. — СПб.: Питер, 2001. — 603 с.
20. *Ван Тассел. Д.* Стиль, разработка, эффективность, отладка и испытание программ. — М.: Мир, 1985. — 332 с.
21. *Дворянкин А. И.* Основы трансляции: учеб. пособие. — Волгоград: ВолгГТУ , 1999. — 80 с.
22. *Дунаев С.* UNIX System V. Release 4.2. Общее руководство. — М.: Диалог-МИФИ, 1995. — 287 с.
23. *Евстигнеев В. А., Мирзуитова И. П.* Анализ циклов: выбор кандидатов на распараллеливание. — Новосибирск: Ин-т систем информатики, 1999. — 48 с.
24. *Жаков В. И.*Синтаксический анализ и генерация кода / В. И. Жаков, В. В. Коровинский, В. В. Фильчаков. — СПб.: ГААП, 1993. — 26 с.
25. *Калверт Ч.* Delphi 4. Энциклопедия пользователя. — Киев: ДиаСофт , 1998. — 800 с.
26. *Карпов Б. И.* Delphi: Специальный справочник. — СПб.: Питер, 2001. — 684 с.
27. *Карпов Б. И.* Visual Basic 6: Специальный справочник. — СПб.: Питер, 2000. — 415 с.
28. *Карпов Б. И., Баранова Т . К.* C++: Специальный справочник. — СПб.: Питер, 2002. — 480 с.
29. *Карпов Ю. Г .* Теория автоматов: учебник для вузов. — СПб.: Питер, 2003. — 208 с.
30. *Карпова Т . С .* Базы данных: модели, разработка, реализация. — СПб.: Питер, 2001. — 304 с.
31. *Керниган Б., Пайк Р.* UNIX — универсальная среда программирования. — М.: Финансы и статистика, 1992. — 420 с.
32. *Кисилев С. Ю.* Технология разработки программного обеспечения информационных систем. — СПб.: СПбГУАП, 1998. — 102 с.
33. *Клочко В. И.* Теория вычислительных процессов и структур: учеб. пособие. — Краснодар: Изд-во КубГТУ, 1999. — 118 с.
34. *Компаниец Р. И.* Системное программирование. Основы построения трансляторов: учеб. пособие для высших и средних учебных заведений / Р. И. Компаниец, Е. В. Маньков, Н. Е. Филатов.— СПб.: КОРОНА принт, 2000. — 256 с.
35. *Костельцев А. В .* Построение интерпретаторов и компиляторов. — СПб.: Наука и Техника, 2001. — 342 с.
36. Концептуальное моделирование информационных систем / под ред. В. В. Фильчакова. — СПб.: СПВУРЭ ПВО, 1998. — 356 с.
37. *Кэнту М.* Delphi 5 для профессионалов. — СПб.: Питер, 2001. — 944 с.
38. *Льюис Ф.* Теоретические основы построения компиляторов / Ф. Льюис [и др.]. — М.: Мир, 1979. — 483 с.
39. *Майерс Дж.* Надежность программного обеспечения. — М.: Мир, 1987. — 360 с.

40. *Маклаков С. В.* BPWin и ERWin. CASE-средства разработки информационных систем. — М.: Диалог-МИФИ, 2000. — 254 с.
41. *Мельников Б. Ф.* Подклассы класса контекстно-свободных языков. — М.: Изд-во МГУ, 1995. — 174 с.
42. *Молчанов А. Ю* . Системное программное обеспечение. Лабораторный практикум. — СПб.: Питер, 2005. — 284 с.
43. *Немнюгин С., Перколаб Л.* Изучаем Turbo Pascal. — СПб.: Питер, 2000. — 320 с.
44. *Непомнящий В. А., Рякин О. М.* Прикладные методы верификации программ / под ред. А. П. Ершова. — М.: Радио и связь, 1988. — 256 с.
45. *Олифер В. Г., Олифер Н. А.* Сетевые операционные системы: учебник для вузов. — СПб.: Питер, 2008. — 672 с.
46. Операционная система СМ ЭВМ РАФОС: справочник / под ред. В. П. Семика. — М.: Финансы и статистика, 1984. — 207 с.
47. *Павловская Т. А.* C/C++: Программирование на языке высокого уровня: Учебник для вузов. — СПб.: Питер, 2003. — 464 с.
48. *Петруцос Э., Хау К.* Visual Basic 6 и VBA. — СПб.: Питер, 2000. — 425 с.
49. *Полетаева И. А.* Методы трансляции: конспект лекций. — Новосибирск: Изд-во НГТУ, 1998. — Ч. 2 — 51 с.
50. *Пратт Т., Зелковиц М.* Языки программирования: разработка и реализация. — СПб.: Питер, 2002. — 688 с.
51. Программирование web-сервисов для .NET. Библиотека Программиста / А. Феррара, М. Мак-Дональд. — Киев: BHV; СПб.: Питер, 2003 — 430 с.
52. Разработка распределенных приложений на платформе Microsoft .Net Framework: учебный курс Microsoft. —М.: Русская редакция; СПб.: Питер, 2008 — 608 с.
53. *Рассел Ч., Кроуфорд Ш.* UNIX и Linux: книга ответов. — СПб.: Питер, 1999. — 297 с.
54. *Рогаткин Д., Федоров А.* Borland Pascal в среде Windows. — Киев: Диалектика, 1993. — 511 с.
55. *Рейчард К., Фостер-Джонсон Э.* UNIX: справочник. — СПб.: Питер, 2000. — 384 с.
56. *Рудаков П. И., Федотов М. А.* Основы языка Pascal: учебный курс. — М.: Радио и связь: Горячая линия — Телеком, 2000. — 205 с.
57. C/C++. Структурное программирование: практикум / Т . А. Павловская, Ю. А. Щупак. — СПб.: Питер, 2002. — 240 с.
58. *Свердлов С. З.* Языки программирования и методы трансляции: учеб. пособие. — СПб.: Питер, 2007. — 638 с.
59. *Серебряков В. И.* Лекции по конструированию компиляторов. — М.: МГУ1997. — 171 с.
60. *Страуструп Б.* Язык программирования Си++. — М.: Радио и связь, 1991. — 348 с.
61. *Таненбаум Э.* Компьютерные сети. — СПб.: Питер, 2002. — 848 с.
62. *Таненбаум Э.* Современные операционные системы. — СПб.: Питер, 2007. — 1038 с.

63. *Федоров В. В.* Основы построения трансляторов: учеб. пособие. — Обнинск: ИАТЭ, 1995. — 105 с.
64. *Финогенов К. Г.* Основы языка ассемблера. — М.: Радио и связь, 1999. — 288 с.
65. *Фомин В. В.* Математические основы разработки трансляторов: учеб. пособие. — СПб.: ИПЦ СПГУВК, 1996 — 65 с.
66. *Чернышов А. В.* Инструментальные средства программирования из состава ОС UNIX и их применение в повседневной практике. — М.: Изд-во МГУП, 1999. — 191 с.
67. *Шапошников И. В.* Интернет-программирование. — СПб.: БХВ-Санкт-Петербург, 2000. — 214 с.
68. *Эпплман Д.* Win32 API и Visual Basic. — СПб.: Питер, 2002. — 1120 с.
69. *Юров В.* Assembler: учебник. — СПб.: Питер, 2000. — 622 с.
70. http://www.codegear.com
71. http://www.corba.ru/
72. http://www.gnu.org/
73. http://java.sun.com/
74. http://www.perl.org/
75. http://www.php.net/
76. http://www.microsoft.com/rus/msdn/vs/default.mspx
77. http://msdn.microsoft.com/ru-ru/aa496123.aspx
78. http://www.citforum.ru/programming/32less/les44.shtml
79. http://www.citforum.ru/cfin/prcorpsys/index.shtml
80. http://www.visual.2000.ru/develop/ms-vb/cp9907-2/msado1-1.htm
81. http://www.interface.ru/fset.asp?Url=/borland/com_dcom.htm
82. http://www.citforum.ru/programming/middleware/midas_4.shtml

Алфавитный указатель

О

П

P

C

Алексей Юрьевич Молчанов

Системное программное обеспечение:
Учебник для вузов. 3-е издание

Руководитель проекта	*П. Маннинен*
Ведущий редактор	*О. Некруткина*
Литературный редактор	*А. Хрипов*
Художественный редактор	*А. Татарко*
Корректоры	*В. Нечаева, И. Тимофеева*
Верстка	*А. Ганчурин*

Подписано в печать 20.08.09. Формат 70x100/16. Усл. п. л. 32,25. Тираж 2500. Заказ
ООО «Лидер», 194044, Санкт-Петербург, Б. Сампсониевский пр., д. 29а.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2;
95 3005 — литература учебная.
Отпечатано по технологии CtP в ОАО «Печатный двор» им. А. М. Горького.
197110, Санкт-Петербург, Чкаловский пр., д. 15.

ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»

предлагают эксклюзивный ассортимент компьютерной, медицинской, психологической, экономической и популярной литературы

РОССИЯ

Санкт-Петербург м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва м. «Электрозаводская», Семеновская наб., д. 2/1, корп. 1, 6-й этаж
тел./факс: (495) 234-38-15, 974-34-50; e-mail: sales@msk.piter.com

Воронеж Ленинский пр., д. 169; тел./факс: (4732) 39-61-70; e-mail: piterctr@comch.ru

Екатеринбург ул. Бебеля, д. 11а; тел./факс: (343) 378-98-41, 378-98-42
e-mail: office@ekat.piter.com

Нижний Новгород ул. Совхозная, д. 13; тел.: (8312) 41-27-31
e-mail: office@nnov.piter.com

Новосибирск ул. Станционная, д. 36
тел.: (383) 363-01-14, факс: (383) 350-19-79; e-mail: sib@nsk.piter.com

Ростов-на-Дону ул. Ульяновская, д. 26; тел.: (863) 269-91-22, 269-91-30
e-mail: piter-ug@rostov.piter.com

Самара ул. Молодогвардейская, д. 33а; офис 223; тел.: (846) 277-89-79
e-mail: pitvolga@samtel.ru

УКРАИНА

Харьков ул. Суздальские ряды, д. 12, офис 10; тел.: (1038057) 751-10-02, 758-41-45
факс: (1038057) 712-27-05; e-mail: piter@kharkov.piter.com

Киев Московский пр., д. 6, корп. 1, офис 33; тел.: (1038044) 490-35-69
факс: (1038044) 490-35-68; e-mail: office@kiev.piter.com

БЕЛАРУСЬ

Минск ул. Притыцкого, д. 34, офис 2; тел./факс: (1037517) 201-48-77
e-mail: gv@minsk.piter.com

Ищем зарубежных партнеров или посредников, имеющих выход на зарубежный рынок. Телефон для связи: **(812) 703-73-73**. **E-mail:** fuganov@piter.com

---

Издательский дом «Питер» приглашает к сотрудничеству авторов. Обращайтесь по телефонам: **Санкт-Петербург** – **(812) 703-73-72**, **Москва** – **(495) 974-34-50**

Заказ книг для вузов и библиотек: (812) 703-73-73. Специальное предложение – e-mail: kozin@piter.com

---

Заказ книг по почте: на сайте **www.piter.com**; по тел.: (812) 703-73-74; по ICQ 413763617 по SKYPE piter_dot_com

---

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

**УВАЖАЕМЫЕ ГОСПОДА!
КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
ВЫ МОЖЕТЕ ПРИОБРЕСТИ
ОПТОМ И В РОЗНИЦУ
У НАШИХ РЕГИОНАЛЬНЫХ ПАРТНЕРОВ.**

Дальний Восток

Владивосток, «Приморский торговый дом книги»,
тел./факс (4232) 23-82-12.
E-mail: bookbase@mail.primorye.ru

Хабаровск, «Деловая книга»,
ул. Путевая, д. 1а,
тел. (4212) 36-06-65, 33-95-31
E-mail: dkniga@mail.kht.ru

Хабаровск, «Книжный мир»,
тел. (4212) 32-85-51, факс 32-82-50.
E-mail: postmaster@worldbooks.kht.ru

Хабаровск, «Мирс»,
тел. (4212) 39-49-60.
E-mail: zakaz@booksmirs.ru

Европейские регионы России

Архангельск, «Дом книги»,
пл. Ленина, д. 3
тел. (8182) 65-41-34, 65-38-79.
E-mail: marketing@avfkniga.ru

Воронеж, «Амиталь»,
пл. Ленина, д. 4,
тел. (4732) 26-77-77.
http://www.amital.ru

Калининград, «Вестер»,
сеть магазинов «Книги и книжечки»,
тел./факс (4012) 21-56-28, 65-65-68.
E-mail: nshibkova@vester.ru
http://www.vester.ru

Самара, «Чакона», ТЦ «Фрегат»,
Московское шоссе, д.15,
тел. (846) 331-22-33.
E-mail: chaconne@chaccone.ru

Саратов, «Читающий Саратов»,
пр. Революции, д. 58,
тел. (4732) 51-28-93, 47-00-81.
E-mail: manager@kmsvrn.ru

Северный Кавказ

Ессентуки, «Россы», ул. Октябрьская, 424,
тел./факс (87934) 6-93-09.
E-mail: rossy@kmw.ru

Сибирь

Иркутск, «ПродаЛитЪ»,
тел. (3952) 20-09-17, 24-17-77.
E-mail: prodalit@irk.ru
http://www.prodalit.irk.ru

Иркутск, «Светлана»,
тел./факс (3952) 25-25-90.
E-mail: kkcbooks@bk.ru
http://www.kkcbooks.ru

Красноярск, «Книжный мир», пр. Мира, д. 86,
тел./факс (3912) 27-39-71.
E-mail: book-world@public.krasnet.ru

Новосибирск, «Топ-книга»,
тел. (383) 336-10-26, факс 336-10-27.
E-mail: office@top-kniga.ru
http://www.top-kniga.ru

Татарстан

Казань, «Таис»,
сеть магазинов «Дом книги»,
тел. (843) 272-34-55.
E-mail: tais@bancorp.ru

Урал

Екатеринбург, ООО «Дом книги»,
ул. Антона Валека, д. 12,
тел./факс (343) 358-18-98, 358-14-84.
E-mail: domknigi@k66.ru

Челябинск, ТД «Эврика», ул.Барбюса, д. 61,
тел./факс (351) 256-93-60.
E-mail: evrika@bookmagazin.ru
http://www.bookmagazin.ru

Челябинск, ООО «ИнтерСервис ЛТД»,
ул. Артиллерийская, д. 124
тел. (351)247-74-03, 247-74-09, 247-74-16.
E-mail: zakup@intser.ru
http://www.fkniga.ru ,www.intser.ru