А. Ю. Молчанов

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

УЧЕБНИК ДЛЯ ВУЗОВ

А. Ю. Молчанов

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Допущено Министерством образования 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 дипломи||©етпнБ1ГсГ1ецйал|1С^ов|. ^ «Информатика и вычислительная техника

300.piter.com

Издательская программа

300 лучших учебников для высшей школы в честь 300-летия Санкт-Петербурга

осуществляется при поддержке Министерства образования РФ

*С^ПТЕР**

Москва • Санкт-Петербург • Нижний Новгород • Воронеж • Ростов-на-Дону Екатеринбург • Самара • Киев • Харьков • Минск

2003

ББК 72.973-018я7 л
УДК 681.3.06(075), %
М76 – –

Рецензенты

Немолочнов О. Ф., Зав кафедрой 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»

М76 **Системное программное обеспечение: Учебник для вузов** / А. Ю. Молчанов. – СПб.: Питер, 2003. – 396 е.: ил.

ISBN 5-94723-562-5

В книге рассматриваются основные теоретические принципы и реализующие их технологии, лежащие в основе современных средств разработки программного обеспечения Содержится вся необходимая информация о трансляторах, компиляторах, интерпретаторах, а также о других составляющих систем программирования, начиная от базовых теоретических сведений до современных технологий разработки распределенных программ

Книга ориентирована прежде всего на студентов, обучающихся в технических вузах по специальностям, связанным с вычислительной техникой Но она будет также полезна всем, чья деятельность так или иначе связана с разработкой программного обеспечения Разработчики системных программ могут почерпнуть в ней для себя немало полезных сведений, а прикладные программисты более детально познакомятся с принципами функционирования инструментов, которыми они пользуются, что в любом случае будет способствовать повышению качества создаваемых ими программных средств

Допущено Министерством образования 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 72 973-018я7
УДК 681.3.06(075)

ISBN 5-94723-562-5

Краткое содержание

Содержание

Посвящается моему сыну
Ленечке Молчанову

Предисловие

Эта книга является логическим продолжением учебника «Системное программное обеспечение», вышедшего в свет в 2001 году, для которого автором книги была подготовлена вторая часть, посвященная трансляторам и формальным грамматикам. Главной целевой аудиторией книги «Системное программное обеспечение» были студенты технических вузов, обучающиеся по специальности «Вычислительные машины, комплексы, системы и сети» и родственным с ней направлениям, поэтому материал книги был подобран исходя из требований стандарта этой специальности.

С момента выпуска прошлой версии учебника в стандарт специальности «Вычислительные машины, комплексы, системы и сети» было внесено изменение: две части, составлявшие ранее дисциплину «Системное программное обеспечение», были разделены на самостоятельные, хотя и взаимосвязанные, курсы. Один из курсов получил название «Операционные системы» (что соответствует первой части прошлой версии учебника), а второй унаследовал название «Системное программное обеспечение».

Это обстоятельство, а также понимание того, что технологии разработки программного обеспечения за время, прошедшее с момента написания предыдущего учебника, получили существенное развитие, побудило автора подготовить новую редакцию учебника.

Данный учебник посвящен компиляторам и системам программирования, что полностью отвечает содержанию дисциплины «Системное программное обеспечение» по новой версии стандарта.

По сравнению со второй частью предыдущей редакции книги «Системное программное обеспечение» автор постарался придать материалу более практическую направленность. 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.

С точки зрения практической направленности претерпело изменение и изложение материала в книге. Естественно, наименьшим изменениям подверглись первые две главы, посвященные теории формальных грамматик и общей структуре трансляторов и компиляторов, — их содержимое осталось практически прежним, поскольку в этой области за прошедшее время не произошло (да и не могло произойти) никаких изменений.

Три следующие главы полностью перестроены автором. Теперь материал структурирован не в соответствии с разделами теории формальных языков, а исходя из общей структуры компилятора. Каждая глава соответствует одной из составляющих структуры компилятора — лексический анализ, синтаксический анализ и генерация кода. Главы дополнены содержательными примерами, которые помогут пользователю на практике усвоить излагаемый материал.

И наконец, полностью перестроена последняя глава, посвященная современным системам программирования. Во-первых, автор дополнил те разделы, на которые, по его мнению, ранее не было обращено должное внимание, а во-вторых, именно в этой части книги нашли свое отражение те самые происшедшие за ближайшее время существенные изменения технологий, о которых говорилось выше.

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. Такими 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 и 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

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

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

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 будем обозначать греческими буквами: а, р, у.

Цепочка символов — это последовательность, в которую могут входить любые допустимые символы. Строка, которую вы сейчас читаете, является примером цепочки, допустимые 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 а и р равны (совпадают), а = р, 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 а обозначается как |а|. Очевидно, что если а = р, то и |а| = |р|.

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 а и р обозначается как ар. Выполнить конкатенацию цепочек просто: например, если а = *аб,* а р = *вг,* то ар = *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 Заир такие, что ар * ра (например, для а - *аб* и р - *вг:* ар - *абвг,* а Ра - *вгаб* и ар * Ра).

Также очевидно, что конкатенация обладает свойством ассоциативности, то есть (ар)у - а(Ру).

Любую цепочку символов языка можно представить как конкатенацию составляющих ее частей — разбить цепочку на несколько подцепочек. Такое разбиение можно выполнить несколькими способами произвольным образом. Например, цепочку у - *абвг* можно представить в виде конкатенации цепочек а - *аб* и р - *вг* (у - **аР),** а можно — в виде конкатенации цепочек и - *а* и со - *бег* (у - исо). 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 у - *абвг,* разобьем ее на три подцепочки: *а- а,* **со-5ир-ег(у- асор), и** выполним подстановку цепочки **и** - *аба* вместо подцепочки **со.** Получим новую цепочку у' - *аабавг* **(у - аир).** 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 **а** обозначается как $а^R$. Если а - *«абвг»,* то $а^R$ = *«гвба».* Для операции обращения справедливо следующее равенство V а,р: $(ар)^R$ - $р^R а^R$.

Итерация (повторение) цепочки п раз, где neN, п > 0 — это конкатенация цепочки самой с собой п раз. Итерация цепочки а п раз обозначается как $а^n$. Для операции повторения справедливы следующие равенства V **а:** $а^1$ = **а,** $а^2$ **= аа,** $а^3$ **- ааа,** ... и т. д. Итерация цепочки символов определена и для п = 0 — в этом случае результатом итерации будет пустая цепочка символов.

Пустая цепочка символов — это цепочка, не содержащая ни одного символа. Пустую цепочку здесь везде будем обозначать греческой буквой X (в литературе ее иногда обозначают латинской буквой е или греческой в).

Для пустой цепочки справедливы следующие равенства:

1. W = O
2. Va: *Xa* - *aX* = a
3. $X^R = X$
4. Vn > 0: $X^n = X$
5. Va: $a^0 = X$

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 не обязательно должен быть конечным множеством, но реально все существующие языки строятся на основе конечных алфавитов.

Цепочка символов а является цепочкой над алфавитом V: а(V), если в нее входят только символы, принадлежащие множеству символов V. Для любого алфавита V пустая цепочка X может как являться, так и не являться цепочкой л(У). Это условие оговаривается дополнительно.

Если V — некоторый алфавит, то:

V^+ — множество всех цепочек над алфавитом V без Л;

V* — множество всех цепочек над алфавитом V, включая X.

Справедливо равенство: $V^* = V^+ u \{/\backslash\}$.

Языком L над алфавитом V: L(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(V) — множеством предложений этого языка.

Для любого языка L(V) справедливо: L(V)cV*.

Язык L(V) включает в себя язык L'(V): L'(V)cL(V), если VaeL(V): aeL'(V). Множество цепочек языка L'(V) является подмножеством множества цепочек

языка L(V) (или эти множества совпадают). Очевидно, что оба языка должны строиться на основе одного и того же алфавита.

Два языка L(V) и L'(V) совпадают (эквивалентны): L'(V) = L(V), если L'(V)c L(V) и L(V)cL'(V); или, что то же самое: VaeL'(V): aeL(V) и V(3eL(V): (3eL'(V). Множества допустимых цепочек символов для эквивалентных языков равны.
Два языка L(V) и L'(V) почти эквивалентны: L'(V)=L(V), если L'(V)u{/\.} = = L(V)u{A,}. 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 Pascal. 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 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 > 0, то получим почти эквивалентный язык L'({0,1}), содержащий пустую цепочку.
Второй способ предусматривает некоторое описание правил, с помощью которых строятся цепочки языка. Тогда 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 упо|то^нщсоташшеь^€£еда*й 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 сейчас этот текст, вы 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 то, 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 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 (a,p). В правилах важен порядок цепочек, поэтому их чаще записывают в виде a p (или a ::= P). Такая запись читается как «a порождает p» или «p по определению есть a».

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)u{/\.} = L(G')u{/\.}.

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 a-»p, где ae$(VNuVT)^+$, pe$(VNuVT)^*$;

S — целевой (начальный) символ грамматики SeVN.

Алфавиты терминальных и нетерминальных символов грамматики не пересекаются: VNnVT - 0. 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 V = VNuVT 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 обычно 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: а p_b а (2> - [a] Pn- Тогда эти правила объединяют вместе и записывают в виде: сх —> j $p_2$1...| P_n. Одной строке в такой записи соответствует сразу п правил. ,

Такую форму записи правил грамматики называют формой Бэкуса—Наура. Форма Бэкуса—Наура предусматривает, как правило, 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 различных правых 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    т | ,т | -т
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 uG[1] 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 (конечно, 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 его самого, и позволяют избежать бесконечного рекурсивного определения (в противном случае этот символ в грамматике был бы просто не нужен). Такими правилами явпяются <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. Тогда определение для 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 понимание способ записи формальных грамматик. Рекурсивные определения хороши для формального анализа цепочек языка, но не удобны с точки зрения человека. Например, то, что правила <чс> <цифра> | <чс><цифра> отражают возможность для построения числа дописывать справа любое число 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 большую 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;
- запятая служит для того, чтобы разделять цепочки символов внутри круглых скобок;
- 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).

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
```

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 любое количество 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 { } (фигурные скобки). 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. Также иногда дополняют смысл уже существующих метасимволов. Например, для метасимвола { } (фигурные скобки) существует удобная форма записи, позволяющая ограничить число повторений цепочки символов, заключенной внутри них: { }n, где n ∈ 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 входной точки дуги могут только выходить, а во входную точку — только входить. В остальные вершины дуги могут как входить, так и выходить (в правильно построенной грам-

матике 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. Тогда, 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 результирующую 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 будет закончено.
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.

Следует подчеркнуть, что представленный рисунок — всего лишь условная схема, отображающая работу алгоритма распознавателя. Ни в коем случае не стоит искать

подобное устройство в составе компьютера. Распознаватель, являющийся частью компилятора, представляет собой часть программного обеспечения компьютера.

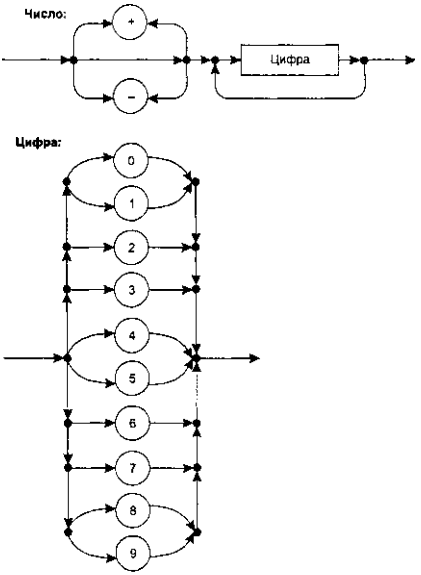


Рис. 1.1. Графическое представление грамматики целых десятичных чисел со знаком

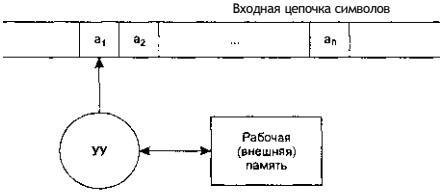


Рис. 1.2. Условная схема распознавателя

Как видно из рисунка, распознаватель состоит из следующих основных компонентов:

- ленты, содержащей входную цепочку символов, и считывающей головки, 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* а, 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 распознавателей при одной и той же входной цепочке символов из начальной конфигурации мо^ут 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.

Виды распознавателей

Распознаватели можно классифицировать в зависимости от вида составляющих их компонентов: считывающего устройства, устройства управления (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, то 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 устройства управления 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.

У 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 составляющих 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.

Задача разбора

Для каждого языка программирования важно не только уметь 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 созда-

нии 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. Она, как правило, четко описана в стандарте языка. 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.

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 зачастую зависит 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 бы анализировал 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 = VNuVT правила имеют вид: a. —> p, где asV$^+$, (ЗeV*. Это самый общий тип грамматик. В него подпадают 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 = VNuVT имеют правила вида: a_1Aa_2 -» cupar, где оц.агбУ*, AeVN, peV$^+$.

Неукорачивающие грамматики G(VT,VN,P,S), V = VNu VT имеют правила вида: a -> p, где a,peV$^+$, |p| > |a|.

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». Цепочки a, и a_2 в правилах грамматики обозначают контекст (a, — левый контекст, а a_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 контекстно-зависимой грамматикой, можно построить неукорачивающую грамматику, которая будет задавать эквивалентный язык, и наоборот: для любого языка, заданного неукорачивающей грамматикой, можно построить контекстно-зависимую 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 контекстно-зависимых грамматик.

Тип 2: контекстно-свободные (КС) грамматики

Контекстно-свободные (КС) грамматики G(VT,VN,P,S), V = VNuVT имеют правила вида: A —> p, где AsVN, psV$^+$. Такие грамматики также иногда называют неукорачивающими контекстно-свободными (НКС) грамматиками (видно, что в правой части правила 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 = VNuVT, правила которых могут иметь вид: A -> p, где AeVN, peV*.

Разница между этими двумя классами грамматик заключается лишь в том, что в УКС-грамматиках в правой части правил может присутствовать пустая цепочка (X), 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 = VNuVT могут иметь правила двух видов: A -> By или A -> y, где A.BeVN, yeVT*.

В свою очередь, праволинейные грамматики G(VT,VN,P,S), V = VNuVT могут иметь правила тоже двух видов: A -» yB или A -> y, где A,BeVN, yeVT*.

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).

Соотношения между типами грамматик

Типы грамматик соотносятся между собой особым образом. Из определения типов 2 и 3 видно, что любая регулярная грамматика является КС-грамматикой, но не наоборот. Также очевидно, что любая грамматика может быть отнесена к типу О, поскольку он не накладывает никаких ограничений на правила. В то же время существуют укорачивающие КС-грамматики (тип 2), которые не являются ни контекстно-зависимыми, ни неукорачивающими (тип 1), поскольку могут содержать правила вида «A Ъ>, недопустимые в типе 1.

Одна и та же грамматика в общем случае может быть отнесена к нескольким классификационным типам (например, как уже было сказано, все без исключения грамматики могут быть отнесены к типу 0). Для классификации грамматики всегда выбирают максимально возможный тип, к которому она может быть отнесена. Сложность грамматики обратно пропорциональна номеру типа, к которому относится грамматика. Грамматики, которые относятся только к типу 0, являются самыми сложными, а грамматики, которые можно отнести к типу 3, — самыми простыми.

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, то для классификации самого языка среди всех его грамматик выбирается грамматика с максимально возможным классификационным типом. Например, если язык L может быть задан грамматиками Gj и G_2, относящимися к типу 1 (контекстно-зависимые), грамматикой G_3, относящейся к типу 2 (контекстно-свободные) и грамматикой G_4, относящейся к типу 3 (регулярные), то сам язык должен быть отнесён 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, то 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 различные роли 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 бы воспринимали 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, допускающие управляющие команды цикла и условия.

В общем случае время на распознавание предложений языка, относящегося к типу 1, 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 4 «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 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 3 «Лексические анализаторы».

Классификация распознавателей

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 с определенным составом компонентов и, следовательно, с заданной сложностью алгоритма работы. 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 и примет решение о том, принадлежит или ие принадлежит входная цепочка заданному языку. 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. Следовательно, и время, необходимое на разбор входной цепочки по заданному алгоритму, экспоненциально зависит от длины входной цепочки символов.

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 контекстно-зависимых языков. Как правило, такие распознаватели применяются для автоматизированного Перевода и анализа текстов на естественных языках, когда временные ограничения на разбор текста несущественны (следует также напомнить, что поскольку естественные языки более сложны, чем контекстно-зависимый тип, то после такой обработки часто требуется вмешательство человека).

В рамках этого учебника контекстно-зависимые 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, распознавателями для которых являются детерминированные автоматы с магазинной (стековой) внешней памятью — ДМП-автоматы. Для таких языков существует алгоритм работы распознавателя с квадратичной сложностью.

Среди всех детерминированных КС-языков существуют такие классы языков, для которых возможно построить линейный распознаватель — распознаватель, у которого время принятия решения о принадлежности цепочки языку имеет линейную зависимость от длины цепочки. Именно эти языки представляют интерес при построении компиляторов. Синтаксические конструкции практически 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 предвари-

тельного 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 контекстно-зависимого 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, конечные автоматы 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 контекстно-свободным, и контекстно-зависимым 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 G1{0.1,2.3.4.5.6.7.8,9.-,+}.{S.T,F},P1,S):

PI:
S -> T I +T I -т
T -> F I TF
F - » 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

По структуре своих правил данная грамматика G1 относится к контекстно-свободным грамматикам (тип 2). Конечно, ее можно отнести и к типу 0, и к типу 1, но максимально возможным является именно тип 2, поскольку к типу 3 эту грамматику отнести никак нельзя: строка T —> F | TF содержит правило T —> TF, которое недопустимо для типа 3, и хотя все остальные правила этому типу соответствуют, одного несоответствия достаточно.

Для того же самого языка (целых десятичных чисел со знаком) можно построить и другую грамматику G1'({0.1,2,3.4.5.6.7.8.9,-.+}.{S.T},P1',S):

РГ •
S -> T I +T I -т
T -» 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ОГ | IT | 2Г | 3Г | 4Г | 5Г | 6Г | 7Г | 8Г | 9Г

По структуре своих правил эта грамматика Gl' является праволинейной и может быть отнесена к регулярным грамматикам (тип 3).

Для этого же языка можно построить эквивалентную леволинейную грамматику (тип 3) Gl"({0.1,2,3.4.5,6.7,8.9.-.+}.{S,T},P1".S):

PI".
T -> + | - \ \
S ТО | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | SO | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 | S9

Следовательно, язык L1 целых десятичных чисел со знаком, заданный грамматиками Gl, Gl' и G1", относится к регулярным языкам (тип 3).

В качестве второго примера возьмем грамматику G2({0,1}, {A.S} .P2.S) с правилами P2:

S -> 0A1
OA -> 00A1
A -> *X*

Эта грамматика относится к типу 0. Она определяет язык, множество предложений которого можно было бы записать так: L(G2) = {Onln | n > 0}.

Для этого же языка можно построить также контекстно-зависимую грамматику G2' ({0,1}, {A.S} ,P2' .S) с правилами P2':

S —> 0A1 | 01
OA -> 00A1 | 0Q1

Однако для того же самого языка можно использовать и контекстно-свободную грамматику G2" ({0.1}, {S} ,P2" ,S) с правилами P2":

S —> 0S1 I 01

Следовательно, язык L2 = {Onln | n > 0} является контекстно-свободным (тип 2).

В третьем примере рассмотрим грамматику G3({a,b.c},{B,C.D,S}.P3,S) с правилами P3:

S -> BO
B -> aBbC | ab
Cb -> bC
CD -> Dc
bDc -> bcc %
abD -> abc

Эта грамматика относится к типу 1. Очевидно, что она является неукорачивающей. Она определяет язык, множество предложений которого можно было бы записать так: L(G3) = {anbncn | n > 0}. Известно, что этот язык не является КС-языком, поэтому для него нельзя построить грамматики типов 2 или 3.

Но для того же самого языка можно построить КЗ-грамматику G3' ({a,Ь.c}. {B, C.D.E.F.S} ,P3', S) с правилами P3':

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 -> be

Язык L3 = {anbncn | 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 при строгом определении типа требуется еще доказать, что две грамматики (первоначально имеющаяся и вновь построенная) эквивалентны, а также то, что для того же языка не существует грамматики с большим по номеру типом. Это нетривиальная задача, которую не так легко решить.

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, 28]. Тогда 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 р = б^ называется *непосредственно выводимой* из цепочки *а* = б^бг в грамматике G(VT,VN,P,S), V = VTuVN, $б_1$ у, $б_2$ е V", со е V^+, если в грамматике G существует правило: со у е Р. Непосредственная выводимость цепочки р из цепочки *а* обозначается так: а => р. Согласно определению, при выводе а => р выполняется подстановка подцепочки у вместо подцепочки со. Иными словами, цепочка р выводима из цепочки *а* в том случае, если можно взять несколько символов в цепочке а, заменить их на другие символы согласно некоторому правилу грамматики и получить цепочку р.

В формальном определении непосредственной выводимости любая из цепочек $5_($ или $б_2$ (а равно и обе эти цепочки) может быть пустой. В предельном случае вся цепочка *а* может быть заменена на цепочку р, тогда в грамматике G должно существовать правило: а —> ре Р.

Цепочка р называется *выводимой* из цепочки *а* (обозначается *а* =>* р) в том случае, если выполняется одно из двух условий:

- р непосредственно выводима из а (а => р);
- 3 у такая, что у выводима из а и р непосредственно выводима из у (а =>* у и у => Р).

Это рекурсивное определение выводимости цепочки. Суть его заключается в том, что цепочка р выводима из цепочки а, если а => р или же если можно построить последовательность непосредственно выводимых цепочек от а к р следующего вида: а => у! =>...=> у, =>...=> $у_n$ => р, п > 1. В этой последовательности каждая последующая цепочка у, непосредственно выводима из предыдущей цепочки у,.].

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 р непосредственно выводима из цепочки а: а => р, то имеется всего один шаг вывода.

Если цепочка вывода из а к (3 содержит одну или более промежуточных цепочек (два или более шагов вывода), то она имеет специальное обозначение а =>+ (3 (говорят, что цепочка (3 *нетривиально выводима* из цепочки а). Если количество шагов вывода известно, то его можно указать непосредственно у знака выводимости цепочек. Например, запись а $\Rightarrow^4$ р означает, что цепочка р выводится из цепочки а 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 т I +T I -т
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 => -TF => -TFF => -FFF => -4FF => -47F => -479
2. S => T => TF => T8 => F8 => 18
3. T => TF => TO => TF0 => T50 => F50 => 350
4. TFT => TFFT => TFFF => FFFF => 1FFF => 1FF4 => 10F4 => 1004
5. F => 5

Получим следующие выводы:

1. S =>* -479 или S =>+ -479 или S $\Rightarrow^7$ -479
2. S =>* 18 или S =>+ 18 или S $\Rightarrow^5$ 18
3. T =>* 350 или T =>+ 350 или T $\Rightarrow^6$ 350
4. TFT =>* 1004 или TFT =>+ 1004 или TFT $\Rightarrow^7$ 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 G3({a.b.c},{B.C.D,S}.P3.S) справилами Р3, которая уже рассматривалась выше:

S -> HD
B -> aBHC | ab
CЬ -> ЬC
CD Dc
bDc bcc
abD abc

[1] В литературе встречается также обозначение: а р, которое означает, что цепочка а выводима из цепочки р за 0 шагов — иными словами, в таком случае эти цепочки равны: а = р. Подразумевается, что обозначение вывода а =>* р допускает и такое толкование — включает в себя вариант *а* р.

Как было сказано ранее, она задает язык L(G3) = {anbncn | n > 0}. Рассмотрим пример вывода предложения aaaabbbbcccc языка L(G3) на основе грамматики G3:

S ⇒ BD ⇒ aBbCD ⇒ aaBbCbCD ⇒ aaaBbCbCbCD ⇒ aaaabbCbCbCD ⇒ aaaabbbCCbCD
aaaabbbCbCCD ⇒ aaaabbbbCCCD ⇒ aaaabbbbCCDc ⇒ aaaabbbbCDcc ⇒ aaaabbbbDccc ⇒
aaaabbbbcccc

Тогда для грамматики G3 получаем вывод: S =>* aaaabbbbcccc.
Иногда, чтобы пояснить ход вывода, над стрелкой, обозначающей каждый шаг вывода, пишут обозначение того правила грамматики, на основе которого сделан этот шаг (для этой цели правила грамматики проще всего просто пронумеровать в порядке их следования). Грамматика, рассмотренная в приведенных здесь примерах, содержит всего 15 правил, и на каждом шаге в цепочках вывода можно понять, на основании какого правила сделан этот шаг (читатели могут легко сделать это самостоятельно), но в более сложных случаях пояснения к шагам вывода с указанием номеров правил грамматики могут быть весьма полезными.

Сентенциальная форма грамматики. Язык, заданный грамматикой

Вывод называется *законченным*, если на основе цепочки (3, полученной в результате вывода, нельзя больше сделать ни одного шага вывода. Иначе говоря, вывод называется законченным, если цепочка (3, полученная в результате вывода, пустая или содержит только терминальные символы грамматики G(VT,VN,P,S): (3eVT*. Цепочка (3, 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 aeV* называется *сентенциальной формой* грамматики G(VT, VN, P,S), V = VTuVN, если она выводима из целевого символа грамматики S: S =>* a. Если цепочка aeVT* 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**h 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, обозна-

чается 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 VTnVT ϕ 0 (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 выводы. 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 G3, задающей язык L(G3) = {Onln | 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 Ae(VTuVN);
- корнем дерева является вершина, обозначенная целевым символом грамматики — S;
- листьями дерева (концевыми вершинами) являются вершины, обозначенные терминальными символами грамматики или символом пустой цепочки ~к\

- если некоторый узел дерева обозначен нетерминальным символом AeVN, а связанные с ним узлы — символами b^-.b,,; n > 0, Vn > i > 0: b,e(VTuVNu{X}), то в грамматике G(VT,VN,P,S) существует правило A—>Ь1Ь2...Ьп е 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 вывода 1 и 2. Эти деревья приведены на рис. 1.3.

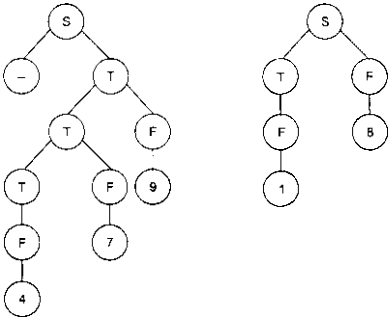


Рис. 1.3. Примеры деревьев вывода для грамматики целых десятичных чисел со знаком

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 относительно полученного слоя 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 выполняются 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 I S/S I (S) I a I b

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

Возьмем цепочку a*Ь+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*Ь+a приведены на рис. 1.4.

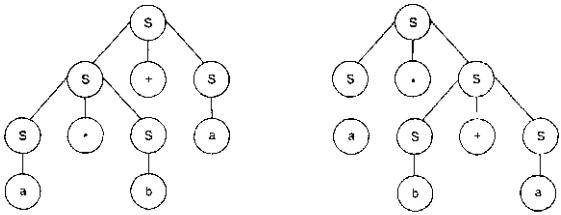


Рис. 1.4. Два варианта дерева цепочки a*b+a вывода для неоднозначной грамматики арифметических выражений

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 того или иного предложения языка.
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 неоднозначной 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 бы допускала 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 количество. 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 Ь существует эквивалентная ей однозначная грамматика вида G' ({+,-,*./.(,). a.b}.{S,T,E}.P',S):

P'

```
S     S+T | S-T I т
T -> T*E I T/E I E
E -> (S) | a | b
```

В этой грамматике для рассмотренной выше цепочки символов языка a*Ь+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, но приоритет операций в данном случае единственно возможный и соответствует их порядку в арифметике.

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: имеются 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').

Рис. 1.5. Дерево вывода для однозначной грамматики арифметических выражений

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 для произвольной заданной грамматики G проверить, является ли она 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].

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 возможно построить распознаватель в виде детерминированного расширенного МП-автомата, а потому можно утверждать, что она является однозначной (см. раздел «Восходящие распознаватели КС-языков 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 —» AA | a
2. A AaA | (3
3. A aA | A|3 | y
4. A aA | aA(3A | y

здесь AeVN; a,p,ye(VNuVT)*.

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 контекстно-свободных 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 Ь — 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 типа (например, два правила: S S+S и S -> a). Поэтому данная грамматика являемся неоднозначной, что и было показано выше.

Контрольные вопросы и задачи

Вопросы

1. Дайте определение цепочки, языка. Какие операции можно выполнять над цепочками символов? Что такое синтаксис и семантика языка?
2. Какие из перечисленных ниже тождеств являются истинными для двух произвольных цепочек символов a и p, а какие нет:

   |ap| = |a| + |p| = |pa|

   ap = pa

   И - loci

$(a^2(3^2)^R = (|3^R a^R)^2$
$(a^2(3^2)^R = ((3^R)^2(a^R)^2$

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? Почему любой язык программирования не является чисто формальным языком?
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 и как она решается?
7. Что такое грамматика языка? Дайте определение 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? Какие виды распознавателей существуют?
11. Как классифицируются распознаватели? Как их классификация соотносится с классификацией языков и грамматик?
12. Какие типы грамматик выделяют по классификации Хомского? Как они между собой соотносятся?
13. Какие типы языков выделяют по классификации Хомского? Как классификация языков соотносится с классификацией грамматик?
14. Дайте определения выводимости цепочки, непосредственной выводимости, нетривиальной выводимости, длины вывода.
15. Что такое сентенциальная форма грамматики? Если язык, заданный грамматикой, представляет собой множество ее конечных сентенциальных форм, то что представляет собой множество всех сентенциальных форм 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 десятичйых чисел с фиксированной точкой. Укажите, к какому типу относится каждая из этих грамматик. К какому типу относится сам язык десятичных чисел с фиксированной точкой?

GI({".",+,-.0.1.2,3.4,5.б,7,8,9}.{<число>,<цел>.<дроб>,<цифра>.<осн>,
<знак>},P1.<число>)
P1.
<ЧИСЛО> <знак><осн>
<знак> X | + | -

<осн> -> <цел> <дроб> | <цел >
<цел> -> <цифра> | <цифра><цифра>
<дроб> -> X | <цел>
<цифра><цифра> -> <цифра><цифра><цифра>
<цифра> - » 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
G2({" ".+,-,0.1,2,3,4,5.6,7,8.9}.{<число>,<часть>.<цифра>,<осн>},P2 ,<нисло>)
P2
<число> -> +<ОСН> | -<ОСН> I <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
G3({".",+,-.О,1,2,3.4,5.6,7,8,9},{<число>,<часть>.<осн>},Р3,<число>)
Р3-
<число> -> +<ОСН> I -<ОСН> I <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
G4({".",+.-,0,1,2,3,4,5.6,7,8,9},{<знак>.<число>,<часть>.<осн>},P4,<число>)
P4:
<число> <часть> | <часть>. | <число>0 | <число>1 | <число>2] <число>3 |
<число>4 I <число>5 | <число>6 I <ЧИСЛО>7 I <число>8 | <число>9
<часть> -> <знак>0 | <знак>1 | <знак>2 | <з'нак>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> -> X | + | -
G5({".",+,-,0,1,2,3.4,5,6.7,8,9}.{<знак>,<число>.<часть>,<осн>},P5,<число>)
P5:
<число> -> <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. Докажите, что если для некоторой цепочки языка, заданного грамматикой, существует единственный левосторонний вывод, то для нее существует и единственное дерево вывода.
3. 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. Какие из предложенных пяти грамматик являются неоднозначными?

4. 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,
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 № 5), содержащего не более 60 вагонов? 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 CHE
   CHE -> CFc
   CFc -> CEc
   aB ab
   bB bb
   bCE -> bCc
   bCc be

8. Дана грамматика условных выражений G({" " л ,f .t.h.e.n.1 .s.b.a}. {E} .P.E) с правилами:

   P'
   E —• i f 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

Определения 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

[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!

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 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 лишь разницей, что сложность языков несколько иная (о том, почему не существует трансляторов с естественных языков, сказано в разделе «Классификация языков и грамматик» в предыдущей главе). Поэтому и само слово «транслятор» (англ.: 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 (как правило, 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 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»). Поскольку первоначально компиляторы ничего другого делать не умели, то этот термин и закрепился за ними.

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 написаиных на нем программ. Хотя в данном случае термин «транслятор» был бы, наверное, более правильным, в литературе употребляется понятие «компилятор», поскольку промежуточный язык является языком очень низкого уровня, будучи родственным машинным командам и языкам ассемблера.

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

Вычислительная система, на которой выполняется 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.

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 совершенно 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 результирующая 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 является результирующей программой для другого компилятора, которая ничем не отличается от всех прочих порождаемых результирующих программ[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.

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 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 программирования, 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 иначе выполнение программ на компьютере невозможно. Он, конечно же, делает это, однако полученные машинные коды не являются доступными — их не видит пользователь интерпретатора. Эти машинные коды порождаются интерпретатором, исполняются и уничтожаются по мере надобности — так, как того требует конкретная реализация интерпретатора. Пользователь же видит результат выполнения этих кодов — то есть результат выполнения исходной программы (требование об эквивалентности исходной программы и порожденных машинных кодов и в этом случае, безусловно, должно выполняться).

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

Этапы трансляции. Общая схема работы транслятора

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 про-

граммы 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 [15, 19, 31].

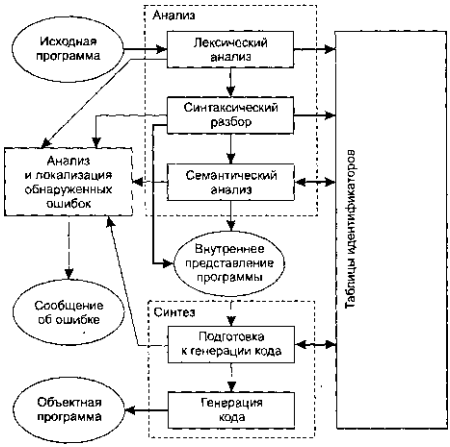


Рис. 2.1. Общая схема работы компилятора

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

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 была построена (поскольку на вопрос о принадлежности сам ответ «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 опреде-

ленным 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 данной книги, 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 является обязательной, необходимой частью компилятора. Однако существуют причины, которые определяют его присутствие практически во всех компиляторах (более подробно они описаны в главе 3 «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 4 «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 5 «Генерация и оптимизация кода»).

Подготовка к генерации кода — это фаза, на которой компилятором выполняются предварительные действия, непосредственно связанные с синтезом текста результирующей программы, но еще 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 5 «Генерация и оптимизация кода»).

Генерация кода — это фаза, непосредственно связанная с порождением команд, составляющих предложения выходного языка и в целом текст результирующей программы. Это основная фаза на этапе синтеза результирующей программы. Кроме непосредственного 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 5 «Генерация и оптимизация кода»).

Таблицы идентификаторов (иногда — «таблицы символов») — это специальным образом организованные наборы 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 набора элементов зависит от используемого входного языка программирования. Понятие «таблицы» вовсе не предполагает, что это хранилище данных должно быть организовано именно в виде таблиц или других 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, на фазе лексического анализа лексемы выделяются из текста входной программы постольку, поскольку они необходимы для следующей 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 результат, 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 результата 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 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 — 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 6 «Современные системы программирования»).

Современные компиляторы и интерпретаторы

Компиляторы с языков высокого уровня

История развития компиляторов

Первые программы, которые создавались еще для ЭВМ первого поколения, писались непосредственно на языке машинных кодов. Это была поистине адская работа. Сразу стало ясно, что человек не должен и не может говорить на языке машинных команд, даже если он специалист по вычислительной технике. Однако все попытки научить компьютер говорить на языках людей успехом не увенчались и вряд ли когда-либо увенчаются (на что есть определенные объективные причины, рассмотренные в первой главе данной книги).

С тех пор все развитие программного обеспечения компьютеров неразрывно связано с возникновением и развитием компиляторов.

Первыми компиляторами были компиляторы с языков ассемблера, или, как они назывались, мнемокодов. Мнемокоды превратили «филькину грамоту» языка машинных команд в более-менее доступный пониманию специалиста язык мнемонических (преимущественно англоязычных) обозначений этих команд. Создавать программы уже стало значительно проще, но исполнять сам мнемокод (язык ассемблера) ни один компьютер не способен, соответственно возникла необходимость в создании компиляторов. Эти компиляторы элементарно просты, но они продолжают играть существенную роль в системах программирования по сей день. Более подробно о языке ассемблера и компиляторах с него рассказано далее в соответствующем разделе.

Следующим этапом стало создание языков программирования высокого уровня. Языки высокого уровня (к ним относится большинство языков программирования) представляют собой некоторое промежуточное звено между чисто формальными языками и языками естественного общения людей. От первых им досталась строгая формализация синтаксических структур предложений языка, от вторых — значительная часть словарного запаса, семантика основных конструкций и выражений (с элементами математических операций, пришедшими из алгебры).
Появление языков высокого уровня существенно упростило процесс программирования, хотя и не свело его до «уровня домохозяйки», как самонадеянно полагали некоторые авторы на заре рождения языков программирования. Сначала таких языков были единицы, затем десятки, сейчас, наверное, их насчитывается более сотни. Процессу этому не видно конца. Тем не менее по-прежнему преобладают компьютеры традиционной, «неймановской» архитектуры, которые

умеют понимать только машинные команды, поэтому вопрос о создании компиляторов продолжает быть актуальным.

Как только возникла массовая потребность в создании компиляторов, стала развиваться и специализированная теория. Со временем она нашла практическое приложение во множестве созданных компиляторов. Компиляторы создавались и продолжают создаваться не только для новых, но и для давно известных языков. Многие производители, от известных, солидных фирм (таких, как Microsoft или Borland/Inprise) до мало кому знакомых коллективов авторов, выпускают на рынок все новые и новые образцы компиляторов. Это обусловлено рядом причин, которые будут рассмотрены далее.

Особенности построения и функционирования компиляторов

Как уже было сказано ранее, для задания языка программирования необходимо решить три задачи:

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

Главную проблему при этом представляет третья задача, поскольку она, в принципе, не относится к теории формальных языков. Чисто формальные языки лишены всякого смысла, а потому для решения этой задачи нужно использовать другие подходы.

В качестве таких подходов можно указать следующие:

1. изложить смысл программы, написанной на языке программирования, на другом языке, который может воспринимать пользователь программы или компьютер;
2. 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 пока универсального 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 для языков программирования.

Следовательно, первый подход, а именно изложение смысла исходной программы на языке машинных команд, составляет основу функционирования компиляторов. Далее в этом учебнике 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 большую часть такого рода ошибок должен обнаруживать человек (разработчик программы или ее пользователь).

Второй подход используется при отладке программы. В качестве «идеальной машины» может выступать интерпретатор, который непосредственно воспринимает и выполняет исходную программу. Однако чаще такой «идеальной машиной» является компьютер, на котором выполняется откомпилированная программа. 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 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; 1 :-0; while i—0 do i —0 может быть легко оценено любой машиной как бессмысленное. Но если в задачу входит обеспечение взаимодействия с другой параллельно выполняемой программой или, например, просто проверка надежности и долговечности процессора или какой-то ячейки памяти, то это же предложение покажется уже не лишенным смысла.

Но 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. Обычно компилятор указывает такие места в виде дополнительных 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.

Но в обоих случаях осмысление исходной программы закладывает в компилятор его создатель (или коллектив создателей) — то есть человек, который руководствуется неформальными методами (чаще всего, описанием входного языка). В теории формальных языков вопрос о смысле программ не решается.

На основании изложенных положений можно сказать, что возможности компиляторов по проверке осмысленности предложений входного языка сильно ограничены. Именно поэтому большинство из них в лучшем случае ограничиваются рекомендациями разработчикам по тем местам исходного текста программ, которые вызывают сомнения с точки зрения смысла. Поэтому большинство компиляторов обнаруживают только незначительный процент от общего числа смысловых («семантических») ошибок, а следовательно, подавляющее число такого рода ошибок всегда, к большому сожалению, остаются 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 трансляторов, поэтому здесь рассматриваться не будет.

Компиляторы в составе систем программирования

С тех пор как большинство теоретических аспектов в области компиляторов получили свою практическую реализацию (а это, надо сказать, произошло довольно быстро, в конце 60-х годов), 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. Появившись, системы программирования быстро завоевали рынок и ныне в массе своей преобладают на нем. Фактически, в настоящее время обособленные компиляторы — это редкость среди современных программных средств. О том, что представляют собой и как организованы современные системы программирования см. в главе 6 «Современные системы программирования».

Ныне компиляторы являются неотъемлемой частью любой вычислительной системы. Без их существования программирование любой прикладной задачи было бы затруднено, а то и просто невозможно. Да и программирование специализированных системных задач, как правило, ведется если не на языке высокого уровня (в этой роли в настоящее время чаще всего применяется язык C), то на языке ассемблера, следовательно, применяется соответствующий компилятор. Программирование непосредственно на языках машинных кодов происходит исключительно редко и только для решения очень узких вопросов.

Первыми компиляторами были компиляторы с мнемокодов. Их потомки — современные компиляторы с языков ассемблера — существуют практически для всех известных вычислительных систем. Они предельно жестко ориентированы на архитектуру. Затем появились компиляторы с таких языков, как FORTRAN, ALGOL-68, PL/1. Они были ориентированы на большие ЭВМ с пакетной обработкой задач. Из вышеперечисленных только FORTRAN, пожалуй, продолжает

использоваться по сей день, поскольку имеет огромное количество библиотек различного назначения [5]. Многие языки, родившись, так и не получили широкого распространения — ADA, Modula, Simula известны лишь узкому кругу специалистов. В то же время на рынке программных систем доминируют системы программирования и компиляторы для языков, которым не прочили светлого будущего. В первую очередь, сейчас это C и C++. Первый из них родился вместе с операционными системами типа UNIX, вместе с ними завоевал свое «место под солнцем», а затем перешел под ОС других типов. Второй удачно воплотил в себе пример реализации идей объектно-ориентированного программирования на хорошо зарекомендовавшей себя практической базе[1]. Ещё можно упомянуть довольно распространенный Pascal, который неожиданно для многих вышел за рамки чисто учебного языка для университетской среды.

Об истории языков программирования и современном состоянии рынка компиляторов можно говорить долго и много. Автор считает возможным ограничиться уже сказанным, поскольку это не является целью данного учебника. Желающие могут обратиться к литературе [5, 12, 15, 17, 35, 42, 45].

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 чем отличается 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. По сути, в таком интерпретаторе не было бы никакого особого смысла — он не давал бы никаких преимуществ по сравнению с аналогичным компилятором[2].

[1] Назвать реализацию «удачной» можно, если судить по достигнутым результатам, а не по качеству заложенных в C++ идей объектно-ориентированного подхода. Последний вопрос способен вызвать споры, которые выходят за рамки данного учебного пособия.

[2] Иногда такая схема используется в современных системах программирования при отладке программ — подробности см. в главе 6 «Современные системы программирования».

Поэтому подавляющее большинство интерпретаторов действуют так, что исполняют 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 по мере поступления [15, 17, 52, 54].

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 оптимизацию исходной программы. Следовательно, фаза оптимизации в общей структуре интерпретатора будет отсутствовать. В остальном же 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.

Отсутствие шага оптимизации определяет еще одну особенность, характерную для многих интерпретаторов: в качестве внутреннего представления программы в них очень часто используется обратная польская запись (см. главу 5 «Генерация и оптимизация кода»). Эта удобная форма представления операций обладает только одним существенным недостатком — она плохо поддается оптимизации. Но в интерпретаторах это как раз и не требуется.

Далеко не все языки программирования допускают построение интерпретаторов, которые могли бы выполнять исходную программу по мере поступления команд. Для этого язык должен допускать возможность существования компилятора, выполняющего разбор исходной программы за один проход. Кроме того, язык не может интерпретироваться по мере поступления команд, если он допускает появление обращений к функциям и структурам данных раньше их непосредственного описания. Поэтому таким методом не могут интерпретироваться такие языки, как C и Pascal.

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 результате 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.

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.

В современных системах программирования существуют реализации программного обеспечения, сочетающие в себе и функции компилятора, и функции интерпретатора — в зависимости от требований пользователя исходная программа либо компилируется, либо исполняется (интерпретируется). Кроме того, 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 6 «Современные системы программирования».

История интерпретаторов пока не столь богата, как история компиляторов. Как уже было сказано, изначально им не придавали большого значения, поскольку почти по всем параметрам они уступают компиляторам. Из известных языков, предполагавших интерпретацию, можно упомянуть разве что Basic, хотя большинству сейчас известна его компилируемая реализация Visual Basic, сделанная фирмой Microsoft [15, 26, 43]. Тем не менее сейчас ситуация несколько изменилась, поскольку вопрос о переносимости программ и их аппаратно-платформенной независимости приобретает все большую актуальность с развитием сети Интернет (более подробно о принципах переносимости программного обеспечения и о программировании для глобальных сетей рассказано далее в главе 6 «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 некоторый 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 и взаимосвязь цепочек этого языка близки 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» или «программа ассемблера».

Язык ассемблера, как правило, содержит мнемонические коды машинных команд. Чаще всего используется англоязычная мнемоника команд, но существуют и другие варианты языков ассемблера (в том числе существуют и русскоязычные варианты). Именно поэтому язык ассемблера раньше носил название «язык мнемокодов» (сейчас это название уже практически не употребляется). Все возможные команды в каждом языке ассемблера можно разбить на две группы: в первую группу входят обычные 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 задача распределения 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 составляющих, расположенных последовательно одна за другой: поле метки, код операции и поле операндов. Компилятор с языка ассемблера обычно предусматривает возможность наличия во входной программе комментариев, которые отделяются от Команд заданным разделителем [53, 58].

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 за-

висит от кода операции — каждая операция языка ассемблера предусматривает строго заданное число своих операндов. Соответственно, 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.

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, идентификаторов 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(O)
loops:  mov     datas[bx+4],cx
        dec     bx
        jnz     loops
```

представляет собой пример последовательности команд языка ассемблера процессоров семейства Intel 8Єx86. Здесь присутствуют команда описания набора данных (db), метка (loops), коды операций (mov, dec и jnz). Операндами являются идентификатор набора данных (datas), обозначения регистров процессора (Ьх и сх), метка (loops) и константа (4). Составной операнд datas[bx+4] отображает косвенную адресацию набора данных datas по базовому регистру Ьх со смещением 4.

Подобный синтаксис языка без труда может быть описан с помощью регулярной 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 то, какие операнды и в каком количестве допустимы для того или иного кода операции.

Поэтому семантический анализ в компиляторе с языка ассемблера также прост, как и синтаксический. Основной его задачей является проверка допустимости операндов для каждого кода операции, а также проверка того, 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.

Схемы синтаксического и семантического анализа в компиляторе с языка ассемблера могут быть, таким образом, реализованы на основе обычного конечного автомата. Именно эта особенность 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 — компилятор с языка ассемблера, в отличие от компиляторов с языков высокого уровня, автоматически такого разделения не выполняет.

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 операнд может быть описан в программе после того, как он первый раз был использован. Тогда его адрес еще не известен на момент построения машинной команды, а поэтому требуется второй проход. Типичным примером такого операнда является метка, предусматривающая переход вперед по ходу последовательности команд.

Для облегчения труда разработчика исходных программ на языке ассемблера используются макрокоманды. Практически все современные компиляторы с язы-

ков 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. Примером может служить последовательность команд, выполняемых каждый раз для организации стекового дисплея памяти при входе в процедуру или функцию.

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* [45, 53, 58].

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 лишь 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 только 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:

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

[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.

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

```
3dd_abx    4,8
```

будет в результате макроподстановки заменена на последовательность команд:

```
add     ax.4
add     bx.4
add     ex.8
push    ax
```

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

```
1oop_ax  macro    xl,x2.yl
         local    loopax
         mov      ax.xl
         xor      bx.bx
1oopax:  add      bx.yl
         sub      ax,x2
         jge      loopax
         endm
```

Здесь метка 1 oopax 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 1 oopax. В таком варианте макрогенератор должен использовать более сложные методы текстовых подстановок, аналогичные тем, что используются в компиляторах нри 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. Там их обрабатывает специальный модуль, называемый препроцессором языка (например, широко известен препроцессор языка С). 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 Pascal, 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 результирующей 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. Результаты вызова функции и макрокоманды могут из-за этого серьезно отличаться.

Рассмотрим пример на языке С.

Если описана функция

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

и аналогичная ей макрокоманда

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

то результаты их вызова не всегда будут одинаковы.

Действительно, вызовы j=fl(i) и j=f2(i) (где i и] - некоторые целочисленные переменные) приведут к одному и тому же результату. Но вызовы j=fl(++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 fl(++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 на фазе лексического анализа. Их характеристики определяются на 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 числу различных идентификаторов, найденных в исходной программе. Каждое поле содержит в себе полную информацию о данном элементе таблицы. Компилятор может работать с одной или несколькими таблицам идентифика-

торов — 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. Например, в таблицах идентификаторов может храниться следующая информация:

- для переменных:
  - имя переменной;
  - тип данных переменной;
  - область памяти, связанная с переменной;
- для констант:
  - название константы (если оно имеется);
  - значение константы;
  - тип данных константы (если требуется);
- для функций:
  - имя функции;
  - количество и типы формальных аргументов функции;
  - тип возвращаемого результата;
  - адрес кода функции.

Приведенный выше состав хранимой информации, конечно же, является только примерным. Другие примеры такой информации указаны в [17, 32, 50]. Конкретное наполнение таблиц идентификаторов зависит от реализации компилятора. Кроме того, 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.

Вне зависимости от реализации компилятора принцип его работы с таблицей идентификаторов остается одним и тем же — 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 [23].

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), то для таблицы, содержащей N элементов, в среднем будет выполнено N/2 сравнений [12].

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 (с которыми можно выполнить прямое сравнение искомого 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 $T_п = 0(\log_2 N)$. Для сравнения: при для N = 128 бинарный поиск потребует максимум 8 сравнений, а поиск в неупорядоченной таблице — в среднем 64 сравнения. Метод называют «бинарным поиском», поскольку на каждом шаге объем

рассматриваемой информации сокращается в 2 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. Таблица 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 [12], а поиск места включения осуществлять с помощью алгоритма бинарного поиска, то среднее время, необходимое на помещение всех элементов в таблицу, можно оценить следующим образом:

$$T_3 \sim 0(N*\log_2 N) + k*0(N^2)$$

Здесь к — некоторый коэффициент, отражающий соотношение между временем, затрачиваемым компьютером на выполнение операции сравнения и операции 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.

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.

[1] Как минимум, при добавлении нового идентификатора в таблицу компилятор должен проверить, существует ли 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.

Дерево называется бинарным, так как каждая вершина в нем может иметь не более двух ветвей. Для определенности будем называть 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 очередной идентификатор меньше, то перейти к шагу 5, иначе — перейти к шагу 6.

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.

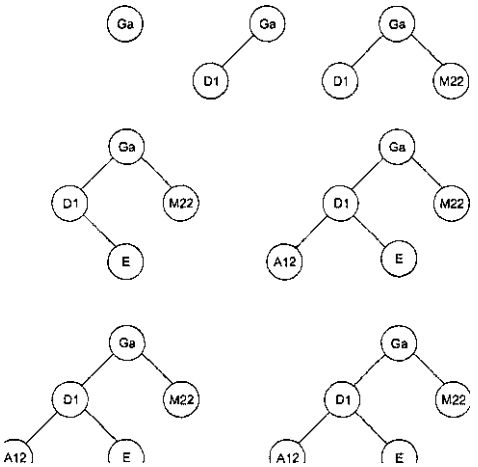


Рис. 2.2. Заполнение бинарного дерева для последовательности идентификаторов GA, D1, M22, E, A12, BC, F

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 Dl. Опять сравниваем идентификаторы. Искомый идентификатор меньше — текущим узлом становится левая вершина A12. При следующем сравнении искомый идентификатор найден.

Если искать отсутствующий идентификатор — например, All, — то поиск опять пойдет от корневой вершины. Сравниваем идентификаторы GA и All. Искомый идентификатор меньше — текущим узлом становится левая вершина 01. 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 рассмотренном выше примере вместо последовательности идентификаторов 3A, Dl, M22, E, A12, BC, F взять последовательность A12, GA, Dl, M22, E, BC, F, то полученное дерево будет иметь иной вид. А если в качестве примера взять последовательность идентификаторов A, B, C, D, E, F, 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_3) и на поиск элемента в нем ($T_{„}$) можно оценить следующим образом [4, т.2]:

$T_3 = N*0(\log_2 N)$

$T_{„} - 0(\log_2 N)$

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 [17, 50].

Хэш-функции и хэш-адресация

Принципы работы хэш-функций

Логарифмическая зависимость времени поиска и времени заполнения таблицы идентификаторов — это самый хороший результат, 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 хэш-функций и хэш-адресации.

Хэш-функцией F называется некоторое отображение множества входных элементов R на множество целых неотрицательных чисел Z: F(r) = n, reR, neZ. Сам термин «хэш-функция» происходит от английского термина «hash function» (hash — «мешать», «смешивать», «путать»). Вместо термина «хэширование» иногда используются термины «рандомизация», «переупорядочивание».

Множество допустимых входных элементов R называется областью определения хэш-функции. Множеством значений хэш-ф'ункции F называется подмножество M из множества целых неотрицательных чисел Z: McZ, содержащее все возможные значения, возвращаемые функцией F: VreR: F(r)eM и VmeM: 3reR: F(r) = m. Процесс отображения области определения хэш-функции на множество значений называется «хэшированием».

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

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

Метод организации таблиц идентификаторов, основанный на использовании хэш-адресации, заключается в размещении каждого элемента таблицы в ячейке, адрес которой возвращает хэш-функция, вычисленная для этого элемента. Тогда в идеальном случае для размещения любого элемента в таблице идентификаторов достаточно только вычислить его хэш-функцию и обратиться к нужной ячейке массива данных. Для поиска элемента в таблице необходимо вычислить хэш-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 хэш-адресации. Трем различным идентификаторам Al, A2, A3 соответствуют на рисунке три значения хэш-функции ml, n2, n3. В ячейки, адресуемые nl, n2, n3, помещается информация об идентификаторах Al, A2, A3. При поиске идентификатора A3 вычисляется значение адреса n_3 и выбираются данные из соответствующей ячейки таблицы.

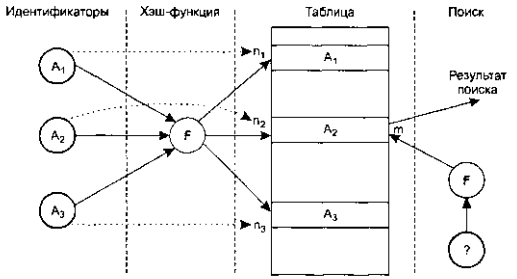


Рис. 2.3. Организация таблицы идентификаторов с использованием хэш-адресации

Этот метод весьма эффективен, поскольку как время размещения элемента в таблице, так и время его поиска определяются только временем, затрачиваемым на вычисление хэш-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 хэш-функции, в то время как реально хранимых в таблице идентификаторов может быть существенно меньше. Второй недостаток — необходимость соответствующего разумного выбора хэш-функции. Этому существенному вопросу посвящены следующие два подпункта.

Построение таблиц идентификаторов на основе хэш-функций

Существуют различные варианты хэш-функций. Получение результата хэш-функции — «хэширование» — обычно достигается за счет выполнения над цепочкой символов некоторых простых арифметических и логических операций. Самой простой хэш-функцией для символа является код внутреннего представления в компьютере литеры символа. Эту хэш-функцию можно использовать и для цепочки символов, выбирая первый символ в цепочке. Так, если двоичное ASCII-представление символа A есть двоичный код 001000012, то результатом хэширования идентификатора ATaBle буДет код 001000012.

Хэш-функция, предложенная выше, очевидно не удовлетворительна: при использовании такой хэш-функции возникнет проблема — двум различным идентификаторам, начинающимся с одной и той же буквы, будет соответствовать одно и то же значение хэш-функции. Тогда при хэш-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 функции, называется *коллизией.*

Естественно, что хэш-функция, допускающая коллизии, не может быть напрямую использована для хэш-адресации в таблице идентификаторов. Причем достаточно получить хотя бы один случай коллизии на всем множестве идентификаторов, чтобы такой хэш-функцией нельзя было пользоваться непосредственно. Но в примере взята самая элементарная хэш-функция. А возможно ли построить хэш-функцию, которая бы полностью исключала возникновение коллизий?
Очевидно, что для полного исключения коллизий хэш-функция должна быть взаимно однозначной: каждому элементу из области определения хэш-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. Тогда любым двум произвольным элементам из области определения хэш-функции будут всегда соответствовать два различных ее значения. Теоретически для идентификаторов такую хэш-функцию построить можно, так как и область определения хэш-функции (все возможные имена идентификаторов), и область ее значений (целые неотрица-

тельные числа) являются бесконечными счетными множествами. Теоретически можно организовать взаимно однозначное отображение одного счетного множества на другое, но практически это сделать исключительно сложно[1].

Практически существует ограничение, делающее создание взаимно однозначной хэш-функции для идентификаторов невозможным. Дело в том, что в реальности область значений любой хэш-функции ограничена размером доступного адресного пространства в данной архитектуре компьютера. При организации хэш-адресации значение, используемое в качестве адреса таблицы идентификаторов, не может выходить за пределы, заданные разрядностью адреса компьютера[2]. 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 хэш-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 хэш-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 *рехэширования* (или расстановка). Согласно этому методу, если для элемента A адрес h(A), вычисленный с помощью хэш-функции h, указывает на уже занятую ячейку, то необходимо вычислить значение функции $n_i = h_i(A)$ и проверить занятость ячейки по адресу n_i. 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 хэш-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 ^ пустая, то поместить в нее элемент A и завершить алгоритм, иначе перейти к шагу 4.

[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 хэш-адресации. Практическая ценность такого отображения весьма сомнительна.

[2] 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 4. Если $п = п_н$ то сообщить об ошибке и завершить алгоритм, иначе i := i + 1 и вернуться к шагу 3.

Тогда поиск элемента А в таблице идентификаторов, организованной таким образом, будет выполняться по следующему алгоритму:

Шаг 1. Вычислить значение хэш-функции п = h(A) для искомого элемента А.

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

Шаг 3. Вычислить $n_i = h_i(A)$. Если ячейка по адресу n_i пустая или $п = n_i$, то элемент не найден и алгоритм завершен, иначе сравнить имя элемента в ячейке n_i с именем искомого элемента А. 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 хэш-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 рехэширования необходимо определить все хэш-функции h_i для всех i. Чаще всего функции h_i определяют как некоторые модификации хэш-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 хэш-функции h. В свою очередь, самым простым подходом здесь будет положить Pi = i. Тогда получаем формулу $h_i(A) = (h(A) + i) \bmod N_m$. В этом случае при совпадении значений хэш-функции для каких-либо элементов поиск свободной ячейки в таблице начинается последовательно от текущей позиции, заданной хэш-функцией h(A).

Этот способ нельзя признать особенно удачным — при совпадении хэш-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 [17]:

$T_{ср} \approx 0{,}5 \cdot (1 - Lf/2)/(1 - Lf))$.

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: $n^{\wedge}$ n_2, n_3, n_4, n_5 и ряд идентификаторов, которые надо разместить в ней: A_k A_2, A_3, A_4, A_5 при условии, что $h(A_1) = h(A_2) = h(A_5) = n_{(}$; $h(A_3) = n_2$; $h(A_4) = n_4$. Последовательность размещения идентификаторов в таблице при использовании простейшего метода рехэширования показана на рис. 2.4. В итоге после размещения в таблице для поиска идентификатора $A_{(}$ потребуется 1 сравнение, для A_2 — 2 сравнения, для A_3 — 2 сравнения, для A_4 — 1 сравнение и для A_5 — 5 сравнений.

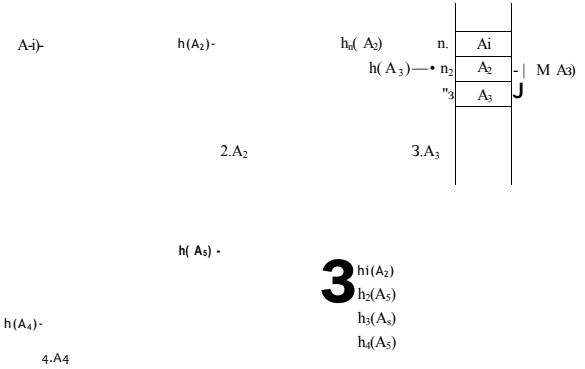


Рис. 2.4. Заполнение таблицы идентификаторов при использовании простейшего рехэширования

Даже такой примитивный метод рехэширования является достаточно эффективным средством организации таблиц идентификаторов при частном заполнении таблицы. Имея, например, заполненную на 90% таблицу для 1024 идентификаторов, в среднем необходимо выполнить 5.5 сравнений 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.

Среднее время на помещение одного элемента в таблицу и на поиск элемента в таблице можно снизить, если применить более совершенный метод рехэширования. Одним из таких методов является использование в качестве p, для функции $h_s(A) = (h(A) + p_,) \bmod N_m$ последовательности псевдослучайных целых чисел p_k p_2,..., p_k. При хорошем выборе генератора псевдослучайных чисел длина последовательности к будет $к = N_m$. Тогда среднее время поиска одного элемента в таблице можно оценить следующим образом [17]:

$$E_{,,} = 0((l/Lf)*log_2(1 - Lf)).$$

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

Построение таблиц идентификаторов по методу цепочек

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

В ячейках хэш-таблицы может храниться либо пустое значение, либо значение указателя на некоторую область памяти из основной таблицы идентификаторов. Тогда хэш-функция вычисляет адрес, по которому происходит обращение сначала к хэш-таблице, а потом уже через нее по найденному адресу — к самой таблице идентификаторов. Если соответствующая ячейка таблицы идентификаторов пуста, то ячейка хэш-таблицы будет содержать пустое значение. Тогда вовсе не обязательно иметь в самой таблице идентификаторов ячейку для каждого возможного значения хэш-функции — таблицу можно сделать динамической так, чтобы ее объем рос по мере заполнения (первоначально таблица идентификаторов не содержит ни одной ячейки, а все ячейки хэш-таблицы имеют пустое значение). Такой подход позволяет добиться двух положительных результатов: во-первых, нет необходимости заполнять пустыми значениями таблицу идентификаторов — это можно сделать только для хэш-таблицы; во-вторых, каждому идентификатору будет соответствовать строго одна ячейка в таблице идентификаторов (в ней не будет пустых неиспользуемых ячеек). Пустые ячейки в таком случае будут только в хэш-таблице, и объем неиспользуемой памяти не будет зависеть от объема информации, хранимой для каждого идентификатора, — для каждого значения хэш-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 хэш-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 хэш-таблицы поместить пустое значение, таблица идентификаторов пуста, переменная FreePtr (указатель первой свободной ячейки) указывает на начало таблицы идентификаторов; i := 1.

Шаг 2. Вычислить значение хэш-функции n, для нового элемента A_i. Если ячейка хэш-таблицы по адресу n, 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 хэш-таблицы адрес ячейки таблицы идентификаторов mj и перейти к шагу 4.

Шаг 4. Для ячейки таблицы идентификаторов по адресу mj 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 mj и повторить шаг 4.

Шаг 5. Добавить в таблицу идентификаторов новую ячейку, записать в нее информацию для элемента A, (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 хэш-функции n для искомого элемента A. Если ячейка хэш-таблицы по адресу n пустая, то элемент не найден и алгоритм завершен, иначе положить] := 1, выбрать из хэш-таблицы адрес ячейки таблицы идентификаторов mj.

Шаг 2. Сравнить имя элемента в ячейке таблицы идентификаторов по адресу mj 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_r Если оно пустое, то искомый элемент не найден и алгоритм завершен; иначе j := j + 1, выбрать из поля ссылки адрес mj 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 хэш-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 хэш-таблицы и таблицы идентификаторов для примера, который ранее был рассмотрен на рис. 2.4 для метода простейшего рехзширования. После размещения в таблице для поиска идентификатора A1 потребуется 1 сравнение, для A2 — 2 сравнения, для A3 — 1 сравнение, для A4 — 1 сравнение и для A5 — 3 сравнения (сравните с результатами простого рехэширования).

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, возни-

кающих 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.

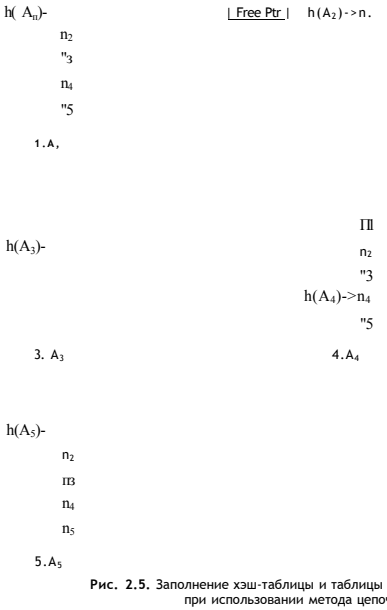


Рис. 2.5. Заполнение хэш-таблицы и таблицы идентификаторов при использовании метода цепочек

Комбинированные способы построения таблиц идентификаторов

Выше в примере была рассмотрена весьма примитивная хэш-функция, которую никак нельзя назвать удовлетворительной. Хорошая хэш-функция распределяет

поступающие на ее вход идентификаторы равномерно на все имеющиеся в распоряжении адреса, так что коллизии возникают не столь часто. Существует большое множество хэш-функций. Каждая из них стремится распределить адреса под идентификаторы по своему алгоритму, но, как было показано выше, идеального хэширования достичь невозможно.

В реальных компиляторах практически всегда так или иначе используется хэш-адресация. Алгоритм применяемой хэш-функции обычно составляет «ноу-хау» разработчиков компилятора. Обычно при разработке хэш-функции создатели компилятора стремятся свести к минимуму количество возникающих 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 хэш-функция — это шаг к значительному ускорению работы компилятора, поскольку обращение к таблицам идентификаторов выполняются многократно на различных фазах компиляции.

То, какой конкретно метод применяется в компиляторе для организации таблиц идентификаторов, зависит от реализации компилятора. Один и тот же компилятор может иметь даже несколько разных 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 хэш-функция, поле ссылки остается пустым. Если же возникает коллизия, то с помощью поля ссылки организуется поиск идентификаторов, для которых значения хэш-функции совпадают, по одному из рассмотренных выше методов: неупорядоченный список, упорядоченный список или же бинарное дерево. При хорошо построенной хэш-функции коллизии будут возникать редко, поэтому количество идентификаторов, для которых значения хэш-функции совпали, будет не столь велико. Тогда и время поиска одного среди них будет незначительным (в принципе, при высоком качестве хэш-функции подойдет даже перебор по неупорядоченному списку).

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

Хэш-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, 17, 50].

Контрольные вопросы и задачи

Вопросы

1. Что такое трансляция, компиляция, транслятор, компилятор? Из каких процессов состоит компиляция? Расскажите об общей структуре компилятора.
2. Как решается вопрос о смысле исходной программы в современных компиляторах?
3. Какую роль выполняет лексический анализ в процессе компиляции? Как могут быть связаны между собой лексический и синтаксический анализы?
4. Компилятор можно построить, не используя лексический анализатор. Объясните, почему все-таки подавляющее большинство современных компиляторов используют фазу лексического анализа.
5. Постройте общую схему работы интерпретатора. В чем ее отличие от общей схемы работы компилятора, приведенной в этом пособии?
6. Постройте общую схему работы компилятора с языка ассемблера. Какие особенности присущи данному компилятору? Какие фазы компиляции в нем могут отсутствовать?
7. 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?
8. Почему в глобальной сети Интернет используются в основном языки программирования, предусматривающие интерпретацию исходного кода? Какие проблемы проистекают из этого факта?
9. 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?
10. Зачем в описании макрокоманды #define f2(a) ((a) + (a)) параметр a взят в скобки, хотя операция этого не требует?
11. Что такое таблица идентификаторов и для чего она предназначена? Какая информация может храниться в таблице идентификаторов?
12. Какие цели преследуются при организации таблицы идентификаторов? Исходя из каких характеристик оценивается эффективность того или иного метода организации таблицы?
13. Какие существуют способы организации таблиц идентификаторов?
14. Что такое коллизия? Почему она происходит при использовании хэш-функций для организации таблиц идентификаторов? В чем заключаются преимущества и недостатки метода цепочек?
15. Метод логарифмического поиска позволяет значительно сократить время поиска идентификатора в таблице. Однако он же значительно увеличивает время на помещение нового идентификатора в таблицу. Почему, тем не менее,

. можно говорить © преимуществах этосометода по сравнению с поиском методом прямого перебора?

16. Проблемы создания хорошей хэш-функции связаны с ограниченной разрядной сеткой ЭВМ и, следовательно, ограниченным размером доступного адресного пространства. Но, как сказано в первой части данного учебника, размер адресного пространства можно увеличить, испойьзуя внешние накопители данных (прежде всего жесткие диски) и механизм виртуальной памяти. Почему эти возможности не используются компиляторами при организации хэш-функций?

17. Как могут быть скомбинированы различные методы организации таблиц идентификаторов ? • >

18. Чем различаются таблица лексем и таблица идентификаторов? В какую из этих таблиц лексический анализатор не должен помещать ключевые слова, разделителен знаки операций?

Задачи

1.. В программе на языке С макрокоманда 1вс1(а,Ь) и функция Inc2(a,b), выполняющие действие а + 2*Ь, описаны следующим образом;

```
#defir»e  Incl(a.b)  {(aЖМИМ)
int Inc2(unt a.  Int Ь) (return a+b+b;}      ;
```

К каким результатам приведут следующие вызобы для целочисленных переменных1,|йк:

```
k-Incl(i.l): к- Incld.Jh- к- Incl(i.j+1); к- IncUl.J**):
к» Incl(T++Jnc2(j.1)):
k-Inc2<1.1): k- Inc2(1vJ): к- Inc2(i.j+1>; к- Inc2(i J++);
к- Inc2(i"++.Incl(j,l>);
```

' В каких из перечисленных случаев будет получен логически неверный результат? . •

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

```
fdefine 1nc1Ц.Ь)  ((a)+(2*(Ь)))
```

Приведет ли это к логически правильным результатам во всех вышеприведенных случаях?

Будет ли верным следующее определение макрокоманды:

```
#define Incl(a.b)  ((a)+(2*b))
```

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

3. 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.

4. 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.
5. Напишите программу, создающую таблицу идентификаторов с помощью хэш-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 легко подменять используемую хэш-функцию. Подсчитывая число коллизий и среднее количество сравнений для поиска идентификатора, сравните результаты для различных хэш-функций. В качестве исходных данных для хэш-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.
6. Выполните действия, описанные в задаче № 5, для таблицы идентификаторов, организованной на основе метода цепочек. Сравните результаты.
7. 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 хэш-функций, предложенных в задаче № 5. Попробуйте предложить свой вариант хэш-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 т. п. 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, а выходная информация передается для дальнейшей обработки компилятором на этапе синтаксического анализа и разбора.

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:

[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.

- применение лексического анализатора упрощает работу с текстом исходной программы на этапе синтаксического разбора и сокращает объем обрабатываемой информации, так как лексический анализатор структурирует поступающий на вход исходный текст программы и отбрасывает всю незначащую информацию;
- для выделения в тексте и разбора лексем возможно применять простую, эффективную и теоретически хорошо проработанную технику анализа, в то время как на этапе синтаксического анализа конструкций исходного языка используются достаточно сложные алгоритмы разбора;
- сканер отделяет сложный по конструкции синтаксический анализатор от работы непосредственно с текстом исходной программы, структура которого может варьироваться в зависимости от версии входного языка — при такой конструкции компилятора для перехода от одной версии языка к другой достаточно только перестроить относительно простой лексический анализатор.

Функции, выполняемые лексическим анализатором, и состав лексем, которые он выделяет в тексте исходной программы, могут меняться в зависимости от реализации компилятора. То, какие функции должен выполнять лексический анализатор и какие типы лексем он должен выделять во входной программе, а какие оставлять для этапа синтаксического разбора, решают разработчики компилятора. В основном лексические анализаторы выполняют исключение из текста исходной программы комментариев, незначащих 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.

Результатом работы лексического анализатора является перечень всех найденных в тексте исходной программы лексем с учетом характеристик каждой лексемы. 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.

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. Также можно отметить, что лексемы в таблице лексем обязательно располагаются в том же порядке, как 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	XI
for	Ключевое слово	X2
i	Идентификатор	i: 1
:-	Знак присваивания	**SI**
1	Целочисленная константа	**1**
to	Ключевое слово	X3
N	Идентификатор	N: 2
do	Ключевое слово	X4
fg	Идентификатор	fg:3
	Знак присваивания	SI
fe	Идентификатор	fg:3
*	Знак арифметической операции	Al
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

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.

Существуют правила, с помощью которых для любой регулярной грамматики может быть построен конечный автомат, распознающий цепочки языка, заданного этой грамматикой (эти правила рассмотрены далее в этой главе).

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.

Определение границ лексем

Выделение границ лексем является нетривиальной задачей. Ведь в тексте исходной программы лексемы не ограничены никакими специальными символами. 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 l=1 невозможно определить тип оператора языка (а соответственно, и границы лексем). В случае DO 10 1=1.15 это присвоение вещественной переменной D0l0I значения константы 1.15 (пробелы в языке FORNTAN игнорируются), а в случае DO 10 1=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: к = = т+++++J,. Существует только одна единственно верная трактовка этого оператора: к = 1++ + ++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, ч 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 организации 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, но в дальнейшем не изменяет. Дальнейшую обработку таблицы лексем выполняют следующие фазы компиляции. Если в процессе разбора лексический анализатор не смог правильно определить тип лексемы, то считается, что исходная программа содержит ошибку.

При параллельном варианте лексический анализ текста исходной программы выполняется поэтапно, по шагам. Лексический анализатор выделяет очередную лексему в исходном коде и передает ее синтаксическому анализатору. Синтаксический анализатор, выполнив разбор очередной конструкции языка, 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, какого типа лексему следует ожидать. Взаимодействуя между собой таким образом, лексический и синтаксические 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 (обычно такой конструкцией является оператор 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. 3.1.

Последовательная работа лексического и синтаксического анализаторов представляет собой самый простой вариант их взаимодействия. Она проще в реализации и обеспечивает более высокую скорость работы компилятора, чем их параллельное взаимодействие. 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. Важ-

[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 не обязателен.

но отметить, что знаки операций сами также являются лексемами, и необходимо не пропустить их при распознавании текста.

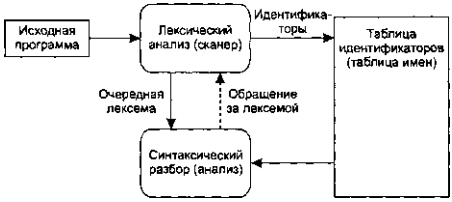


Рис. 3.1. Параллельное взаимодействие лексического и синтаксического анализаторов

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 данных добавляется в лексему всегда, когда он может быть туда добавлен. Как только символ не может быть добавлен в лексему, то считается, что он является границей лексемы и началом следующей лексемы (если символ не является пустым разделителем — пробелом, символом табуляции или перевода строки, знаком комментария).

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 = 1+++++J: будет разбита на лексемы следующим образом: < = -,++++ + $j_и$ 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 ••'сложнить лексический распознаватель неизбежно приведут к необходимости его взаимосвязи с синтаксическим разбором. Это потребует организации их паэаллельной работы и неизбежно снизит эффективность работы всего компилятора. Возникшие накладные расходы никак не оправдываются достигаемым эффектом — распознаванием строк с сомнительными лексемами. Достаточно :бязать пользователя явно указать с помощью пробелов (или других незначалих символов) границы лексем, что значительно проще[1].

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

Не для всех входных языков тёкой подход возможен. Например, для рассмотренного выше примера с язьща FORTRAN невозможно применить указанный метод — разница между оператором цикла и оператором присваивания слишком существенна, чтобы «о можно было пренебречь. Здесь придется прибегнуть к связи с синтаксическим разбором[1]. В таком случае приходится организовывать параллельную работу лексического и синтаксического анализаторов.

Очевидно, что последовательный вариант организации взаимодействия лексического и синтаксического анализаторов является более эффективным, так как он не требует организации сложных механизмов обмена данными и не нуждается в повторном прочтении уже разобранных лексем. Этот метод является и более простым. Однако не для всех языков программирования возможно организовать такое взаимодействие. Это зависит в основном от синтаксиса языка, заданного его грамматикой. Большинство современных широко распространенных языков программирования, таких как C и Pascal, 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, 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, 28, 32]. 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 ее в таблице идентификаторов и запись нотой лексемы в таблицу идентификаторов. Набор 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 переходсш на графе автомата (или в функции переходов автомата) связать выполнение некоторой произвольной функции f(q,a), где q — текущее состояние автомата, а — 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;

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

- 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, как уже было сказано, относятся два типа грамматик: леволинейные и праволинейные.

Леволинейные грамматики G(VT,VN,P,S), V = VNuVT могут иметь правила двух видов: A By или A -» y, где A,BeVN, yeVT*.

В свою очередь, праволинейные грамматики G(VT,VN,P,S), V = VNuVT могут иметь правила также двух видов: A yB или A -» y, где A,BeVN, yeVT.

Доказано, что эти два класса грамматик эквивалентны. Для любого регулярного языка, заданного праволинейной грамматикой, может быть построена леволинейная грамматика, определяющая эквивалентный язык; и наоборот — для любого регулярного языка, заданного леволинейной грамматикой, может быть построена праволинейная грамматика, задающая эквивалентный язык.

Разница между леволинейными и праволинейными грамматиками заключается в основном в том, в каком порядке строятся предложения языка: слева направо для леволинейных, либо справа налево для праволинейных. Поскольку предложения языков программирования строятся, как правило, в порядке слева направо, то в дальнейшем в разделе регулярных грамматик будет идти речь в первую очередь 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(VT,VN,P,S), V = VNuVT могут иметь правила двух видов: A Bt или A -> t, где A.BeVN, teVT.

Праволинейные автоматные грамматики G(VT,VN,P,S), V = VNuVT могут иметь правила двух видов: A tB или A t, где A,BeVN, teVT.
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
где S — целевой символ грамматики. При этом символ S не должен встречаться в правых частях других правил грамматики. Тогда язык, заданный автоматной грамматикой G, может включать в себя пустую цепочку: XeL(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 -> к 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(VT,VN,P,S), необходимо преобразовать ее в почти эквивалентную автоматную грамматику G'(VT,VN',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 -> Ba_t, A,BeVN, a_teVT или вида A -> a_b AeVN, a_teVT, то они переносятся во множество P' правил грамматики G' без изменений.

Если встречаются правила вида A -> Ba^-.a,,, n > 1, A,BeVN, Vn > i > 0: a^ VT, то во множество нетерминальных символов VN' грамматики G' добавляются символы A!,A_2....A_{n_j}, а во множество правил P' грамматики G' добавляются правила:

A -> A^a,,

$A_{n\_}$! -> $A_{n\_2}a_n$.!

A2 AЙ2

A[Ba!

Если встречаются правила вида A a^-.a,,, n > 1, AeVN, Vn > i > 0: a,eVT, то во множество нетерминальных символов VN' грамматики G' добавляются символы AьAг.-.Aп,!, а во множество правил P' грамматики G' добавляются правила:

A $A_{n\_1}a$,,

$A_{n\_}$! $A_{n\_2}a_{n\_}$!

A2 ^ Aj3.2

A, ->a_t

Если встречаются правила вида A B или вида A X, 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 B или вида A —> к.

Если находится правило вида A B, то просматривается множество правил P' грамматики G'. Если в нем присутствуют правила вида B -» C, B -» Ca, B a или B -> X, то в него добавляются правила вида A -» C, A -» Ca, A a и A A. соответственно, VA,B,Ce VN', Vae 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 -» B удаляется из множества правил P'.

Если находится правило вида A X (и символ A не является целевым символом S), то просматривается множество правил P' грамматики G'. Если в нем присутствуют правила вида B A или B -» Aa, то в него добавляются правила вида B -> X и B -> a соответственно, VA,BeVN', VaeVT (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 -> X удаляется из множества правил P'.

Шаг 4. Если на шаге 3 было найдено хотя бы одно правило вида A B или *A—>X* зо 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 -» X

(такие правила называются ^-правилами). Реальные регулярные грамматики обычно не содержат правил такого вида. Тогда алгоритм преобразования грамматики к автоматному виду существенно упрощается. Кроме того, эти правила можно было бы устранить предварительно с помощью специальных алгоритмов преобразования (эти алгоритмы рассмотрены дальше, в главе 4 «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 S1, а само правило разбиваем на два: S -> S1) и S1 -> 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 C1, а само правило разбиваем на два: C -> C1* и C1 -» (; 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.

Шаг3. Правил вида A-»B или A->л 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.S1.C.C1.K}.P',S):**
P':
S -> S1) | K}
S1 -> C*
C C1* | Ca | C{ (C} | C(| C* | C)
C1 -> (
к { | Ka | K(| к* | K) | к{

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:

$M(Q\ V,\ 5,\ q_0,\ F)$,

где

Q. — конечное множество состояний автомата;

V — конечное множество допустимых входных символов (алфавит автомата);

8 — функция переходов, отображающая V*Q, (декартово произведение множеств) во множество подмножеств Q.: R(Q), то есть 8(a,q)=R, aeV, qeQ, RcQ;

q_0 — начальное состояние автомата Q, q_0eQ;

F — непустое множество конечных состояний автомата, FcQ, F * 0.

КА называют *полностью определенным,* если в каждом его состоянии существует функция перехода для всех возможных входных символов, то есть: VaeV, VqeQ. 38(a,q) = R, RcQ.

Работа конечного автомата представляет собой последовательность шагов (или тактов). На каждом шаге работы автомат находится в одном из своих состояний 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 8. Она зависит не только от текущего состояния, но и от символа из алфавита 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 coeV^+.

Видно, что конфигурацию КА на каждом шаге работы можно определить в виде (q,co,n), где q — текущее состояние автомата, qeQ; co — цепочка входных символов, coeV^+; n — положение указателя в цепочке символов, neNu{0}, n < |co| (N — множество натуральных чисел). Конфигурация автомата на следующем шаге — это (q',co,n+l), если q'e8(a,q) и символ aeV находится в позиции n + 1 цепочки co. Начальная конфигурация автомата: (q_0,co,0); заключительная конфигурация автомата: (f, co, n), feQ, n = |co|, она является конечной конфигурацией, если feF.

Язык, заданный конечным автоматом

КА M(Q,V,8,q_0,F) *принимает цепочку символов* coeV^+, если, получив на вход эту цепочку, он из начального состояния q_0 может перейти в одно из конечных состояний feF. В противном случае КА не принимает цепочку символов.

Язык L(M), заданный КА MCQjVAqo.F), — это множество всех цепочек символов, которые принимаются этим автоматом. Два КА эквивалентны, если они задают один и тот же язык.

Все КА являются распознавателями для регулярных языков [4, т. 1, 15, 28].

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.qeQ, помеченная символом aeV, если в КА определена 5(a,p) и qeS(a,p). Начальное и конечное 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} ,8.H, {S}) ;8: 8(H.b)=B. 8(B.a)=A, 8(A,b)={B,S}. На рис. 3.2 приведен пример графа состояний для этого КА.

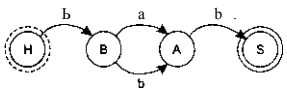


Рис. 3.2. Граф переходов недетерминированного конечного автомата

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} ,8.H, {S}): 8: 8(H,a)=E, 8(H,b)=B. 8(B.a)-A, 5(B.b)=E, 8(A,a)={E}. 8(A.b)={B.S}. 8(E,a)={E}. 8(E.b)={E}. 5(S.a)={E}. 8(S,b)={E}. Состояние E как раз соответствует состоянию «ошибка». Граф переходов этого КА представлен на рис. 3.3.

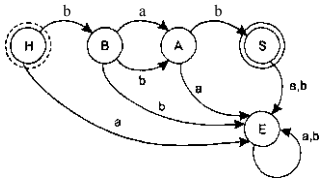


Рис. 3.3. Граф переходов полностью определенного недетерминированного конечного автомата

Детерминированные и недетерминированные конечные автоматы

Определение детерминированного конечного автомата

Конечный автомат M(Q,V,8,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 VaeV, VqeQ: либо 8(a,q) - {r} reft либо 8(a,q) - 0.**

В противном случае конечный автомат называют *недетерминированным.*

Из этого определения видно, что автоматы, представленные ранее на рис. 3.2 и 3.3 являются недетерминированными КА.

ДКА может быть задан в виде пятерки:

M(Q V, 8, qo, F),

где

Q, — конечное множество состояний автомата;

V — конечное множество допустимых входных Символов;

8 — функция переходов, отображающая V*Q, в множество Q; 5(a,q) - г, aeV, q,reQ;

q_0 — начальное состояние автомата Q, q_0eQ;

F — непустое множество конечных состояний автомата, FcQ, F * 0.

Если функция переходов ДКА определена для каждого состояния автомата, то автомат называется *полностью определенным* **ДКА: VaeV, VqsQ; либо 3S(a,q) - г, reQ. 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,5,q_0,F) в эквивалентный ему ДКА M'(Q',V,5',q'o,F') заключается в следующем:

1. **Множество состояний Q,' автомата M' строится из комбинаций всех состояний множества Q, автомата M. Если qi,q2,-,q,,, n > 0 — состояния автомата M, V0 < i < n q,eQ, то всего будет 2"-1 состояний автомата M'. Обозначим их так: [q,,q_2....q j , 0 < m < n.**
2. **Функция переходов 8' автомата M' строится так: 5'(a,[q_1,q_2,—,q_m]) - [r_1,r_2,...,r_k], где VO < i < m 30 < j < к так, что 8(a,^) = r_j.**
3. **Обозначим q'$_0$ - [qj.**
4. **Пусть f,,f_2,....fi, 1 > 0 — конечные состояния автомата M, V0 < i < 1 f^eF, тогда множество конечных состояний F автомата M' строится из всех состояний, имеющих вид [...,fj,...], f,eF.**

Доказано, что описанный выше алгоритм строит ДКА, эквивалентный заданному произвольному КА.

После построения из нового ДКА необходимо удалить все недостижимые состояния.

Состояние q e Q в КА M(Q,V,8,q_0,F) называется недостижимым, если ни при какой входной цепочке coe 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 R и множество текущих активных состояний на каждом шаге алгоритма P_i. Результатом работы алгоритма является полное множество достижимых состояний R. Рассмотрим работу алгоритма по шагам:

1. **R :- {q_0}; i :- 0; P_0 := {q_0}.**
2. P_{1+1} :- 0.
3. **VaeV, VqeP,: P_{v+1} :- P_{v+1}u8(a,q).**
4. **Если P_{v+1} - R = 0, то выполнение алгоритма закончено, иначе R RuP$_{1+1}$, i ;= i + 1 и перейти к шагу 3.**

После выполнения данного алгоритма из КА можно исключить все состояния, не входящие в построенное множество R.,

Рассмотрим работу алгоритма преобразования произвольного КА в ДКА на примере автомата M({H,A,B.S}.{a,Ь},8,H,{S}): 8: 8(H,b)=B, 8(B,a)-A. 8(A.b)={B,S}. Видно, что это недетерминированный КА (из состояния А возможны два различных перехода по символу Ь). Граф переходов для этого автомата был изображен выше на рис. 3.2.

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:

8'([H].Ь)-[B]
6'([A].bMBS]
б'([B].a)-[A]
6'([HA].b)=[BS]
б'([HB].a)-[A]
6'([HB].ЬMB]
6'([HS].bMB]
8'([AB],a)-[A]
S'([AB].b)=[BS]
6'([AS],b)=[BS]
S'([BS].a)=[A]
6'([HAB].a)=[A]
8'([HAB],b)=[BS]
8'([HAS],b)-[BS]

6'([HBS],b)-[B]
8'([HBS],a)=[A]
8'([ABS].b)-[BS]
6'([ABS].a)=[A]
8'([HABS],a)=[A]
8'([HABS],b)=[BS]

Начальное состояние эквивалентного ДКА:

qO' = [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]}),
8([H].ЬМВ], 8([B],a)-[A]. 8([A],b)=[BS]. 8([BS].a)-[A].

Ничего не изменяя, переобозначим состояния ДКА. Получим:

M'({H,B.A,S},{a.b}.H,{S}),
8(H,b)=B, 5(B,a)=A, 8(A.b)=S, 5(S.a)=A.

Граф переходов полученного ДКА изображен на рис. 3.4.

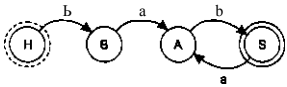


Рис. 3.4. Граф переходов детерминированного конечного автомата

Этот автомат можно преобразовать к полностью определенному виду. Получим граф состояний, изображенный на рис. 3.5 (состояние E — это состояние «ошибка»).

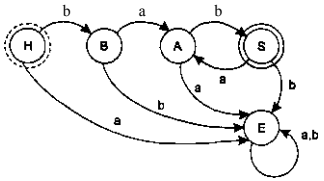


Рис. 3.5. Граф переходов полностью определенного детерминированного конечного автомата

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 существенно возрасти и в худшем случае составит 2" - 1, где п — количество состояний исходного КА. В этом случае затраты на моделирование ДКА окажутся больше, чем на моделирование исходного КА.

Поэтому не всегда выполнение преобразования автомата к детерминированному виду 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 $M(Q,V,5,q_0,F)$ qeQ и $q'eQ$ называются п-эквивалентными (п-неразличимыми), $n > O$ $neNu\{0\}$, если, находясь в одном из этих состояний и получив на вход любую цепочку символов со: $coeV^*$ $|co| < n$, автомат может перейти в одно и то же множество конечных состояний. Очевидно, что О-эквивалентными состояниями автомата $M(Q,V,5,q_0,F)$ являются два множества его состояний: F и Q-F. Множества эквивалентных состояний автомата называют классами эквивалентности, а всю их совокупность — множеством классов эквивалентности R(n), причем R(0) - - {F,Q-F}.

Рассмотрим работу алгоритма построения эквивалентных состояний по шагам:

1. **На** первом шаге **п** := **0,** строим **R(0).**
2. На втором шаге **п** := **п** + **1,** строим **R(n)** на основе **R(n-l): R(n) = {r,(n): {q_4eQ:** $VaeV$ **5**$(a^{\wedge}{}_{n})cr/n-1)\}$ **Vi,jeN}.** То есть в классы эквивалентности на шаге п входят те состояния, которые по одинаковым символам переходят в п-1 эквивалентные состояния.
3. **Если R(n) = R(n-l), то работа алгоритма закончена, иначе необходимо вернуться к шагу 2.**

Доказано, что алгоритм построения множества классов эквивалентности завершится максимум для п = ш-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}.5.A.{D,E}). 5(A.0)={B}. 5(A.1)={C}, 5(B,1)={D}, 5(C.1)={E}. 5(D.0)={C}, 5(D.1)={E}, 5(E.0)={B}, 5(E.1)-{D}. 5(F,0)={D}, 5(F.1)={G}, 5(G,0)={F}. 5(G,1)={F}; необходимо построить эквивалентный ему минимальный КА.

Граф переходов этого автомата приведен на рис. 3.6.

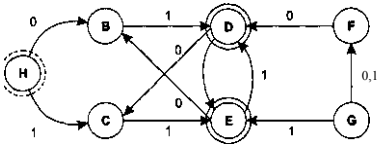


Рис. 3.6. Граф переходов конечного автомата до его минимизации

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}.5'.A. {DE}). 5'(A,0)-{BC}. 5' (A. 1)={BC}. 5'(BC,1)={DE}. 5'(DE,0)={BC), 5'CDE,1)={DE}.

Граф переходов минимального КА приведен на рис. 3.7.

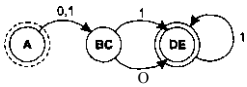


Рис. 3.7. Граф переходов конечного автомата после его минимизации

Минимизация конечных автоматов позволяет при построении распознавателей получить автомат с минимально возможным числом состояний и, тем самым, в дальнейшим 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:

P Q - конкатенация PeV* и QeV*: P Q - {pq VpeP, VqeQ};

P* - итерация PeV*: P* - {p* VpeP}.

Тогда для алфавита V регулярные множества определяются рекурсивно:

1. 0 — регулярное множество;
2. {X} — регулярное множество;
3. {a} — регулярное множество VaeV;
4. если P и Q — произвольные регулярные множества, то множества PuQ, 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.

Регулярные выражения. Свойства регулярных выражений

Регулярные множества можно обозначать с помощью регулярных выражений.

Эти обозначения вводятся следующим образом:

1. О — регулярное выражение, обозначающее 0;
2. К — регулярное выражение, обозначающее {X};
3. а — регулярное выражение, обозначающее {a} VaeV;
4. если р и q — регулярные выражения, обозначающие регулярные множества P и Q, то р + q, pq, p* — регулярные выражения, обозначающие регулярные множества PuQ, PQ и P* соответственно,

ПРИМЕЧАНИЕ

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

Два регулярных выражения а и р равны а - р, 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 a, P и *y* — регулярные выражения, то свойства регулярных выражений можно записать в виде следующих формул:

1. X + aa = X + a'a = a* (Я + a^+ - a")
2. a + p = p + a3
3. a + (p + *y)* - (a + P) + *y*
4. a(P + *y)* = ap + *ay*
5. (P + *y)a* - pa + *ya*
6. a(Py) - (aP)y
7. a + a - a
8. a + a" - a*
9. *X* + *a* - *a* + *X*- *a*
10. 0*-Я.
11. Oa-aO-O
12. 0 + a - a + 0 - a
13. *Xa* - *aX* - a
14. (a')' - a'.

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 aP - Pa, то есть операция конкатенации не обладает свойством коммутативности. Это и не удивительно, поскольку для этой операции важен порядок следования символов.

Уравнения с регулярными коэффициентами

На основе регулярных выражений можно построить уравнения с регулярными коэффициентами [15, 22]. Простейшие уравнения с регулярными коэффициентами будут выглядеть следующим образом:

X - aX + p,

X= Xa + p,

где

a,PeV" — регулярные выражения над алфавитом V, а переменная XgV.

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 (правосторонняя

и левосторонняя записи) 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. Обе записи равноправны.

Решением первого уравнения является множество, обозначенное регулярным выражением a'p. Проверим это решение, подставив его в уравнение вместо переменной X:

$$aX + (3 - a(a'P) + g -^{6} (aa^*)P + P -^{13} (aa)P + Xp -^{5} (aa + X)p -^{1} a'p - X.$$

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

Решением второго уравнения является множество, обозначенное регулярным выражением Pa*.

$$Xa + p - (pa')a + P -^{6} P(a a) + p -^{13} p(a'a) + pA. «^{4} p(a'a + к) -^{1} pa^* - X.$$

ПРИМЕЧАНИЕ

Указанные решения уравнений не всегда являются единственными. Однако доказано, что X - a'p и X - pa* — это наименьшие из возможных решений для данных двух уравнений. Эти решения называются наименьшей подвижной точкой [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):

$$^{x}i \sim «ю^{+} <^{*}iiXi + a_{12}X_2 + ... + dj_nX_n$$

$$X_2 - a_{20} + a_2iXj + a_{22}X_2 + ... + a^X,,$$

$$X, - a_{i0} + a_dX, + cs_{i2}X_2 + ... + a_{,,}X_n$$

$$X_{,,} - «_{,,}o + a_{n1}X, + a_{12}X_2 + ,. + a_{,,}Д_n$$

или (левосторонняя запись):

$$Xj - a_{10}'+ Xion + X_2a_{12} + ... + X_na_{1n}$$

$$X_2 — a_{20} + X|<x_2i + X_2a_{22} + ... + X_{,,}a_{2n}$$

$$X_f - a_{10} + X.Oj, + X_2a_{12} + ... + X_na_{,n}$$

$$X_{,,} - a_{n0} + Xjctn, + X_2a_{12} + ... + X_na_{,,n}.$$

В системе уравнений с регулярными коэффициентами все коэффициенты a_u являются регулярными выражениями над алфавитом V, а переменные не входят в алфавит V: Vi XjjSV. 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 X_n при подстановке которых в систему все уравнения превращаются в тождества. Иными словами, решением системы является некоторое отображение $f(X)$ множества переменных уравнения Д = {X,: n > i > 0} на множество языков над алфавитом 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, = a,X, + P„ где $a_2 = a_{,,}$ p, = P_{s0} + $P_{I1+1}X_{s+1}$ + ... + $P_{s1}X_{,,}$. Решить уравнение и получить X, = a.'Pr Затем для всех уравнений с переменными $X_{|+1}$. . . .X„ подставить в них найденное решение вместо X„.**

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

Шаг 4. **После всех подстановок уравнение для X„ будет иметь вид X„ = a„X„ + p, где $a_n = a_{nn}$. Причем p будет регулярным выражением над алфавитом V" (не содержит в своем составе переменных системы уравнений X,). Тогда можно найти окончательное решение для X„: X„ = a„*p. 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, = a,X, + p>, где a] = $a_{,,}$ P, = $P_{,0}$ + $P_{,,+1}X_{1+1}$ + + ... + $P_{s1}X_{,,}$ и подставляем в него окончательные решения для переменных $X_{,+1,}$..., $X_{,,}$. Получаем окончательное решение для X„. Перейти к шагу 5.**

Для левосторонней записи системы уравнений алгоритм решения будет аналогичным, С разницей только в порядке записи коэффициентов (справа от переменных).

Система уравнений с регулярными коэффициентами всегда имеет решение, но это решение не всегда единственное. Для рассмотренного алгоритма решения системы уравнений с регулярными коэффициентами доказано, что он всегда находит решение $f(X)$, которое является наименьшей неподвижной точкой системы уравнений. То есть если существует любое другое решение $g(X)$, то всегда $f(X) c g(X)$ [4, т. 1, 15].

В качестве примера рассмотрим систему уравнений с регулярными коэффициентами над алфавитом V ={"-", "+", "." 0, 1} (для ясности записи символы -,+ и . взяты в кавычки, чтобы не путать их со знаками операций):

```
XI = ("-" + "+" + X)
X2 = XI" "(O + 1) + X3"." + X2(0 + 1)
X3 = X1(0 + 1) + X3(0 + 1)
X4 = X2 + X3
```

Решим эту систему уравнений.

Шаг 1.

```
i •= 1
```

Шаг 2.

Имеем i - 1 < 4.

Берем уравнение для i - 1. Имеем XI = ("-" + "+" + X). Это уже и есть решение для XI.

Подставляем его в другие уравнения. Получаем:

X2 - ("-" + "+" + >.)"."(0 + 1) + X3"." + X2(0 + 1)
xз = с-» + ••+•• + а.)(о + 1) + хз(0 + 1)
X4 = X2 + X3

Шаг 3.

i i + 1 - 2

Возвращаемся к шагу 2.

Шаг 2.

Имеем i = 2 < 4.

Берем уравнение для i - 2. Имеем X2 = ("-" + "+" + *X)"."(0* + 1) + X3"." + X2(0 + 1). Преобразуем уравнение к виду: X2 - X2(0 + 1) + (("-" + "+" + Я.)". "(О + 1) + X3"."). Тогда а2 = (0 + 1), (32 = ("-" + "+" + X)". "(О + 1) + X3"."*.* Решением для X2 будет: X2 = P2a2* = (("-" + "+" + X)"."(0 + 1) + X3"."*)*(0 + 1)* = ("-" + "+" + X)"."(0 + + 1)(0 + 1)* + X3"."(0 + 1)*.
Подставим его в другие уравнения. Получаем:

X3 = ("-» + "+" + JL)(0 + 1) + X3(0 + 1)
X4 = ("-» + ••+" + ХГ.'ЧО + 1)(0 + 1)* + X3". "(О + 1)* + X3

Шаг 3.

i i +1 = 3

Возвращаемся к шагу 2.

Шаг 2.

Имеем i = 3 < 4.

Берем уравнение для i - 3. Имеем X3 - ("-" + "+" + Я.)(О + 1) + X3(0 + 1). Преобразуем уравнение к виду X3 = X3(0 + 1) + ("-" + "+" + X)(0 + 1). Тогда а3 - (0 + 1), Р3 - *("-" + "+" + X)(0* + 1). Решением для X3 будет: X3 - р3а3* = ("-" + "+" + *X)* (О + 1X0 + 1)*

Подставим его в другие уравнения. Получаем:

X4 = ("-" + "+" + *X)"."(0* + 1)(0 + 1)* + ("-" + "+" + X)(0 + 1)CO + 1)*"."a* + + ("-" + "+" + X)(0 + 1X0 + 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 X4 теперь имеет вид X4 - ("-" + "+" + A,)"."(O + 1>CO + 1)* + ("-" + "+•• + X)(0 + 1X0 + 1)*"."(0 + 1)*+ ("-" + "+" + A.K0 + 1)(0 + 1)*. Оно не нуждается в преобразованиях и содержит окончательное решение для X4. Переходим к шагу 5.

Шаг 5.

i := i - 1 = 3 > 0

Переходим к шагу 6.

Шаг 6.

Уравнение для X3 имеет вид X3 = ("-" + "+" + ***X)(0*** + 1)(0 + 1)*. Оно уже содержит окончательное решение для X3. Переходим к шагу 5.

Шаг 5.

i := i - 1 - 2 > 0

Переходим к шагу 6.

Шаг 6.

Уравнение для X2 имеет вид X2 = ("-" + "+" + X)"."аа* + X3"."а *. Подставим в него окончательное решение для X_3. Получим окончательное решение для X2: X2 = ("-" + -+" + A.)"."(0 + 1)(0 + 1)* + ("-" + "+" + X)(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 XI имеет вид XI = ("-" + "+" + X). Оно уже содержит окончательное решение для XI. Переходим к шагу 5.

Шаг 5.

i i - 1 - 0 - 0

Алгоритм завершен.

В итоге получили решение:

XI = ("." + "+» + x)
X2 = ("-" + "+" + X)"."(0 + 1)(0 + 1)* + ("-" + "+" + X)(0 + 1) (0 + 1)*"."(0 + 1)*
X3 = ("-" + ••+" + Л)(0 + 1)(0 + 1)*
X4 = ("-" + "+" + X)"."(0 + 1>C0 + 1)* + ("-" + "+" + X)(0 + 1)(0 + 1)*"."(0 + 1)*
+ (»_» + '•+•• + X)(0 + 1)(0 + 1)*

Выполнив несложные преобразования, это же решение можно представить в более простом виде:

XI = (••_» + ••+•• +
12 = ("-" + "+" + 0 + 1) + (0 + 1)(0 + 1)*"."")(0 + 1)*
X3 = ("." + ••+" + X)(0 + 1)+
X4 = ("-" + "+" + X) C" -" C 0 + 1) + (0 + 1)(0 + 1)*"." + (0 + 1) HO + 1)*'

Можно заметить, что регулярное выражение для X4 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, то 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,(V) и L_2(V). Необходимо проверить, являются ли эти два языка эквивалентными.**
- ***Проблема принадлежности цепочки языку.*** **Дан регулярный язык L(V) и цепочка символов aeV*. 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 a * X, такую что aeL(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 грам-

матик 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.

Лемма о разрастании для регулярных языков

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].

Лемма о разрастании для регулярных языков формулируется следующим образом: если дан регулярный язык и достаточно длинная цепочка символов, принадлежащая этому языку, то в этой цепочке можно найти непустую подцепочку, которую можно повторить сколь угодно много раз, и все полученные таким способом новые цепочки будут принадлежать тому же регулярному языку[2].

Формально эту лемму можно записать так: если дан язык L, то 3 константа р > О, такая, что если а е L и |а| > р, то цепочку а можно записать в виде а = 5(38, где О < |р| < р, и тогда а' = б^е, а'е L VI > 0.

Используя лемму о разрастании регулярных языков, докажем, что язык L = {а"Ь" | п > 0} не является регулярным.

Предположим, что этот язык регулярный, тогда для него должна выполняться лемма о разрастании. Возьмем некоторую цепочку этого языка а = а"Ь" и запишем ее в виде а = 508. Если (Зба$^+$ или (ЗеЬ$^+$, то тогда для i = 0 цепочка бр°£ = 5е не принадлежит языку L, что противоречит условиям леммы; если же Реа$^+$Ь$^+$, тогда для i = 2 цепочка 5р^2£ = 5рре не принадлежит языку L. Таким образом, язык L не может быть регулярным языком.

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 регу-

[1] Возможны и другие способы представления регулярных множеств, а для них разрешимость указанных проблем будет уже не очевидна.

[2] Если найденную подцепочку повторять несколько раз, то исходная цепочка как бы «разрастается» — отсюда и название «лемма о разрастании языков».

лярные 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 справедливо для других способов, которыми можно определить регулярные языки [4, т. 1,9, 15, 19, 28].

Из всех возможных преобразований практический интерес представляют два преобразования:

- построение регулярного выражения, задающего язык, на основе регулярной грамматики;
- 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, то 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^ X_2. . . . $X_п$}. Тогда все правила грамматики будут иметь вид: Xj -> XjY или Xj -> у Xj.XjeVN, yeVT*; целевому символу грамматики S будет соответствовать некоторое обозначение $X_к$.
2. Построим систему уравнений с регулярными коэффициентами на основе переменных Xj, X_2,..., $X_п$:

   $X_1 = a_{0i} + X_t a_{i1} + X_2 a_{21} + ... + X_п a_{п1}$

   $X_2 = a_{0г} + X_t a_{i2} + X_2 a_{22} + ... + X_п a_{п2}$

   $X_п = a_{0п} + X_1 a_{1п} + X_2 a_{2п} + ... + X_п a_{пп}$

   Коэффициенты $a_{0}i$, a_{02},..., $a_{0п}$ выбираются следующим образом:

   $a_{0i} = (y_1 + y_2 + ... + y_т)$.

   Если во множестве правил P грамматики G существуют правила X, $y_t \mid y_2 1 ... \mid y_т$;

   $a_{oi} = 0$.

   если правил такого вида не существует.

   Коэффициенты a^, a_{j2},..., $a_{jп}$ для некоторого| выбираются следующим образом:

   $a_{ji} = (Y1 + y2 + ... + Ут)$.

   если во множестве правил P грамматики G существуют правила X, -> X^ | Xyy_2 I - IX_{jYm};

   «л - 0.

   если правил такого вида не существует.

3. Находим решение построенной системы уравнений.

Доказано, что решение для $X_к$, 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 $X_к$.

Пример построения регулярного выражения для языка, заданного леволинейной грамматикой

Например, рассмотрим леволинейную грамматику, определяющую язык двоичных чисел с плавающей точкой G({".". "-", "О". "1"}, {<знак>, <дробное>, <целое>, <число>}, P, <число>):

P:
<знак> -> - | + | A.
<дробное> <знак>.O | <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** = {Xj, X_2. X_3, X_4}.

Построим систему уравнений на основе правил грамматики 6:

XI = ("-" + "+" + X)
X2 = **XI".**"(0+1) + X3"." + X2C0+1)
X3 = X1C0+1) + X3C0+1)
X4 = X2 + X3

Эта система уравнений уже была решена выше. В данном случае нас интересует только решение для X_4, **которое соответствует целевому символу грамматики 6** <число>.

Решение для X_4 **может быть записано в виде:**

X_4 = ("-" + "+" + X)("."(0+1) + (0+1)(0+1)*"." + (0+1Ж0+1)*,

то есть

<число> = ("-" + "+" + X)(\"(0+1) + (0+1)(0+1)*"." + (0+1))(0+1)*.

Это и есть регулярное выражение, определяющее язык двоичных чисел с плавающей точкой, заданный грамматикой 6.

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 автомат на основе известной грамматики, мы получаем распознаватель для лексических конструкций данного языка. Таким образом удается решить задачу разбора для лексических конструкций языка, заданных произвольной регуляр-

ной 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,VAq$_0$,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(O,У,8^$_0$,E).

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(O,У,8^$_0$,E) 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, для множества состояний автомата M можно записать: Q = VNu{H}.

Шаг 2. Входным алфавитом автомата M является множество терминальных символов грамматики G: V = VT.

Шаг 3. Просматриваем все множество правил исходной грамматики.

Если встречается правило вида A -» teP, где AeVN, teVT, то в функцию переходов 6(H,t) автомата M добавляем состояние A: Ae6(H,t).

Если встречается правило вида A -» BteP, где A,BeVN, teVT, то в функцию переходов 6(B,t) автомата M добавляем состояние A: Ae6(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","C,"*"."{","}"},{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*) I 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» в этой главе).

Получим леволинейную автоматную грамматику следующего вида: G'({"a"."(", "*".")"."{"."}"}. {S.Sl.C.Cl.KJ.P*.S):

P':
S -> S1) I K}
S1 c*
C C1* | Ca | C{ | C} | C(| C* | C)
C1 -> (
K { | Ka | K(| K* | K) | K{

Для удобства переобозначим нетерминальные символы C1 и S1 символами D и E. Получим 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,8,q_0,F), эквивалентный указанной грамматике.

Шаг 1.

Строим множество состояний автомата. Получаем: Q = VNu{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 8(E.")") = {S}: 8(K."}") = {S}.

Для правила E -» C* имеем 8(C,"*") = {E}.

Для правил C D* | Ca | C{ | C} | C(| C* | C) имеем 8(0."*") = {C}: 8(C."a") = {C}: 8(C."{") = {C}: 8(C.")") = {C}: 8(C."Г) = {C}: 8(C."*") = {E.C}; 8(C.")") = {C}.
Для правила D -» (имеем 8(H."(") = {D}.

Для правил К { | Ка | К(| К* | К) | К{ имеем 5(Н."{") - {К}; S(K,"a") * {К}: 8(К."(") - {К}; 5(К."*") - {К}: 6(<,")") - {<}; 8 «."{"> - {К}.

Шаг 4.

Начальным состоянием автомата является состояние qO - И.

Шаг 5.

Множеством конечных состояний автомата является множество F - {S}.

Выполнение алгоритма закончено.

В итоге получаем автоматМ({S.E.C.D.K.H}, {"а"."Г."Г."{"."}"} .8.Н.{S}) с функцией переходов:

5(Н."{") = {К}
5(Н,"(") - {D}
5(К."а") - {К}
5(К."(") - {К}
5(К."*") = {К}
5(К.")") = {К}
5(К."{") = {К}
5(К."}"> - {S}
5(D."*") - {С}
8(С."а") - {С}
5(С,"{") - {С}
5(С."}") - {С}
5(С."(") - {С}
8(С."*") - {E.C}
5(С.")") - {С}
5(Е.")") - {S}

Граф переходов этого автомата изображен на рис. 3.8.

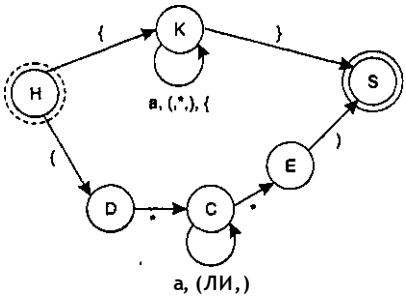


Рис. 3.8. Недетерминированный КА для языка комментариев в Borland Pascal

Это недетерминированный конечный автомат, поскольку существует состояние, в котором множество, получаемое с помощью функции переходов по одному и тому же символу, имеет более одного следующего состояния. Это состояние С и функция 8(С."*") = {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"."(".".*",")","{"."}"} ,8' .H.{S}) с функцией переходов:

```
6'(H."{") = {K}
6ЧН.'Ч") = {D}
8'(К,"а") = {К}
8ЧК.Ч") = {К}
8'(К."*") = {К}
6ЧК.")") = {К}
5ЧК."{") = {К}
8ЧК."}") = {S}
8'(D."*") = {C}
6"(С."а") = {C}
8ЧС."{") = {C}
8ЧС."}") = {C}
8ЧС.Ч") = {C}
8ЧС.")") = {C}
8'(С."*") = {E}
8' (Е."а") = {C}
8ЧЕ."{") = {C}
8ЧЕ."}") = {C}
8ЧЕ."(") = {C}
8'(Е."*") = {E}
8ЧЕ.")") = {S}
```

Граф переходов этого автомата изображен на рис. 3.9.

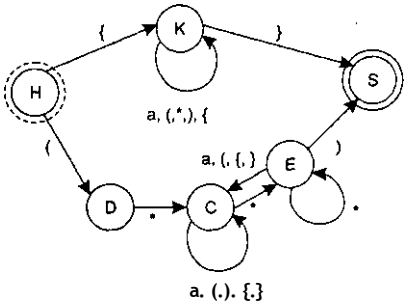


Рис. 3.9. Детерминированный КА для языка комментариев в Borland Pascal

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 а может означать любой алфавитно-цифровой символ, кроме символов (, *), {, }.

Построение леволинейной грамматики на основе конечного автомата

Задача формулируется так: имеется конечный автомат M(Q,V,8,q_0,F), необходимо построить эквивалентную ему леволинейную грамматику G(VT,VN,P,S).

Построение выполняется по следующему алгоритму:

Шаг 1. **Множество терминальных символов грамматики G строится из алфавита входных символов автомата M: VT = V.**

Шаг 2. **Множество нетерминальных символов грамматики G строится на основании множества состояний автомата M таким образом, чтобы каждому состоянию автомата, за исключением начального состояния, соответствовал один нетерминальный символ грамматики: VN = Q\{qo}-**

Шаг 3. **Просматриваем функцию переходов автомата M для всех возможных состояний из множества Q для всех возможных входных символов из множества V. Если имеем 8(A,t) = 0, то ничего не выполняем.**

Если имеем 8(A,t) = {B_1(B_2....B_n}, n > 0, где AeQ, Vn > i > 0: B.eQ, te V, тогда для всех состояний Bj выполняем следующее:

- **добавляем правило Bj -» t во множество P правил грамматики G, если A = q_0;**
- **добавляем правило B, -» At во множество P правил грамматики G, если A * q_0.**

Шаг 4. **Если множество конечных состояний F автомата M содержит только одно состояние F - {F_0}, то целевым символом S грамматики G становится символ множества VN, соответствующий этому состоянию: S = F_0; иначе, если множество конечных состояний F автомата M содержит более одного состояния F = {Fj^.-mF,,}, n > 1, тогда во множество нетерминальных символов VN грамматики G добавляется новый нетерминальный символ S: VN = VNu{S}, а во множество правил P грамматики G добавляются правила: S -» Fj | $F_2$1...| F_n.**

На этом построение грамматики заканчивается.

Примеры построения лексических анализаторов

Теперь можно рассмотреть практическую реализацию лексических анализаторов. В принципе, компилятор может иметь в своем составе не один, а несколько лексических анализаторов, каждый из которых предназначен для выборки и проверки определенного типа лексем.

Таким образом, обобщенный алгоритм работы простейшего лексического анализатора в компиляторе можно описать следующим образом:

- **из входного потока выбирается очередной символ, в зависимости от которого запускается тот или иной сканер (символ может быть также проигнорирован либо признан ошибочным);**
- **запущенный сканер просматривает входной поток символов программы на исходном языке, выделяя символы, входящие в требуемую лексему, до обнаружения очередного символа, который может ограничивать лексему, либо до обнаружения ошибочного символа;**

- при успешном распознавании информация о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, алгоритм возвращается к первому этапу и продолжает рассматривать входной поток символов с того места, на котором остановился сканер;
- при неуспешном распознавании выдается сообщение об ошибке, а дальнейшие действия зависят от реализации сканера — либо его выполнение прекращается, либо делается попытка распознать следующую лексему (идет возврат к первому этапу алгоритма).

В целом, техника построения сканеров основывается на моделировании работы детерминированных и недетерминированных КА с дополнением функций распознавателя вызовами функций обработки ошибок, а также заполнения таблиц лексем и таблиц идентификаторов. Такая техника не требует сложной математической обработки и принципиально важных преобразований входных грамматик. Для разработчиков сканера важно только решить, где кончаются функции сканера и начинаются функции синтаксического разбора. После этого процесс построения сканера легко поддается автоматизации.

Лексический анализатор целочисленных констант языка С

Рассмотрим пример анализа лексем, представляющих собой целочисленные константы в формате языка С. В соответствии с требованиями языка, такие константы могут быть десятеричными, восьмеричными или шестнадцатеричными. Восьмеричной константой считается число, начинающееся с 0 и содержащее цифры от 0 до 7; шестнадцатеричная константа должна начинаться с последовательности символов Ox и может содержать цифры и буквы от а до f. Остальные числа считаются десятеричными (правила их записи напоминать, наверное, не стоит). Константа может начинаться также с одного из знаков, + или -, а в конце цифры, обозначающей значение константы, в языке С может следовать буква или две буквы, явно обозначающие ее тип (u, U — unsigned; h, H — short; 1, L — long).

При построении сканера будем учитывать, что константы входят в общий текст программы на языке С. Во избежание путаницы и для сокращения объема информации в примере будем считать, что все допустимые буквы являются строчными (читатели легко смогут самостоятельно расширить пример для прописных букв, которые язык С в константах не отличает от строчных).

Рассмотренные выше правила могут быть записаны в форме Бэкуса—Наура в грамматике целочисленных констант для языка С.

G({S, W. U. L. V. D. G. X Q, Z, N}. {0...9. x. a...f. u. 1, h. ±},P,S)
P.

S -> GX | ZX | DX | QX | UX | LX | VX | WX
и -> Lu | Vu | Ul | Uh
и -> Gu | Zu | Hu | Qu
L -» Gl | Z1 | Hl | Ql
V Gh J Zh | Hh | Qh
D l'| 2 | 3 | 4 | 5 | i6 I 7 | 8 | 9 |
N1 | N2 | N3 | N4 | N5 | 1N6 | N7 | N8 | N9 |
DO | D1 | D2 | D3 | D4 | 1D5 | D6 | 07 | D8 | D9 |

Z8 | *19* I Q8 I Q9
G —> XO 1 XI I X2 | X3 | X4 | X5
Xa I Xb I Xc | Xd | Xe 1 Xf 1
GO | G1 | G2 | G3 | G4 | G5 | G6
Ga | Gb | Gc | Gd j Ge 1 Gf
X -> Zx
Q -> Z0 1 zi 1 Z2 1 Z3 1 Z4 1 Z5
QO | Q1 | Q2 I Q3 I Q4 I Q5 | Q6
z —> o I N)
N -> + | -

Эта грамматика является леволинейной автоматной регулярной грамматикой. По ней можно построить КА, который будет распознавать цепочки входного языка (см. раздел «Конечные автоматы»). Граф переходов этого автомата приведен на рис. 3.10. Видно, что построенный автомат является детерминированным КА, поэтому в дальнейших преобразованиях он не нуждается. Начальным состоянием автомата является состояние H.

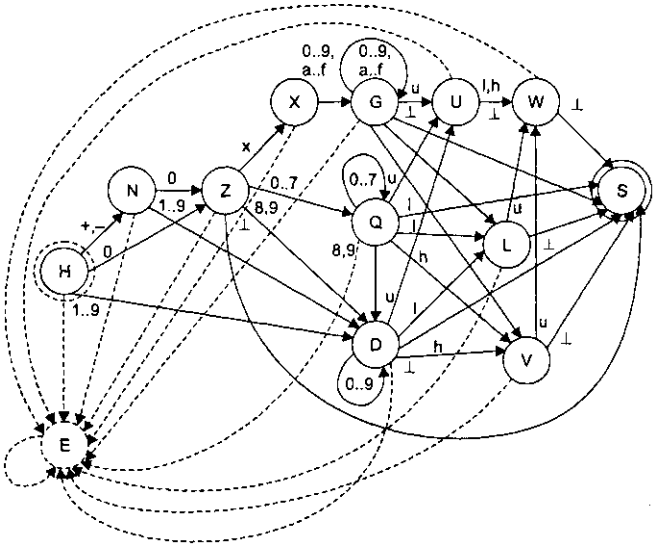


Рис. 3.10. Граф конечного автомата для распознавания целочисленных констант языка С

Приведем автомат к полностью определенному виду — дополним его новым состоянием E, на которое замкнем все дуги неопределенных переходов. На графе автомата дуги, идущие в это состояние, не нагружены символами — они обозначают функцию перехода по любому символу, кроме тех символов, которыми уже помечены другие дуги, выходящие из той же вершины графа (такое соглашение

принято, чтобы не загромождать обозначениями граф автомата). На рис. 3.10 это состояние и связанные с ним дуги переходов изображены пунктирными линиями. При построении КА знаком _L были обозначены любые символы, которыми может завершаться целочисленная константа языка С. Однако в реальной программе этот символ не встречается, поскольку границы лексем в ней явно не указаны. При моделировании КА надо решить проблему определения границ лексем. Следует принять во внимание, какими реальными символами входного языка может завершаться целочисленная константа языка С.

В языке С в качестве границы константы могут выступать:

- **знаки пробелов, табуляции, перевода строки;**
- **разделители (,), [,], {,}, ., :, ;;**
- **знаки операций +, -, *, /, &, |, ?, <, >, =, %.**

Теперь можно написать программу, моделирующую работу указанного автомата. Ниже приводится текст функции на языке Pascal, реализующей данный распознаватель. Результатом функции является текущее положение считывающей головки анализатора в потоке входной информации после завершения разбора лексемы при успешном распознавании либо 0 при неуспешном распознавании (если лексема содержит ошибку).

В программе переменная iState отображает текущее состояние автомата, переменная i является счетчиком символов входной строки. В текст программы вписаны комментарии в те места функции, где возможно выполнить запись найденной лексемы в таблицу лексем и таблицу символов, а также выдать сообщение об обнаруженной ошибке.

```
function RunAuto (const sInput: string; iPos: integer): integer;
(* sInput - входная строка исходного текста;
   iPos - текущее положение считывающей головки во входной
   строке исходного текста *)
type
   TAutoState = (AUT0_H. AUT0_S. AUTOJJ. AUTOJJ. AUT0_L. AUT0_V. AUT0_D, AUT0_G,
AUT0_X. AUT0_Q. AUT0_Z, AUT0_N. AUT0_E):
const
   AutoStop = ['     #10. #13. #7. '('. ')'. '['. ']'. '{'• '}'•        ':'. ':'.
'+•.     . •*'. '/'. '&'. '|'. '?'.     . '<'. '>'.     . '='. '%'];
var
   iState ; TAutoState;
   i.iL : integer;
   sOut : string;
begin
   1 := iPos:
   iL := Length(sInput);
   sOut := '';
   iState := AUT0_H;
   repeat
      case iState of
         AUT0_H:
            case sInput[i] of
```

```
      '+','-': iState := AUT0_N;
      'O':     iState := AUT0_Z;
      '1'..'9': iState := AUT0_D;
      else     iState := AUT0_E;
   end;
AUT0_N:
   case sInput[i] of
      '0':     iState := AUT0_Z;
      '1'..'9': iState := AUTOJ;
      else     iState := AUTO_E;
   end;
AUTO_Z;
   case sInput[i] of
      'x':     iState := AUTOJ;
      '0'..'7': iState := AUTO_Q;
      '8','9': iState := AUTOJ;
      else
      if sInput[i] in AutoStop then
         iState := AUTOJ
         else iState := AUTOJ;
   end;
AUTOJ:
   case sInput[i] of

      'a'..'f': iState := AUTOJ:
      else     iState := AUTOJ;
   end:
AUTOJ:
   case sInput[i] of
      '0'..'7':: iState := AUTO Q
      '8','9':  iState := AUTO D
      'u':      iState := AUTO U
      'V;       iState := AUTO L
      'h":      iState := AUTO V
      else
      if sInput[i] in AutoStop then
         iState := AUTOJ
         else iState := AUTOJ:
   end;
AUTOJ:
   case sInput[i] of
      '0'...'9': iState := AUTOJ;
      V:       iState := AUTOJ;
      'I':      iState := AUTOJ;
      'h':      iState := AUTOJ;
      else
      if sInputCi] in AutoStop then
         iState := AUTOJ
         else iState := AUTOJ;
   end:
```

```
        AUT0_G:
           case sInput[i] of
              'O'..'9'.
              'a'..'f': iState := AUT0_G:
              'u':      iState := AUT0_U:
              •I':      iState := AUTOJ;
              'h' :     iState ;= AUTOJ/;
              else
              if sInput[i] in AutoStop then
                 iState := AUT0_S
                 else iState := AUT0_E;
           end;
        AUTOJJ:
           case sInput[i] of
              '1'.'h':  iState := AUTOJJ:
              else
              if sInput[i] in AutoStop then
                 iState := AUT0_S
                 else iState := AUTOJ:
           end:
        AUTOJ:
           case sInput[i] of
              'u':      iState := AUTOJ:
              else
              if sInput[i] in AutoStop then
                 iState := AUTOJ
                 else iState := AUTOJ:
           end:
        AUTOJ:
           case sInput[i] of
              'u':      iState := AUTOJ:
              else
              if sInput[i] in AutoStop then
                 iState := AUTOJ
                 else iState := AUTOJ;
           end;
        AUTOJ:
           if sInput[i] in AutoStop then
              iState := AUTOJ
              else iState := AUTOJ;
     end {case}:
     if not (iState in [AUTOJ.AUTOJ]) then
     begin
        sOut := sOut + sInput[i];
        i := i + 1;
     end:
until ((iState = AUTOJ) or (iState = AUTOJ)
        or (I > iL));
if (iState = AUTOJ) or (i > iL) then
```

```
    begin
    { Сюда надо вставить вызов функций записи лексемы sOut }
       RunAuto := i;
    end
    else
    begin
    { Сюда надо вставить вызов функции формирования сообщения об ошибке }
       RunAuto := 0;
    end;
end; { RunAuto }
```

Конечно, рассмотренная программа — это всего лишь пример для моделирования такого рода автомата. Она построена так, чтобы можно было четко отследить соответствие между этой программой и построенным на рис. 3.10 автоматом. Конкретный распознаватель будет существенно зависеть от того, как организовано его взаимодействие с остальными частями компилятора. Это влияет на возвращаемое функцией значение, а также на то, как она должна действовать при обнаружении ошибки и при завершении распознавания лексемы.

В данном примере предусмотрено, что основная часть лексического анализатора сначала считывает в память весь текст исходной программы, а потом начинает его последовательный просмотр. В зависимости от текущего символа вызывается тот или иной сканер. По результатам работы каждого сканера разбор либо продолжается с позиции, которую вернула функция сканера, либо прерывается, если функция возвращает 0.

Лексический анализатор для поиска строк в исходном тексте программы, написанной на языке Pascal

Действия, решаемые этим лексическим анализатором, возникли из реальной практической задачи. Эта задача является удачным примером практического применения лексических анализаторов, причем в данном случае лексический анализатор не является частью компилятора, а выступает как самостоятельная программная единица. Такие примеры довольно часто встречаются в реальной жизни, поскольку, как уже было отмечено, лексический анализ находит применение не только в компиляции.

***Задача*: имеется исходный текст программы, написанной на языке Pascal, необходимо найти в этом тексте все строковые константы и записать их в отдельный файл.**

ПРИМЕЧАНИЕ

На практике такая задача может возникнуть, например, для перевода интерфейса и текстовых сообщений программы с одного естественного языка на другой (скажем, с русского на английский). В этом случае выбранные из исходного текста строки надо отдать переводчику, которому весь исходный текст не интересен (да и разбираться в нем он не сможет).

Надо отметить, что в данном случае задача лексического анализатора значительно уже, чем задача лексического анализатора компилятора Pascal. Анализатор

должен найти только все строковые константы, не обращая никакого внимания на все остальные лексемы. Также нет необходимости заполнять таблицу лексем и таблицу идентификаторов — сопоставление лексем и дальнейшая обработка текста не входят в постановку задачи. И, конечно же, нет необходимости проверять синтаксическую правильность исходной программы — это не задача лексического анализатора.

Именно по причине наглядности и простоты решаемой задачи данный лексический анализатор выбран в качестве удачного примера.

Построение лексического анализатора начнем с построения грамматики исходного языка. Исходным языком является язык строковых констант Pascal. Для построения его грамматики следует разобраться, что представляет собой строковая константа в языке Pascal.

Строковая константа в языке Pascal может состоять из одной или более частей, соединенных между собой знаком +. Каждая часть строковой константы начинается с символа ' (одинарная кавычка) и заканчивается символом ' (одинарная кавычка). В состав строковой константы могут входить любые алфавитно-цифровые символы, но если в нее надо включить одинарную кавычку, то она должна быть повторена дважды: ''. Также в строку могут быть включены коды символов. В языке Pascal они записываются восьмеричными цифрами, следующими за знаком #. Коды символов включаются в строку либо сразу после завершающей одинарной кавычки, либо с помощью знака +. Для лексического анализатора строковая константа считается законченной, если после очередной ее части он встретил любой алфавитно-цифровой символ, кроме знаков + и #, или конец файла. Все символы, не входящие в строковые константы, анализатором должны игнорироваться.

Для простоты и удобства работы далее будем символом b обозначать все незначащие символы языка (пробелы, знаки табуляции и перевода строки), символом d — все допустимые восьмеричные цифры (от 0 до 7), а символом a — все остальные алфавитно-цифровые символы, кроме символов, обозначенных через b и d, а также символов ', + и #. Символом J_ будем обозначать конец файла.
Тогда грамматику языка строковых констант Pascal можно записать следующим образом: Gl({a.b.d.' . + .#}. {S1.S2.S3.S4.D1.D2.S} .P.S), а ее правила P:

```
51 -> ' | Sla | Sib | Sid | S1+ | Sl# | S2' | S4' | Dl'
52 -> SI'
53     S2b | S3b
54 -> S2+ | S3+ | S4b | Dl+ | D2+
Dl     # | S2# | S4# | Did | Dl#
D2     Dlb | D2b
S      a | b | d | + | S2a | S3a | S4a | Dla | D2a | S2± | S31 | DU | D21
```

Эта грамматика учитывает структуру всех возможных строковых констант языка Pascal, но она недостаточна для построения лексического анализатора, поскольку не учитывает комментарии, которые могут встречаться в тексте исход-

ной программы. Естественно, анализатор должен игнорировать комментарии и все сороковые константы, которые могут быть внутри комментариев.

Язык Pascal предусматривает два независимых варианта комментариев. Первый вариант: комментарий начинается последовательностью символов (* и заканчивается последовательностью символов *), второй вариант: комментарий начинается с символа { и заканчивается символом }. Комментарии могут содержать любые алфавитно-цифровые символы.

Тогда грамматику комментариев языка Pascal можно записать следующим образом: G2({a.b.d.'.+.#.(.*.).{.}}.{C1.C2.C3.C4.S}.P.S), а ее правила P:

C1 (| СИ
C2 -> C1* | C2a | C2b | C2d | C2' | C2+ | C2# | C2(| C2) | C2{ | C2} | C3a | C3Ь | C3d | C3' | C3+ | C3# | C3(| C3{ | C3}
C3 C2* | C3*
C4 { | C1{ | C4a | C4b | C4d | C4' | C4+ | C4# | C4(| C4* | C4) | C4{
S -> a | b | d | + | * |) | C1a | C1b | C1d | C1+ | C1) | C3) | C4}

Здесь a обозначает уже все алфавитно-цифровые символы, кроме символов, обозначенных ранее через b и d, а также символов ',+,#,(,*,),{ и }.

ПРИМЕЧАНИЕ

Эта грамматика построена таким образом, чтобы игнорировать исходный текст программы, не входящий в комментарии.

На основе двух построенных грамматик G! и G_2 можно реализовать лексический анализатор, решающий задачу в два этапа (он как бы будет состоять из двух лексических анализаторов). На первом этапе из исходной программы исключаются все комментарии анализатором, на основе грамматики G_2, и полученный в результате текст записывается в промежуточное хранилище данных (файл). На втором этапе исходный текст из промежуточного хранилища обрабатывается анализатором на основе грамматики G_t и в нем ищутся все строковые константы. Получим двухпроходный лексический анализатор.

Однако анализатор можно упростить, если построить грамматику языка, который будет включать в себя и строковые константы, и комментарии языка Pascal. Такую грамматику можно построить на основе G_t и G_2. Надо только учесть, что строки и комментарии могут следовать в тексте исходной программы друг за другом в произвольном порядке, а также то, что символы, ограничивающие комментарии — (,*,),{ и }, — могут входить в состав строки.

Грамматику строковых констант и комментариев языка Pascal можно записать следующим образом:

3({a,b,d,',+,#,(,*.).{.}},{C1.C2.C3.C4.S1,S2,S3,S4.D1.D2.S}.P.S), а ее правила P:
Л (| СИ | S2(| S3C | S4(| Dl(| D2(
12 -> CI* | C2a | C2b | C2d | C2' | C2+ | C2# | C2(| C2) | C2{ | C2} | C3a | C3Ь | :3d I C3' I C3+ I C3# I C3(| C3{ I C3}
: з - > C2* | c3*

```
C4 -» { | C1{ | S2{ | S3{ | S4{ | Dl{ | D2{ | C4a | C4b | C4d | C4' | C4+ | C4# |
C4( | C4* | C4) | C4{
51 -> ' | Sla | Sib | Sid | S1+ | Sl# | Sl( | SI* | SI) | Sl{ | SI} | S2' | S4' |
Dl' | CI`
52     SI'
53 -> S2b | S3b
54     S2+ | S3+ | S4b | Dl+ | D2+
Dl —> # | S2# | S4# | Did | Dl# | Cl#
D2 -> Dlb | D2b
S    a | b j d j + j * j ) j Cla j Clb | Cld j C1+ j CI) | C3) j C4} j S2a | S2)
| S3a | S3) | S4a | Dla | Dl) | D2a | D2) | S21 | S31 | D1l | D21
```

На основе этой грамматики можно строить однопроходный лексический анализатор, выполняющий поиск строковых констант в исходной программе на языке Pascal. Данный анализатор игнорирует комментарии, а также весь остальной текст исходной программы, не содержащий строковых констант.

Эта грамматика является автоматной грамматикой, поэтому на ее основе элементарно просто построить КА. Он будет иметь 12 состояний. Можно заметить, что построенный КА будет детерминированным, поэтому его сразу можно реализовать в программном коде лексического анализатора.

Лексический анализатор, моделирующий работу данного КА, начинает свое функционирование с начала просмотра исходного текста программы. При этом КА находится в начальном состоянии. Если символ на входе не относится ни к комментарию, ни к строковой константе, то анализатор просто пропускает его — при этом КА сразу переходит в конечное состояние, а потом анализатор должен вернуть его в начальное состояние. Если символ относится к строковой константе, то КА начинает переходить по всем состояниям, относящимся к строковой константе (все состояния Si и Di). Лексический анализатор должен при этом запоминать все символы константы (строку). При переходе КА в конечное состояние S анализатор записывает строку в результирующий файл, а КА возвращает в начальное состояние. Если встречается комментарий, то лексический анализатор должен пропустить все символы до конца комментария (КА при этом находится в состояниях Ci).

Таким образом, граф переходов КА лексического анализатора поиска строковых констант языка Pascal должен быть нагружен функциями:

- создания пустой строки для записи очередной строковой константы;
- добавления символа или подстроки в конец строковой константы;
- записи строковой константы в результирующий файл и очистка строки для следующей строковой константы.

Нагрузив граф переходов КА соответствующими функциями, получим конечный преобразователь, преобразующий исходную программу на языке Pascal в файл строковых констант этой программы.

Можно заметить, что построенный КА не является полностью определенным. Это вызвано тем, что существуют ошибки, которые могут быть обнаружены с по-

мощью данного лексического анализатора (хотя это и не является его основной задачей). Примером такой обнаруживаемой ошибки является незакрытый комментарий. Ошибки можно обозначить отдельным «ошибочным» нетерминальным символом грамматики E и соответствующим состоянием КА. Правило обнаружения ошибок можно записать в виде:

E -» } | CI} | S2d | S2* | S2} | S3d | S3' | S3# | S3* | S3} | S4d | S4+ | S4* | S4) | S4} | D1* | D1} | D2d | D2' | D2# | D2* j 02} | СИ | C21 | C3l | C41 | S11 | S41

Дополнив грамматику G этим правилом, можно построить полностью определенный детерминированный КА. Дуги переходов в состояние E этого автомата надо нагрузить функцией выдачи пользователю соответствующего сообщения об ошибке.

СОВЕТ

Рекомендуем читателям самостоятельно реализовать лексический анализатор на основе построенной грамматики и проверить его работу на любом исходном тексте программы для языка Pascal.

Данный пример достаточно наглядно иллюстрирует, как просто можно решать задачи лексического анализа, используя математический аппарат регулярных грамматик и КА. После построения КА задача разработчика заключается только в моделировании его работы (что элементарно просто) и реализации всех необходимых функций при переходе КА из одного состояния в другое[1].

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)[2].

[1] Рассмотренный выше пример при реализации без использования математического аппарата лексического анализа привел к многочисленным ошибкам разработчика, в результате исправления которых изначально несложный текст программы превратился в путаницу «заплаток», разобраться в которых не смог в конце концов и сам разработчик.

[2] 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.

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, то 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. Результатом 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.

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). Текст 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, 8, 20, 29, 46, 48].

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 детерминированный КА после преобразования из недетерминированного КА? Всегда ли это количество состояний можно сократить?**
7. **Являются ли регулярными множествами следующие множества:**
   - **О множество целых натуральных чисел;**
   - **О множество вещественных чисел;**
   - **О множество всех слов русского языка;**
   - **О множество всех возможных строковых констант в языке Pascal;**
   - **О множество иррациональных чисел;**
   - **О множество всех возможных вещественных констант языка С?**
8. **На основании свойств регулярных выражений докажите следующие тождества для произвольных регулярных выражений а, р, у и 5:**
   1. (a + P)(5 + y) = a5 + ay + p5 + py
   2. 5(a + p)y - 5ay + 5py
   3. p + Pa + pa' - pa
   4. p + paa* + paaa* - pa*
   5. 5ay0* + 5a*y + 8y = 5a*y
   6. (O* + aa* + aaa*)* - a*.
9. **Используя свойства регулярных множеств, проверьте, являются ли истинными следующие тождества для произвольных регулярных выражений а и р:**
   1. **(a + P)'- (a'PT**
   2. (aP)--a'P'
   3. (a + A,)* = a*
   4. a'P* + p*a* - a'P'a*
   5. a'a' = a*.

10. **Почему возможны две формы записи уравнений с регулярными коэффициентами? Как они соотносятся с двумя классами регулярных грамматик?**
11. **Можно ли для языка, заданного леволинейной грамматикой, построить праволинейную грамматику, задающую эквивалентный язык?**
12. **Всякая ли регулярная грамматика является однозначной? Если нет, то приведите пример неоднозначной регулярной грамматики.**
13. **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:**

G1({".".+.-.O.1}.{<число>.<часть>,<цифра>,<осн.>}.P1,<число>)
P1:
<число> -> +<осн.> | -<осн.> | <осн.>
<осн.> -> <часть>.<часть> | <часть>. | <часть>
<часть> -> <цифра> | <часть><цифра>
<цифра> —> 0 1 1
G2({".",+.-.0,1}.{<число>.<часть>,<осн.>},P2.<число>)
P2:
<число> +<осн.> | -<осн.> | <осн.>
<осн.> <часть>. | <часть> | <осн.>0 | <осн.>1
<часть> -> 0 | 1 | <часть>0 | <часть>1
G3C{".",+,-.O.1}.{<число>.<часть>,<осн.>},P3.<число>)
P3:
<число> -> +<осн.> | -<осн.> | <осн.>
<осн.> -> 0 | 1 | 0.<часть> | 1.<часть> | 0<осн.> | 1<осн.>
<часть> -> 0 | 1 | 0<часть> | 1<часть>
G4({".".+.-.0,1},{<знак>.<число>.<часть>,<осн.>},P4.<число>)
P4:
<число> -> <знак>0 | <знак>1 | <часть>. | <число>0 | <число>1
<часть> -> <знак>0 | <знак>1 | <часть>0 | <часть>1
<знак> -> X | + | -
G5C{".".+.-.0.1}.{<знак>.<число>.<часть>.<осн.>}.P5,<число>)
P5:
<число> -> <часть>. | <осн.>0 | <осн.>1 | <часть>0 | <часть>1
<осн.> <часть>. | <осн.>0 | <осн.>1
<часть> _> **о** | **1** | <знак>0 | <знак>1 | <часть>0 | <часть>1
<знак> -> + | -

2. **Докажите (любым способом), что грамматики G_3, G_4 и G_5 из задачи № 1 задают один и тот же язык.**
3. **Преобразуйте к автоматному виду грамматики G_3 и G_4 из задачи № 1.**
4. **Постройте КА на основании грамматик G_4 или G_5 из задачи № 1. Преобразуйте построенный автомат к детерминированному виду.**

5. Задан КА: M({H.I.S.E.F.G}.{b.d}. 8.H.{S}). S(H.b) = {I.S}. 5(H.d) = {E}. 8(1.b) = {I.S}. 8(1.d) = {I.S}. 8(E.b) = {E.F}, 8(E.d) = {E.F}. 8(F,b) = {E}. d(F.d) = {E}. 8(G.b) = {F.S}. d(G.d) = {S}; минимизируйте его и постройте эквивалентный ДКА.
6. Задан КА: M({S,R,Z}.{a.b},5.S.{Z}). 8(S.a) = {S.R}. 5(R.b) - {R}. 8(R.a) = {Z}. Преобразуйте его к детерминированному виду и минимизируйте полученный КА.
7. Постройте и решите систему уравнений с регулярными коэффициентами для грамматики G_4 из задачи № 1. Какой язык задает эта грамматика?
8. Докажите, что уравнение с регулярными коэффициентами X = aX + p имеет решение X = a*(P + y)> где *у* обозначает произвольное множество, в том случае, когда множество, обозначенное выражением а, содержит пустую цепочку. (Указание: поскольку множество, обозначенное выражением а, содержит пустую цепочку, то выражение а можно представить в виде а = 8 + X, при этом а* = 8*.)
9. Решите систему уравнений с регулярными коэффициентами над алфавитом V - {0,1}:
   1. $X, = 0X_2 + IX, + X$
   2. $X_2 = 0X_3 + 1X_2$
   3. $X_3 = OX, + 1X_3$
10. 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.
11. С помощью леммы о разрастании для регулярных языков докажите, что язык $L, = \{a^n,b^n \mid n > 2\}$ не является регулярным, а язык $L_2 = \{a^nb^m \mid n > 1, m > 2\}$ является регулярным.
12. 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++.
13. Постройте сканер, который выделял бы из текста входной программы на языке С все содержащиеся в ней строковые константы и записывал их в отдельный файл. Строковые константы в языке С состоят из строк, одиночных символов и кодов символов. Длинные строковые константы могут продолжаться на нескольких строках исходной программы подряд. Строки ограничены символами "" (двойные кавычки), одиночные символы — символами '' (одинарные кавычки), а коды символов начинаются со знака \ (обратная косая черта) и содержат цифры[1].

[1] Следует заметить, что эта задача отнюдь не так проста, как может показаться. Программа должна уметь не только находить в исходном тексте строки и объединять их между собой, но и правильно игнорировать комментарии (которых в С два типа). Поэтому лучше сначала корректно описать грамматику, а потом на ее основе построить автомат.

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 входит:**

- **поиск и выделение синтаксических конструкции в тексте исходной программы;**
- **установка типа и проверка правильности каждой синтаксической конструкции;**
- **представление синтаксических конструкций 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.

Выходом лексического анализатора является таблица лексем. Эта таблица образует вход синтаксического анализатора, который исследует только один компонент каждой лексемы — ее тип. Остальная информация о лексемах используется на более поздних фазах компиляции при семантическом анализе, подготовке к генерации и генерации кода результирующей 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 распозна-

ются 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 3 «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 1 «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. Далее в этой главе рассмотрены технические аспекты, связанные с реализацией синтаксических анализаторов.

Автоматы с магазинной памятью

Определение МП-автомата

Контекстно-свободными (КС) называются языки, определяемые грамматиками типа G(VT,VN,P,S), в которых правила P имеют вид: A p, где Ae VN и PeV*, V - VTuVN. Распознавателями КС-языков служат автоматы с магазинной памятью *(МП-автоматы).*

В общем виде МП-автомат можно определить следующим образом:

$R(QV,Z,5,q_0,z_0,F)$,

где

Q — множество состояний автомата;

V — алфавит входных символов автомата;

Z — специальный конечный алфавит магазинных символов автомата, VcZ;

8 — функция переходов автомата, которая отображает множество Qx(Vu{A,})xZ на конечное множество подмножеств P(QxZ');

q_0€Q — начальное состояние автомата;

z_0eZ — начальный символ магазина;

FcQ — множество коне<йших 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, показанной на рис. 4.1.

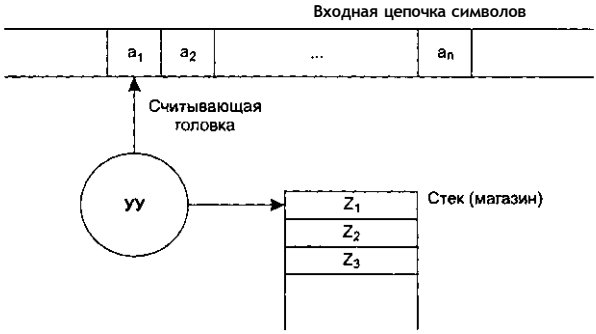


Рис. 4.1. Общая условная схема автомата с магазинной памятью (МП-автомата)

Конфигурация МП-автомата описывается в виде тройки (q,a,(o)eQxV*xZ*, которая определяет текущее состояние автомата q, цепочку еще не прочитанных символов а на входе автомата и содержимое магазина (стека) со. Вместо а в конфигурации можно указать пару ((3,п), где (3eV* — вся цепочка входных символов, а neNu{0}, п > 0 — положение считывающего указателя в цепочке.

Тогда один такт работы автомата можно описать в виде (q,aa,zco) 4- (q',a,yco), если (q',y)e5(q,a,z), где: q,q'eQ, aeVu{X,}, aeV*, z<=Zu{>*}, y,coeZ\ 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 A,-переходами (^-тактами). Аналогично, автомат не обязательно должен извлекать символ из стека — когда z = к, этого не происходит.

Начальная конфигурация МП-автомата, очевидно, определяется как (q_0,a,z_0), aeV*. Множество конечных конфигураций автомата — (c]Д,co), qeF, coeZ*.
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 определенную цепочку. Иначе цепочка символов не принимается.
Язык, определяемый МП-автоматом, — это множество всех цепочек символов, которые допускает данный автомат. Язык, определяемый МП-автоматом R, обозначается как L(R). Два МП-автомата называются эквивалентными, если они определяют один и тот же язык: L(Rj) = L(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 (qA,A), qeF. Если язык задан МП-автоматом R, который допускает цепочки с опустошением стека, это обозначается так: $L_{\cdot}$(R). Доказано, что для любого МП-автомата всегда можно построить эквивалентный ему МП-автомат, допускающий цепочки заданного языка с опустошением стека [4, т. 1]. То есть V МП-автомата R: 3 МП-автомат R', такой, что L(R) = L_X(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 8 для расширенного МП-автомата отображает множество Qx(Vu{X})xZ* на конечное множество подмножеств P(QxZ').
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 расширенным МП-автоматом) [4, т. 1]. 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, 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. Таким образом, 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(Q,V,Z,5, q_0,z_0,F) функция переходов 8 VqeQ, VaeV, VzeZ может иметь один из следующих трех видов:

1. S(q,a,z) содержит один элемент: 8(q,a,z) = {(q',y)}, y^{E}**Z*** и 8(q,^,z) = 0;
2. 8(q,a,z) = 0 и 8(q,^,z) содержит один элемент: 5(q,^,z) = {(q',y)}, *yeZ'*
3. 8(q,a,z) - 0 и **5(q**,A,**z)** = 0.

Класс ДМП-автоматов и соответствующих им языков значительно уже, чем весь класс МП-автоматов и КС-языков.

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.

Детерминированные МП-автоматы бпределяют очень важный класс среди всех КС-языков, называемый детерминированными КС-языками. Доказано, что все языки, принадлежащие к классу детерминированных КС-языков, могут быть построены с помощью однозначных КС-грамматик. Поскольку однозначность — это важное и обязательное требование в грамматике любого языка программирования, ДМП-автоматы представляют особый интерес для создания компиляторов.

ПРИМЕЧАНИЕ

Любой детерминированный КС-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—>AB, A-»aAb|ab, B—>aBbb|abb}, S) задает КС-язык, который не является детерминированным.

Все без исключения синтаксические конструкции языков программирования задаются с помощью однозначных КС-грамматик. Чаще всего синтаксические структуры этих языков относятся к классу детерминированных КС-языков и могут распознаваться с помощью ДМП-автоматов. Поэтому большинство распознавателей, которые будут рассмотрены далее, относятся к классу детерминированных КС-языков.

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, 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 исходной программы, сложнее, чем регулярные грамматики и КА, лежащие в основе лексического анализа.

Алгоритм создания МП-автомата на основе произвольной КС-грамматики прост [4, т. 1, 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, то 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 несложные 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, 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, то 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, 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 (квадратичная зависимость требуемых вычислительных ресурсов от длины входной цепочки неприемлема для компилятора).

Варианты синтаксических анализаторов

Построение синтаксического анализатора — это более творческий 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 тот, для 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, с целью 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, — в лучшем случае удается достичь квадратичной зависимости времени работы распознавателя от длины входной цепочки. 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, 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.
Далее будут рассмотрены формальные преобразования, применимые к КС-грамматикам, которые позволяют строить распознаватели на основе произвольной КС-грамматики. Затем изучаются основы двух названных алгоритмов — алгоритма с подбором альтернатив и алгоритма «сдвиг-свертка». Вначале они даны для распознавателей с возвратами — хотя такие распознаватели КС-языков имеют ограниченное применение, но их рассмотрение позволяет понять принципы, лежащие в основе этих алгоритмов. А потом следуют разделы, посвященные некоторым наиболее известным классам КС-грамматик. Для каждого класса дается его определение, основные ограничения, а также принципы и алгоритмы, лежащие в основе распознавателя для этого класса КС-языков.
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, 17, 52].

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 нескольких 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. В первую очередь, речь идет именно о восходящих и нисходящих распознавателях: в основе первых лежит алгоритм подбора альтернатив, в основе вторых — алгоритм «сдвиг-свертка».

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 или каким образом будут обработаны 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 часто легче построить правосторонний (восходящий) распознаватель. Класс языков, заданных восходящими распозна-

вателями, шире, чем класс языков, заданных нисходящими распознавателями (хотя следует сказать, что не все здесь столь однозначно).

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 (проанализировать) входную программу на входном языке, но и построить (синтезировать) результирующую программу. Более подробную информацию об этом можно получить в главе 5 «Генерация и оптимизация кода» или в работе [4, т, 1, 2]. Левосторонний анализ, основанный на нисходящем распознавателе, оказывается предпочтительным также при учете вопросов, связанных с обнаружением и локализацией ошибок в тексте исходной программы [4, т. 1, 2, 54].

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 5 «Генерация и оптимизация кода»). Поэтому бывает удобным использовать исходную грамматику такой, как она есть, не стремясь преобразовать ее к более простому классу.

В целом, следует отметить, что с учетом всего сказанного выше 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

Преобразование грамматик. Цель преобразования

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 основных 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 [9, 13, 22].

Все преобразования условно можно разбить на две группы:

- первая группа — это преобразования, связанные с исключением из грамматики избыточных правил и символов, без которых она может существовать (именно эти преобразования позволяют выполнить упрощения правил);
- 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(VT,VN,P,S) GXVr.**VN'.P'**.S'): L**(G)** = L**(G')**

Приведенные грамматики

Приведенные грамматики — это КС-грамматики, которые не содержат недостижимых и бесплодных символов, циклов и ^-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 Х-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 AeVN называется *бесплодным,* если для него выполняется: {а | A=>*a, aeVT*} = 0. 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 **Y,.** 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

1. $Y_0 = 0$, i :- 1;
2. Y, - {A | (A -> a)eP, a e ^ и У Т П и У , . , ;
3. если Y, * Y,_„ то i := i + 1 и перейти к шагу 2, иначе перейти к шагу 4;
4. VN' = Y„ VT' = VT, в P' входят те правила из P, которые содержат только символы из множества (VTuY,), S' = S.

Удаление недостижимых символов

Символ xe(VTuVN) 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) — V,. 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

1. $V_0 = \{S\}$, i := 1;
2. V, - {x | xe(VTuVN) и (A -» axp)eP, AeV,.,, a,pe(VTuVN)'}uV,-i;
3. если V, *ф* V,_i, то i := i + 1 и перейти к шагу 2, иначе перейти к шагу 4;
4. VN' = VNnV,, VT' = VTnV„ в P' входят те правила из P, которые содержат только символы из множества V„ S' = S.

Пример удаления недостижимых и бесплодных символов

Рассмотрим работу алгоритмов удаления недостижимых и бесплодных символов на примере грамматики:

```
K{a,b,c),{A.B.C,D,E.F.G,S).P.S)
э
5 -> aAB | E
A -> aA | ЬB
3 -> ACЬ | b
: -> A I ЬA I cC I aE
I —> cE | aE | Eь | ED j FG
:    a I c I Fb
- -> BC | EC | AC
;    Ga I Gb
```

ВНИМАНИЕ

Для правильного выполнения преобразований необходимо сначала удалить бесплодные символы, а потом — недостижимые символы, но не наоборот.

Удалим бесплодные символы:

1. Y0 = 0, i:=1 (шаг 1);
2. Y1 = {B.D}, Y1*Y0: т=2 (шаги 2 и 3);
3. Y2 = {B.D.A}, Y2 *Y1: i.=3 (шаги 2 и 3);
4. Y3 = {B.D.A.S.C}, Y3 Ф Y2: т :=4 (шаги 2 и 3);
5. Y4 = {B,D,A,S,C,F}, Y4*Y3: i :=5 (шаги 2 и 3);
6. Y5 = {B.D.A.S.C.F}, Y5 = Y4 (шаги 2 и 3);
7. строим множества VN' = {A.B.C.D.F.S}, VT' = (a,Ь.c} и P' (шаг 4).

Получили грамматику:

G'({a.b,c}.{A,B,C.D,F,S},P',S)
P'.
S -> aAB
A -> aA | ЬB
B -> ACЬ | b
C -> A I ЬA I cC
D -> a | c | Fb
F -> BC | AC

Удалим недостижимые символы:

1. V0 - {S}, 1 :=1 (шаг 1);
2. VI = {S.A.B}, VI * V0: т.=2 (шаги 2 и 3);
3. V2 = {S.A.B.C}, V2 * VI: i :=2 (шаги 2 и 3);
4. V3 - {S.A.B.C}, V3 = V2 (шаги 2 и 3);
5. строим множества VN" = {A.B.C.S}, VT" = {a.Ь.c} и P'(шаг 4).

В итоге получили грамматику:

G"({a,b,c}.{A.B.C.S},P",S)
P":
S -> aAB
A —> aA | ЬB
B -> ACЬ | b
C -> A | ЬA | 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 ^.-правил

^.-правилами называются все правила грамматики вида A —> X, где AeVN.

КС-грамматика G(VT,VN,P,S) называется грамматикой без ^.-правил, если в ней не существует правил (A X.)eP, A Ф S и существует только одно правило (S -» eP, в том случае, когда X.eL(G), и при этом S не встречается в правой части ни одного правила.

Для того чтобы упростить построение распознавателей языка L(G), грамматику G часто целесообразно преобразовать к виду без ^.-правил. Существует алгоритм преобразования произвольной КС-грамматики к виду без ^.-правил. Он работает с некоторым множеством нетерминальных символов W,.

Алгоритм устранения ^.-правил по шагам

1. W_0 = {A: (A Я.)eP }, i := 1;
2. W, = W,_1u{A: (A -» a)eP, **aeW,,.'};**
3. если W, ϕ W,_,, то i := i + 1 и перейти к шагу 2, иначе перейти к шагу 4;
4. VN' = VN, VT = VT, в P' входят все правила из P, кроме правил вида A X;
5. если (A —» a)eP и в цепочку a входят символы из множества W„ тогда на основе цепочки a строится множество цепочек {a'} путем исключения из a всех возможных комбинаций символов W„ все правила вида A —> a' добавляются в P' (при этом надо учитывать дубликаты правил и бессмысленные правила);
6. если SeW„ то значит XeL(G), и тогда в VN' добавляется новый символ S', который становится целевым символом грамматики G', а в P' добавляются два новых правила: S' -> X,S; иначе S' = S.

Данный алгоритм часто ведет к увеличению количества правил грамматики, но позволяет упростить построение распознавателя для заданного языка.

Пример устранения ^.-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 | A
C —> AB | c

Удалим ^.-правила:

1. W0 = {B}. T.-1 (шаг 1);
2. W1 = {B,A}, W1 Ф WQ 1 .=2 (шаги 2 и 3);
3. W2 = {B.A.C}, W2 *W1. т =3 (шаги 2 и 3);
4. W3 = (B.A,C}. W3 = W2 (шаги 2 и 3);
5. построим VN' = {A.B.C.S}, VT' = {a.b.c} и множество правил 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 W3 = {B,A,C} и получим новые правила S —> Aa | aB | a | a j 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 W3 = {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 W3 = {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 W3 = {B,A.C} и получим новые правила C -» A] B, добавим их в P', получим: C -» AB | A | B | c.
7. SgW3, поэтому в грамматику G' не надо добавлять новый целевой символ S', 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 j b | B
B Ba | a
C —^ AB j A j B | c

Устранение цепных правил

Циклом (циклическим выводом) в грамматике G(VT,VN,P,S) называется вывод вида A => *A, AeVN. 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,BeVN. 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 XeVN строится специальное множество цепных символов 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. N^x_0 = {X}, i := 1.

Шаг 3. N* - N^*_4u{B: (A B)eP, BeN*,.,}.

Шаг 4. Если N^x, Ф N^x,.u to i := i + 1 и перейти к шагу 3, иначе $N^x = N^*$ - {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 a)eP', если BeN^A, B Ф A, то в P' добавляются правила вида B —^ ct.

Данный алгоритм, так же как и алгоритм устранения ^-правил, ведет к увеличению числа правил грамматики, но упрощает построение распознавателей.

Пример устранения цепных правил

Рассмотрим в качестве примера грамматику арифметических выражений над символами a и Ь, которая уже рассматривалась ранее в этом учебном пособии:

G({+.-./.*.a.b}.{S.T.E},P.S):
P:
s S+T I S-T I T
T T*E I T/E I E
E (S) | a | b

Устраним цепные правила:

1. NSO = {S}. i :=l;
2. NSl = {S.T}, NSl ф NSO. i:=2;
3. NS2 = {S.T.E}. NS2 * NSl, l =3;
4. NS3 = {S.T.E}. NS3=NS2, NS = {T,E};
5. NlO = {T}. l:=1;
6. NTl = {T.E}, NTl #NT0. i:=2;
7. NT2 = {T.E}. NT2 = NT1. NT = {E};
8. NEO = {E}. i.-1;
9. NEl = {E}. NEl = NEO. NE = 0;
10. получили: NS = {T.E}, NT = {E}, NE = 0, 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 NS = {T.E} и NT = {E}:
    - О для правил T —> T*E | T/E имеем новые правила S T*E | T/E, поскольку TeNS;
    - О для правил E (S) | a | b имеем новые правила S (S) | a | b и T (S) | a | Ь, поскольку EeNS и Ee NT.

Получим новую грамматику:

G'({+.-./.*.a,b},{S,T,E}.P'.S)
P':
S S+T | S-T I T*E I T/E I (S) I a I 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 AeVN в КС-грамматике G(VT,VN,P,S) называется рекурсивным, если для него существует цепочка вывода вида A => +aAp, где a,Pe(VTuVN)\
Если a = X и p * X, то рекурсия называется левой, а грамматика **G** — леворекурсивной; если a ϕ X и p = X, то рекурсия называется правой, а грамматика **G** — праворекурсивной. Если a = X и P - X, то рекурсия представляет собой цикл. Алгоритм исключения циклов был рассмотрен выше, поэтому далее циклы рассматриваться не будут.

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.

Доказано, что любую КС-грамматику можно преобразовать к нелеворекурсивному или неправорекурсивному виду [4, т. 1].

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 со 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!,A_2....A_n}, i:= 1.

Шаг 2. Рассмотрим правила для символа A,. Если эти правила не содержат левой рекурсии, то перенесем их во множество правил P' без изменений, а символ A, добавим во множество нетерминальных символов VN'.

Иначе запишем правила для A, в виде A, A,^ | A,$a_2$1...| A,a_m | p, | $p_2$1...| P_p, где VI < j < p ни одна из цепочек Pj не начинается с символов A^, таких, что k < i.

Вместо этого правила во множество P' запишем два правила вида:

Л p, **I** p_2**1...|** p_P **I** р д ' **I** p_2**a/** p_P**a;**

A/ СЦ | $a_2$1...| a_m | a,A,' | a_2A,' a_TA,'

Символы A, и A,' включаем во множество VN'.

Теперь все правила для A, начинаются либо с терминального символа, либо с нетерминального символа $A_к$, такого, что k > i.

Шаг 3. Если i = п, то грамматика G' построена, перейти к шагу 6, иначе i := i + 1, j .— 1 и перейти к шагу 4.

Шаг 4. Для символа Aj во множестве правил P' заменить все правила вида A, -» AjCi, где aE(VTuVN)*, на правила вида A, -> p^ | p_2a $p_т$a, причем Aj P! | $p_2$1...| $P_т$ — все правила для символа $A_т$

Так как правая часть правил Aj p_t | $p_2$1...| $p_т$ уже начинается с терминального символа или нетерминального символа A_k, к > j, то и правая часть правил для символа A, 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 $A_к$, соответствующий символу S исходной грамматики G.

Рассмотрим в качестве примера грамматику для арифметических выражений над символами а и Ь:

G({+,-,/,*,a,b}.{S.T,E},P,S):
P.
S -> S+T | S-T I т
T -> T*E I T/E I E
E -> (S) | a | b

Эта грамматика является леворекурсивной. Построим эквивалентную ей нелеворекурсивную грамматику G'.

Шаг 1. Обозначим VN = {Al, A2. A3}, i := 1.

Тогда правила грамматики G будут иметь вид:

Al -> Al+A2 | Al-A2 | A2
A2 -> A2*A3 | A2/A3 | A3
A3 -> (A1) | a | b

Шаг 2. Для A1 имеем правила Al Al+A2 | Al-A2 | A2. Их можно записать в виде Al A1a1 | Ala2 | pi, где al = +A2, a2 = -A2. pi = A2.

Запишем новые правила для множества P':

Al —> A2 | A2A1'
AГ -> +A2 | -A2 | +A2AГ | -A2A1'

Добавив эти правила в P', а символы A1 и A1' во множество нетерминальных символов, получим: VN = {Al ,A1'}.

Шаг3. 1 = 1 < 3. Построение не закончено: i := т+1 = 2. j := 1.

Шаг 4. Для символа A2 во множестве правил P' нет правила вида A2 Ala, поэтому на этом шаге никаких действий не выполняем.

Шаг 5. з = 1 = 1-1, переходим опять к шагу 2.

Шаг 2. Для A2 имеем правила A2 -> A2*A3 | A2/A3 | A3. Их можно записать в виде A2 -> A2al | A2a2 | pi, где al = *A3, a2 = /A3, pi = A3.

Запишем новые правила для множества P':

A2 A3 | A3A2'
A2' *A3 | /A3 1 *A3A2' | /A3A2'

Добавим эти правила в P', а символы A2 и A2' во множество нетерминальных символов, получим:VN' = {A1.AГ .A2.A2'}.

Шаг3. 1 = 2 < 3. Построение не закончено: i := i + 1 = 3. j .= 1.

Шаг 4. Для символа A3 во множестве правил P' нет правила вида A3 -» Ala, поэтому на этом шаге никаких действий не выполняем.

Шаг 5. j = 1 < 1 -1, з .= 3 + 1 = 2, переходим к шагу 4.

Шаг 4. Для символа A3 во множестве правил P' нет правила вида A3 -> A2a, поэтому на этом шаге никаких действий не выполняем.

Шаг 5. J = 2 = i - 1, переходим опять к шагу 2.

Шаг 2. Для A3 имеем правила A3 -> (Al) | a | Ь. Эти правила не содержат левой рекурсии. Переносим их в P', а символ A3 добавляем в VN'. Получим: VN' = {A1, A1[1].A2.A2'.A3}.

Шаг 3. л =3 = 3. Построение грамматики G' закончено.

В результате выполнения алгоритма преобразования получили нелеворекурсивную грамматику G({+.-,/.*. a, b}. {Al,Al' .A2.A2'. A3}. P' ,A1) с правилами:

P':
Al —> A2 | A2A1'
AГ +A2 | -A2 | +A2A1' | -A2A1'
A2 -> A3 | A3A2'
A2' *A3 | /A3 1 *A3A2' | /A3A2[1]
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 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 одного 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 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 — 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: a, aeVT', n = |a|. Конкретная зависимость определяется вариантом реализации алгоритма [4, т. 1].

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: $T_э = 0(e^n)$ 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 $M_э = 0(e^n)$.

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 простых 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.

Нисходящий распознаватель с возвратом

Принцип работы нисходящего распознавателя с подбором альтернатив

Нисходящий распознаватель с подбором альтернатив моделирует работу МП-автомата с одним состоянием q: R({q},V,Z,8,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 = VTuVN.

Начальная конфигурация автомата определяется так: (q,a,S) — автомат пребывает в своем единственном состоянии q, считывающая головка находится в начале входной цепочки символов aeVT*, в стеке лежит символ, соответствующий целевому символу грамматики S.

[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.

Конечная конфигурация автомата определяется так: ^ДД) — 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,a)e8(q,^,A), AeVN, ae(VTuVN)", если правило A -> a содержится во множестве правил P грамматики G: A a e P;
2. (яД)б8(ч,а,а) VaeVT.

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 a, если в грамматике языка есть правило A -> a, 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 a, тогда функция 8(q,X,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 альтернатив и хранение информации о том, какие альтернативы на каком шаге уже были выбраны, чтобы иметь возможность вернуться к этому шагу и подобрать другие альтернативы. 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.

Реализация алгоритма распознавателя с подбором альтернатив

Существует масса способов реализации алгоритма с подбором альтернатив [4, т. 1, 31]. Рассмотрим один из примеров реализации алгоритма нисходящего распознавателя с возвратом.

Для удобства работы все правила из множества P в грамматике G представим в виде A -» сц | $a_2$1...| a_n, то есть пронумеруем все возможные альтернативы для каждого нетерминального символа AeVN. Входная цепочка символов имеет вид a = a^.-.$a_{,,,}$ |a| = п. Алгоритм использует также дополнительное состояние b (от «back » — «назад»), которое сигнализирует о выполнении возврата к уже

прочитанной части входной цепочки[1]. Для хранения уже выбранных альтернатив используется дополнительный стек L_2, который может содержать следующую информацию:

- символы aeVT входного языка автомата;
- символы вида A_r где AeVN, — это означает, что среди всех возможных правил для символа А была выбрана альтернатива с номером j.

В итоге алгоритм работает с двумя стеками: L] — стек МП-автомата и L_2 — стек возвратов. Оба они представлены в виде цепочек символов. Символы в цепочку стека L] помещаются слева, а в цепочку стека L_2 — справа. В целом, состояние алгоритма на каждом шаге определяется четырьмя параметрами: (Q,i,L_1,L_2), где

- Q — текущее состояние автомата (q или Ь);
- i — положение считывающей головки во входной цепочке символов а;
- L_1 — содержимое стека МП-автомата;
- L_2 — содержимое дополнительного стека.

Начальным состоянием алгоритма является состояние (q,l,S,^), где S — целевой символ грамматики. Алгоритм начинает свою работу с начального состояния и циклически выполняет шесть шагов до тех пор, пока не перейдет в конечное состояние или не обнаружит ошибку. На каждом шаге алгоритма проверяется, соответствует ли текущее состояние алгоритма заданному для данного шага исходному состоянию и выполняются ли заданные дополнительные условия. Если это требование выполняется, алгоритм переходит в следующее состояние, установленное для этого шага, если нет — шаг пропускается, алгоритм переходит к следующему шагу.

Алгоритм предусматривает циклическое выполнение следующих шагов:

Шаг 1 (Разрастание), (q, i, Ар, *а)* (q, i, уД aA_t), если А у, — это первая из всех возможных альтернатив для символа А.

Шаг 2 (Успешное сравнение), (q, i, ар, а) -» (q, i+1, р, аа), если а = a_i, aeVT.

Шаг 3 (Завершение). Если состояние соответствует (q, n+1, *X,* а), то разбор завершен, алгоритм заканчивает работу, иначе (q, i, X, *А)* -» (b, i, X, а), когда i Ф n + 1.

Шаг 4 (Неуспешное сравнение), (q, i, ар, а) (b, i, ар, а), если а Ф a_i, aeVT.

Шаг 5 (Возврат по входу), (b, i, р, аа) -» (q, i-1, ар, а), V aeVT.

Шаг 6 (Другая альтернатива). Исходное состояние (b, i, уД aAj), действия:

- перейти в состояние (q, i, y_{j+1}p, aA_{j+1}), если еще существует альтернатива А -» y_{j+1} для символа AeVN;
- сигнализировать об ошибке и прекратить выполнение алгоритма, если A^S и не существует больше альтернатив для символа S;
- иначе перейти в состояние (xj, i, Ар, а).

[1] Сам автомат имеет только одно состояние q, которого достаточно для его функционирования, однако нет возможности моделировать на компьютере работу недетерминированного автомата, поэтому приходится выполнять возврат к уже прочитанной части цепочки и вводить для этой цели дополнительное состояние.

В случае успешного завершения алгоритма цепочку вывода можно построить на основе содержимого стека L_2, полученного в результате выполнения алгоритма. Цепочка вывода строится следующим образом: поместить в цепочку номер правила т, соответствующий альтернативе А -» y_k если в стеке содержится символ Aj (все символы aeVT, содержащиеся в стеке L_2, игнорируются).

ВНИМАНИЕ

Следует помнить, что для применения этого алгоритма исходная грамматика не должна быть леворекурсивной. Если это условие не выполняется, то грамматику предварительно надо преобразовать к нелеворекурсивному виду.

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 j /EF
E (S) | a | b

Это нелеворекурсивная грамматика для арифметических выражений (ранее в разделе «Устранение левой рекурсии» она была построена с помощью алгоритма устранения левой рекурсии).

Проследим разбор цепочки a+(a*Ь) из языка этой грамматики с помощью алгоритма нисходящего распознавателя с возвратами. Работу алгоритма будем представлять в виде последовательности его состояний, взятых в скобки {} (фигурные скобки используются, чтобы не путать их с круглыми скобками, предусмотренными в правилах грамматики). Альтернативы будем нумеровать слева направо, правила — слева направо и сверху вниз. Для пояснения каждый шаг работы сопровождается номером шага алгоритма, который был применен для перехода в очередное состояние (записывается через символ : — двоеточие).

.Алгоритм работы нисходящего распознавателя с возвратами при разборе цепочки a+(a*Ь) будет выполнять следующие шаги:

1. 0: {q. 1 .SД}
2. 1: {q.l.T.Sl}
3. 1: {q.l.E.SlTl}
4. 1: {q.l,{S}.SlTlEl}
5. 4: {b.l,{S}.SlTlEl}
6. 6: {q.l,a.SlTlE2}
7. 2: {q,2.A..SlTlE2a}
8. 3: {b.2.X.SlTlE2a}
9. 5: {b.l.a.S1T1E2}
10. 6: {q,l.b,SlTlE3}

```
И.  4: | b,l.b.SITlE3}
12.  6: | b.1.E.S1T1}
13.  6: | q.1.EF.S1T2}
14.  1: | q,1,{S}F.S1T2E1}
15.  4: | b.1,{S}F.S1T2E1}
16.  6: | q,l,aF.SlT2E2}
17.  2: | q.2.F.SlT2E2a}
18.  1: | q,2,*E.SlT2E2aFl}
19.  4: | b,2,*E,SlT2E2aFl}
20.  6: | q.2./E.SlT2E2aF2}
21.  4: | b,2./E,SU2E2aF2}
22.  6: | q.2,*EF,SlT2E2aF3}
23.  4: | b.2,*EF,SlT2E2aF3}
24.  6: | q,2,/EF,SlT2E2aF4}
25.  4: | b.2./EF.SlT2E2aF4}
26.  6: | b.2.F.SlT2E2a}
27.  5: | b.1.aF.S1T2E2}
28.  6: | q.l,bF,SlT2E3}
29.  4: | b.l.bF.SlT2E3}
30.  6: | b.l.EF.SlT2}
31.  6: | b.1.T.S1}
32.  6: | q.l,TR,S2}
33.  1: | q.1.ER.S2T1}
34.  1: | q,l.{S}R.S2T1E1}
35.  4: | b.1.{S}R,S2T1E1}
36.  6: | q.1.aR.S2T1E2}
37.  2: | q.2.R.S2TlE2a}
38.  1: | q,2,+T,S2TlE2aRl}
39.  2: | q,3.T.S2TlE2aRl+}
40.  1: | q.3.E.S2TlE2aRl+Tl}
41.  1: | q,3.(S),S2TlE2aRl+TlEl}
42.  2: | q.4.S),S2TlE2aRl+TlEl(}
43.  1: | q,4,T),S2TlE2aRl+TlEl(Sl}
44.  1: | q,4,E),S2TlE2aRl+TlEl(SlTl}
45.  1: | q,4,(S)),S2TlE2aRl+TlEl(SlTlEl}
46.  4: | b.4.(S)),S2TlE2aRl+TlEl(SlTlEl}
47.  6: | q.4,a),S2TlE2aRl+TlEl(SlTlE2}
```

48. 2: {q.5.),S2TlE2aRl+TlEl(SITlE2a}
49. 4: {b,5,),S2TlE2aRl+TlEl(SITlE2a}
50. 5: {b,4,a),S2TlE2aRl+TlEl(SlTlE2}
51. 6: {q,4,b),S2TlE2aRl+TlEl(SITlE3}
52. 4: {b.4,b).S2TlE2aRl+TlEl(SITlE3}
53. 6: {b,4,E),S2TlE2aRl+TlEl(SlTl}
54. 6: {q.4,EF).S27TE2aRl+TlEl(SlT2}
55. 1: {q.4,(S)F),S2TlE2aRl+TlEl(SlT2El}
56. 4: {b.4,(S)F),S2TlE2aRl+TlEl(SlT2El}
57. 6: {q,4,aF),S2TlE2aRl+TlEl(SlT2E2}
58. 2: (q.5,F).S2TlE2aRl+TlEl(SlT2E2a}
59. 1: {q.5,*E).S2TlE2aRl+TlEl(SlT2E2aFl}
60. 2: {q,6,E),S2TlE2aRl+TlEl(SlT2E2aFl*}
61. 1: {q.6. (S)),S2TlE2aRl+TlEl(SlT2E2aFl*El}
62. 4: {b,6,(S)),S2TlE2aRl+TlEl(SlT2E2aFl*El}
63. 6: {q.6.a),S2TlE2aRl+TlEl(SlT2E2aFl*E2}
64. 4: {b,6,a),S2TlE2aRl+TlEl(SlT2E2aFl*E2}
65. 6: {q,6,b), S2TlE2aRl+TlEl(SlT2E2aFl*E3}
66. 2: {q.7,),S2TlE2aRl+TlEl(SlT2E2aFl*E3b}
67. 2: {q.8 Л,S2TlE2aRl+TlEl(SlT2E2aFl*E3b)}
68. 3: Разбор закончен, алгоритм завершен.

На основании полученной цепочки номеров альтернатив S2T1E2R1T1E1S1T2E2F1E3 построим последовательность номеров примененных правил: 2, 7, 14, 3, 7, 13, 1, 8, 14, 9, 15. Получаем левосторонний вывод: S => TR => ER => aR => a+T => a+E => a+(S) => a+(T) => a+(EF) => a+(aF) => a+(a*E) => a+(a*b). Соответствующее ему дерево вывода приведено на рис. 4.2.

Из приведенного примера очейиден недостаток алгоритма нисходящего разбора с возвратами — значительная временная емкость: для разбора достаточно короткой входной цепочки (всего 7 символов) потребовалось 68 шагов работы алгоритма. Такого результата и следовало ожидать.

Преимуществом данного алгоритма можно считать простоту его реализации. Практически этот алгоритм разбора можно использовать только тогда, когда известно, что длина исходной цепочки символов заведомо не будет большой. Для реальных компиляторов такое условие невыполнимо, но для некоторых небольших распознавателей вполне допустимо, и тогда данный алгоритм разбора может найти применение именно благодаря своей простоте.

Еще одно преимущество алгоритма — его универсальность. На его основе можно распознавать входные цепочки языка, заданного любой КС-грамматикой, достаточно лишь привести ее к нелеворекурсивному виду (а это можно сделать с любой грамматикой). Интересно, что грамматика даже не обязательно должна быть

однозначной — для неоднозначной грамматики алгоритм найдет один из возможных левосторонних выводов.

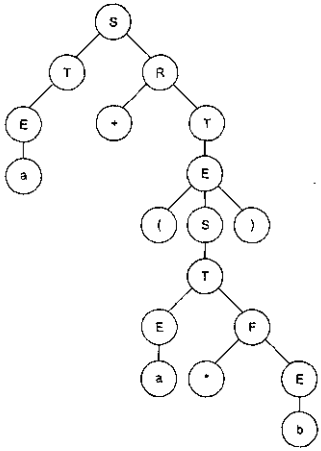


Рис. 4.2. Дерево вывода для грамматики без левых рекурсий

Сам по себе алгоритм разбора с подбором альтернатив, использующий возвраты, не находит применения в реальных компиляторах. Однако его главные 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.

Распознаватель на основе алгоритма «сдвиг-свертка»

Принцип работы восходящего распознавателя по алгоритму «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 = VTuVN.

Начальная конфигурация автомата определяется так: (я,аД) — автомат пребывает в своем единственном состоянии q, считывающая головка находится в начале входной цепочки символов aeVT, стек пуст.

Конечная конфигурация автомата определяется так: (q,^,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)e5(q,/Wy), AeVN, ye(VTuVN)', если правило A -» у содержится во множестве правил P грамматики G: A —> yeP;
2. (q,a)e8(q,a,^) VaeVT.

Неформально работу этого расширенного автомата можно описать так: если на верхушке стека находится цепочка символов у, то ее можно заменить на нетерминальный символ А, если в грамматике языка существует правило вида A -> у, не сдвигая при этом считывающую головку автомата (этот шаг работы называется «свертка»); с другой стороны, если считывающая головка автомата обозревает некоторый символ входной цепочки а, 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 ^-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 ^-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 у выбрать для поиска правил (цепочка у должна встречаться в правой части правил грамматики);
3. какое правило выбрать для свертки, если окажется, что существует несколько правил вида A -> у (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.

Реализация распознавателя с возвратами на основе алгоритма «сдвиг-свертка»

Существует несколько реализаций алгоритма «сдвиг-свертка» с возвратами [4, т. 1, 15]. Один из вариантов рассмотрен ниже.

Для работы алгоритма всем правилам грамматики G(VT,VN,P,S), на основе которой построен автомат, необходимо дать порядковые номера. Будем нумеровать правила грамматики в направлении слева направо и сверху вниз в порядке их записи в форме Бэкуса—Наура. Входная цепочка символов имеет вид a = a^-.a,,, |a| = n. Алгоритм моделирования расширенного МП-автомата, аналогично алгоритму нисходящего распознавателя, использует дополнительное состояние b и дополнительный стек возвратов L_2. В стек помещаются номера правил грамматики, использованных для свертки, если на очередном шаге алгоритма была выполнена свертка, или 0, если на очередном шаге алгоритма был выполнен сдвиг.

В итоге алгоритм работает с двумя стеками: L, — стек МП-автомата и L_2 — стек возвратов. Первый представлен в виде цепочки символов, второй — цепочки целых чисел от 0 до т, где т — количество правил грамматики G. Символы в цепочку стека L, помещаются справа, числа в стек L_2 — слева. В целом, состояние алгоритма на каждом шаге определяется четырьмя параметрами: (Q, i, L,, L_2), где

- Q — текущее состояние автомата (q или Ь);
- i — положение считывающей головки во входной цепочке символов a;
- L, — содержимое стека МП-автомата;
- L_2 — содержимое дополнительного стека возвратов.

Начальным состоянием алгоритма является состояние (q, 1, X, X). Алгоритм начинает свою работу с начального состояния и циклически выполняет пять шагов до тех пор, пока не перейдет в конечное состояние или не обнаружит ошибку.

Алгоритм предусматривает циклическое выполнение следующих шагов:

Шаг 1 (Попытка свертки), (q, i, оф, y) (q, i, aA, jy), если A p — это первое из всех возможных правил из множества правил P с номером j для подцепочки p, причем оно есть первое подходящее правило для цепочки ap, для которой правило вида A p существует. Если удалось выполнить свертку — возвращаемся к шагу 1, иначе — переходим к шагу 2.

Шаг 2 (Перенос — сдвиг). Если i < n + 1, то (q, i, a, y) (q, i+1, aa,, Oy), a,eVT. Если i = n + 1, то перейти к шагу 3, иначе перейти к шагу 1.

Шаг3 (Завершение). Если состояние соответствует (q, n+1, S, y), то разбор завершен, алгоритм заканчивает работу, иначе перейти к шагу 4.

Шаг 4 (Переход к возврату), (q, n+1, a, y) (Ь, n+1, a, y).

Шаг 5 (Возврат). Если исходное состояние (b, i, aA, jy), то:

- перейти в состояние (q, i, a'B, ky), если j > 0, и A —> p — это правило с номером j, и существует правило B p' с номером к, к > j, такое, что ap = a'p'; после чего надо вернуться к шагу 1;

- перейти в состояние (Ь, п+1, оф, у), если i - n + 1, j > 0, A p — это правило с номером j, и не существует других правил из множества Р с номером k > j, таких, что их правая часть является правой подцепочкой из цепочки оф; после этого вернуться к шагу 5;
- перейти в состояние (q, i+1, ара,, Оу), a,e VT, если i * n + 1, j > O, A p — это правило с номером j, и не существует других правил из множества Р с номером k > j, таких, что их правая часть является правой подцепочкой из цепочки аР; после этого перейти к шагу 1;
- иначе сигнализировать об ошибке и прекратить выполнение алгоритма.

Если исходное состояние (b, i, аа, Оу), а eVT, то если i > 1, тогда перейти в следующее состояние (b, i-1, a, y), и вернуться к шагу 5; иначе сигнализировать об ошибке и прекратить выполнение алгоритма.

В случае успешного завершения алгоритма цепочку вывода можно построить на основе содержимого стека L_2, полученного в результате выполнения алгоритма. Для этого достаточно удалить из стека L_2 все цифры 0.

ВНИМАНИЕ

Следует помнить, что для применения этого алгоритма исходная грамматика не должна содержать циклов и /.-правил. Если это условие не удовлетворяется, то грамматику надо предварительно преобразовать к приведенной форме.

Возьмем в качестве примера грамматику G({+.-,/,*,a,b}, {S,T,E} ,P,S):

P:
S -» S+T | S-T I T*E I T/E I (S) I a I b
T —> T*E I T/E I (S) I a I b
E -> (S) | a | b

Это грамматика для арифметических выражений, в которой устранены цепные правила (ее уже рассматривали в разделе «Преобразование КС-грамматик. Приведенные грамматики»). Следовательно, в ней не может быть циклов[1]. Кроме того, видно, что в ней нет ^.-правил. Таким образом, цепочки языка, заданного этой грамматикой, можно распознавать с помощью алгоритма «сдвиг-свертка». Проследим разбор цепочки a+(a*b) из языка этой грамматики. Работу алгоритма будем представлять в виде последовательности его состояний, взятых в скобки {} (фигурные скобки используются, чтобы не путать их с круглыми скобками, предусмотренными в правилах грамматики). Правила будем нумеровать слева направо и сверху вниз (всего в грамматике получается 15 правил). Для пояснения каждый шаг работы сопровождается номером шага алгоритма, который был применен для перехода в очередное состояние (записывается перед состоянием через символ «:» — двоеточие).

[1] На самом деле исходная грамматика для арифметических выражений, которая была рассмотрена в разделе «Проблемы однозначности и эквивалентности грамматик» главы 1, тоже не содержит циклов (это легко заметить из вида ее правил), однако для чистоты утверждения цепные правила были исключены.

Алгоритм работы восходящего распознавателя с возвратами при разборе цепочки a+(a*b) будет выполнять следующие шаги:

1. 0: {q.l**.a..*.}**
2. 2: {q,2,a,[0]}
3. 1: {q,2,S.[6,0]}
4. 2: {q.3,S+.[0.6,0]}
5. 2: {q.4,S+(,[0,0,6.0]}
6. 2: {q.5.S+(a.[0.0.0.6.0]}
7. 1: {q.5,S+CS.[6.0.0,0.6,0]}
8. 2: {q.6,S+(S*,[0.6,0,0,0.6.0]}
9. 2: {q.7,S+(S*b,[0,0,6,0.0.0,6,0]}
10. 1: {q,7,S+(S*S,[7.0,0,6,0,0,0,6,0]}
11. 2: {q.8.S+(S*S).[0.7,0,0,6.0.0,0,6,0]}
12. 4: {b.8,S+(S*S),[0,7,0,0,6,0,0,0,6,0]}
13. 5: {b,7.S+(S*S,[7.0.0,6,0.0.0.6,0]}
14. 5: {q,7,S+(S*T,[12.0,0.6,0,0,0.6,0]}
15. 2: (q,8.S+(S*T),[0,12,0,0,6,0,0,0,6,0]}
16. 4: {b,8,S+(S*T),[0,12,0,0,6,0,0,0.6,0]}
17. 5: {b.7,S+(S*T,[12,0,0,6.0,0,0,6,0]}
18. 5: {q,7,S+(S*E,[15,0,0,6,0,0,0,6,0]}
19. 2: {q,8,S+(S*E).[0,15,0,0,6,0,0.0.6,0]}
20. 4: {b.8.S+(S*E),[0.15.0.0.6,0.0.0.6,0]}
21. 5: {b,7,S+(S*E,[15.0.0.6.0.0,0.6,0]}
22. 5: {q.8.S+(S*a).[0,0,0,6,0,0,0,6.0]}
23. 4: {b.8.S+(S*a).[0.0.0,6.0.0.0.6.0]}
24. 5: {b.7,S+(S*a.[0,0.6,0,0,0,6,0]}
25. 5: {b.6.S+(S*.[0.6.0.0.0,6.0]}
26. 5: {b.5.S+(S,[6,0.0.0,6.0]}
27. 5: {q,5,S+(T,[11,0.0,0,6,0]}
28. 2: {q.6.S+(T*.[0.11.0.0.0.6.0]}
29. 2: {q.7,S+(T*b.[0,0,ll,0.0.0,6,0]}
30. 1: {q,7,S+(T*S,[7,0,0,ll,0,0,0,6,0]}
31. 2: {q,8.S+(T*S),[0.7,0,0,11,0,0,0,6,0]}
32. 4: {b.8,S+(T*S).[0,7,0,0,11,0,0,0.6,0]}
33. 5: {b,7,S+(T*S,[7.0.0,11.0,0,0,6,0]}
34. 5: {q,7,S+(T*T,[12,0,0,ll,0.0,0,6,0]}

35. 2:{q.8.S+(T*T).[0,12,0.0.11,0,0.0,6.0]}
36. 4:{b.8,S+(T*T).[0.12.0,0.11,0,0,0,6,0]}
37. 5:{b.7,S+(T*T,[12,0,0,ll,0.0.0,6,0]}
38. 5:{q.7,S+(T*E,[15,0,0,ll,0.0,0,6,0]}
39. 1:{q,7,S+(S,[3,15,0,0,ll,0.0,0,6,0]}
40. 2:{q.8,S+(S).[0,3,15,0.0,11,0,0,0,6,0]}
41. 1:{q.8.S+S,[5,0.3,15,0,0,11.0.0.0,6,0]}
42. 4:{b,8.S+S.[5,0.3,15,0,0,11,0,0,0,6,0]}
43. 5:{q,8,S+T,[10,0,3,15,0.0,11,0,0,0,6,0]}
44. 1:{q.8.S.[1,10,0,3,15,0,0,11,0,0,0,6,0]}
45. 3: Разбор закончен, алгоритм завершен.

На основании полученной цепочки номеров правил: 1, 10, 3, 15, 11, 6 получаем правосторонний вывод: S ⇒ S+T ⇒ S+(S) ⇒ S+(T*E) ⇒ S+(T*b) ⇒ S+(a*b) ⇒ a+(a*b). Соответствующее ему дерево вывода приведено на рис. 4.3.

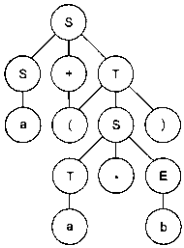


Рис. 4.3. Дерево вывода для грамматики без цепных правил

В приведенном примере очевиден тот же недостаток алгоритма восходящего разбора с возвратами, что и у алгоритма нисходящего разбора с возвратами — значительная временная емкость: для разбора достаточно короткой входной цепочки (всего 7 символов) потребовалось 45 шагов работы алгоритма.

Преимущество у данного алгоритма то же, что и у алгоритма нисходящего разбора с возвратами, — простота реализации. Поэтому и использовать его можно практически в тех же случаях — когда известно, что длина исходной цепочки символов заведомо не будет большой.

Этот алгоритм также универсален. На его основе можно распознавать входные цепочки языка, заданного любой КС-грамматикой, достаточно лишь преобразовать ее к приведенному виду, чтобы она не содержала цепных правил и ^.-правил.

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 требующимся вычислительным ресурсам и одинаково просты в реализации. То, какой из них лучше взять для реализации простейшего распознавателя в том или ином случае, зависит прежде всего от грамматики языка. В рассмотренном примере восходящий алгоритм смог построить вывод за меньшее число шагов, чем нисходящий, — но это еще не значит, что он во всех случаях будет эффективнее для рассмотренного языка арифметических выражений. Вопрос о выборе типа распознавателя — нисходящий либо восходящий — был рассмотрен выше.

Нисходящие распознаватели КС-языков без возвратов

Стремление улучшить алгоритм с подбором альтернатив 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 aeVT, 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 5(q,a,a) = {(q,^)}, VaeVT 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 AeVN грамматики G(VN,VT,P,S) строится процедура разбора, которая получает на

вход цепочку символов a 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 ay, aeVT, ye(VNuVT)*, первый символ правой части которого совпадал бы с текущим символом входной цепочки a = a,. Если такого правила не найдено, то цепочка не принимается. Иначе (если найдено правило A -> ay или для символа A в грамматике G существует только одно правило A -> y), запоминается номер правила и, когда a = считывающая головка передвигается (увеличивается i), а для каждого нетерминального символа в цепочке y рекурсивно вызывается процедура разбора этого символа.

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(VN,VT,P,S) VAeVN возможны только два варианта правил:

A -> y, ye(VNuVT)* и это единственное правило для A;

A $a_1pi \mid a_2p_21... \mid a_np_n$, Vi: a,eVT, p,e(VNuVT)* и если i * j, то a, * a,.

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 бы позволил преобразовать произвольную КС-грамматику к указанному выше виду, равно как не существует и алгоритма, который бы позволял 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).

Эти преобразования заключаются в следующем:

1. исключение ^.-правил;
2. исключение левой рекурсии;
3. добавление новых нетерминальных символов (левая факторизация), например, если правило имеет вид: A aa, | $aa_21... \mid aa_n, \mid Ь,р, \mid b_2p_21... \mid b_mp_m$ и ни одна цепочка символов Pj не начинается с символа a, то заменяем его на два: A aA' | Ь,р! | $Ь_2p_21.,. \mid b_mp_m$ и A' -> a, | $a_21... \mid a_n$;

4. замена нетерминальных символов в правилах на цепочки их выводов, например, если имеются правила:

   A -> B, | $B_2$1...| B_n | b,p, | $b_2p_2$1...| b_mp_m,

   Bi -> a,, | a_{12}1...| $a_{1к}$,

   B_n a„i | a_{n2}1...| $a_{пр}$.

   заменяем первое правило на:

   A -> $a_и$ | a_{12}1—| $a_{1к}$ $a_{п1}$ | a_{n2}1...| $a_{пр}$ | b^ | $b_2p_2$1...| b_mp_m.

Левую факторизацию можно применять к правилам грамматики несколько раз с целью исключить для каждого нетерминального символа правила, начинающиеся с одних и тех же терминальных символов.

В целом алгоритм рекурсивного спуска эффективен и прост в реализации, но имеет очень ограниченную применимость.

Пример реализации метода рекурсивного спуска

Дана грамматика GC{a ,b.c}. {A.B.C.S} .P.S):

P:
S -> aA | ЬВ
A -> a | ЬА | 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 С, 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)
{
   pi Rul [*iP] = iRule;
   *iP = *iP + 1;
}

int proc_S (char* szS, int iN. mt* piRul. int* iP)
{
   switch (szS[iN])
   {
      case 'a':
         WriteRules(piRul ,iP,1):
         return proc_A(szS,iN+l,piRul,iP):
      case 'b'.
         WriteRules (pi Rul ,iP.2);
         return proc_B(szS,iN+l.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+l;
      case 'b':
         WriteRules (pi Rul .iP,4);
         return proc_A(szS,iN+l,piRul,iP);
      case 'c':
         WriteRules(piRul,iP,5);
         return proc_C(szS,iN+l,piRul,iP):
   }
   return 0;
}

int proc_B (char* szS, int iN, int* piRul, int* iP)
{
   switch (szS[iN])
   {
      case 'b':
         WriteRules (pi Rul л P. 6);
         return iN+l;
      case 'a':
         WriteRules(piRul .iP,7);
         return proc_B(szS,iN+l.piRul,iP):
```

```
        case 'c':
            WriteRules(piRul .1P.8);
            return proc_B(szS,lN+l.piRul ,iP);
    }
    return 0;
}

int proc_C (char* szS. int iN, int* piRul, int* iP)
{ int i.
    WriteRules(piRul лР,9):
    i = proc_A(szS,iN.piRul,iP);
    if (l == 0) return 0,
    if (szS[i] != 'a') return 0;

    l = proc_B(szS,i.pi Rul,iP);
    if (i == 0) return 0:
    if (szS[i] != 'b') return 0:
    return l+l;
}
```

Теперь для распознавания входной цепочки необходимо иметь целочисленный массив Rules достаточного объема для хранения номеров правил. Тогда работа распознавателя заключается в вызове процедуры proc_S(Str,0,Rules,&Ю, 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 — цепочка 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. Во избежание 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-авто-

[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.

мата. 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  I  S-T  I  T
T     T*E  I  T/E  I  E
E -» (S) | a | b
```

Это грамматика для арифметических выражений, которая уже была рассмотрена в разделе «Проблемы однозначности и эквивалентности грамматик» главы 1 и служила основой для построения распознавателей в разделе «Синтаксические распознаватели с возвратом».

Запишем правила этой грамматики в форме с применением метасимволов. Получим:

```
P:
S     T{(+T,-T)}
T     E{(*E,/E)}
E -» (S) | a | b
```

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 (, а или Ь, 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 Borland Delphi, 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.l,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.l.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].

```
procedure proc_S (S  string; i,n- integer, var pr string): integer;
var si • string:
begin
   i  := proc_T(S,i.n.sl):
   if l > 0 then
   begin
      pr  .= '3.' + si:
      while (i <= n) and (i <> 0) do
      case S[i] of
         '+': begin
            if i = n then i  := 0
            else
            begin
               l  .= proc_T(S.i+l.n.sl);
               pr  := '1,' + pr +','+ si;
            end:
         end:
         '-': begin
            if i = n then l  •= 0
            else
```

[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.

```
            begin
                i := proc_T(S.i+l.n.sl):
                pr : = '2.' + pr +','+ si:
            end;
        end:
        else break;
    end;{case}
end; {if}
proc_S := i;
end;

procedure proc_S CS: string: i,n: integer; var pr: string): integer:
var si : string;
begin
    i := proc_E(S.i.n.sl);
    if l > 0 then
    begin
        pr := '6.' + si:
        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+l.n.sl);
                    pr := 4,' + pr +','+ si;
                end:
            end;
            '/': begin
                if i = n then i := 0
                else
                begin
                    l := proc_E(S.i+l.n.sl):
                    pr := '5.' + pr +','+ si;
                end:
            end;
            else break;
        end:{case}
    end;{if}
    proc_S := i;
end:

orocedure proc_E (S: string; i.n: integer; var pr: string): integer;
jar si : string:
segin
    case S[i] of
        'a': begin
            pr := '8':
            proc_E := i+l;
        end;
```

```
        'b"•  begin
            pr  .=  '9';
            proC_E  :=  1+1;
        end;
        '('   begin
            proc_E  :=  0;
            if i  <  n  then
            begin
                i  :=  proc_S(S,i+l.n.sl);
                if  (i  >  0)  and  (l  <  n)  then
  -             begin
                    pr  :=  '7,'  +  si:
                    if S[i]  =  ')'  then  proc_E  ;=  i+l;
                end;
            end:
        end;
        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 различных 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, 13, 15, 17, 19. 50].

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 вне зависимости от объема грамматики (количества правил и символов в ней). Эти предварительные действия можно выполнить на компьютере, в то время как расширенная трактовка рекурсивного спуска предполагает неформальный анализ грамматики человеком, и в этом серьезный недостаток метода.

Щк)-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. Тем не менее

существует класс грамматик, основанный именно на этом принципе — выборе одной альтернативы из множества возможных на основе нескольких очередных символов в цепочке. Это ЬЦк)-грамматики. Правда, алгоритм работы распознавателя для них не так очевидно прост, как рассмотренный выше алгоритм рекурсивного спуска.

Грамматика обладает свойством LL(k), к > 0, если на каждом шаге вывода для однозначного выбора очередной альтернативы достаточно знать символ на верхушке стека и рассмотреть первые к символов от текущего положения считывающей головки во входной цепочке символов.

Грамматика называется *LL(k)-грамматикой,* если она обладает свойством LL(k) для некоторого к > О[1].

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 «к» в названии класса грамматики стоит некоторое число, которое показывает, сколько символов надо рассмотреть, чтобы однозначно выбрать одну из множества альтернатив. Так, существуют ЬЦ1)-грамматики, ЬЬ(2)-грамматики и другие классы. В совокупности все IX(к)-грамматики для всех к > 0 образуют класс LL-грамматик. На рис. 4.4 схематично показано частичное дерево вывода для некоторой LL(k)-грамматики. В нем со обозначает уже разобранную часть входной цепочки а, которая построена на основе левой части дерева у. Правая часть дерева х — это еще не разобранная часть, а А — текущий нетерминальный символ на верхушке стека МП-автомата. Цепочка х представляет собой незавершенную часть цепочки вывода, содержащую как терминальные, так и нетерминальные символы. После завершения вывода символ А раскрывается в часть входной цепочки и, а правая часть дерева х преобразуется в часть входной цепочки *х.* Свойство LL(k) предполагает, что однозначный выбор альтернативы для символа А может быть сделан на основе к первых символов цепочки их, являющейся частью входной цепочки а. Алгоритм разбора входных цепочек для ЬЦк)-грамматики носит название «к-предсказывающего алгоритма». Принципы его выполнения во многом соответствуют функционированию МП-автомата с той разницей, что на каждом шаге работы этот алгоритм может просматривать к символов вперед от текущего положения считывающей головки автомата.

Для ЬЦк)-грамматик известны следующие полезные свойства:

- всякая ЬЬ(к)-грамматика для любого к > 0 является однозначной;

И существует алгоритм, позволяющий проверить, является ли заданная грамматика ЬЦк)-грамматикой для строго определенного числа к.

Требование к > 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 к = 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.

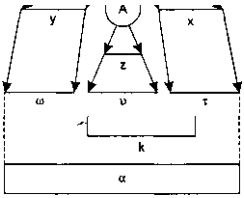


Рис. 4.4. Схема построения дерева вывода для Щк)-грамматики

Кроме того, известно, что все грамматики, допускающие разбор по методу рекурсивного спуска, являются подклассом ЬЦ1)-грамматик. То есть любая грамматика, допускающая разбор по методу рекурсивного спуска, является LL(^-грамматикой (но не наоборот!).

Есть, однако, неразрешимые проблемы для ЬЦк)-грамматик:

- не существует алгоритма, который бы мог проверить, является ли заданная КС-грамматика ЬЦк)-грамматикой для некоторого произвольного числа к;
- не существует алгоритма, который бы мог преобразовать произвольную КС-грамматику к виду ЬЬ(к)-грамматики для некоторого к (или доказать, что преобразование невозможно).

Это общие проблемы, характерные для всех классов КС-грамматик.
Для ЬЦк)-грамматики при k > 1 совсем не обязательно, чтобы все правые части правил грамматики для каждого нетерминального символа начинались с к различных терминальных символов. Принципы распознавания предложений входного языка такой грамматики накладывают менее жесткие ограничения на правила грамматики, поскольку к соседних символов, по которым однозначно выбирается очередная альтернатива, могут встречаться в нескольких правилах грамматики (эти условия рассмотрены ниже).

ПРИМЕЧАНИЕ

Грамматики, у которых все правые части правил для всех нетерминальных символов начинаются с к 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 ЬЬ(к)-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 детерминированные КС-языки, которые не могут быть заданы LL(k)-грамматикой ни для каких к. Однако LL-грамматики удобны для использования, поскольку позволяют построить линейные распознаватели.
Очевидно, для каждого нетерминального символа ЬЬ(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, в принципе, ЬЦ1)-грамматика допускает в правой части правил цепочки, начинающиеся с нетерминальных символов, а также Х-правила. ЬЬ(1)-грамматики позволяют построить достаточно простой и эффективный распознаватель, поэтому они рассматриваются далее отдельно в соответствующем разделе.

Методы построения распознавателей для LL(k)-грамматик при k > 1 в данной книге не рассматриваются, с ними можно ознакомиться в работах [4, т. 1, 15].

Синтаксический разбор для Щ1)-грамматик

Для построения распознавателей языков, заданных ЬЬ(к)-грамматиками, используются два множества, определяемые следующим образом:

- FIRST(k,a) — множество терминальных цепочек, выводимых из ae(VTuVN)', укороченных до к символов;
- FOLLOW(k,A) — множество укороченных до к символов терминальных цепочек, которые могут следовать непосредственно за AeVN в цепочках вывода.

Формально эти два множества могут быть определены следующим образом:

FIRST(k,a) = {coeVT* | либо |ю| < к и а => * со, либо |ю| > к и а => * сох, xe(VTu VN)'}, ae(VTuVN)', к > 0.

FOLLOW(k,A) = {coeVT* | S => * aAy и coeFIRST(y,k), aeVT*}, AeVN, к > 0.

Эти множества используются не только для построения распознавателей языков, заданных ЬЬ(к)-грамматиками, но и для ряда других классов грамматик, которые будут рассматриваться далее. В случае ЬЦ1)-грамматик используются множества FIRST(l,a) и FOLLOW(l,A).

Для ЬЬ(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 aeVT, обозреваемый считывающей головкой, и символ AeVN, находящийся на верхушке стека.
Эти условия можно сформулировать так:

1. необходимо выбрать в качестве альтернативы правило A -> a, если aeFIRST(l,a);

2. необходимо выбрать в качестве альтернативы правило *A к,* если aeFOLLOW(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) ЬЦ1)-грамматикой, необходимо и достаточно проверить следующее условие: для каждого символа AeVN, для которого в грамматике существует более одного правила вида A a! | a_2 a_n, должно выполняться требование FIRST(l,a,FOLLO'U" (1,A)) n FIRST(l,a,FOLLOW(l,A)) = 0 Vi * j, n > i > 0, n > j > 0. Очевидно, что если для символа AeVN отсутствует правило вида A —» *к,* то согласно этому требованию все множества FIRST^a,), FIRST(l,a_2), ..., FIRST(l,a_n) должны попарно не пересекаться, если же присутствует правило A *к,* то они не должны также пересекаться со множеством FOLLOW(l,A).

ВНИМАНИЕ

Очевидно, что ЬЦ1)-грамматика не может содержать для любого нетерминального символа AeVN двух правил, начинающихся с одного и того же терминального символа.

Условие, накладываемое на правила ЬЦ1)-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 ЬЦ1)-грамматик. Например, приведенная выше грамматика может быть преобразована к виду G'({a}. {S, A}, {S aAA *к* \ S} .S)[1]. В такой форме она уже является ЬЦ1)-грамматикой. Но формального метода преобразовать произвольную КС-грамматику к виду ЬЦ1)-грамматики или убедиться в том, что такое преобразование невозможно, не существует.

СОВЕТ

Для приведения произвольной грамматики к виду ЬЬ(1)-грамматики можно рекомендовать преобразования, рассмотренные выше при знакомстве с методом рекурсивного спуска.

Но применение преобразований не гарантирует, что произвольную КС-грамматику удастся привести к виду ЬЦ1)-грамматики.

[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' /.-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.

Для того чтобы запрограммировать работу МП-автомата, выполняющего разбор входных цепочек символов языка, заданного LL(1)-грамматикой, надо научиться строить множества символов FIRST(l,a) и FOLLOW(l,A). Для множества FIRST(l,a) все очевидно, если цепочка а начинается с терминального символа b (a = b(3, beVT, ae(VTbVN)$^+$, pe(VTuVN)*) - в этом случае FIRST(l,a) - = {b}, если же она начинается с нетерминального символа B (a = Bp, BeVN, ae(VTuVN)$^+$, Pe(VTuVN)*), то FIRST(l,a) = FIRST(1,B). Следовательно, для LL (1)-грамматик остается только найти алгоритм построения множеств FIRST(1,B) и FOLLOW(l,A) для всех нетерминальных символов A,BeVN.
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), AeVN. Для выполнения алгоритма надо предварительно преобразовать исходную грамматику G(VT,VN,P,S) в грамматику G'(VT,VN',P',S'), не содержащую Л-правил. На основании полученной грамматики **G'** и выполняется построение множеств FIRST(1,A) для всех AeVN (если AeVN, то, согласно алгоритму преобразования, также справедливо AeVN'). 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 AeVN: FIRST0(1,A) = {X | A —> XaeP, Xe(VTuVN), ae(VTuVN)*} (первоначально вносим во множество первых символов для каждого нетерминального символа А все символы, стоящие в начале правых частей правил для этого символа А); i := 0.

Шаг 2. Для всех AeVN: $FIRST_{1+1}$(1,A) = FIRST,(l,A)uFIRST,(l,B), для всех нетерминальных символов Be (FIRST,(1,A)nVN).

Шаг 3. Если 3 AeVN: $FIRST_{1+1}$(1,A) * FIRST,(1,A), то i := i + 1 и вернуться к шагу 2, иначе перейти к шагу 4.

Шаг 4. Для всех AeVN: FIRST(1,A) = FIRST,(1,A) \ VN (исключаем из построенных множеств все нетерминальные символы).

Алгоритм построения множества FOLLOW(1,A)

.Алгоритм строит множества FOLLOW(l,A) сразу для всех нетерминальных лшволов грамматики G(VT,VN,P,S), AeVN. Для выполнения алгоритма предварительно надо построить все множества FIRST(1,A), V AeVN. Множества строятся методом последовательного приближения. Алгоритм состоит из нескольких шагов:

Шаг 1. Для всех AeVN: $FOLLOW_0$(1,A) = {X | 3 B -> aAXpe P, BeVN, Xe(VTu VN), a,pe(VTuVN)*} (первоначально вносим во множество последующих символов для каждого нетерминального символа А все символы, которые в правых частях правил встречаются непосредственно за символом А); i := 0.

Шаг 2. $FOLLOW_0(1,S) = FOLLOW_0(1,S) \cup \{/.\}$ (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: $\perp_K$).

Шаг 3. Для всех AeVN: FOLLOW', (1,A) - FOLLOW,(1,A)uFIRST(l,B), для всех нетерминальных символов Be(FOLLOW,(l,A)nVN).

Шаг 4. Для всех AeVN: FOLLOW"; (1,A) = $FOLLOW'_1$(1,A)uFOLLOW$'_1$(l,B) для всех нетерминальных символов Be(FOLLOW',(l,A)nVN), если существует правило B —> X.

Шаг 5. Для всех AeVN: $FOLLOW,_{+1}$(l,A) - $FOLLOW'_1$(1,A)uFOLLOW"$_1$(l,B). для всех нетерминальных символов BeVN, если существует правило B —> aA. ae(VTuVN)*.

Шаг 6. Если 3 AeVN: $FOLLOW_{1+1}$(1,A) * FOLLOW,(1,A), то i := i + 1 и вернуться к шагу 3, иначе перейти к шагу 7.

Шаг 7. Для всех AeVN: FOLLOW(l,A) = FOLLOW,(1,A)\VN (исключаем из построенных множеств все нетерминальные символы).

Пример построения распознавателя для ЬЬ(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 I /EF
E -> (S) | a | Ь

Это нелеворекурсивная грамматика для арифметических выражений (ранее она была построена в разделе «Синтаксические распознаватели с возвратом»).

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 X-правила. В результате получим новую грамматику G'Ct+.-./Aa.bJ.fS.R.T.F.EJ.P' ,S) с правилами:

P':
S —> TR
я *x* | +TR | -TR-
T —> EF
F -» X | *EF | /EF
E -> (S) | a | b

Построенная грамматика G' эквивалентна исходной грамматике G. В этом можно убедиться, если воспользоваться алгоритмом устранения X-правил из раздела «Преобразование КС-грамматик» главы 4. 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 ЬЬ(к)-грамматику для заданного к). Эта грамматика является ЬЬ(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 л-правил.

Построение множества FIRST:

Шаг 1. FIRSTO(S) - {T};
FIRSTO(R) = {+.-}:
FIRSTO(T) = {E}:
FIRSTO(F) - {*./}:
FIRSTO(E) = {(.a.b):
i = 0.

Шаг 2. FIRST1(S) = {T.E}:
FIRST1(R) = {+.-}:
FIRST1(T) = {E.C.a.b};
FIRST1(F) - {*./}.
FIRST1(E) = {(.a.b}.

Шаг 3. i = l, возвращаемся к шагу 2.

Шаг 2. FIRST2CS) - {T.E. (.a.b}.
FIRST2CR) = {+.-};
FIRST2CT) = (E.(.a.b};
FIRST2CF) = {*./}:
FIRST2CE) = {(.a.b}.

Шаг 3- 1=2, возвращаемся к шагу 2.

Шаг 2. FIRST3CS) = {T.E.(.a.b}:
FIRST3CR) - {+,-}:
FIRST3CT) = {E,(.a.b}:
FIRST3C F) = {*,/}:
FIRST3CE) = {(.a.b}.

Наг 3. т=2, переходим к шагу 4.

Лаг 4. FIRST(S) = {(.a.b}:
FIRST(R) = {+.-}:
FIRST(T) = { (.a.b};
FIRST(F) = {*./};
FIRST(E) - {(.a.b};
Построение закончено.

Построение множества FOLLOW:

Шаг 1. FOLLOW0(S) = {)}:
FOLLOW0(R) = 0;
FOLLOW0(T) = {R};
FOLLOW0(F) = 0;
FOLLOW0(E) = {F};
1 = 0.

Шаг 2. FOLLOW0(S) = {) Д}:
FOLLOW0(R) = 0:
FOLLOW0(T) = {R};
FOLLOW0(F) = 0:
FOLLOW0(E) = {F}.

Шаг3. FOLLOW'0(S) = {)Д}:
FOLLOW' OCR) = 0:
FOLLOW'OCT) - {R.+.-}:
FOLLOW'0(F) = 0:
FOLLOW'0(E) = {F,*./}.

Шаг 4. FOLLOW'O(S) = *{).X}:*
FOLLOW'O(R) = 0;
FOLLOW'O(T) = {R.+,-}:
FOLLOW'O(F) = 0;
FOLLOW'O(E) = {F,*,/};

Шаг 5. FOLLOWl(S) = {)Д}.
FOLLOWl(R) = {).M:
FOLLOWl(T) = {R.+.-}:
FOLLOWl(F) = {R.+.-}:
FOLLOWl(E) = {F.*./};

Шаг 6. т=1, возвращаемся к шагу 3.

Шаг3. FOLLOW'1(S) = {) Л}:
FOLLOW' 1 (R) = {)Л}:
FOLLOW'1(T) = {R.+.-}:
FOLLOW'1(F) = {R.+.-}:
FOLLOW'1(E) = {F.*./}:

Шаг 4. FOLLOW 1(S) = {)Д}:
FOLLOW'l(R) = {) Л}:
FOLLOW'l(T) = {R.+.-.)A}:
FOLLOW" 1(F) - {R.+.-.) Л}:
FOLLOW" 1(E) = {F.R.*/.+.-.)Д}:

Шаг 5. FOLLOW2(S) = {)A}:
FOLLOW2(R) = {).*X}.
FOLLOW2(T) = {R+.-.).*.}:
FOLLOW2(F) = {R,+.-.)A}:
FOLLOW2(E) = {F.R.*./.+.-.)Л}:

Шаг 6. 1=2, возвращаемся к шагу 3.

Шаг3. FOLLOW2(S) = {)A}.
FOLLOW 2(R) = {),*.}:
FOLLOW2(T) = {R.+,-.)A};
FOLLOW2(F) = {R.+.-.)A};
FOLLOW2(E) = {F,R.*./.+.-.)A}:

Шаг 4. FOLLOW"2(S) = {)A}:
FOLLOW"2(R) = *{).X}:*
FOLLOW"2(T) - {R.+,-.)A}:
FOLLOW"2(F) = {R.+.-JA}:
FOLLOW"2(E) = {F,R,*,/,+,-,)A};

Шаг 5. FOLLOW3(S) = {)A}:
FOLLOW3(R) = {)A}:
FOLLOW3(T) = {R.+.-.)A}:
FOLLOW3(F) = {R,v,),M;
FOLLOW3(E) = {F.R.*./.+.-.),*X}:*

Шаг 6. т=2, переходим к шагу 7.

Шаг 7. FOLLOW(S) = {)A};
FOLLOW(R) - {)A}:
FOLLOW(T) = {+.-.)A}:
FOLLOW(F) = {+,-.)A}:
FOLLOW(E) = {*,/.+.-.)A};
Построение закончено.

В результате выполнения построений можно увидеть, что необходимое и достаточное условие принадлежности КС-грамматики к классу ЬЬ(1)-грамматик выполняется.

Построенные множества FIRST и FOLLOW можно представить в виде таблицы. Результат выполненных построений отражает табл. 4.1.

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, казанное в его конфигурации, можно опустить, так как оно единственное: (a,Z,y),

де

. — непрочитанная часть входной цепочки символов;

Z — содержимое стека (верхушка стека находится слева);

y — последовательность номеров примененных правил (последовательность дс полняется слева, так как автомат порождает левосторонний вывод).

Таблица 4.1. Множества FIRST и FOLLOW для грамматики G'

Символ AeVN	FIRST(1,A)	FOLLOW(l,A)
S	(a b	И
R	-+ -	)X
T	(a b	
F	* /	
E	(a b	

Примем, что правила в грамматике нумеруются в порядке слева направо и сверху вниз. На основе номеров примененных правил при успешном завершенш-разбора можно построить цепочку вывода и дерево вывода.

В качестве примера возьмем две правильных цепочки символов a+a*Ь и (a+a)*: и две ошибочных цепочки символов a+a* и (+a)*Ь.

Разбор цепочки a+a*Ь:

1. (a+a*b.S.W
2. (a+a*b.TR, 1), так как aeFIRST(1,TR)
3. (a+a*b,EFR, 1.5), так как aeFIRST(l,EF)
4. (a+a*b.aFR.l,5.10), так как aeFIRST(l,a)
5. (+a*b, FR. 1,5.10)
6. (+a*b.R,l,5.10,6), так как +eFOLLOW(1,F)
7. (+a*b.+TR.l,5,10,6.3), так как +eFIRST(l,+TR)
8. (a*b.TR.1.5,10.6.3)
9. (a*b.EFR.1.5.10.6.3.5), так как aeFIRST(l,EF)
10. (a*b.aFR.1.5.10.6,3.5.10), так как aeFIRST(l,a)
11. (*b.FR.l.5.10.6.3.5.10)
12. (*b. *EFR, 1,5.10,6,3.5.10,7), так как *e 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 beFIRST(l,b)
15. a.FR.1.5,10.6.3,5.10.7,11)
16. a.R. 1.5.10.6,3,5,10.7,11,6), так как *.eFOLLOW(1,F)
17. (Я.Д.1.5.10.6,3,5,10.7,11,6.2), так как ^.eFOLLOW(l,R), разбор закончен. Цепочка принимается.

Получили цепочку вывода:

S ⇒ TR ⇒ EHR ⇒ aFR ⇒ 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. 4.5.

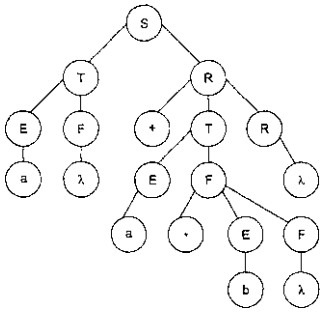


Рис. 4.5. Дерево вывода в LL(1 (-грамматике для цепочки «a+a*b»

Разбор цепочки (a+a)*b:

1. ((a+a)*b.SA)
2. ((a+a)*b,TR. 1), так как (eFIRST(1.TR)
3. ((a+a)*b.EFR.1.5), так как (eFIRST(1,EF)
4. ((a+a)*b.(S)FR.1,5.9), так как (eFIRST(1,(S))
5. (a+a)*b.S)FR,1.5,9)
6. (a+a)*b.TR)FR.1.5.9.1), так как aeFIRST(1.TR)
7. (a+a)*b.EFR)FR,1,5,9,1.5), так как aeFIRST(1,EF)
8. (a+a)*b.aFR)FR,1.5,9,1.5,10), так как aeFIRST(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 +eFOLLOW(1,F)
11. (+a)*b.+TR)FR,1.5,9,1,5.10.6.3), так как +eFIRST(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 aeFIRST(1,EF)
14. (a)*b,aFR)FR, 1.5.9.1.5.10.6.3.5.10), так как aeFIRST(1,a)
15. (q.)*b.FR)FR,1,5,9,1,5,10,6,3,5,10)
    ()*b.R)FR. 1.5.9.1.5.10.6.3,5.10.6), так как)eFOLLOW(1,F)
" O*b,)FR,1.5.9.1.5,10.6.3,5,10,6.2), так как)eFOLLOW(1,R)
- (*b.FR.1.5.9.1.5.10.6.3.5.10.6.2)
? (*b.*EFR.1.5.9.1.5.10.6.3.5.10.6.2.7), так как *eFOLLOW(1,*EF)
30. (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 beFIRST**(l.b)**
22. a.FR. 1.5.9,1,5,10,6,3,5,10,б,2,7.11)
23. (Я..R. 1,5,9.1,5,10,6.3,5,10,б,2,7,11,6), так как Xe FOLLOW(1,F)
24. а д . 1.5.9.1.5,10.б,3,5,10.6,2,7,11.б,2), так как ^.eFOLLOWd.R), разбор закончен. Цепочка принимается.

Получили цепочку вывода:

S ⇒ TR ⇒ EHR ⇒ (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. 4.6.

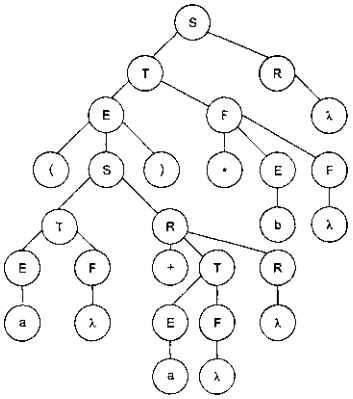


Рис. 4.6. Дерево вывода в Щ1)-грамматике для цепочки «<a+a)*b»

Разбор цепочки a+a*:

1. (a+a*.БД)
2. (a+a*,TR. 1), так как aeFIRST(l,TR)
3. (a+a*.EFR.1.5), так как aeFIRST(l,EF)
4. (a+a*.aFR,1,5,10), так как aeFIRST(l,a)
5. (+a*,FR,l,5,10)
6. (+a*,R,l,5,10,6), так как +eFOLLOW(1,F)
7. (+a*,+TR,l,5,10.6,3), так как +eFIRST(l,+TR)

8. (a*,TR,1.5,10.6,3)
9. Ca*.EFR, 1.5.10.6,3.5), так как aeFIRST(1,EF)
10. (a*.aFR.1.5,10.6.3,5.10), так как aeFIRST(1.a)
11. (*.FR.1.5.10.6.3.5.10)
12. C*.*EFR.l.5.10.6.3.5.10.7), так как *eFIRST(l,*EF)
13. (Я.,EFR.1.5.10,6,3.5.10.7)
14. Ошибка, так как A,eF0LL0W(l,E), но нет правила вида E -> к . Цепочка не принимается.

Разбор цепочки (+a)*Ь:

1. ((+a)*Ь.БД)
2. ((+a)*b.TR.l), так как (eFIRST(1,TR)
3. ((+a)*b.EFR.1.5), так как (eFIRST(1,EF)
4. ((+a)*b,(S)FR. 1.5.9), так как (eFIRST(1,(S))
5. (+a)*b,S)FR,1.5,9)
6. Ошибка, так как нет правил для S вида S -> a таких, чтобы +eFIRST(l,a) и +gF0LL0W(l,S). Цепочка не принимается.

Из рассмотренных примеров видно, что алгоритму разбора цепочек, построенному на основе распознавателя для ЬЦ1)-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 1X(1)-грамматик сужают возможности его применения.

Восходящие распознаватели КС-языков 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.

ЦЗ(к)-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 а выбрать из стека для выполнения свертки;
- какое правило выбрать для выполнения свертки (в том случае, если существует несколько правил вида Aj —> а, A_2 —> а, ... A_n —> а).

Тогда восходящий алгоритм распознавания цепочек КС-языка не требовал бь 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 ЬЦк)-грамматик. Тогда мы получим другой класс КС-грамматик, который носит название LR(k)-грамматик.

КС-грамматика обладает свойством LR(k), к > О, 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 к символов от текущего положения считывающей головки автомата во входной цепочке символов.

Грамматика называется *LR(k)-грамматикой,* если она обладает свойством LR(k) для некоторого к > О[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 работы распознавателя получается правосторонний вывод. Вместо «к» 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 1Л(0)-грамматики, LR(1)-грамматики и другие классы.

В совокупности все LR(k)-граММаТНКН для всех к > О образуют класс LR-грамматик.

На рис. 4.7 схематично показано частичное дерево вывода для некоторой LR(k)-грамматики. В нем <о обозначает уже разобранную часть входной цепочки а, на основе которой построена левая часть дерева у. Правая часть дерева х — это еще не разобранная часть, а А — это нетерминальный символ, к которому на очередном шаге будет свернута цепочка символов z, находящаяся на верхушке стека

[1] Существование Ы(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 к = О 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.

МП-автомата. В эту цепочку уже входит прочитанная, но еще не разобранная часть входной цепочки и. Правая часть дерева х будет построена на основе части входной цепочки т. Свойство LR(k) предполагает, что однозначный выбор действия, выполняемого на каждом шаге алгоритма «сдвиг-свертка», может быть сделан на основе цепочки и и к первых символов цепочки т, являющихся частью входной цепочки а. Этим очередным действием может быть свертка цепочки z к символу А или перенос первого символа из цепочки т и добавление его к цепочке *г*.

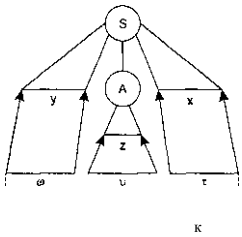


Рис. 4.7. Схема построения дерева вывода для 1_Р(к)-грамматики

Рассмотрев схему построения дерева вывода для ЬК(к)-грамматики на рис. 4.7 и сравнив ее с приведенной выше на рис. 4.4 схемой для ЬЦк)-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 ЬК(к)-грамматики обрабатывается больше информации, чем на шаге работы распознавателя ЬЦк)-грамматики. Действительно, для принятия решения на каждом шаге алгоритма распознавания ЬЦк)-грамматики используются первые к символов из цепочки ит, а для принятия решения на шаге распознавания ЬК(к)-грамматики — вся цепочка и и еще первые к символов из цепочки т. 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]. То есть для каждого КС-языка, заданного LL-грамматикой, может быть построена LR-грамматика, задающая тот же язык, но не наоборот[1].

[1] Говоря о соотношении классов LL-грамматик и LR-грамматик, мы не затрагиваем вопрос о значениях к для этих грамматик. Если для некоторой LL(k)-граММаТНКН всегда существует эквивалентная ей LR-грамматика, то это вовсе не значит, что она будет LR(k)-граММатикой с тем же значением к. Но существуют LR-грамматики, для которых нет эквивалентных им LL-грамматик для всех возможных значений к > 0.

Для ЬК(к)-грамматик известны следующие полезные свойства:

- всякая ЬК(к)-грамматика для любого к > О является однозначной;
- существует алгоритм, позволяющий проверить, является ли заданная грамматика ЫЦк)-грамматикой для строго определенного числа к.

Есть, однако, неразрешимые проблемы для ЬК(к)-грамматик:

- не существует алгоритма, который бы мог проверить, является ли заданная КС-грамматика ЫЦк)-грамматикой для некоторого произвольного числа к;
- не существует алгоритма, который бы мог преобразовать (или доказать, что преобразование невозможно) произвольную КС-грамматику к виду LR(**k)-граМ**матики для некоторого к.

Это общие проблемы, характерные для всех классов КС-грамматик.

Формальное определение ЬК(к)-свойства для КС-грамматик можно найти в [4. т. 1]. 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,VNu{S'},Pu{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 ЬЯ(к)-грамматик работают с пополненными грамматиками.

Распознаватель для ЬК(к)-грамматик функционирует на основе управляющей таблицы Т. Эта таблица состоит из двух частей, называемых «Действия» и «Переходы». 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 к (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 LI1(к)-грамматики использует также два специальных символа ±н и **±к.** Считается, что входная цепочка символов всегда начинается символом ±н и завершается символом **±к.** Тогда в начальном состоянии работы распознавателя символ **±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 ±н, а считывающая головка автомата должна обозревать символ **Хк.**

Алгоритм функционирования распознавателя LK(к)-грамматики можно описать следующим образом:

Шаг 1. Поместить в стек символ ±н и начальную (нулевую) строку управляющей таблицы T. В конец входной цепочки поместить символ **Хк.** 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 ЬК(к)-грамматики является линейным распознавателем.

На практике используются 1Л1(0)-грамматики и LR(1)-грамматики. LR(k)-граМматики со значениями k > 1 практически не применяются. Это вызвано двумя причинами:

1. построение распознавателей для 1Л1(к)-грамматик при k > 1 связано со значительными сложностями, а управляющая таблица T имеет большой объем;
2. доказано, что для любого детерминированного КС-языка может быть построена LR(1)-грамматика, задающая этот язык, поэтому не имеет смысла использовать LR(k)-**граММатнкн** 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 детерминированный КС-язык (это очевидно следует из однозначности всех LR-грамматик), и для любого детерминированного КС-языка можно построить LR(1)-грамматику, задающую этот язык. Второе утверждение уже не столь очевидно, но доказано в теории формальных языков [4, т. 1].

ПРИМЕЧАНИЕ

Класс Ы1(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 детерминированных КС-языков. Но детерминированный КС-язык может быть задан грамматикой, которая не относится к классу 1Л(1)-грамматик. В таком случае совсем не очевидно, что для этого языка удастся построить распознаватель на основе Ы1(1)-грамматики, потому что в общем случае нет алгоритма, который бы позволил 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 (^-грамматики для каждого детерминированного КС-языка играет важную роль — всегда есть смысл попытаться построить такую грамматику.

Синтаксический разбор для 1_П(0)-грамматик

Построение управляющей таблицы для LR(0)-граММаТНК

Простейшим случаем Ы1(к)-грамматик являются LR(())-грамматики. При к = О 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 Т заданной 1^(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 левой части правила A —> *a,* где ае(VTuVN)' специальным символом •, который может стоять в произвольном месте цепочки а: а = у«р (причем любая из цепочек у и р 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 —> aj | a_2 a_n входят во множество правил этой грамматики, то ситуация, содержащая множество правил {S —> S —> •a_2; ... S —> *a_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 А -» у*Вр, где А,BeVN, у,ре(VNuVT)' и при этом правила В -» a_4 | a_2 $a_т$ входят во множество правил грамматики, то правила вида В —> »a_4; В —> «a_2; ... В —> »$a_т$ должны быть добавлены в ситуацию;
- если ситуация содержит правила вида А у«хр, где AeVN, у,ре(VNuVT)*, а х - произвольный символ (хеVNuVT), то из нее может быть построена новая ситуация, содержащая правила вида А —> ух»р, при этом новая ситуация должна проистекать из исходной ситуации на основании символа х.

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 **R„** где i — номер ситуации от 0 до N - 1 (где N — общее количество ситуаций в последовательности). Тогда начальная ситуация будет обозначаться **R_0.**

После построения последовательности ситуаций на ее основе строится управляющая таблица Т. Построение таблицы происходит следующим образом:

- количество строк в таблице **Т** соответствует количеству N ситуаций **R** в последовательности ситуаций, причем каждой строке Т, в таблице Т соответствует ситуация **R,** в последовательности ситуаций;
- для всех ситуаций в последовательности от **R_0** до **$R_{N\_}$,** выполняется следующее:
  - О если ситуация **R,** содержит правило вида S -> у«, ye(VNuVT)*, где S — целевой символ грамматики, то в графе «Действия» строки Т, таблицы Т должно быть записано *«успех»;*
  - О если ситуация **R,** содержит правило вида A у», где AeVN, ye(VNuVT)', А не является целевым символом и грамматика содержит правило вида A у с номером *т,* то в графе «Действия» строки Т, таблицы Т должно быть записано число *т* (*свертка* по правилу с номером *т);*
  - О если ситуация **R,** содержит правила вида A —» у»хр, где AeVN, а х - произвольный символ (xeVNuVT), y,pe(VNuVT)", то в графе «Действия» строки Т, таблицы Т должно быть записано *«сдвиг»]*
  - О если ситуация Rj проистекает из ситуации **R,** и связана с нею через символ х (xeVNuVT), то в графе «Переходы», соответствующей символу х, строки Т, таблицы Т должно быть записано число;';
- клетки таблицы, оставшиеся пустыми после заполнения таблицы Т на основании последовательности ситуаций, соответствуют состоянию *«ошибка».*

Если удается непротиворечивым образом заполнить управляющую таблицу Т на основании последовательности ситуаций по описанным выше правилам, то рассматриваемая грамматика является Ы1(0)-грамматикой. Иначе, когда при заполнении таблицы возникают противоречия, грамматика не является LR**(0)-граМ**-матикой, и для нее нельзя построить распознаватель типа LR(0) [4, т. 1, 13, 50].

Пример построения распознавателя для 1.Н(0)-грамматики

Рассмотрим КС-грамматику GC {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 R0 будет содержать правило S' »S, но в нее также должны быть включены правила S -» *aSS и S «Ь, поскольку в грамматике для символа S есть правила S —> aSS | Ь. Получим ситуацию R0 = {S' —> »S,S —> «aSS.S —> *b}.
Из начальной ситуации R0 по символу S можно получить ситуацию Rl, содержащую правило S' S». Других правил в нее не может быть добавлено, поэтому получаем Rl = {S' —> S»}.

Из начальной ситуации R0 по символу a можно получить ситуацию R2, содержащую правило S ->a *SS, но в нее также должны быть включены правила S -» «aSS и S *Ь, поскольку в грамматике для символа S есть правила S —> aSS | Ь. Получаем R2 = {S —> a»SS.S »aSS,S -> »b}.

Из начальной ситуации R0 по символу b можно получить ситуацию R3, содержащую правило S —> Ь«. Других правил в нее не может быть добавлено, поэтому получаем R3 = {S —> Ь«}.

Из ситуации R2 по символу S можно получить ситуацию R4, 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 | Ь. Получаем R4 = = {S -> aS-S.S -> -aSS.S -> -b}.

Из ситуации R4 по символу S можно получить ситуацию R5, содержащую правило S -» aSS*. Других правил в нее не может быть добавлено, поэтому получаем R5 = {S —> aSS*}.

Из ситуации R2 по символу a можно получить ситуацию R6, содержащую правило S ->a *SS, но в нее также должны быть включены правила S «aSS и S «Ь, поскольку в грамматике для символа S есть правила S -» aSS | Ь. Получаем R6 = = {S —>a *SS, S —> »aSS, S —> »b}. Можно заметить, что эта ситуация совпадает с ситуацией R2: R6 = R2.

Из ситуации R2 по символу b можно получить ситуацию R7, содержащую правило S ->Ь •. Других правил в нее не может быть добавлено, поэтому получаем R7 = = {S -»Ь •}. Можно заметить, что эта ситуация совпадает с ситуацией R3: R7 = R3.
Из ситуации R4 по символу a можно получить ситуацию R8, содержащую правило S ->a »SS, но в нее также должны быть включены правила S -» «aSS и S -> «Ь, поскольку в грамматике для символа S есть правила S —> aSS | Ь. Получаем R8 = = {S ->a »SS,S -> «aSS.S -> »b}. Можно заметить, что эта ситуация совпадает с ситуацией R2: R8= R2.

Из ситуации R4 по символу b можно получить ситуацию R9, содержащую правило S -»Ь •. Других правил в нее не может быть добавлено, поэтому получаем R9 = = {S —»Ь •}. Можно заметить, что эта ситуация совпадает с ситуацией R3: R9 = R3.
Всего получилось 6 различных ситуаций от R0 до R5. Граф взаимосвязи ситуаций показан на рис. 4.8.

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 ЬИ(0)-грамматикой. Управляющая таблица для нее приведена в табл. 4.2.

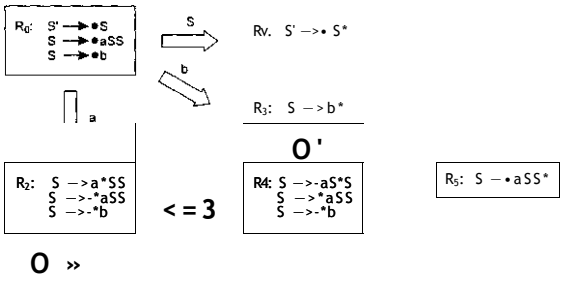


Рис. 4.8. Пример графа взаимосвязи ситуаций для 1_P(0)-грамматики

Таблица 4.2. Пример управляющей таблицы для 1.H(0)-грамматики

№	Стек	Действия	Переходы		
			S	a	Ь
0	J„	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 компонентов: 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_aьaьaьЬ:

1. (abababblK.{J-н.O}Д)
2. (bababbi-K. {_Lh.0}{a.2} .X)

3. (ababbJ.K, {J_h ,0}{a ,2}{b,3} .*X)*
4. (ababbIK. {1H.0}{a,2}{S,4},3)
5. (babbJ-K, {J.H ,0}{a ,2}{S,4}{a ,2} ,3)
6. (abbi-K, {J-H.0}{a,2}{S.4}{a,2}{b,3},3)
7. (abbJ-к, {J_H,0}{a,2}{S,4}{a,2}{S,4}.3,3)
8. (bbJ.K.{J-H,o}{a,2}{S,4}{a,2}{S,4}{a,2},3,3)
9. (bJ_K. {_Lh,0}{a,2}{S,4}{a,2}{S,4}{a,2}{b,3} ,3,3)
10. (b±K.{±H,0}{a,2}{S,4}{a.2}{S,4}{a.2}{S.4},3.3.3)
11. (J-K, {J.H,0}{a,2}{S,4}{a,2}{S,4}{a,2}{S,4}{b,3},3,3,3)
12. (J-K, {J-H,0}{a,2}{S.4}{a.2}{S.4}{a,2}{S,4}{S,5}.3.3.3,3)
13. (J-K, {J-H,0}{a,2}{S.4}{a,2}{S,4}{S,5}.3,3,3,3,2)
14. (J-K, {J-H,0}{a,2}{S,4}{S,5} .3,3,3,3,2.2)
15. (J-K.{J-H,0}{S.1}.3.3.3.3,2,2.2)
16. (1k,{1h,o}{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±K,{±H,0} *,X)*
2. (aЬЬЫк,{1н,0}{a.2}Л)
3. (ЬЬЫк. {IH.O}{a,2}{a,2} Л)
4. (bblK,{±H,0}{a,2}{a,2}{b,3} *,X)*
5. (bb±K.{±H.0}{a.2}{a.2}{S,4},3)
6. (bJ-K,{J-H,0}{a,2}{a,2}{S,4}{b,3},3)
7. (bJ-к, {J-H,0}{a ,2}{a,2}{S,4}{S.5} ,3.3)
8. (bJ-к,{J-H,0}{a,2}{S,4},3.3,2)
9. (±K,{±H,0}{a,2}{S,4}{b,3},3,3,2)
10. (J-K, {J-H,0}{a.2}{S,4}{S,5}.3,3,2,3)
11. (J-K, {J-H.o}{S, 1} .3,3.2.3,2)
12. (J-K, {_Lh,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. (aabbJ-к. {J.H.0} Д)
2. (abbJ-K, {JH ,0} {a ,2} Д)
3. (bblK.{lH.0}{a.2}{a,2}.*X)*
4. (bJ-K,{J-H,0}{a,2}{a.2}{b,3}Д)

5. (blK.{lH.0}{a,2}{a.2}{S.4}.3)
6. (_Lk.{_Lh,0}{a,2}{a,2}{S,4}{b,3} .3)
7. (J-K,{J_H,0}{a,2}{a,2}{S,4}{S,5},3,3)
8. (lK,{lH,0}{a,2}{S,4},3,3,2)
9. ошибка, невозможно выполнить сдвиг.

Распознаватель для ЬИ(0)-грамматики достаточно прост. Приведенный выше пример можно сравнить с методом рекурсивного спуска или с распознавателем для LL(1)-грамматики — оба эти 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 1_Я(0)-грамматик

Управляющая таблица Т для LR(0)-грамматики строится на основании понятия «левых контекстов» нетерминальных символов: после выполнения свертки для нетерминального символа А в стеке МП-автомата ниже этого символа будут располагаться только те символы, которые могут встречаться в цепочке вывода слева от А. Эти символы и составляют «левый контекст» для А. Поскольку выбор между сдвигом или сверткой, а также между типом свертки в Ы1(0)-грамматиках выполняется только на основании содержимого стека, то LR(0)-граММаТНКа должна допускать однозначный выбор на основе левого контекста для каждого символа [4, т. 2].

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 | *к) ,S).* Пополненная грамматика для нее будет иметь вид G'({a,b}, {S.S'}. {S' -» S.S SaSb | Л.}.S'). Начнем построение последовательности ситуаций для грамматики G'.

Начальная ситуация R0 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 -> SaSbj Я. (в правиле S • пустое место соответствует пустой цепочке *X,* и это правило в равной степени можно рассматривать как S —> *"X* или S -> X»). Получим ситуацию R0 = {S" -» «S.S -> «SaSb.S -> •}.

При построении управляющей таблицы Т для этой грамматики уже на шаге для ситуации R0 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 *X* (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 Х-правила, не может быть LR(0)-грамматикой.

Можно попытаться выполнить преобразования (например, исключить ^-правила), чтобы привести грамматику к виду ЬЯ(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 ЬК(1)-грамматики'.

Синтаксический разбор для 1_П(1)-грамматик

Построение управляющей таблицы для LR(1)-грамматики

Как и для 1Л(0)-грамматик, управляющую таблицу Т для 1Л(1)-грамматик можно построить на основании последовательности ситуаций [4, т. 2]. Однако ситуация для LR(1)-грамматики имеет более сложную форму, чем для L R (0) - г р а м м а т и к и. Правило в ситуации для LR(1)-грамматики должно иметь вид: А -> у«р/а, где AeVN, у,ре(VNuVT)*, aeVT. Символ •, а также цепочки у и р имеют тот же смысл, что и для LR(0)-граММаТНК, а символ ае VT определяет правый контекст. Начальная ситуация в последовательности ситуаций для 1Л(1)-грамматики должна содержать правила {S -» «оц/Хк, S -> «о^/Ik, ... S -> «ап/Хк}, если правила S -> оц | $a_2$1...| a_n входят в множество правил этой грамматики, а символ S является целевым символом. Символ $\pm_к$ обозначает символ конца входной цепочки.

Правила построения последовательности ситуаций для LR(1)-грамматики следующие:

- если ситуация содержит правило вида А -» у»ВЬр/а, где A,BeVN, a,beVT, у,ре(VNuVT)*, и при этом правила В -> a_1 | a_2 $a_т$ входят во множество правил грамматики, то правила вида В -> *a_1/Ь; В «a_2/Ь;... В -> *a_m/b должны быть добавлены в ситуацию;
- если ситуация содержит правило вида А -> у«В/а, где A,BeVN, aeVT, уе(VNu VT)*, и при этом правила В a! | $a_2$1...| $a_т$ входят во множество правил грамматики, то правила вида В —> «оц/а; В —> *a_2/а;... В —> *$a_т$/а должны быть добавлены в ситуацию;
- если ситуация содержит правило вида А у»ВСр/а, где A,B,CeVN, a,beVT, у,ре(VNuVT)*, и при этом правила В a_1 | a_2 $a_т$ входят во множество правил грамматики, тогда VceVT, таких, что ceFIRST(l,C), правила вида В *a_1/с; В «a_2/с; ... В «$a_т$/с должны быть добавлены в ситуацию;

[1] Как уже было сказано выше, распознаватели для ЬК(к)-грамматик с k > 1 практически не используются.

- если ситуация содержит правила вида A -> у«хр/а, где AeVN, aeVT, а х - произвольный символ (xeVNuVT), у,ре(VNuVT)*, то из нее может быть построена новая ситуация, содержащая правила вида A -> ух«р/а, при этом нова? ситуация должна проистекать из исходной ситуации на основании символа х

Здесь FIRST(1,C) обозначает множество первых символов для нетерминального символа CeVN. 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, то общее количество возможных ситуаций в LR(^-грамматике будет существенно больше, чем количество ситуаций в LR(0)-грамматике Управляющая таблица Т для распознавателя на основе LR(1)-грамматики строится на основе последовательности ситуаций так же, как и для 1^(0)-грамматики, с учетом того, что графа «Действия» в этой таблице разбита на несколько подграф пс количеству возможных терминальных символов, каждая из которых соответствует определенному символу правого контекста. Для всех ситуаций в последовательности от $\mathbf{R_0}$ до $\mathbf{R_{N1}}$ С учетом правого контекста выполняются следующие действия:

- если ситуация R, содержит правило вида S -» у«/а, уе(VNuVT)*, aeVT, где S — целевой символ грамматики, то в графе «Действия» строки Т, таблицы Т для символа *а* должно быть записано *«успех»;*
- если ситуация **R,** содержит правило вида A -> у*/а, где AeVN, уе(VNuVT)' aeVT, А не является целевым символом и грамматика содержит правило вида A у с номером *т,* то в графе «Действия» строки Т, таблицы Т для символа *а* должно быть записано число *т* (*свертка* по правилу с номером *т);*
- если ситуация **R,** содержит правила вида A -> у«хр/а, где AeVN, aeVT, а х - произвольный символ (xeVNuVT), у,ре(VNuVT)*, то в графе «Действия» строки Т, таблицы Т для символа *а* должно быть записано *«сдвиг*
- если ситуация **Rj** проистекает из ситуации **R,** и связана с нею через символ х (xeVNuVT), то в графе «Переходы», соответствующей символу х, строки Т таблицы Т должно быть записано число *j.*

Если удается непротиворечивым образом заполнить управляющую таблицу Т на основании последовательности ситуаций по описанным выше правилам, 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].

Пример построения распознавателя для 1_Я(1)-грамматики

Возьмем КС-грамматику G({a.b}, {S}, {S -» SaSb | /.}.S), которая уже рассматривалась выше. Пополненная грамматика для нее будет иметь вид G' ({a, b}, {S. S'}. {S' -> S.S SaSb | X}.S").

Построим последовательность ситуаций для грамматики G'.

Начальная ситуация R0 будет содержать правило S' »S/±k, но в нее также должны быть включены правила S —> «SaSb/J-К и S —> */Лк, поскольку в грамматике для символа S есть правила S —> SaSb | X, а затем на основании правила S ->

•SaSb/Xк еще и правила S -» »SaSb/a и S—>*/a. Получим ситуацию R0={S' -» ·S/_LK,S •SaSb/J_K,S «/XK.S -> «SaSb/a.S -> »/a}.

Из начальной ситуации R0 по символу S можно получить ситуацию Rl, содержащую правила S' —> S*/XK, S —> S»aSb/XK и S —> S*aSb/a. Других правил в нее не может быть добавлено, поэтому получаем RL = {S" -» S»/XK,S ->'S«aSb/XK,S -» S*aSb/a}. Из ситуации Rl по символу а можно получить ситуацию R2, содержащую правила S ->Sa «Sb/Xк 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 | Я, а затем на основании правила S • SaSb/b еще и правила S -» *SaSb/a и S -> «/a. Получим ситуацию R2 = {S —> Sa«Sb/XK,S Sa«Sb/a,S «SaSb/b.S «/b.S •SaSb/a,S -» «/a}.

Продолжив построения, получим 8 различных ситуаций от R0 до R7, граф взаимосвязи которых показан на рис. 4.9.

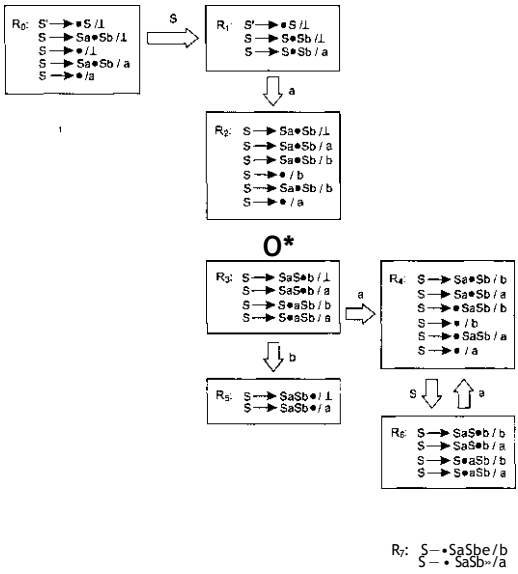


Рис. 4.9. Пример графа взаимосвязи ситуаций для LR(1)-грамматики

На основе найденной последовательности ситуаций можно построить управляющую таблицу распознавателя для ЬК(1)-грамматики. Поскольку при построена управляющей таблицы не возникает противоречий, то рассмотренная грамматик^ 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	Дк	a	Ь	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 соответствуют 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 1Л1(0)-грамматик.
Разбор цепочки abababb:

1. (abababblK,{1н,0}Д)
2. (abababb±**K,{±H**.0}{**S,I**}.3)
3. (bababblK.{J.**H**,0}{S,l}{a,2}.3)
4. (bababblK.**{±H**.0}{S,l}{a.2}{S,3},3,3)
5. (ababbi.**K,**{J_H,0}{S,l}{a.2}{S,3}{b**,5}**,3,3)
6. (ababb_LK.**{±H**,0}{S,l}**}**,**3.3**.2)
7. (babbJ-**K,{±H**,0}{S.**L**}{a,2},3,3,2)
8. (babb±**K.{±H**,0}{S**,I**}{a.**2**}{S,3},3.3,2.3)
9. (abblK, {J.**H**,0}{S.l}{a.2}{S,3}{b,5} ,3,3,2.3)
10. (abb_LK. {J_h.0}{S. 1} ,3,3.2.3.2)
11. (bbi.**K,** {-LH,0}{S,l}{a,2} ,3,3,2,3.2)
12. (bb-LK,{_LH,0}{S,1}{a,2}{S,3},3.3.2,3,2.3)
13. (b-LK,{.LH,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. (aaЬaЬЫк,{1н.0}Л)
2. (aaЬaЬЫк,{±H.0}{S, 1},3)
3. (ababblK.{J_H,0}{S.l}{a,2},3)
4. (ababb-LK, {_LH,0}{S, 1}{a,2}{S,3} ,3,3)
5. (babb±K,{lH,0}{S,l}{a.2}{S,3}{a.4}.3,3)
6. (babblK,{±H,0}{S,l}{a,2}{S,3}{a,4}{S,6},3,3,3)
7. (abb±K,{LH,0}{S,l}{a,2}{S,3}{a,4}{S,6}{b,7},3,3,3)
8. (abb-LK, {±н.0}{S.l}{a,2}{S,3} ,3,3,3,2)
9. (bblK.{lH.-0}{S.l}{a,2}{S,3}{a.4},3,3,3,2)
10. (bblK,{±H,0}{S,l}{a,2}{S,3}{a,4}{S,6},3,3,3,2,3)
11. (b±K, {XH,0}[S, 1][A,2]{S,3}{a,4][S,6}{b,7] .3,3,3,2,3)
12. (Ык, {J.H,0}{S,l}{a,2}{S,3} .3,3,3,2,3,2)
13. (lK,{lH,0}{S,l}{a,2}{S,3}{b,5}.3,3,3,2,3,2)
14. (J_K.(J_H,0]{S. 1] .3.3,3,2,3,2,2)
15. (±K,[1H,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(^-грамматикой. И если язык задан грамматикой, не являющейся 1^(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(l)-граММаТНК можно с уверенностью сказать, что если не удалось преобразовать произвольную КС-грамматику к виду LR(l)-граММаТНКН, то нет большого смысла пытаться преобразовать ее к виду 1^(к)-грамматики с k > 1.

Дело в том, что даже для LR(l)-граММаТНК 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 1^(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)-граММаТНК при k > 1 будет увеличиваться с ростом к в геометрической прогрессии пропорционально количеству терминальных символов грамматики. Поэтому 1Л(к)-грамматики с k > 1 практического применения не имеют.

Например, неоднократно рассмотренная выше грамматика арифметических выражений для символов а и Ь 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) 1 a | Ь
```

является LR(1)-грамматикой, но последовательность ситуаций для нее слишком велика, чтобы ее можно было проиллюстрировать в данном учебном пособии (желающие могут построить последовательность ситуаций и управляющую таблиц} 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)-гра\iMaTHKit 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(l)-граММаТНКН, но многие из них очень удобны и практичны для построения лексических анализаторов.

Далее в этой главе будут рассмотрены еще два класса КС-грамматик — SLR (1)-грамматики и ЬАЬИ(1)-грамматики, являющиеся незначительной модификацией LR(l)-граММаТНК. Отдельная глава посвящена грамматикам предшествования. Остальные грамматики из всего множества классов КС-грамматик в данном учебном пособии не рассматриваются, их можно найти в [4, т. 2].

SLR(1)- и LALR(1)-грамматики

Распознаватели для обоих этих классов грамматик представляют собой модификацию распознавателя для LR(l)-граММаТНК.

Синтаксический разбор для SLR(1)-грамматик

Идея распознавателей для SLR(k)-граММаТНК заключается в том, чтобы сократить множество рассматриваемых ситуаций и упростить построение управляющей таблицы Т. Ведь увеличение количества рассматриваемых ситуаций в 1Л(1)-грамматике по сравнению с 1^(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 Т.

Все SLR(k)-граММагаКН для всех k > 1 образуют класс SLR-грамматик (k > 1. поскольку SLR(0)-граММамКа совпадает по определению с 1Л(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)-граММаТНК построение управляющей таблицы Т выполняется следующим образом:

- строится последовательность ситуаций, соответствующая ЬК(0)-грамматике;
- на основе построенной последовательности ситуаций заполняется управляющая таблица Т, причем если при заполнении таблицы возникают конфликты, то для их разрешения используются правый контекст длиной к и множества символов FIRST(k,a) и FOLLOW(k,A) для правил вида А-»а.

Здесь множества FIRST(k,a) и FOLLOW(k,A) представляют собой соответственно множество к первых терминальных символов цепочки вывода из а и к первых терминальных символов, которые могут следовать в цепочках вывода за символом А. Эти множества были определены выше в разделе, посвященном LL(1)-грамматикам.

На практике чаще всего применяются SLR(l)-граММаТНКН, поскольку для них построение множеств символов FIRST(1,a) и FOLLOW(1,A) выполняется элементарно просто.

Управляющая таблица Т для распознавателя на основе 81Л(1)-грамматики строится на основе последовательности ситуаций так же, как и для LR(0)-граММаТНки, но графа «Действия» в этой таблице разбита на несколько подграф по количеству возможных терминальных символов, аналогично LR(l)-граММаТНКе. Для всех ситуаций в последовательности от $\mathbf{R_0}$ до $\mathbf{R_{M1}}$ с учетом правого контекста выполняются следующие действия:

- если ситуация R, содержит правило вида S —» у», уе(VNuVT)', где S — целевой символ грамматики, то в графе «Действия» строки Т, таблицы Т должно быть записано *«успех»* для всех символов aeFOLLOW(1,S) — фактически, таким символом является символ конца строки $\underline{I}_K$;
- если ситуация R, содержит правило вида А -» у», где AeVN, уе(VNuVT)*, А не является целевым символом и грамматика содержит правило вида А-»у с номером *т,* то в графе «Действия» строки Т, таблицы Т должно быть записано число *т (свертка* по правилу с номером *т)* для всех символов aeFOLLOW(l,A);
- если ситуация R, содержит правила вида А -» у*хр, где AeVN, а х - произвольный символ (хеVNuVT), у,ре(VNuVT)*, то в графе «Действия» строки Т, таблицы Т должно быть записано *«сдвиг»* для всех символов aeFIRST(l.x)[1];
- если ситуация Rj проистекает из ситуации **R,** и связана с нею через символ х (хеVNuVT), то в графе «Переходы», соответствующей символу х, строки Т, таблицы Т должно быть записано число *j.*

Если удается непротиворечивым образом заполнить управляющую таблицу Т на основании последовательности ситуаций по описанным выше правилам, то рассматриваемая грамматика является SLR(1)-грамматикой. Иначе, когда при заполнении таблицы возникают противоречия, грамматика не является SLR(^-грамматикой и для нее нельзя построить распознаватель типа SLR(l) [4, т. 2].

[1] Напоминаем, что для терминального символа b справедливо FIRST(1,b) = {b}.

Например, рассмотрим грамматику арифметических выражений для символе; a и b G({+.-,/.*.a,b}.{S",S,T.E},P,S'):

P:
S" -> S
S -> S+T I S-T I T
T -> T*E | T/E I E
E -> (S) | a | b

Если начать строить множество ситуаций ЬК(0)-грамматики для этой грамматики, то начальная ситуация будет выглядеть так: R0 = {S" -» *S,S *S+T,S — •S-T.S »T.T *T*E,T «T/E.T -> »E.E »(S).E «a.E «Ь}. Из нее г. символу S можно получить ситуацию Rl = {S' -» S».S -» S»+T. S -•• S*-T}, а п: символу T — ситуацию R2 = {S -> T».T T**E,T T*/E}.

При построении управляющей таблицы T для этой грамматики на основе алгорит • ма для ЫЗ1(0)-грамматик в состояниях R1 и R2 возникнут противоречия, поэтом данная грамматика не является ЬК(0)-грамматикой. Однако если воспользоваться алгоритмом для SLR(l)-граММаТМОi, то противоречий удается избежать:

- для ситуации R1, приняв во внимание, что FOLLOWdS') = {±к}, а FIRST(1.+) = и FIRSTQ.-) = {-}, получим, что для правого контекста Хк («конец строки» необходимо сигнализировать об успехе, а для правых контекстов + и — выполнять сдвиг (остальные варианты правого контекста в этой ситуации предполагают ошибку);
- для ситуации R2 найдем FOLLOW(lS), получим FOLLOWdS) = {+,-.)}, FIRST(1,*) = = {*} и FIRSTd,/) = {/}, 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, чтс рассмотренная грамматика является SLR(l)-граММаТiiKoft.

Ограничения SLR(1)-грaММaniK

Языки, заданные SLR(l)-граММаТНКаМН, представляют собой более широкий класс КС-языков, чем языки, заданные 1^(0)-грамматйками.
С другой стороны, SLR(1)-грамматики позволяют сократить объем управляющей таблицы T распознавателя за счет того, что количество строк в ней соответствует количеству состояний ЬИ(0)-грамматики, которое зачастую существенно меньше, чем количество состояний LR(l)-граММаТНКН. В то же время они используют более простой алгоритм исключения конфликтных ситуаций при построении этой таблицы, чем LR(l)-граММаТНКН. В этом их преимущество перед LR(1)-грамматиками.

Однако не всякая 1Л1(1)-грамматика является SLR(l)-гра\iМаТНКоft. Более того, оказывается, что класс языков, заданных 51Л1(1)-грамматиками уже, чем класс языков, заданных LR(1)-грамматиками. Это значит, что он уже, чем класс детерминированных КС-языков.

ВНИМАНИЕ

Не всякий детерминированный КС-язык может быть задан SLR(1)-грамматикой.

Возьмем рассмотренную ранее КС-грамматику G({a ,b}. {S}, {S -» SaSb | A.}.S) с пополненной грамматикой G'({a,b},{S,S'},{S' —> S.S —> SaSb | A.J.S'). Как было показано выше, начальная ситуация для этой грамматики имеет вид R0 = {S" -» *S.S -> *SaSb,S •}. При построении управляющей таблицы Т на основе алгоритма для 1Л(0)-грамматики возникают противоречия. Если попытаться воспользоваться алгоритмом для 8Ы1(1)-грамматики, то необходимо вычислить множества FIRST(l.S) и FOLLOW(l.S). Получим FIRST(l.S) = {a} и FOLLOW(lS) = {a.b}. Оказывается, что для правого контекста а конфликта избежать не удается: с одной стороны, в этой ситуации для него необходимо выполнять сдвиг, поскольку S «SaSb входит в ситуацию и aeFIRST(l.S), но, с другой стороны, для него надо выполнять свертку по правилу S -» *X,* поскольку S -» • входит в ситуацию, и aeF0LL0W(l,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 — ЬАЬК(1)-грамматики.

Синтаксический разбор для LALR(1)-грамматик

Идея распознавателей для LALR(k)-граММаТНК та же самая, что и для SLR(k)-грамматик: сократить множество рассматриваемых ситуаций до множества ситуаций 1^(0)-грамматики и упростить построение управляющей таблицы Т, но при этом найти алгоритм разрешения конфликтов при построении для ситуаций, в которых эти конфликты возникают, на основании правого контекста длиной к. Все LALR(k)-граММаТНКН для всех к > 1 образуют класс LALR-грамматик (k > 1, поскольку LALR(0)-граММаТНКа совпадает по определению с LR(0)-граММаТНкой). 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(l)-граММаТНКН, так как применяемый для них метод разрешения конфликтов достаточно сложно распространить на LALR(k)-**граММагаКН** с k > 1.

В распознавателях на основе LALR(l)-граММаТНК построение управляющей таблицы Т выполняется так же, как и для SLR(l)-граММаТНК, но разрешение конфликтов выполняется не на основании множества FOLLOW(l,A), а путем анализа ipanoro 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 Т записывается действие *«свертка»* на основании конфликтного правила. Действие *«сдвиг»* записывается для всех остальных символов правого контекста, которые входят в множество FIRST(1,p) правил зида А -» а*р и не допускают выполнение свертки (более подробное описание анализа правого контекста для LALR(l)-граММаТНК можно найти в [4, т. 2]).

Возьмем КС-грамматику G({a,b}, {S} ,{S ->• SaSb | .S) с пополненной грамматикой G'({a.b},{S,S'}.{S" S.S SaSb | A.}.S').

Построим для нее последовательность ситуаций:

R0 = {S' ->.S.S -> -SaSb.S •}:
R1 = {S* -> S«,S-> S*aSb} (порождена из RO по символу S):
R2 = {S -> Sa»Sb,S -> «SaSb.S -> •} (порождена из R1 и из R3 по символу a):
R3 = {S -> SaS»b. S -> S»aSb} (порождена из R2 по символу S).
R4 = {S -> SaSb*} (порождена из R3 по символу Ь).

Конфликты присутствуют в ситуациях R1 и R2.

Рассмотрим ситуацию R1. В ней возможна свертка по правилу S' -> S* или сдви в соответствии с правилом S -» S*aSb. Если проанализировать правило S' S* то видно, что оно появилось в ситуации R1 из правила S' »S ситуации R0, следовательно, правым контекстом для этого правила может быть только символ 1- Тогда для ситуации R1 имеем, что для правого контекста ±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 а - сдвиг (FIRST(l.aSb) = {a}).

Аналогично, для ситуации R2 возможна свертка по правилу S • или сдвиг в соответствии с правилами S -» Sa *Sb и S ->• «SaSb. Если проанализировать правиле S —> •, то видно, что оно могло появиться в ситуации R2 из правила S —> Sa »S: этой же ситуации — тогда правым контекстом для него должен быть символ г но это же правило могло появиться в ситуации R2 также на основании правила S -» «SaSb из этой же ситуации — тогда правым контекстом для него будет символ a. Получаем, что в ситуации R2 для правых контекстов а и b должна выполняться свертка по правилу S -> *X.* Сдвиг в этой ситуации выполняться не должен, поскольку FIRST(l.Sb) = FIRST(l.SaSb) = {a.b}, но для символов а и b уже выполняется свертка.

Все конфликтные ситуации удалось разрешить. Таким образом, данная грамматика является LALR(l)-граММаТНКоfl.

Построим управляющую таблицу для данной грамматики, используя найденнук последовательность ситуаций и правила разрешения конфликтов, полученные для ситуаций R1 и R2.

Таблица 4,4. Пример управляющей таблицы для LALR(1)-грамматики

№	Стек	Действия			Переходы		
			b	±к	a	b	S
0	-Iм	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			

Как и в ранее построенных таблицах, колонка «Стек» не нужна распознавателю, а введена для пояснения состояния стека. Пустые клетки в таблице соответству-

ют состоянию *«ошибка».* Правила грамматики пронумерованы от 1 до 3. Колонка «Действия» в таблице содержит перечень действий, соответствующих текущему входному символу, обозреваемому считывающей головкой автомата.

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 abababb:

1. (abababb±K.{±H.O}*X)*
2. (abababbln. {1h,0}{S,ı} ,3)
3. (bababblK.{lH.0}{S.l}{a,2}.3)
4. (bababblK.{1h.0}{S,1}{a.2}{S.3}.3.3)
5. (ababblK,{lH,0}{S.l}{a.2}{S.3}{b,4}.3.3)
6. (ababblK.{1h.0}{S, 1}.3.3,2)
7. (babblK.{lH,0}{S.l}{a,2}.3.3,2)
8. (babblK.{1h.0}{S,1}{a,2}{S.3}.3,3.2,3)
9. (abblK.{lH.0}{S.l}{a.2}{S,3}{b,4}.3.3,2.3)
10. (abblK. {1h.0}{S. 1} .3,3.2,3,2)
11. (bblK.{lH,0}{S.l}{a.2}.3.3,2,3.2)
12. (ЬЬ1к.{1h.0}{S,1}{a.2}{S.3}.3,3,2,3,2.3)
13. (Ь1к,{lH.0}{S,l}{a,2}{S.3}{b.4},3,3,2.3.2.3)
14. (blK,{lH.0}{S.l}.3.3.2,3,2.3,2)
15. **ошибка, нет действий для b в строке 1.**

Разбор цепочки aababb:

1. (aababblK,{1н,0}.A.)
2. (aababblK, {1h.0}{S,1} .3)
3. (ababblK,{lH.0}{S,l}{a,2}.3)
4. (ababbJLK. {JLh.0}{S.1}{a,2}{S.3}.3.3)
5. (babblK.{lH,0}{S,l}{a,2}{S.3}{a,2}.3.3)
6. (babblK. {lH,0}{S,l}{a.2}{S.3}{a.2}{S.3}.3,3,3)
7. (abblK.{lH.0}{S.l}{a,2}{S.3}{a.2}{S.3}{b,4}.3.3,3)
8. (abblK, {lH,0}{S.l}{a,2}{S.3}.3.3.3.2)
9. (ЬЬ1к.{lH,0}{S,l}{a.2}{S.3}{a,2}.3.3,3,2)
10. (ЬЬ1к, {1h,0}{S.l}{a,2}{S,3}{a,2}{S,3}.3.3,3,2,3)
11. (blK,{lH,0}{S,l}{a.2}{S,3}{a.2}{S,3}{b,4},3,3,3.2.3)
12. (Ь1к,(1h,0}{S,1}{a.2}{S.3}.3,3,3,2,3,2)
13. (Ik,{1h,0}{S,l}{a,2}{S.3}{b,4},3,3,3,2,3,2)
14. (1K.{1H.0}{S,1},3.3,3.2,3,2,2)
15. (JLk. {JLh.o}{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 ЬАЫ1(1)-грамматики, соответствует выводу, найденному распознавателем на основе LR(1)-грайматики.

Особенности иЫ.Н(1)-грамматик

Как и SLR(l)-граММаТНКН, ЬА1Л(1)-грамматики позволяют сократить объем управляющей таблицы Т распознавателя за счет того, что количество строк в ней. соответствует количеству состояний 1Л(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 ЬАЬК(1)-грамматики проще, чем распознаватель на основе LR(^-грамматики, но немного сложнее, чем распознаватель на основе SLR(l)-граММаraKii Класс LALR(l)-граММаraK шире, чем класс SLR(l)-граММаraK. Например, рассмотренная выше пополненная КС-грамматика G' ({a.b}.{S.S'},{S' —> S.S -> SaSb | *X)* S') является LALR(l)-грамматикой, но не является SLR(l)-граММаraKoft. Однакс важно, что и языки, заданные ЬА1^(1)-грамматиками, представляют собой более широкий класс КС-языков, чем языки, заданные 8Ы1(1)-грамматиками.

ПРИМЕЧАНИЕ

Всякая SLR(l)-**граММаraKa** является ЬАЬ11(1)-грамматикой, но не наоборот.

Доказано, что любой детерминированный КС-язык может быть задан LALR(l)-грамматикой, а это значит, что класс языков, заданных ЬАЫ1(1)-грамматиками. совпадает с классом языков, заданных 1Л(1)-грамматиками [4, т. 2]. Хотя известно, что класс LR(l)-граММаТНК шире, чем класс ЬА1Л(1)-грамматик.

ВНИМАНИЕ

Всякий язык, заданный Ь11(1)-грамматикой, может быть задан LALR(^-грамматикой, но не всякая ЬЯ(1)-грамматика является ЬАЬ11(1)-грамматикой.

LALR(l)-граММаТНКН представляют собой более удобный инструмент для построения распознавателей, чем LR(l)-граММаraKH. Кроме того, этот класс грамматик достаточно мощный, чтобы на его основе можно было построить синтаксический анализатор для любого языка программирования, представляющего практический интерес (а такой интерес представляют только детерминированные КС-языки). Однако этот класс грамматик уже, чем класс 1Л(1)-грамматик, а значит, существуют Ы1(1)-грамматики, которые не являются LALR(^-грамматиками. Как и для других классов КС-грамматик, не всегда удается найти преобразование, позволяющее построить LALR(l)-грамматику для языка, заданного 1^(1)-грамматикой. В ряде случаев этот факт может ограничить применение распознавателей на основе LALR(l)-граММаТНК.

Но распознаватели на основе LALR(l)-граММагаК имеют еще один недостаток по сравнению с распознавателями на основе ЬК(1)-грамматик. Дело в том, что метод анализа правого контекста, применяемый при построении распознавателя для ЬАЬИ(1)-грамматики, несколько ограничивает возможности распознавателя по обнаружению ошибок во входной цепочке символов. Ошибка, конечно же, будет обнаружена, но распознавателю на основе ЬАЫ1(1)-грамматики потребуется для этого больше шагов, чем распознавателю на основе Ы1(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(l)-граММагаК упрощают выполнение синтаксического анализа, но при этом уменьшают возможности компилятора по диагностике ошибок, по сравнению с распознавателями на основе ЬК(1)-грамматик.

Тем не менее несмотря на указанные недостатки распознаватели на основе ЬАЬК(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.

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 : 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, 8, 29. 48]. Как и программа LEX, служащая для автоматизации построения лексических анализаторов, программа YACC 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 гинтаксического распознавателя на языке С. Однако существуют версии YACC,

[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(^-грамматики.

выполняющиеся под управлением ОС, отличных от UNIX, и порождающие исхо~ ный код на других языках программирования (например, Pascal).
Принцип работы YACC похож на принцип работы LEX: на вход поступает фанг содержащий описание грамматики заданного КС-языка, а на выходе получаем текст программы синтаксического распознавателя, который, естественно, можк дополнять и редактировать, как и любую другую программу на заданном язып - программирования.

Исходная грамматика для YACC состоит из трех секций, разделенных двойным символом % — %% секции описаний, секции правил, в которой описываете - грамматика, и секции программ, содержимое которой просто копируется в вь ходной файл.

Например, ниже приведено описание простейшей грамматики для YACC, коте - рая соответствует грамматике арифметических выражений, многократно использовавшейся в качестве примера в данном пособии:

```
Xtoken a
Ztoken b
*start e
%%
e : e '+' m | e      m
m : m '*' t | m '/' 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 метаязык;. 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, YACC не всегда может построить распознаватель, даже если входной язык задан правильной КС-грамматикой. Ведь заданная грамматика может и не принадлежать класс} LALR(l)-rpaMMaraK. В этом случае YACC выдаст сообщение об ошибке (о наличии неразрешимого конфликта в LALR(l)-rpaMMaraKe) при построении синтаксического анализатора. Тогда пользователь должен либо преобразовать грамматику, либо задать YACC 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 поро-

ждается исходный текст программы распознавателя (обычно это язык С). Последовательность должна располагаться после правой части соответствующего правила. Также 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 [20, 29, 46].

Синтаксические распознаватели на основе грамматик предшествования

Общие принципы грамматик предшествования

Еще одним распространенным классом КС-грамматик, для которых возможно 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 *с*\ ствует между этими двумя символами. В зависимости от найденного отноик выполняется либо сдвиг, либо свертка. При отсутствии отношения предшс вания между символами алгоритм сигнализирует об ошибке.

Задача заключается в том, чтобы иметь возможность непротиворечивым образ определить отношения предшествования между символами грамматики. Ecr это возможно, то грамматика может быть отнесена к одному из классов грам[1] тик предшествования.

Существует несколько видов грамматик предшествования. Они различаются Tt какие отношения предшествования в них определены и между какими типа' символов (терминальными или нетерминальными) могут быть установлены отношения. Кроме того, возможны незначительные модификации функционир вания самого алгоритма «сдвиг-свертка» в распознавателях для таких граммат. (в основном на этапе выбора правила для выполнения свертки, когда возможн- неоднозначности) [4, т. 1, 2].

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-грам- матику[1] G(VN,VT,P,S), V = VTuVN, в которой:

1. Для каждой упорядоченной пары терминальных и нетерминальных символе: выполняется не более чем одно из трех отношений предшествования:
   - O B, =• Bj (VB$_1$,B$_J$eV), если и только если В правило A -> xBjBjyeP, гл AeVN, x,yeV*;
   - O B, <• Bj (\/B„B^У), если и только если 3 правило A -» xB,DyeP и вывол D => *SjZ, где A,DeVN, x.y.zeV;
   - O B, •> Bj (VBpBjeV), если и только если 3 правило A -> xCBjyeP и выво" C *zB, или 3 правило A -> xCDyeP и выводы C *zB, и D *BjW, гд A,C,DeVN, x,y,z,weV*.
2. Различные правила в грамматике имеют разные правые части (то есть в грам- матике не должно быть двух различных правил с одной и той же право!" частью).

[1] Напоминаем, что КС-грамматика называется приведенной, если она не содержит циклов, бесплодных и недостижимых символов и Х-правил.

Отношения =•, <• и •> 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 предшествования — это значит, что они не могут находиться рядом ни в одном элементе разбора синтаксически правильной цепочки. Отношения предшествования зависят от порядка, в котором стоят символы, и в этом смысле их нельзя путать со знаками математических операций — они не обладают ни свойством коммутативности, ни свойством ассоциативности. Например, если известно, что B_i •> B_j, то не обязательно выполняется B_j < B_i (поэтому знаки предшествования иногда помечают специальной точкой: =•, <•, •>).

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 (для этого достаточно проверить рассмотренные выше свойства грамматик простого предшествования или воспользоваться алгоритмом построения матрицы предшествования, который рассмотрен далее).**

Как и для многих других классов грамматик, для грамматик простого предшествования не существует алгоритма, который бы мог 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 <• 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 •> 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_j =• 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. 4.10. На нем изображена входная цепочка символов ауРб в тот момент, когда выполняется свертка цепочки у. Символ а является последним символом подцепочки а, а символ b — первым символом подцепочки р. Тогда, если в грамматике удастся установить непротиворечивые отношения предшествования, то 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 <• или =•. А как только между этими символами будет обнаружено отношение •>, так сразу надо выполнять свертку. Причем для выполнения свертки из стека надо выбирать все символы, связанные отношением =•. То, что

все различные правила в грамматике предшествования имеют различные прагчасти, гарантирует непротиворечивость выбора правила при выполнении сверт Таким образом, установление непротиворечивых отношений предшествова! между символами грамматики в комплексе с несовпадающими правыми частяразличных правил дает ответы на все вопросы, которые надо решить для орган, зации работы алгоритма «сдвиг-свертка» без возвратов.

Рис. 4.10. Отношения между символами входной цепочки в грамматике предшествован. -

На основании отношений предшествования строят матрицу предшествование 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 непосредс: венно на определения отношений предшествования. Удобнее воспользоваться дв;. мя дополнительными множествами — множеством крайних левых и множество(крайних правых символов относительно нетерминальных символов граммати G(VN,VT,P,S), V = VTuVN. Эти множества определяются следующим образ<

- L(A) = {X | 3 A => *Xz}, AeVN, XeV, zeV* — множество крайних левых си волов относительно нетерминального символа A (цепочка z может быть и пустой цепочкой);
- R(A) = {X | 3 A => *zX}, AeVN, XeV, zeV* — 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:

- Bj =• Bj (VB„BjeV), если 3 правило A -> xBДyeP, где AeVN, x,yeV;
- B, <• Bj (VBj.BjeV), если 3 правило A -» xB;DyeP и BJeL(D), где A,DeVN x,yeV';
- B, •> Bj (VB„BjeV) , если 3 правило A -> xCBjyeP и B,eR(C) или 3 правиле A xCDyeP и B,eR(C), B^ЦБ), где A,C,DeVN, x,yeV\

Такое определение отношений удобнее на практике, так как не требует построения выводов, а множества L(A) и R(A) могут быть построены для каждогс нетерминального символа AeVN грамматики G(VN,VT,P,S), V = VTuVN ii очень простому алгоритму:

Шаг 1. VAeVN:

$R_0(A) = \{X \mid A \wedge yX, XeV, yeV^*\}$, $L_0(A) = \{X \mid A \wedge Xy, XeV, yeV^*\}$, i := 1.

Для каждого нетерминального символа А ищем все правила, содержащие А 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. VAeVN:

$Ri(A) \ll R_{i.1}(A) \cup R_{i.1}(B)$, $VBe(R_M(A) \cap VN)$,

$L,(A) = L \cup CAM \wedge B)$, $VBe(L_M(A) \cap VN)$.

Для каждого нетерминального символа А: если множество L(A) содержит нетерминальные символы грамматики А', А", ..., то его надо дополнить символами, входящими в соответствующие множества L(A'), L(A"),... и не входящими в L(A). Ту же операцию надо выполнить для R(A).

Шаг 3. Если 3 AeVN: $R_;(A) * R_M(A)$ или Ц(А) * Ц.^А), то i := i + 1 и вернуться к шагу 2, иначе построение закончено: R(A) = R,(A) и L(A) = L,(A).

Если на предыдущем шаге хотя бы одно множество L(A) или R(A) для некоторого символа грамматики изменилось, то надо вернуться к шагу 2, иначе построение закончено.

После построения множеств L(A) и R(A) по правилам грамматики создается матрица предшествования. Матрицу предшествования дополняют символами $\underline{I}_{„}$ и $1_К$ (начало и конец цепочки). Для них определены следующие отношения предшествования:

$\pm_н$ <• X, VaeV, если 3 S => *Xy, где SeVN, yeV* или (с другой стороны) если XeL(S);

$\pm_к$ •> X, VaeV, если 3 S => *yX, где SeVN, yeV* или (с другой стороны) если XeR(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. Однозначный выбор правила при свертке обеспечивается за счет различия правых частей всех правил грамматики. В начальном состоянии автомата считывающая головка обозревает первый символ входной цепочки, в стеке МП-автомата находится символ $\pm_{„}$ (начало цепочки), в конец цепочки помещен символ $1_К$ (конец цепочки). Символы $\underline{I}_Н$ и $\underline{I}_К$ введены для удобства работы алгоритма, в язык, заданный исходной грамматикой, они не входят.

Разбор считается законченным (алгоритм завершается), если считывающая : ловка автомата обозревает символ J_K, и при этом больше не может быть выпс - нена свертка. Решение о принятии цепочки зависит от содержимого стека. Автомат принимает цепочку, если в результате завершения алгоритма в стеке находя ~ начальный символ грамматики S и символ J_H. Выполнение алгоритма мо - быть прервано, если на одном из его шагов возникнет ошибка. Тогда входная почка не принимается.

Алгоритм состоит из следующих шагов:

Шаг 1. **Поместить в верхушку стека символ ±н, считывающую головку — в на ло входной цепочки символов.**

Шаг 2. **Сравнить с помощью отношения предшествования символ, находящи" на вершине стека (левый символ отношения), с текущим символом входной _ почки, обозреваемым считывающей головкой (правый символ отношения).**

Шаг 3. **Если имеет место отношение <• или =, то произвести сдвиг (перенос тек щего символа из входной цепочки в стек и сдвиг считывающей головки на од шаг вправо) и вернуться к шагу 2. Иначе перейти к шагу 4.**

Шаг 4. **Если имеет место отношение •>, то произвести свертку. Для этого н найти на вершине стека все символы, связанные отношением =• («основу»), \ лить эти символы из стека. Затем выбрать из грамматики правило, имею, правую часть, совпадающую с основой, и поместить в стек левую часть выб^ ного правила (если символов, связанных отношением =, на верхушке стека :- то в качестве основы используется один, самый верхний символ стека). Е. правило, совпадающее с основой, найти не удалось, то необходимо прервать полнение алгоритма и сообщить об ошибке, иначе, если разбор не закончен вернуться к шагу 2.**

Шаг **5. Если не установлено ни одно отношение предшествования между тек щим символом входной цепочки и символом на верхушке стека, то надо пр рвать выполнение алгоритма и сообщить об ошибке.**

Ошибка в процессе выполнения алгоритма возникает, когда невозможно выпол- нить очередной шаг — например, если не установлено отношение предшествова- ния между двумя сравниваемыми символами (на шагах 2 и 4) или если не удаетс - найти нужное правило в грамматике (на шаге 4). Тогда выполнение алгоритм прерывается.

Пример распознавателя для грамматики простого предшествования

Рассмотрим в качестве примера грамматику G({+,-./,*,**a,**b}.{S,R,T,F.E}.**P.**S) **с пр. вилами:**

P:
S TR | т
R -» +т | -T !I +TR | -TR
T -» EF | E
F -> *E | /E II *ef I /EF
E -» (S) I a |I b

Эта нелеворекурсивная грамматика для арифметических выражений над символами a и b уже несколько раз использовалась в качестве примера для построения распознавателей. Хотя эта грамматика и содержит цепные правила, легко увидеть, что она не содержит циклов, совпадающих правых частей правил и A,-правил, следовательно, по формальным признакам ее можно отнести к грамматикам простого предшествования. Осталось определить отношения предшествования. Построим множества крайних левых и крайних правых символов относительно нетерминальных символов грамматики.

Шаг 1.

L0(S) = {T} RO(S) = {R.T}

L0(R) = {+.-} ROCR) = {R.T}

LOCI) = {E} ROCI) = {E.F}

L0(F) = {*./} R0(F) = {E,F}

L0(E) = {(.a.b} ROCE) = {).a.b}

Шаг 2.

MS) = {T,E} R1(S) = {R.T.E.F}

L1(R) = {+.-} R1(R) = {R.T.E.F}

L1(T) = {E.(.a.b} R1(T) = {E.F.).a.b}

L1(F) = {*./} R1(F) = {E.F.).a.b}

L1(E) = {(.a.b} R1(E) = {).a.b}

Шаг3. **Имеется L0(S) L1(S), возвращаемся к шагу 2.**

Шаг 2.

L2(S) = {T.E.(,a,b} R2(S) = {R.T.E.F,).a.b}

L2(R) = {+.-} R2(R) = {R.T.E.F,),a.b}

L2(T) = {E,(.a.b} R2(T) = {E.F,),a.b}

L2(F) = {*./} R2(F) = {E.FJ.a.b}

L2(E) = {(.a.b} R2(E) = {),a.b}

Шаг 3. **Имеется L1(S) * L2(S), возвращаемся к шагу 2.**

Шаг 2.

L3(S) « {T.E.(.a.b}R3(S) = {R.T.E.F.).a.b}

L3(R) « {+.-} R3(R) = {R.T.E.F.).a.b} [1]

L3(T) = {E.C.a.b} R3(T) = {E.FJ.a.b}

L3(F) = {*./} R3(F) = {E.FJ.a.b}

L3(E) = {(.a.b} R3(E) = {).a.b}

Шаг 3. **Ни одно множество не изменилось, построение закончено. Результат:**

L(S) = {T.E.(.a.b} R(S) = {R.T.E.F.).a.b}

L(R) = {+.-} R(R) = {R.T.E.FJ,a.b}

L(T) = {E.(.a.b}R(T) = {E.FJ.a.b}

LCF) = {*./} R(F) = {E.FJ.a.b}

L(E) = {(.a.b} R(E) = {).a.b}

На основании построенных множеств крайних левых и крайних правых сим; лов и правил грамматики построим матрицу предшествования. Результат приз ден в табл. 4.5.

Таблица 4.5. Таблица предшествования для грамматики простого предшествования

	+	-	*	/	(	)	a	b	S	R	T	F	E	-K
+					<•		<•	<•					<•	
					<•		<•	<•					<•	
-					<•		<•	<•						
/					<•		<•	<•					—	
(					<•		<•		-		<•		<•	
)	>	•>		•>		•>				•>		>		•>
a	>	•>				•>				•>		>		•>
b	>	•>		•>		•>				>		•>		•>
S														
R						•>								•>
T	<•	<•				•>								•>
F	>	>				•>				•>				•>
E	•>	>	<•	<•		•>				>				•>
-L„					<•		<•				<•			

Поясним построение таблицы на примере символа +.

Во-первых, в правилах грамматики R —> -ЬТ | +IR символ + стоит слева от символа ~ Значит, в строке символа + в столбце, соответствующем символу Т, ставим знак = Кроме того, во множество ЦТ) входят символы Е, (, а, Ь. Тогда в строке символа - во всех столбцах, соответствующих этим четырем символам, ставим знак <•.

Во-вторых, символ + входит во множество L(R), а в грамматике имеются правила вида S TR и R -» +IR | -TR. Следовательно, надо рассмотреть также множество R(T). Туда входят символы Е, F,), а, Ь. Значит, в столбце символа + во всех строках, соответствующих этим пяти символам, ставим знак •>. Больше символ + ни в какие множества не входит и ни в каких правилах не встречается.

Продолжая эти рассуждения для остальных терминальных и нетерминальных символов грамматики, заполняем все ячейки матрицы предшествования, приведенной выше. Если окажется, что согласно логике рассуждений в какую-либо клетку матрицы предшествования необходимо поместить более чем один знак = <• или •>, то это означает, что исходная грамматика не является грамматико!. простого предшествования.

Отдельно можно рассмотреть заполнение строки для символа _Ih (начало строки) и столбца для символа _Ik (конец строки). Множество L(S), **где** S **— целевой**

символ, содержит символы Т, Е, (, а, Ь. Помещаем знак <• в строку, соответствующую символу JH для всех пяти столбцов, соответствующих этим символам. Аналогично, множество R(S) содержит символы R, T, E, F,), a, Ь. **Помещаем знак •> в столбец, соответствующий символу ±к для всех семи строк, соответствующих этим символам.**

Рассмотрим работу алгоритма распознавания на примерах. Последовательность разбора будем записывать в виде последовательности конфигураций МП-автомата из трех составляющих:

1. **не просмотренной автоматом части входной цепочки;**
2. **содержимого стека;**
3. **последовательности примененных правил грамматики.**

Так как автомат имеет только одно состояние, то для определения его конфигурации достаточно двух составляющих — положения считывающей головки во зходной цепочке и содержимого стека. Последовательность номеров правил несет дополнительную полезную информацию, на основании которой можно построить цепочку или дерево вывода. Правила в грамматике нумеруются в направлении слева направо и сверху вниз (всего в грамматике имеется 15 правил).

Будем обозначать такт автомата символом -н. Введем также дополнительное обозначение -мд, если на данном такте выполнялся перенос, и -к:, если выполнялась свертка.

Последовательности разбора цепочек входных символов будут, таким образом, иметь вид:

Пример 1. **Входная цепочка а+а*Ь:**

1. {а+а*Ь±к:±н;0} -hi
2. {+а*Ь±к;±на:0} -к:
3. {+а*Ь±к;±нЕ;14} ^C
4. {+а*Ь±к:±нТ;14,8}
5. {а*Ь±к;±нТ+:14,8}
6. {*Ь±к;±нТ+а;14.8} -к:
7. {*Ь_1_к;_LhT+E:14.8,14} -нп
8. {Ь-LK:_LHT+E*;14,8,14} -hi
9. {_LK;_1_нТ+Е*Ь:14.8.14} -к:
10. {±к;±нТ+Е*Е:14.8.14.15} -к:
11. {1K;1hT+EF;14.8.14,15.9} -к:
12. {±к:±нТ+Т:14,8.14,15,9,7}
13. {J.K;_LhTR;14,8,14,15,9.7,3} **-к:**
14. {_Lk;_LhS;14,8,14,15.9.7,3,1} **алгоритм завершен, цепочка принята.**

Соответствующая цепочка вывода будет иметь вид (используется правосторонний вывод): S => TR => T+T => T+EF => T+E*E => T+E*b => T+a*b => E+a*b => a+a*Ь.

Дерево вывода, соответствующее этой цепочке, приведено на рис. 4.11.

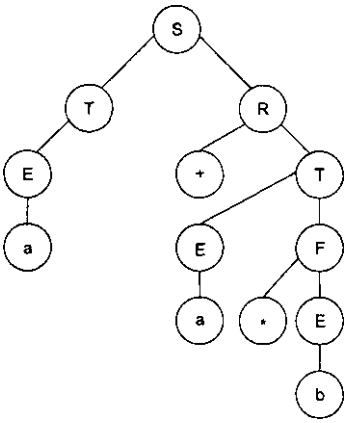


Рис. 4.11. Пример дерева вывода для грамматики простого предшествования

Пример 2. **Входная цепочка (a+a)*b:**

1. **{(a+a)*bJ_K:J_H;0}** -hi
2. {a+a)*bJ-K;J_H(;0} -HI
3. **{+a)*b-|_к:_]_н(a:0}** -ec
4. **{+a)*bJ_K;J_H(E;14}** -c
5. **{+a)*bJ_K;J_h(T; 14.8}** -sn
6. **{a)*bJ_K:J_h(T+; 14.8}** +n
7. **{)*bJ_K;J_H(T+a:14,8}** *c
8. **{)*bJ_K;J_h (T+E; 14.8,14}** -sc
9. **{)*bJ_K:J_H(T+T;14.8.14.8}** +c
10. **{)*bJ_K;J_H(TR; 14,8.14.8,3}** -sc
11. **{)*bJ_K;±H(S:14.8,14.8,3.1}** -hi
12. **{*bJ_K; J_h(S); 14.8.14,8.3.1}** -не
13. **{*bJ_K :_LhE: 14.8.14.8.3.1.13}**
14. **{bJ_K; _LHE*; 14,8.14.8.3.1.13}** -Hi
15. **{±к;±нЕ*Ь:14.8.14,8.3.1,13}** -не
16. {±k;_LhE*E;14.8.14,8.3,1.13,15} +c
17. {J_k;J_HEF; 14.8.14.8.3,1.13.15,9} -не
18. **{J_k;J_hT;14.8,14,8,3.1.13,15,9.7}** -не
19. **{_Lk;J_hS: 14.8.14.8,3.1.13.15.9.7.2} алгоритм завершен, цепочка принята.**

Соответствующая цепочка вывода будет иметь вид (используется правосторонний вывод): S => T => EF => E*E => E*b => (S)*b => (TR)*b => (T+T)*b => (T+E)*b = (T+a)*b => (E+a)*b => (a+a)*b.

Дерево вывода, соответствующее этой цепочке, приведено на рис. 4.12.

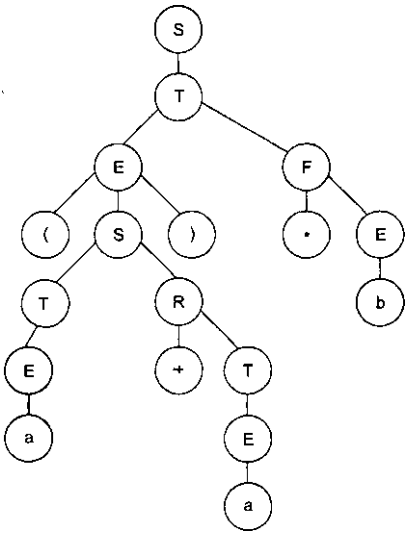


Рис. 4.12. Пример дерева вывода для грамматики простого предшествования

Пример 3. **Входная цепочка** a+a*:

1. {a+a*_1_к;_1_н;0} -5-п
2. {+a*-]_к;_]_на;0}
3. {+a*_1_к;_]_нЕ;14} -с
4. {+a*1к;_]_нТ;14.8} н-п
5. {a*±к;±нТ+;14.8} -hi
5. {*1к;1нТ+а;14.8} -не
7. {*J_k;J_HT+E: 14,8,14} -s-n
S. {J_k;J_HT+E*; 14.8,14} *
9. ошибка! **(нет отношений предшествования между символами * и ±к).**

Пример 4. **Входная цепочка a+a)*Ь:**

1. {a+a)*b_]_к;_]_н;0} -гП
2. {+a)*Ык;1на;0} н-С
3. {+a)*bJ_K;J_hE; 14} н-с
4. {+a)*bJ_K;J_hT; 14,8} -s-n
5. {a)*bJ_K: _LhT+; 14.8} -г-п
6. {)*b_LK; _1_нТ+а; 14.8} ч-с

7. {)*Ь±к;J_hT+E; 14,8,14} -s-c
8. {}*bJ_K; J_hT+T ; 14,8,14,8}
9. {}*bJ_K;J_hTR;14,8.14,8.3} -sc
10. {}*bJ_K;J_hS ; 14.8,14.8.3,1} +n
11. {*bJ_K;J_HS): 14,8.14.8,3.1} -s-c
12. ошибка! (невозможно выбрать правило для свертки на этом шаге).

Грамматики простого предшествования являются удобным механизмом для лиза входных цепочек КС-языков. Распознаватель для этого класса грамл строить легче, чем для рассмотренных выше LR-грамматик. Однако класс ков, заданных грамматиками простого предшествования, уже, чем класс яз! заданных LR-грамматиками. Отсюда ясно, что не всякий детерминирова! КС-язык может быть задан грамматикой простого предшествования, а еле. тельно, не для каждого детерминированного КС-языка можно построить р< знаватель по методу простого предшествования.

У грамматик простого предшествования есть еще один недостаток — при С шом количестве терминальных и нетерминальных символов в грамматике рица предшествования будет иметь значительный объем (при этом следует о тить, что значительная часть ее ячеек может оставаться пустой). Поиск в -к матрице может занять некоторое время, что существенно при работе распоз; теля — фактически время поиска линейно зависит от числа символов грам.\ ки, а объем матрицы — квадратично. Во избежание хранения и обработки т. матриц можно выполнить «линеаризацию матрицы предшествования». Т каждый раз, чтобы установить отношение предшествования между двумя сп лами, будет выполняться не поиск по матрице, а вычисление некой специ но организованной функции. Вопросы линеаризации матрицы предшеств; здесь не рассматриваются, с применяемыми при этом методами можно озн миться в [3, 4, т. 1, 2, 17, 22].

Грамматики операторного предшествования

Операторной грамматикой называется КС-грамматика без ^.-правил, в коте правые части всех правил не содержат смежных нетерминальных символов. . операторной грамматики отношения предшествования можно задать на мне стве терминальных символов (включая символы $J_{\_}$, и $J_{\_K}$).

Грамматикой операторного предшествования называется операторная КС-грам тика G(VN,VT,P,S), V = VTuVN, для которой выполняются следующие условие

1. Для каждой упорядоченной пары терминальных символов выполняется более чем одно из трех отношений предшествования:
   - а =• Ь, если и только если существует правило A -> xabyeP или правил: A хаСЬу, где a,beVT, A,CeVN, x.yeV*;
   - а <• b, если и только если существует правило A —xaCyeP и вывод C = *bz или вывод C => *Dbz, где a,beVT, A,C,DeVN, x,y,zeV;

О a •> Ь, если и только если существует правило A xCЬyeP и вывод C => *za или вывод C *zaD, где a,beVT, A,C,DeVN, x,y,zeVV

2. Различные порождающие правила имеют разные правые части, ^.-правила отсутствуют.

Этношения 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. То есть в грамматике операторного предшествования G(VN,VT,P,S), V = VTuVN не может быть ни одного правила вида: A -> xBCy, где A,B,CeVN, x,yeV* (здесь x и y — это произвольные цепочки символов, которые могут быть и пустыми).

Для грамматик операторного предшествования также известны следующие свойства:

- всякая грамматика операторного предшествования задает детерминированный КС-язык (но не всякая грамматика операторного предшествования при этом является однозначной!);
- легко проверить, является или нет произвольная КС-грамматика грамматикой операторного предшествования (точно так же, как и для простого предшествования).

Как и для многих других классов грамматик, для грамматик операторного предшествования не существует алгоритма, который бы мог 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 — $L^1(A)$ или $R^1(A)$:

- $L^1(A)$ = {t | B A => *tz или B A => *Ctz }, где teVT, A,CeVN, zeV*;
- $R^1(A)$ = {t | 3 A => *zt или 3 A => *ztC }, где teVT, A,CeVN, zeV\

Тогда определения отношений операторного предшествования будут выглядеть так:

- a =• Ь, если 3 правило A -> xaby eP или правило U xaCЬy, где a,beVT, A,CeVN, x,ysV*;

[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.

- **a <• Ь, если В правило А -> хаСу eP и beL^{l}(C), где a,beVT, A,CeVN, x,yeY**
- **a •> b, если 3 правило А -> хСЬу eP и aeR^{l}(C), где a,beVT, A,CeVN, x,yeY**

В данных определениях цепочки символов x, y, z могут быть и пустыми цепочкам». Для нахождения множеств L^{l}(A) и R^{f}(A.) предварительно необходимо выпалнить построение множеств L(A) и R(A), как это было рассмотрено ранее. Далее для построения L^{l}(A) и R^{C}(A) используется следующий алгоритм:

Шаг 1. **VAeVN:**

К*о(А) = {t | A -> ytB или A yt, teVT, BeVN, yeV*},

L$^{l}_{0}$(A) - {t | A -> Bty или A ty, teVT, BeVN, yeV*}.

Для каждого нетерминального символа А ищем все правила, содержащие А в левой части. Во множество L(A) включаем самый левый терминальный символ из правой части правил, игнорируя нетерминальные символы, а во множество R(A) — самый крайний правый терминальный символ из правой части правил. Пере дим к шагу 2.

Шаг 2. **VAeVN:**

R^{l}i(A) = RVi(A)uRVi(B), VBe(R(A)nVN),

L\(A) **= 1^{1}$_{н}$(A)иЦ.,(B), VBe(L(A)nVN).**

Для каждого нетерминального символа А: если множество L(A) содержит нетерминальные символы грамматики А', А",..., то его надо дополнить символами, вхсдящими в соответствующие множества Ь^{Г}(А'), Ь'(А"),... и не входящими в L'(A Ту же операцию надо выполнить для множеств R(A) и R^{l}(A).

Шаг 3. **Если 3 AeVN: R^A) * К^{1}$_{М}$(А) или Ц(А) * LVi(A), то i := i + 1 и вернуться к шагу 2, иначе построение закончено: R(A) = R\(A) и L(A) = L^{l}j(A).**

Если на предыдущем шаге хотя бы одно множество L^{l}(A) или R^{l}(A) для некоторого символа грамматики изменилось, то надо вернуться к шагу 2, иначе построение закончено.

Для практического использования матрицу предшествования дополняют символами J$_{„}$, и J$_{К}$ (начало и конец цепочки). Для них определены следующие отношения предшествования:

1„ <• a, VaeVT, если 3 S => *ах или 3 S => *Сах, где S,CeVN, xeV* или если aeU(S);

J$_{К}$ •> a, VaeVT, если 3 S => *ха или 3 S *хаС, где S,CeVN, xeV* или если aeR4S).

Здесь S — целевой символ грамматики.

Матрица предшествования служит основой для работы распознавателя языка, заданного грамматикой операторного предшествования. Поскольку она содержит только терминальные символы, то, следовательно, будет иметь меньший размер, чем аналогичная матрица для грамматики простого предшествования. Следует отметить, что напрямую сравнивать матрицы двух грамматик нельзя — не всякая грамматика простого предшествования является грамматикой операторного предшествования и наоборот. Например, рассмотренная далее в приме-

ре грамматика операторного предшествования не является грамматикой простого предшествования (читатели могут это проверить самостоятельно).

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, с той лишь разницей, что объем хранимой матрицы будет меньше.

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 этличие 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 $\_I_{4b}$ 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 <• или =•, 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 •>, то произвести свертку. Для этого надо найти на вершине стека все терминальные символы, связанные отношением =• («основу»), а также все соседствующие с ними нетерминальные символы (при определении отношения нетерминальные символы игнорируются). Если терминальных символов, связанных отношением =•, 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 : - кончен, то вернуться к шагу 2.

Шаг 5. Если не установлено ни одно отношение предшествования между текут, символом входной цепочки и самым верхним терминальным символом в сте- то надо прервать выполнение алгоритма и сообщить об ошибке.

Конечная конфигурация данного МП-автомата совпадает с конфигурацией п: распознавании цепочек грамматик простого предшествования.

Пример построения распознавателя для грамматики операторного предшествования

Рассмотрим в качестве примера грамматику для арифметических выражений символами a и b G({+.-./.*.a.b}.{S.T.E}.P.S):

P:

S -> S+T I S-T I T
T -> T*E I T/E I E
E -> (S) | a | b

Эта грамматика уже много раз использовалась в качестве примера для пост] ния распознавателей. [1]

Видно, что эта грамматика является грамматикой операторного предшествования Построим множества крайних левых и крайних правых символов L(A), R(A) от- носительно всех нетерминальных символов грамматики. Рассмотрим работу ал- горитма построения этих множеств по шагам.

Шаг 1.

L0(S) = {S.T}. R0CS) = {T}.

L0(T) = {T,E}, R0(T) = {E}.

L0(E) = {(.a.b}. R0CE) = {).a,b}. i = 1

Шаг 2.

L1(S) = {S.T.E},R1(S) = {T}.

L1(T) = {T.E.(.a.b}. R1(T)={E}.

L1(E) = {(.a.b},R1(E) = {).a.b}

Шаг 3. Так как $L_0(S)$ * Li(S), то i = 2, и возвращаемся к шагу 2.

Шаг 2.

L2(S) = {S.T.E.(.a.b}. R2(S) = {T},

L2(T) = {T.E.(.a.b}. R2(T) = {E},

L2(E) = {(.a.b}.R2(E) = {).a.b}

Шаг 3. Так как $L_j(S)$ * $L_2(S)$, то i = 3, и возвращаемся к шагу 2.

Шаг 2.

L3(S) = {S.T.E.(.a.b}. R3(S) = {T}.

L3(T) = {T.E.(.a.b}. R3(T) = {E},

L3(E) = {(.a.b}.R3(E) = {).a.b}

Построение закончено.

Получили результат:

US) - {S.T.E.(.a.b}. R(S) = {T}.

L(T) = {T.E.(.a.b}. R(T) = {E}.

L(E) = {(.a.b}. R(E) = {).a.b}

На основе полученных множеств построим множества крайних левых и крайних правых терминальных символов Lt(A), Rt(A) относительно всех нетерминальных символов грамматики. Рассмотрим работу алгоритма построения этих множеств по шагам.

Шаг 1.

LtO(S) = {+.-}. RtO(S) « {+.-}.

LtO(T) = {*./}. RtO(T) = {*./}.

LtO(E) = {(.a.b}. RtO(E) = {).a.b}. i = 1

Шаг 2.

Ltl(S) = {+.-.*./.(.a.b}. Rtl(S) = {+,-.*./}.

Ltl(T) = {*./.(,a.b}. Rtl(T) = {*./,).a.b}.

Ltl(E) = {(.a.b}. Rtl(E) = {).a.b}

Шаг 3. **Так как $L^{l}_{0}(S)$ * LS(S), то i = 2, и возвращаемся к шагу 2.**

Шаг 2.

Lt2(S) = {+.-.*./.(.a.b}. Rt2(S) = {+.-.*./.).a.b}.

Lt2(T) = {*./.(.a.b}. Rt2(T) = {*./.).a.b}.

Lt2(E) = {(.a.b}. Rt2(E) = {).a.b}

Шаг 3. **Так как RS(S) * $R^{t}_{2}(S)$, то i = 3, и возвращаемся к шагу 2.**

Шаг 2.

Lt3(S) = {+.-.*./.(.a.b}. Rt3(S) = {+.-.*./.).a.b}.

Lt3(T) = {*./.(.a.b}. Rt3(T) = {*./.).a.b}.

Lt3(E) = {(.a.b}. Rt3(E) = {).a.b}

Построение закончено.

Получили результат:

Lt(S) = {+.-.*./.(.a.b}. Rt(S) = {+.-.*./.).a.b}.

Ltd) = {*./.(.a.b}. Rtd) = {*./.).a.b}.

LtCE) = {(.a.b}. RUE) = {).a.b}

На основе этих множеств и правил грамматики G построим матрицу предшествования грамматики (табл. 4.6).

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 Ltd) входят символы: * /, (, а, Ь. Ставим знак <• 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 Rt(S) входят символы: +, -, *, /,), а, Ь. Ставим знак •>

в клетках матрицы, соответствующих этим символам, в столбце для символа - Больше символ + ни в каком правиле не встречается, значит, заполнение матр; цы для него закончено, берем следующий символ и продолжаем заполнять ма: рицу таким же методом, пока не переберем все терминальные символы.

Таблица 4.6. Матрица предшествования грамматики

Символы	+	-	*	/	(	)	a	b	
+	·>	>		<·	<·	·>	<	<·	·>
-	·>	>	<	<·	<·	·>	<·	<·	>
*	·>	>		·>	<·	·>	<·	<	·>
/	>	>		·>		·>	<·	<·	·>
(	<·		<·	<·	<·		<·	<·	
)	>	·>	·>	·>		·>			>
a	·>	>	>	·>		>			·>
b	·>	·>	·>	·>		·>			>
-L,,	<·	<·	<·	<·	<·		<·	<·	

Отдельно рассмотрим символы 1н и 1к. В строке символа 1h ставим знак <• в клетках символов, входящих во множество Lt(S). Это символы +,-,*,/,(, а, Ь. В столбце символа 1k ставим знак •> в клетках символов, входящих во множество Rt(S Это символы +, —, *, /,), а, Ь.

Еще можно отметить, что в клетке, соответствующей открывающей скобке (символ () слева и закрывающей скобке (символ)) справа, помещается знак =• («составляют основу»). Так происходит, поскольку в грамматике присутствует правило E-> (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 (=•) верно, а отношение)=•(— 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 I S-S I S (правила 1, 2 и 3)
S -> S*S | S/S I S (правила 4, 5 и 6)
S -> (S) | a | b (правила 7, 8 и 9)

Если теперь исключить бессмысленные правила вида S—>S, то получим следующее множество правил (нумерацию правил сохраним в соответствии с исходной грамматикой):

P:
S S+S | S-S (правила 1, 2)

S -> S*S I 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 (который при выборе правил все равно игнорирует нетерминальные символы) после построения матрицы предшествования. Полученная в результате преобразования грамматика не является однозначной, но в алгоритм распознавания уже были заложены все необходимые данные о порядке применения правил при создании матрицы предшествования, 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].

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, напротив, 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 просмотренной автоматом части входной цепочки;**
2. **содержимого стека;**
3. **последовательности 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 или дерево вывода.

Как и ранее, будем обозначать такт автомата: -hi, если на данном такте выполнялся перенос, и -не, если выполнялась свертка.

Последовательности разбора цепочек входных символов будут, таким образом, иметь вид, приведенный ниже.

[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.

Пример 1. **Входная цепочка a+a*Ь:**

1. {a+a*b_1_к;_Lh ; 0}-гп
2. {+a***Ык:±на:0**}+c
3. {+a*b±K:±HS:8}+n
4. {a*b_LK:J_HS+:8}-e-n
5. {*b±K;lHS+a:8}4-c
6. {*b±K:±HS+S:8.8}+n
7. {**Ь-LK** ; **_LHS+S*** ; 8,8}-ГП
8. {lK;lHS+S*b:8.8}4-c
9. {±K;±HS+S*S:8.8.9}-H:
10. {1K;1HS+S:8.8.9.4}-HC
11. {1K:J.HS: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. 4.13.

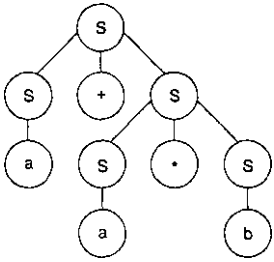


Рис. 4.13. Пример дерева вывода для грамматики операторного предшествования

Пример 2. **Входная цепочка** (a+a)*b:

1.	{(a+a)*b1к;1н	0}+n
2.	{a+a)*b_1_к;_1_н(	0}-m
3.	{+a)*b±к;±н(a	0}-нс
4.	{+a)*b±**K;±H**(S	8}-m
5.	{a)***blK;lH**(S+	8}-m
6.	{)***b**±K;±**H**(S+a	8}-H:
7.	{)*bJ_K :_LH(S+S	8,8}-нс
8.	{)*b_LK:_LH(S ; 8	8.1}+n
9.	{*blK;LH(S):8	8.1}+c
10.	{*blK;lHS:8.8	1.7}+n
И.	{**b±K:±HS***;**8.8**	1,7}-m

12. {±K:±HS*b:8.8.1.7}*c

13. {1_к;_LHS*S ;8.8,1.7,9}-5-c

14. {_LK;_LHS;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. 4.14.

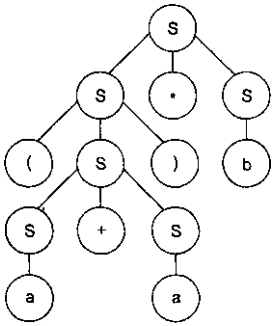


Рис. 4.14. Пример дерева вывода для грамматики операторного предшествования

Пример 3. **Входная цепочка** a+a*:

1. {a+a*_1_к;_1_н;0}-г-п
2. {+a*±к;±на;0}-н:
3. {+a*±K;±HS:8}-m
4. {a*±K:±HS+;8}-Hi
5. {*±K;±HS+a;8Hc
6. {*±K;±HS+S;8.8}-HI
7. {_LK;_LHS+S*;8.8}-H:
8. ошибка! (нет правила для выполнения свертки на этом шаге).

Пример 4. **Входная цепочка a+a)*Ь:**

1. {a+a)*Ь_1_к;_1_н:0}-т
2. {+a)*Ык;±на:0}ч-с
3. {+a)*b±K;±HS;8}-m
4. {a)*Ь:1нБ+:7}н-п
5. {)*blK:±HS+a:8}H-c
6. {)*blK:lHS+S:8.8}-c
7. {)*b_LK:_LHS;8.8.1}
8. ошибка! (нет отношений предшествования между символами 1н и)).

Два первых примера наглядно демонстрируют, что приоритет операций, установленный в грамматике, влияет на последовательность разбора и последовательность применения правил несмотря на то, что нетерминальные символы распознавателем не учитываются.

Как было сказано выше, матрица для грамматики операторного предшествования всегда имеет меньший объем, чем матрица для эквивалентной ей грамматики простого предшествования. Кроме того, распознаватель грамматики операторного предшествования игнорирует нетерминальные символы в процессе разбора, а значит, не учитывает цепные правила, делает меньше шагов и порождает более короткую цепочку вывода. Поэтому распознаватель для грамматики операторного предшествования всегда проще, чем распознаватель для эквивалентной ей грамматики простого предшествования.

Интересно, что поскольку распознаватель на основе грамматик операторной: предшествования не учитывает типы нетерминальных символов, то 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 aAЬ | 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 | Б будет применено на первом шаге вывода).

ПРИМЕЧАНИЕ

Грамматики, содержащие правила, различающиеся только типами нетерминальных символов, практического значения не имеют, а потому интереса для компиляторов не представляют.

Хотя классы грамматик простого и операторного предшествования несопоставимы[1], класс языков операторного предшествования уже, 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 опреде-

[1] В том, что эти два класса грамматик несопоставимы, можно убедится, 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.

лить далеко не каждый детерминированный КС-язык. Грамматики операторного предшествования — это очень удобный инструмент для построения распознавателей, но они имеют ограниченную область применения.

Контрольные вопросы и задачи

Вопросы

1. Какие из следующих утверждений справедливы:
   - О если язык задан КС-грамматикой, то он может быть задан с помощью МП-автомата;
   - О если язык задан КС-грамматикой, то он может быть задан с помощью ДМП-автомата;
   - О если язык задан ДМП-автоматом, то он может быть задан КС-граматикой;
   - О если язык задан однозначной КС-грамматикой, то он может быть задан с помощью ДМП-автомата;
   - О если язык задан расширенным МП-автоматом, то он может быть задан КС-грамматикой?
2. Алфавит магазинных символов МП-автомата включает в себя алфавит входных символов автомата. Почему?
3. Какие из приведенных ниже МП-автоматов являются детерминированными:

   R1({q1.q2}.{a.b}.{a.b.A}.81.q1.A.{q2}).81(q1.a.A) = {(q1.AA)}.81(q2.a.A) = {(q2 A)}.81(q1.b.A) = {(q2.A)}.51(q2.b.A) = {(q2.Aab)}.S1(q1A.a) = {(q1A)}.81(q1A.b) = {(q1A)};
   R2({q},{a.b},{a.b.A},82,q.A,{q}),82(q,a,A) = {(q.AA)},52(q,b.A) « {(q.A)},82(qA.a) = {(q A)} ,82(q A.b) = {(q A)} .82(q A.A) = {(q.A)}:
   R3({q}.{a.b}.{a.b.A},83.q,A,{q}),83(q.a.A) = {(q.AA).(q.a).(qA)}.S3(q.b.A) = {(q.A)}.83(qA.a) = {(q.A)};
   R4({q}.{a.b}.{a.b.A}.84.q.A.{q}).84(q.a.A) = {(q,AAb)}.84(q.b.A) = {(q.a)}.84(qA.a) = {(q A)}.84(q A.b) = {(qA)}
4. Почему синтаксические конструкции языков программирования могут быть распознаны с помощью ДМП-автоматов? 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. Если проблема однозначности неразрешима для произвольной КС-грамматики, то почему можно утверждать, что некоторые КС-грамматики являются однозначными? Всегда ли должны быть однозначными грамматики синтаксических конструкций языков программирования?
6. Какой цели служит преобразование правил КС-грамматик? Всегда ли это преобразование ведет к упрощению правил?
7. Почему при преобразовании КС-грамматики к приведенному виду сначала необходимо удалить бесплодные символы, а потом — недостижимые символы?

8. Почему для языка, заданного грамматикой, содержащей ^.-правила и циклы существуют трудности с моделированием работы расширенного МП-автом та, выполняющего восходящий разбор с правосторонним выводом? Что мо^ но сказать об обычном МП-автомате, выполняющем нисходящий разбор д..-. цепочек, построенных с помощью таких грамматик?
9. Почему необходимо устранить именно левую рекурсию из правил грамматик! Для моделирования работы какого (восходящего или нисходящего) распозн^ вателя представляет сложность левая рекурсия? Можно ли полностью устр нить рекурсию из правил грамматики, записанных в форме Бэкуса—Наур.
10. Сравните восходящий и нисходящий распознаватели. Какими преимуществ^ ми и недостатками обладает каждый из этих методов?
11. В чем главное преимущество линейных распознавателей для КС-языков rv ред всеми другими видами распознавателей? Какие существуют сложност в построении линейных распознавателей для КС-языков?
12. На алгоритме какого распознавателя основан метод рекурсивного спуск:. В чем заключается принципиальное изменение, внесенное в алгоритм? Веде ли это изменение к ограничению применимости данного алгоритма?
13. Какие преобразования возможно выполнить над грамматикой, чтобы к не был применим метод рекурсивного спуска? Можно ли формализовать и авк: матизировать выполнение этих преобразований? Можно ли реализовать pat - познаватель по методу расширенного рекурсивного спуска для грамматик; содержащей левую рекурсию?
14. За счет чего класс ЬЦ1)-грамматик является более широким, чем класс КС-грамматик, для которых можно построить распознаватель по методу рекурсивного спуска?
15. На каком алгоритме основана работа распознавателя для ЬЦк)-грамматию Какие изменения внесены в алгоритм? Насколько они сузили применимост данного алгоритма?
16. Почему класс языков, заданных LR-грамматиками, является более широкиv чем класс языков, заданных LL-грамматиками?
17. На каком алгоритме основана работа распознавателя для LR(k)-граММагаКi! Какие изменения внесены в алгоритм? Насколько они сузили применимост данного алгоритма?
18. Почему не существует языков, заданных ЬЦО)-грамматиками, но существ)-ют языки, заданные 1^(0)-грамматиками?
19. Почему можно утверждать, что любая LL-грамматика и любая LR-грамматика являются однозначными?
20. В чем особенности реализации алгоритма типа «сдвиг-свертка» для грамматик простого и операторного предшествования? Какие ограничения эти особенности накладывают на правила грамматики?
21. Почему в грамматике простого предшествования не могут присутствовать цепные правила, а в грамматике операторного предшествования — могут? Почему в обоих классах грамматик не могут присутствовать ^.-правила?

!2. Распознаватели на основе грамматик операторного предшествования имеют массу преимуществ перед распознавателями на основе грамматик простого предшествования. Почему же, тем не менее, используются оба класса КС-грамматик?

!3. Класс языков, заданных LR(1)-грамматиками совпадает с классом детерминированных КС-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 | X
   B ACb | b | X
   C A | B | bA | aB | cC | aE | bE
   E -> Ea | Eb | Ee | ED | FG | DG
   D -> c | Fb | Fa | X
   F ^ BC | AC | DC | EC
   G -> Ga | Gb | Gc | GD

   Преобразуйте ее к приведенному виду.

3. **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

   О Преобразуйте ее к виду без цепных правил.

   О Исключите левую рекурсию в правилах грамматики.

4. **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)».**

5. **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)».**

6. **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.

7. **Постройте распознаватель, основанный на расширенном методе рекурсивного спуска для грамматики: G({a,<,=,>.+,-./,*},{S.T.E}.P.S)**

P:
S -> T<T | T>T I T<=T I т>=т
T -> T+E I T-E I E
E —> a*a | a/a | a

Выполните разбор цепочки символов a*a+a<a/a-a*a.

8. **Дана грамматика G({(.). + .*.a}.{S.T.F}.P.S)**

P:
S -> S+T | т
T T*E I F
F (S) | a

Постройте для нее распознаватель на основе 8ЬК(1)-грамматики.

9. **Дана грамматика G({(a}.{S.T.E.F}.P.S)**

P:
S -»• S^AT | т
T -»• T&E | E
E —> —E j F
F -» (S) | a

Постройте для нее распознаватель на основе грамматики операторного предшествования. Выполните разбор цепочки символов a^Лa&-a&(a^A~~a).

10. **Придумайте функцию линеаризации матрицы предшествования для грамматики, заданной в задаче № 8.**

11. **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 S j a

Постройте для нее распознаватель на основе грамматики операторного предшествования, считая if, then и else едиными терминальными символами. Является ли данная грамматика однозначной? Выполните 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?

12. **Перестройте правила грамматики, данной в задаче № 10 так, чтобы else всегда относилось к объемлющему i f, а не к ближайшему.**

13. **Придумайте функцию линеаризации матрицы предшествования для грамматики, заданной в задаче № 10.**

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, 15]. 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 инфор-

[1] **Примером контекстной зависимости, часто встречающейся во многих языках программирования, может служить необходимость предварительно описать идентификатор до его первого использования.**

мация помещается в таблицу лексем на основе конструкций, найденных синтаксическим распознавателем. Примерами таких конструкций являются блоки описания 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.
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. Как правило, на этапе семантического анализа используются различные варианты синтаксических деревьев, построенных в результате синтаксического разбора, поскольку семантический анализатор интересует прежде всего структура исходной программы. 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 имеющихся в таблице идентификаторов данных (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, 31, 50, 52].

Этапы семантического анализа

Семантический анализатор выполняет следующие основные действия:

- проверку соблюдения во входной программе семантических соглашений входного языка;
- дополнение внутреннего представления программы в компиляторе операторами и действиями, неявно предусмотренными семантикой входного языка;
- проверку элементарных семантических (смысловых) норм языков программирования, напрямую не связанных со 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 заданные и 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, 50].

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 будет 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 им ют различный смысл, хотя и обозначаются в рассмотренном примере одним зн_ ком операции — +. Следовательно, от семантического анализа зависит также ке ^ результирующей программы.

Если какое-либо из семантических требований входного языка не выполняете = то компилятор выдает сообщение об ошибке и процесс компиляции на этом, на- правило, прекращается.

Дополнение внутреннего представления программы

Это дополнение внутреннего представления программы связано с добавление[1]- в него операторов и действий, неявно предусмотренных семантикой входног языка. Как правило, эти операторы и действия связаны с преобразованием типе: 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 операции присвоения результата. Соответствующим образом должен быть порожден и ко.: 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 выполняет эти преобразования? Это может сделать разработчик программы — но тогда преобразования ТИПОЕ в явном виде должны будут присутствовать в тексте входной программы. Для рассмотренного примера это выглядело бы примерно так:

```
a := double(reaKb) + 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 отличаться в зависимости от используемых преобразований типов. Например, если предположить, что перед выполнением рассматриваемого оператора его операнды будут иметь значения b = 1 ис = 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 функций компилятора описан в главе 6 «Современные системы программирования»). Вызовы этих функций как раз и будут включены компилятором в текст результирующей программы для удовлетворения семантических соглашений о преобразованиях типов во входном языке, хотя в тексте исходной программы в явном виде они не присутствуют. Чтобы это произошло, эти функции должны быть встроены 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 весьма нетривиальных функций преобразования типов. Разработчик программы должен помнить об этом, 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. Другим примером такого рода операций могут служить операции вычисления адреса, когда происходит обращение к элементам сложных структур данных. 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, напрямук не связанных со входным языком, — это сервисная функция, которую предоставляют разработчикам большинство современных компиляторов. Эта функция обеспечивает проверку компилятором соглашений, выполнение которых связанс 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 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 в семантическом анализаторе: 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

[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.

«предупреждение» (warning). Компилятор выдает пользователю сообщение об обнаружении несоблюдения одного из требований, не прерывая сам процесс компиляции, — то есть он просто обращает внимание пользователя на то или иное место в исходной программе. То, как реагировать на «предупреждение» (вносить изменения в исходную программу или проигнорировать этот факт) — это уже забота и ответственность разработчика исходной программы.

Необязательность указанных соглашений объясняется тем, о чем уже говорилось выше в главе 2 «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[1].

Задача проверки выполнения семантических соглашений входного языка во многом связана с проблемой верификации программ. Эта проблема детально рассмотрена в [1, И, 18, 30, 33, 36, 40].

Рассмотрим в качестве примера функцию, представляющую собой фрагмент входной программы на языке С:

```
int f_test(1nt a)
{   i nt b. c;
        b = 0;
        c = 0:
        if (b=l) { return a; }
        c = a + b;
}
```

Практически любой современный компилятор языка С обнаружит в данном месте входной программы массу «неточностей». Например, переменная с описана, ей присваивается значение, но она нигде не используется; значение переменной Ь, 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+Ь; никогда выполнен не будет. Скорее всего, компилятор выдаст еще одно предупреждение, характерное именно для языка С — в операторе if(b=l) присвоение стоит в условии (это не запрещено ни синтаксисом, ни семантикой языка, но является очень распространенной семантической ошибкой в языке С). В принципе, смысл (а точ-

[1] 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.

нее, бессмысленность) этого фрагмента будет правильно воспринят и обработа:-компилятором.

Однако если взять аналогичный по смыслу, но синтаксически более сложны? 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 значени; переменных и выполнение условий в данном фрагменте из двух функций (обг они сами по себе независимо вполне осмысленны!). Единственное предупреждение, которое, скорее всего, получит в данном случае разработчик, — это то, чт: функция f test не всегда корректно возвращает результат (отсутствует операто return перед концом функции). И то это предупреждение на самом деле не буде 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 лексически?, 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 всегс этого бывает недостаточно, чтобы установить однозначное соответствие межд;> 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 сильно затруднена, 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 C 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 лексичесь единицам языка, используются только во внутреннем представлении исходи программы, и разработчик исходной программы не сталкивается с ними. Но о:-: могут потребоваться ему в некоторых случаях — например, при отладке програ мы, при порождении текста результирующей программы на языке ассембле: - или при использовании библиотеки, созданной на другом языке программирования (или даже с помощью другой версии компилятора). Тогда разработчик дс - жен знать, по каким правилам компилятор порождает уникальные имена дл лексических единиц исходной программы[1].

СОВЕТ

Правила, по которым происходит модификация имен, достаточно просты. Их мс> но выяснить для конкретной версии компилятора и использовать при разработ- - Однако лучше этого не делать, так как нет гарантии, что при переходе к другой Вс: сии компилятора эти правила не изменятся — тогда код станет неработоспособны > Правильным средством будет аккуратное использование механизма отключен именования лексических единиц, предоставляемое синтаксисом исходного язы:-_

Во многих современных языках программирования предусмотрены специальные настройки и ключевые слова, которые позволяют отключить процесс порождения компилятором уникальных имен для лексических единиц языка. Эти слобь 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 уникальньа имен для указанных лексических единиц. В этом случае разработчик программы сам отвечает за уникальность имени данной лексической единицы в пределах всей исходной программы или даже в пределах всего проекта (если используются несколько исходных модулей). Если требование уникальности не будет выполняться, могут возникнуть синтаксические или семантические ошибки на ст^ дии компиляции либо же другие ошибки на более поздних этапах разработк программного обеспечения (эти этапы описаны далее в главе 6 «Современны- 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 язык*2 (тут важно еще учесть соглашения о передаче параметров, которые рассмотрены далее).

[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, особенис когда пользователь сам создал библиотеку и точно знает, что эта функция в ней есть Чаще всего проблема связана с наименованием функций — компилятор дает функция в библиотеке не совсем то имя, которое дал ей пользователь в исходном коде.

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; кроме того, перед распределением памяти надо выполнить идентификацию лексических единиц языка. Поэтому распределение памяти выполняется уже после семантического анализа текста исходной программы.

Процесс распределения памяти в современных компиляторах, как правило, работает с относительными, а не абсолютными адресами ячеек памяти (разница между абсолютными и относительными адресами описана в разделе «Трансляция адресов. Настраивающий загрузчик» в главе 6 «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.

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 элемента 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 испог зуются одновременно.

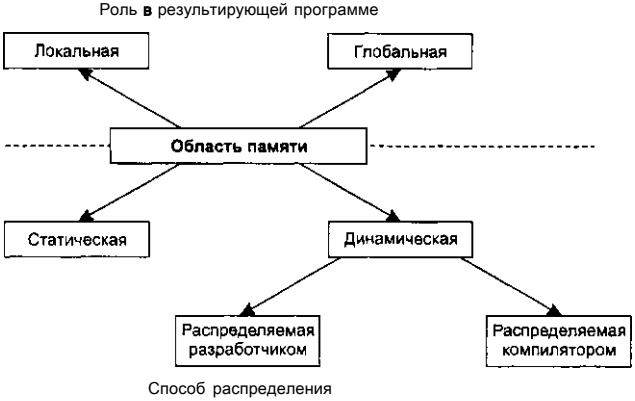


Рис. 5.1. Область памяти в зависимости от ее роли и способа распределения

Виды переменных и областей памяти

Распределение памяти для переменных скалярных типов

Во всех языках программирования существует понятие так называемых «баз. вых типов данных», которые также называют основными или *скалярными* тиг.1 ми). Размер области памяти, необходимый для лексической единицы скалярно: типа, считается заранее известным. Он определяется семантикой языка и арх; тектурой целевой вычислительной системы, на которой должна выполняться созданная компилятором результирующая программа.

Идеальным вариантом для разработчиков программ был бы такой компилятс: у которого размер памяти для базовых типов зависел бы только от семантик, языка. Но чаще всего зависимость результирующей программы от архитектура целевой вычислительной системы полностью исключить не удается. Создатели компиляторов и языков программирования предлагают механизмы, позволяк>- щие свести эту зависимость к минимуму.

СОВЕТ

Размер области памяти каждого скалярного типа данных фиксирован и известе - для определенной целевой вычислительной системы. Однако не рекомендуется неп> средственно использовать его в тексте исходной программы, так как это ограничивав переносимость программы. Вместо этого нужно использовать функции определи ния размера памяти, предоставляемые входным языком программирования.

Например, в языке программирования С 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 С 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 С 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: $V_{Мас} = П_{..1п}(пм) * У_{эл}$,

где

п — размерность массива, п^ — количество элементов i-й размерности, $У_{эл}$ — объем памяти для одного элемента;

для структур: $V_{СТр} = Т_{..ш1}У_{ноия}$.,

где

п — общее количество полей в структуре, $У_{ПОЛЯ}$. — объем памяти для i-го поля Структуры;

для объединений: $V_{СТр} = тах_{Н1п}У_{П0Ля}$.,

где

п — общее количество полей в объединении, $У_{ПЮЛЯ}$ — 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 памят? для вычисления объема памяти, необходимой для одного элемента массива будет вызвана процедура вычисления памяти структуры. Такой подход опреде ления объема занимаемой памяти очень удобен, если декларативная часть язы:-. представлена в виде дерева типов. Тогда для вычисления объема памяти, зан; маемой типом из каждой вершины дерева, нужно вычислить объем памяти д.т- всех потомков этой вершины, а потом применить формулу, связанную непс- средственно с самой вершиной (этот механизм подобен механизму СУ-перевод! применяемому при генерации кода). Как раз такого типа древовидные конструк- ции строит синтаксический анализатор для декларативной части языка.

Выравнивание границ областей памяти

Говоря об объеме памяти, занимаемой различными лексическими единицам:- и структурами данных языка, следует упомянуть еще один момент, связанны} с выравниванием границ областей памяти, отводимых для различных лексиче- ских единиц. Архитектура многих современных вычислительных систем пред;, сматривает, что обработка данных выполняется более эффективно, если адрес по которому выбираются данные, кратен определенному числу байтов (как пра- вило, это 2, 4, 8 или 16 байтов)[1]. Современные компиляторы учитывают особен- ности целевых вычислительных систем. При распределении данных они могут размещать области памяти под лексические единицы наиболее оптимальным образом. Поскольку не всегда размер памяти, отводимой под лексическую еди- ницу, кратен указанному числу байтов, то в общем объеме памяти, отводимой под результирующую 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 памя- ти. Однако если кратность адресов для доступа к памяти установлена 4 байта, тс под эти переменные будет отведено в целом 12 байт памяти, из которых 9 не бу- дут использоваться.

Как правило, разработчику исходной программы не нужно знать, каким образом компилятор распределяет адреса под отводимые области памяти. Чаще всегс компилятор сам выбирает оптимальный метод, и, изменяя границы выделенных областей,-он всегда корректно осуществляет доступ к ним. Вопрос об этом может встать, если с данными программы, написанной на одном языке, работают про- граммы, написанные на другом языке программирования (чаще всего на языке ассемблера), реже такие проблемы возникают при использовании двух различ- ных компиляторов с одного и того же входного языка. Большинство компиля- торов позволяют пользователю в этом случае самому указать, использовать или нет кратные адреса и какую границу кратности установить (если это вообще воз- можно с точки зрения архитектуры целевой вычислительной системы).

[1] Некоторые вычислительные системы вообще не могут работать с адресами, не кратными определенному числу байтов.

Виды областей памяти. Статическое и динамическое связывание

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 даже для глобальных областей памяти. При этом сами области памяти и связанные с ними лексические единицы остаются глобальными, ограничения налагаются только на возможность их использования в тексте исходной программы (и эти ограничения, в принципе, возможно 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 языка исходной программы. Только зная смысл синтаксических конструкций исходного 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
        GlobalJ = 1;
        Global_2 :
var                integer = 2;
        Global I :
                   integer:
function Test (Param: integer): pointer:
const
        Local_l = 1:
```

```
        Local_2 : integer = 2:
var
        Local_I : integer:

begin

end:
```

Согласно семантике языка Pascal, переменная Global_I является глобальной *г.*-ременной языка и размещается в глобальной области памяти, константа Globe _ также является глобальной, но язык не требует, чтобы компилятор обязательн размещал ее в памяти — значение константы может быть непосредственно noi-ставлено в код результирующей программы там, где она используется. Типизи-рованная константа Global_2 является глобальной, но, в отличие от констант: Global_l, семантика языка предполагает, что эта константа обязательно будс размещена в памяти, а не в коде программы. Доступность идентификатор: Global_I, Global_2 и Global_l из других модулей зависит от того, где и как оь описаны. Например, для компилятора Borland Pascal переменные и константа описанные в заголовке модулей, доступны из других модулей программы, а пе-ременные и константы, описанные в теле модулей, недоступны, хотя и те и др> -гие являются глобальными.

Параметр Param функции Test, переменная Local_1 и константа Local_1 являкт локальными элементами этой функции. Они не доступны вне пределов даннс функции и располагаются в локальной области памяти. Типизированная кон-станта Local_2 представляет собой очень интересный элемент программы. С ол-ной стороны, она не доступна вне пределов функции Test и потому является с локальным элементом. С другой стороны, как типизированная константа она б> дет располагаться в глобальной области памяти, хотя ниоткуда извне не може~ быть доступна (аналогичными конструкциями языка C, например, являются лс -кальные переменные функций, описанные как static).

Семантические особенности языка должны учитывать создатели компиляте ра, когда разрабатывают модуль распределения памяти. Безусловно, это должс-знать и разработчик исходной программы, чтобы не допускать семантически, (смысловых) ошибок — этот тип ошибок сложнее всего поддается обнаружение Так, в приведенном выше примере в качестве результата функции Test може~ выступать адрес переменной Global_I или типизированной константы Global l Результатом функции не может быть адрес констант Global_l или Local_l — эт: запрещено семантикой языка. Гораздо сложнее вопрос с переменной Local_1 ил; параметром функции Param. Будучи элементами локальной области памяти функ-ции Test, они никак не могут выступать в качестве результата функции, потом;-что он будет использован вне самой функции, когда область ее локальной памят:-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 факт такого использования адреса локальной переменной). Эти особенности языка должен учитывать разра-ботчик программы. А вот адрес типизированной константы Local_2, в принци-пе, может быть результатом функции Test. Ведь хотя она и является локально;:

константой, но размещается в глобальной области памяти (другое дело, что этим пользоваться не рекомендуется, поскольку нет гарантии, что принцип размещения локальных типизированных констант не изменится с переходом к другой версии компилятора)[1].

Статическая и динамическая память

Статическая область памяти — это область памяти, размер которой известен на этапе компиляции.

Поскольку для статической области памяти известен ее размер, компилятор всегда может выделить эту область памяти и связать ее с соответствующим элементом программы. Поэтому для статической области памяти компилятор порождает некоторый адрес (как правило, это относительный адрес в программе — см. подраздел «Трансляция адресов. Настраивающий загрузчик» в главе 6 «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 непосредственно выделить адрес — для нее он порождает фрагмент кода, который отвечает за распределение памяти (ее выделение и освобождение). Как правило, с динамическими областями памяти связаны многие 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 другие). 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

[1] Рассуждения о возможности или невозможности использования адресов 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.

и освобождение памяти отвечает сам разработчик, а не компилятор. Компилятс: должен только построить код вызова соответствующих функций и сохранен}-:; результата — в принципе, для него работа с динамической памятью пользователя ничем не отличается от работы с любыми другими функциями и дополнительных сложностей не вызывает.

Другое дело — динамические области памяти, выделяемые компилятором. Он:* появляются тогда, когда пользователь использует типы данных, операции над котсрыми предполагают перераспределение памяти, не присутствующее в явном вил: в тексте исходной программы[1]. Примерами таких типов данных могут служит: строки в некоторых языках программирования, динамические массивы и, конечно, многие операции над экземплярами объектов (классов) в объектно-ориенпрованных языках (наиболее характерный пример — тип данных string в версия Borland Delphi языка Object Pascal). В этом случае сам компилятор отвечает :_ порождение кода, который будет всегда обеспечивать своевременное выделен; памяти под элементы программы и освобождать ее по мере использования.
Как статические, так и динамические области памяти сами по себе могут быт-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 требе - вании на выделение оперативной памяти он запрашивает у ОС область памят. намного большего объема, чем это первоначально необходимо результирующее программе. Зато для всех последующих требований на выделение оперативно, памяти менеджер памяти стремится выделить фрагменты из уже имеющейс.- 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 рас - поряжении память распределена, менеджер памяти вновь запрашивает у ОС область памяти заведомо большего размера, чем требуется результирующей программе. По мере выполнения результирующей программы менеджер памят; следит за использованием всех находящихся в его распоряжении областей памят; 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 выде-

[1] В принципе, в этом случае разработчик исходной программы может даже и не подозревать о том, что используемая им операция предполагает перераспределение памяти. Однакс это не лучшим образом будет сказываться на эффективности созданной разработчиком программы.

ленной 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 в результирующей программе 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 результирующей 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. :- должен быть сохранен на все время ее выполнения.

Процедура должна работать со своими данными однотипным образом вне завис мости от того, из какого места результирующей программы она была вызвана (от:-: да произошло обращение к этой процедуре). Поэтому компилятор должен со^. - ветствующим образом организовать обращение к данным, доступным процедура С глобальными данными никаких проблем не возникает — их адреса известит, они все устанавливаются на этапе распределения памяти. Даже если память г.. _ эти данные распределяется динамически, компилятору однозначно известно, к^- к этой памяти обратиться, и способ обращения одинаков для всей программа Поэтому он всегда может создать код для обращения к этим данным вне завис; мости от того, откуда это обращение происходит.

Сложнее с локальными параметрами и переменными процедуры. Они относятс; к локальной памяти процедуры и потому должны быть как-то связаны с нее» Существует несколько вариантов связывания локальных переменных и паре метров с кодом соответствующей процедуры или функции [15]. Современны: вычислительные системы ориентированы, главным образом, на стековую орга- низацию дисплея памяти процедуры (функции). Поэтому компиляторы при со:- - дании результирующей программы используют именно этот вариант.

Стековая организация дисплея памяти процедуры (функции)

Стековая организация дисплея памяти процедуры (функции) основана на TON: что для хранения параметров (аргументов) процедур и функций, их локальны переменных, а также адреса возврата в результирующей программе выделяете - одна на всю программу специальная область памяти, организованная в виде стек! Этот стек называют стеком передачи параметров, или просто стеком параметроЕ При стековой организации дисплея памяти процедуры или функции в момент е вызова все параметры и адрес возврата помещаются в стек параметров. При за- вершении выполнения процедуры или функции параметры извлекаются («вы- талкиваются») из стека, а управление передается по адресу возврата.
Внутри вызываемой процедуры или функции на верхушке стека может быть вы- делено место для хранения всех ее локальных переменных. Объектный код про- цедуры или функции адресуется к параметрам и своим локальным переменных, по смещениям относительно верхушки стека.

В этой схеме компилятор организует единый для всей программы стек пара- метров. Верхушка стека адресуется одним из регистров процессора — регистров стека. Для доступа к данным удобно использовать еще один регистр процессо- ра, 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;
- ~ ить в стеке адрес возврата и передать управление вызываемой процедуре.

В начале выполнения процедура или функция 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 другими. 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.

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 воз-

врата из стека). Такой подход значительно упрощает и ускоряет выполнение вызовов при стековой организации дисплея памяти процедуры (функции).

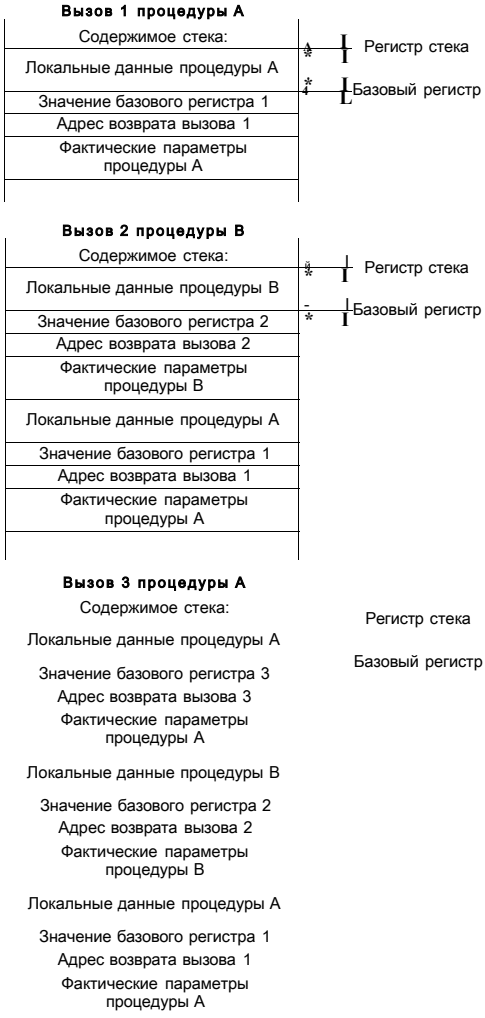


Рис. 5.2. Содержание стека параметров при выполнении трех последовательных вызовов процедур

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, большую часть времени своей работы результирующая программа не использует значительную часть стека, а значит, выделенная для этой цели часть оперативной памяти компьютера 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 отличаться 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.

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[2]. 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.

[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.

[2] Существуют 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.

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 выполнения функции. Как правило, для этой цели компилятор использует определенный регистр процессора, чаще всего — регистр аккумулятора, если он существует в процессоре целевой вычислительной архитектуры. Если разные компиляторы используют разные регистры для этой цели могут возникнуть проблемы с передачей результата выполнения функции, вызванной из библиотеки, созданной другим компилятором. Однако разработчики компиляторов стараются соблюдать общепринятые соглашения о вызове, поэтому проблем с передачей результата выполнения функции при использовании известных компиляторов не возникает.

Исключительные ситуации и их обработка

Понятие исключительной ситуации

Понятие *исключительной ситуации* (exception) появилось в современных объектно-ориентированных языках программирования.

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 вызывае-

мых 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 ут нию разработчика.

Таким образом, в механизме порождения и обработки исключительных сит>\ может быть задействован не только объектный код результирующей програ> но и код других компонентов целевой вычислительной системы (ОС, систем библиотек, драйверов устройств), где выполняется результирующая програ* Следовательно, обработка исключительных ситуаций зависит от типа целеЕ вычислительной системы. То есть компилятор должен включать в результир\ • щую программу код порождения и обработки исключительных ситуаций в за: симости от типа вычислительной системы, на которую она ориентирована.

Обработчики исключительных ситуаций

За обработку исключительных ситуаций отвечают специальные синтаксичес конструкции, предусмотренные в объектно-ориентированных языках прогр мирования. Примерами таких конструкций являются блоки типа try ... excep: и try ... finally ... в языке программирования Pascal, блоки типа throw ... catc 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 обработчик* исключительных ситуаций, специальные синтаксические конструкции и семаЕтические правила в каждом языке программирования служат для определена? 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 люба^ 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 разработчю 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 ИЛЕ 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, корректнс

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 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_t;
- помещает в ячейку E_t текущее значение регистра стека.

Для конца блока обработки исключительных ситуаций (если исключительная ситуация не произошла) компилятор доложен порождать объектный код, который выполняет следующее:

- извлекает из стека предыдущий адрес, хранившийся в ячейке E_n и помещает его в ячейку E^
- 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_t помещается в регистр стека;
- из стека извлекается предыдущий адрес, хранившийся в ячейке E_n и записывается в ячейку E_t;
- 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_t, 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 исключитель-

ных ситуаций выполняет действия по освобождению выделенных компилятором динамических областей памяти для всех типов исключительных ситуаций.

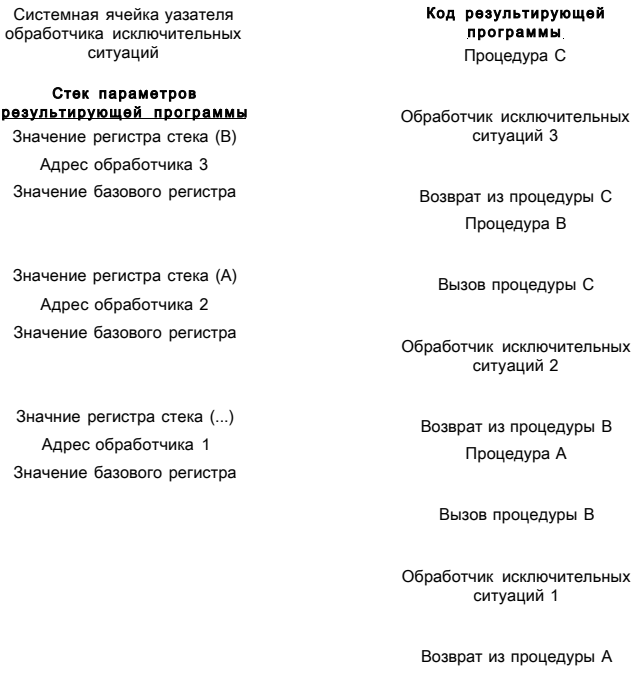


Рис. 5.3. Простейший механизм обработки исключительных ситуаций через стек параметров

Например, в языке Object Pascal 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 EZeroOivide 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 или 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, 6, 15, 25, 26, 27, 34, 42, 43, 51]. Эти структуры данных напрямую не доступны 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 [25, 26, 27, 34, 42, 43, 51].

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 взаимо-

связи 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.
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 большие 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 результирующей программы. В общем случае организовать такого рода взаимодействие между двумя фрагментами кода, порожденного двумя различными компиляторами, напрямую практически невозможно (о других возможностях такого рода взаимодействия см. в главе 6 «Современные системы программирования»).

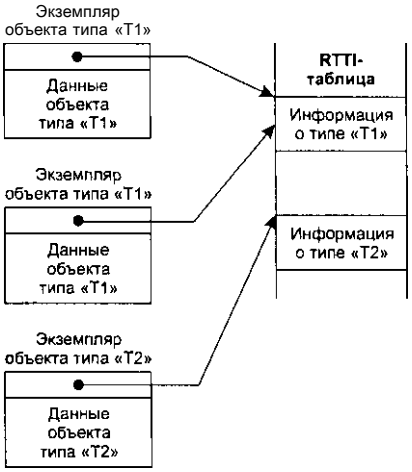


Рис. 5.4. Взаимосвязь RTTI-таблицы с экземплярами объектов в результирующей программе

Генерация кода. Методы генерации кода

Общие принципы генерации 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 машинном языке (в машинных кодах). 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: распределено адресное пространство под функции и переменные, проверено соответствие имен и типов переменных, констант и функций в синтаксических конструкциях исходной программы и т. д. Характер отображения входной программы в последовательность команд, выполняемую генерацией, зависит от входного языка, архитектуры вычислительной системы, на которую ориентирована результирующая программа, 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 типовой результирующий код.

Синтаксически управляемый перевод

Чтобы компилятор мог построить код результирующей программы для синтаксической конструкции входного языка, 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 синтаксически управляемого 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, значение которых человек обычно понимает из того контекста, в котором эти фразы встречаются (и то он не всегда 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, посвященным 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 ассемблера, 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. То 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 результирующей программы на основании результатов синтаксического анализа. Для удобства понимания сути метода можно считать, что результат 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)

[1] Фамилия этого известного лингвиста уже встречалась, когда речь шла о типах языков и грамматик и их классификации.

в фиксированном порядке последовательности кода C(N) и последовательностей кодов, связанных со всеми вершинами, являющимися прямыми потомками N. Д 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 бы встречалась 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 из широко распространенных классов КС-языков, в частности для LR и LL языков [4, т. 1, 2].

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 машинных кодов или команд ассемблера, представляющих собой результат работы (выход) компилятора;
- выдачу пользователю сообщений об обнаруженных ошибках и предупреждениях (которые должны помещаться в 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 найденной 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 несут никакой смысловой нагрузки и 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» и «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.

Известны 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);**

[1] **Существуют 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.**

- обратная (постфиксная) польская запись операций;
- ассемблерный код или машинные команды.

В каждом конкретном компиляторе может использоваться одна из этих форм, 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. 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 — иначе он не сможет построить результирующую программу. Далее будут рассмотрены различные формы внутреннего представления программы, начиная от связных списочных структур.

Синтаксические деревья. Преобразование дерева разбора в дерево операций

Связные списочные структуры, представляющие 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 программирования, 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 опе-

раторов, но после построения дерева никакой смысловой нагрузки 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» — языка входной программы. 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 (а+а)*Ь 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 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 (а и Ь) 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 итоге всегда имеет одну и ту же структуру, зависящую только от семантики входного языка.

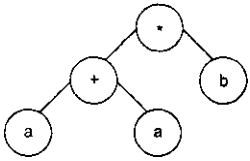


Рис. 5.5. Пример дерева операций для языка арифметических выражений

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 результирующей 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, 15].
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 результатов 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 и ее реализации).

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 (либо последовательность команд ассемблера). В этом их преимущество перед синтаксическими деревьями. А в отличие от команд ассемблера, тетрады не зависят от архитектуры вычислительной системы, на которую ориентирована результирующая программа. Поэтому они 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:

1. * (B, C. T1)
2. + (T1, D. T2)
3. * (B, 10. T3)
4. - (T2. T3. T4)
5. := (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 результат выполнения другой триады. Поэтому триады при записи последовательно нумеруют для удобства указания ссылок одних триад на другие (в реализации компилятора в качестве ссылок можно использовать не номера триад, а непосредственно ссылки в виде указателей —

тогда при изменении нумерации и 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 и ее реализации). 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 для них несложно написать тривиальный алгоритм, который будет преобразовывать последовательность триад в последовательность команд результирующей программы (либо последовательность команд ассемблера). В этом их преимущество перед синтаксическими деревьями. Однако здесь требуется также и алгоритм, отвечающий за распределение памяти, необходимой для хранения промежуточных результатов вычисления, так как временные переменные для этой цели не используются. В этом отличие триад от тетрад.

Так же как и тетрады, триады не зависят от архитектуры вычислительной системы, на которую ориентирована результирующая программа. Поэтому они 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 удобным. Необходимость иметь алгоритм, отвечающий за распределение памяти для хранения промежуточных результатов, не является недостатком, так как удобно распределять результаты не только по доступным ячейкам временной памяти, но и по имеющимся регистрам процессора. Это дает определенные преимущества. Триады ближе к двухадресным машинным командам, чем тетрады, а именно эти 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:

1. * (B, C)
2. + Г1, D)
3. * (B. 10)
4. - Г2, ^{A}3)
5. := (A. ^{A}4)

Здесь операции обозначены соответствующим знаком (при этом присвоение также является операцией), а знак ^A означает ссылку операнда одной триады на результат другой.

Обратная польская запись операций

Обратная (постфиксная) польская запись — очень удобная для вычисления выражений форма записи операций и операндов. Эта форма предусматривает, что знаки операций записываются после операндов.

Более подробно обратная польская запись 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»), 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 вычислительной системы, на которую ориентирован результирующий код. 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 работать с ними. Кроме того, только обрабатывая машинные команды (или их представление в форме команд ассемблера), можно добиться наиболее эффективной результирующей программы (см. далее раздел «Оптимизация кода. Основные методы оптимизации» главы 2). 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

Обратная польская запись — это постфиксная запись операций. Она была предложена польским математиком Я. Лукашевичем, откуда и происходит ее название [4, т. 2, 17, 52][1].

[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.

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. 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[1]. Для этого достаточно ввести знаки, предусматривающие вычисление соответствующих операций. В том числе, возможно ввести операции условного и безусловного перехода, предполагающие изменение последовательности хода вычислений и перемещение вперед или назад на некоторое количество шагов в зависимости от результата на верхушке стека [4, т. 2, 50, 52]. 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 (попадает на верхушку стека);
2. если встречается знак унарной операции (операции, требующей одного операнда), то операнд выбирается с верхушки стека, операция выполняется, и результат помещается в стек (попадает на верхушку стека);

[1] Язык программирования Forth является хорошим примером языка, где для вычисления любых выражений явно используются стек и обратная польская запись операций.

3. если встречается знак бинарной операции (операции, требующей двух операндов), то два операнда выбираются с верхушки стека, операция выполняется, и результат помещается в стек (попадает 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  I  T
T    T*E  I  T/E  I  E
E     (S)  |  a  |  b
```

Схему СУ-компиляции будем строить из расчета, 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. 1,2]. 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 (X):

```
S    S+T;  R(p) = "+".  p = p + 1
S -> S-T;  R(p) =       p = p + 1
S    T: X
T    T*E;  R(p) = "*",  p = p + 1
T    T/E;  R(p) = "/",  p = p + 1
T     E;  X
E    (S): X
E    a:  R(p) = "a",  p = p + 1
E -> b:  R(p) = "b".  p - p + 1
```

Эту схему СУ-компиляции можно использовать для любого распознавателя без возвратов, допускающего разбор входных цепочек на основе правил данной грамматики (например, для распознавателя на основе грамматики операторного предшествования, рассмотренного в разделе «Восходящие распознаватели КС-языков без возвратов»). 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]). 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 сам принцип СУ-компиляции.

Схемы СУ-перевода

Выше был описан принцип СУ-перевода, позволяющий получить линейную последовательность команд результирующей программы или внутреннего пред-

ставления программы в компиляторе на основе результатов синтаксического анализа. Теперь построим вариант алгоритма генерации кода, который получает на входе дерево операций и создает по нему фрагмент объектного кода для линейного участка результирующей программы. Рассмотрим примеры схем СУ-перевода для арифметических операций. Эти схемы достаточно просты, и на их основе можно проиллюстрировать, как выполняется СУ-перевод в компиляторе при генерации кода.

В качестве формы представления результатов синтаксического разбора возьмем дерево операций (метод построения дерева операций рассмотрен выше). Для построения внутреннего представления объектного кода (в дальнейшем — просто кода) по дереву операций может использоваться простейшая рекурсивная процедура обхода дерева. Можно использовать и другие методы обхода дерева — важно, чтобы соблюдался принцип, что нижележащие операции в дереве всегда выполняются перед вышележащими операциями (порядок выполнения операций одного уровня не важен, он не влияет на результат и зависит от порядка обхода вершин дерева).

Процедура генерации кода по дереву операций, прежде всего, должна определить тип узла дерева. Он соответствует типу операции, символом которой помечен узел дерева. После определения типа узла процедура строит код для узла дерева в соответствии с типом операции. Если все узлы следующего уровня для текущего узла есть листья дерева, то в код включаются операнды, соответствующие этим листьям, и получившийся код становится результатом выполнения процедуры. Иначе процедура должна рекурсивно вызвать сама себя для генерации кода нижележащих узлов дерева и результат выполнения включить в свой порожденный код. Фактически, процедура генерации кода должна для каждого узла дерева выполнить конкатенацию цепочек команд, связанных с текущим узлом и с нижележащими узлами. Конкатенация цепочек выполняется таким образом, чтобы операции, связанные с нижележащими узлами, выполнялись до выполнения операции, связанной с текущим узлом.

Поэтому для построения внутреннего представления объектного кода по дереву вывода в первую очередь необходимо разработать формы представления объектного кода для четырех случаев, соответствующих видам текущего узла дерева вывода:

- оба нижележащих узла дерева — листья, помеченные операндами;
- только левый нижележащий узел является листом дерева;
- только правый нижележащий узел является листом дерева;
- оба нижележащих узла нё являются листьями дерева.

Рассмотрим построение двух видов внутреннего представления по дереву вывода:

1. построение списка триад по дереву вывода;
2. построение ассемблерного кода по дереву вывода.

Далее рассматриваются две функции, реализующие схемы СУ-перевода для каждого их этих случаев.

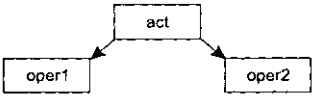
Пример схемы СУ-перевода дерева операций на язык ассемблера

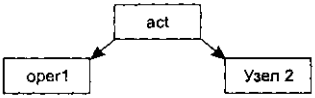
В качестве языка ассемблера возьмем язык ассемблера процессоров типа Intel 80x86. При этом будем считать, что операнды могут быть помещены в 16-разрядные регистры процессора, и в коде результирующей объектной программы могут использоваться регистры AX (аккумулятор) и DX (регистр данных), а также стек для хранения промежуточных результатов.

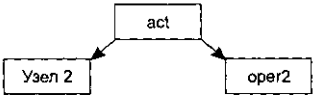
Функцию, реализующую перевод узла дерева в последовательность команд ассемблера, назовем Code. Входными данными функции должна быть информация об узле дерева операций. В ее реализации на каком-либо языке программирования эти данные можно представить в виде указателя на соответствующий узел дерева операций. Выходными данными является последовательность команд языка ассемблера. Будем считать, что она передается в виде строки, возвращаемой в качестве результата функции.

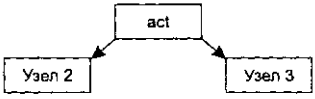
Тогда четырем формам текущего узла дерева для каждой арифметической операции будут соответствовать фрагменты кода на языке ассемблера, приведенные в табл. 5.1. I

Таблица 5.1. Преобразование узлов дерева вывода в код на языке ассемблера для арифметических операций

Вид узла дерева	Результирующий код	Примечание
	`mov ax, oper1` `act ax, oper2`	act — команда соответствующей операции oper1, oper2 — операнды (листья дерева)

	`Code (Узел 2)`
	`mov dx, ax`
	`mov ax, oper1`
	`act ax, dx`
		Узел 2 — нижележащий узел (не лист!) дерева
		Code (Узел 2) — код, порождаемый процедурой для нижележащего узла

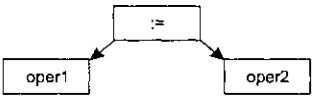
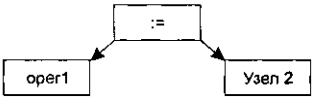
	`Code (Узел 2)` `act ax, oper2`	Code (Узел 2) — код, порождаемый процедурой для нижележащего узла

	`Code (Узел 2)`
	`push ax`
	`Code (Узел 3)`
	`mov dx, ax`
	`pop ax`
	`act ax, dx`
		Code (Узел 2) — код, порождаемый процедурой для нижележащего узла
		Code (Узел 3) — код, порождаемый процедурой для нижележащего узла
		push и pop — команды сохранения результатов в стеке и извлечения результатов из стека

Каждой допустимой арифметической операции будет соответствовать своя команда на языке ассемблера. Если взять в качестве примера операции сложения (+), вычитания (-), умножения (*) и деления (/), то им будут соответствовать команды add, sub, mul и div. Причем в ассемблере Intel 80x86 от типа операции зависит не только тип, но и синтаксис команды (операции mul и div в качестве первого операнда всегда предполагают регистр процессора AX, который не нужно указывать в команде). Соответствующая команда должна записываться вместо act при порождении кода в зависимости от типа узла дерева.

Код, порождаемый для операции присвоения результата, будет отличаться от кода, порождаемого для арифметических операций. Кроме того, семантика языка требует, чтобы в левой части операции присвоения всегда был операнд, поэтому для нее возможны только два типа узлов дерева, влекущие порождение кода (два других типа узлов должны приводить к сообщению об ошибке, которая в компиляторе обнаруживается синтаксическим анализатором). Фрагменты кода для этой операции приведены ниже в табл. 5.2.

Таблица 5.2. Преобразование узлов дерева вывода в код на языке ассемблера для операции присвоения

Вид узла дерева	Результирующий код	Примечание
	mov ax, oper2 mov oper1, ax	oper1, oper2 — операнды (листья дерева)
	Code (Узел 2) mov oper1, ax	Узел 2 — нижележащий узел (не лист!) дерева Code(Узел 2) — код, порождаемый процедурой для нижележащего узла

Теперь последовательность порождаемого кода определена для всех возможных типов узлов дерева.

Рассмотрим в качестве примера выражение A:=B*C + D- B*10. Соответствующее ему дерево вывода приведено на рис. 5.7.

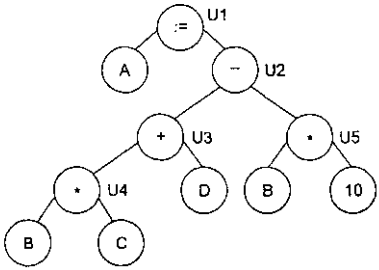


Рис. 5.7. Дерево операций для арифметического выражения «A := B * C + D – B * 10

Построим последовательность команд языка ассемблера, соответствующую дереву операций на рис. 5.7. Согласно принципу СУ-перевода, построение начинается от корня дерева. Для удобства иллюстрации рекурсивного построения последовательности команд все узлы дерева помечены от U1 до U5. Рассмотрим последовательность построения цепочки команд языка ассемблера по шагам рекурсии. Эта последовательность приведена ниже.

```
Шаг 1:Code(U2)
          mov A, ax
Шаг 2:Code(U3)
          push ax
          Code(U5)
          mov dx. ax
          pop ax
          sub ax. dx
          mov A. ax
Шаг 3:Code(U4)
          add ax. D
          push ax
          Code(U5)
          mov dx. ax
          pop ax
          sub ax, dx
          mov A,ax
Шаг 4:mov ax. B
          mul C
          add ax. D
          push ax
          Code(U5)
          mov dx. ax
          pop ax
          sub ax. dx
          mov A, ax
Шаг 5: mov ax, B
          mul C
          add ax, D
          push ax
          mov ax. B
          mul 10
          mov dx. ax
          pop ax
          sub ax, dx
          mov A, ax
```

Полученный объектный код на языке ассемблера, очевидно, может быть оптимизирован. Вопросы оптимизации результирующего кода рассмотрены далее в разделе «Оптимизация кода. Основные методы оптимизации».

В рассмотренном примере при порождении кода преднамеренно не были приняты во внимание многие вопросы, возникающие при построении реальных компиляторов. Это было сделано для упрощения примера. Например, фрагменты кода, соответствующие различным узлам дерева, принимают во внимание тип операции, но никак не учитывают тип операндов. А в языке ассемблера операндам различных типов соответствуют существенно отличающиеся команды, могут использоваться различные регистры (например, если сравнить операции с целыми числами и числами с плавающей точкой). Некоторые типы операндов (например, строки) не могут быть обработаны одной командой языка, а требуют целой последовательности команд. Такую последовательность разумнее оформить в виде функции библиотеки языка, тогда выполнение операции будет представлять собой вызов этой функции.

Кроме того, ориентация на определенный язык ассемблера, очевидно, сводит на нет универсальность метода. Каждый тип операций жестко связывается с последовательностью определенных команд, ориентированных на некоторую архитектуру вычислительной системы.

Все эти требования ведут к тому, что в реальном компиляторе при генерации кода надо принимать во внимание очень многие особенности, зависящие как от семантики входного языка, так и от архитектуры целевой вычислительной системы, на которой будет выполняться результирующая программа. В результате увеличится число различных типов узлов дерева, каждому из которых будет соответствовать своя последовательность команд. Это может осложнить алгоритм генерации кода, что особенно нежелательно в том случае, если порождаемый код не является окончательным и требует дальнейшей обработки.

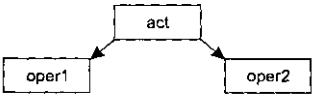
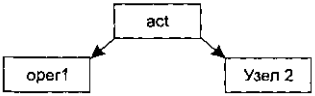
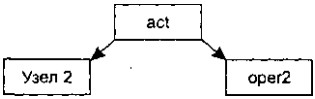
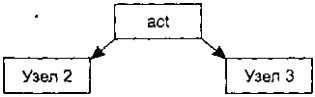
Указанных проблем можно избежать, если порождать код не в виде цепочки команд ассемблера, а в виде последовательности команд машинно-независимого метода внутреннего представления программы — например, тетрад или триад.

Пример схемы СУ-перевода дерева операций в последовательность триад

Как и в случае с порождением команд ассемблера, функцию, реализующую перевод узла дерева в последовательность триад, назовем Code. Входные и выходные данные у этой функции будут те же, что и при работе с командами ассемблера. Триады являются машинно-независимой формой внутреннего представления в компиляторе результирующей объектной программы, а потому не требуют оговорки дополнительных условий при генерации кода. Триады взаимосвязаны между собой, поэтому для установки корректной взаимосвязи функция генерации кода должна учитывать также текущий номер очередной триады (i).

Будем рассматривать те же варианты узлов дерева, что и для команд ассемблера. Тогда четырем формам текущего узла дерева будут соответствовать последовательности триад объектного кода, приведенные в табл. 5.3.

Таблица 5.3. Преобразование типовых узлов дерева вывода в последовательность триад

Вид узла дерева	Результирующий код	Примечание
	i) act (oper1,oper2)	act — тип триады oper1, oper2 — операнды (листья дерева вывода)
	i) Code (Узел 2, i) i+j) act (oper1, ^i+j-1)	Узел 2 — нижележащий узел дерева вывода Code (Узел 2, i) — последовательность триад, порождаемая для Узла 2, начиная с триады с номером i, j — количество триад, порождаемых для Узла 2
	i) Code (Узел 2. i) i+j) act П+j-1. oper2)	Узел 2 — нижележащий узел дерева вывода Code (Узел 2, i) - последовательность триад, порождаемая для Узла 2, начиная с триады с номером i, j — количество триад, порождаемых для Узла 2
	i) Code (Узел 2. i) i+j) Code (Узел 3. i+j) i+j+k) act ("i+j-1, ^i+j+k-1)	Узел 2, Узел 3 - нижележащие узлы дерева вывода Code (Узел 2, i) - последовательность триад, порождаемая для Узла 2, начиная с триады с номером i, j — количество триад, порождаемых для Узла 2 Code (Узел 3, i+j) - последовательность триад, порождаемая для Узла 3, начиная с триады с номером i+j, к — количество триад, порождаемых для Узла 3

Для триад не требуется учитывать операцию присваивания как отдельный тип узла дерева. Она преобразуется в триаду обычного вида.

Рассмотрим в качестве примера то же самое выражение/A := B * C + D - B * 10. Соответствующее ему дерево вывода уже было приведено на рис. 5.7.

Построим последовательность триад, соответствующую дереву операций на рис. 5.7. Согласно принципу СУ-перевода, построение начинается от корня дерева.

```
Шг 1:  1)    Code (U2, 1)
       1)    := (A. Л1-1)
Шг 2:  1)    Code (U3. 1)
       J)    Code (U5. j)
       1-1)  - П-1. Л1-2)
       1)    := (A. Л1-1)
Шг 3:  1)    Code (U4, 1)
       к)    + Гк-1, D)
       J)    Code (U5, j)
       1-1)  - П-1. Л1-2)
       1)    := (A. Л1-1)
Шг 4:  1)    * (B. C)
       2)    + И, D)
       3)    Code (U5, 3)
       1-1)  - rj-1. Л1-2)
       1)    := (A. A1-1)
Шг 5:  1)    * (B. C)
       2)    + (A1. D)
       3)    * (B. 10)
       4)    - Г2. A3)
       5)      (A. Л4)
```

Следует обратить внимание, что в данном алгоритме последовательные номера триад (а следовательно, и ссылки на них) устанавливаются не сразу. Это не имеет значения при рекурсивной организации процедуры, но если используется другой способ обхода дерева операций, при реализации генерации кода лучше увязывать триады между собой именно по ссылке (указателю), а не по порядковому номеру. Использование триад позволяет сократить количество рассматриваемых вариантов узлов дерева операций. В этом случае не требуется принимать во внимание типы операндов и ориентироваться на архитектуру целевой вычислительной системы. Все эти аспекты необходимо будет принимать во внимание уже на этапе преобразования триад в последовательность команд ассемблера или непосредственно в последовательность машинных команд результирующей программы.

Как и последовательность команд ассемблера, последовательность триад можно оптимизировать. Для триад разработаны универсальные (машинно-независимые) алгоритмы оптимизации кода. Уже после их выполнения (оптимизации внутреннего представления) триады могут быть преобразованы в команды на языке ассемблера.

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 синтаксиче-

ской 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 бы являлся 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]. 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, и 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).

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, 50, 52]. Таким образом, оптимизация в компиляторе может выполняться несколько раз на этапе генерации кода.

[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.

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. Так, 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 предусматривают 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]. 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 присваивания 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 число раз, 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 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 присваивания значения некоторой произвольной переменной A с номером i, а также операция присваивания 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 присваивания 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 присвоения, 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 присваивания 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 присваивания A := B * C; уже не является бесполезной, хотя это и не столь очевидно. В этом случае неверно следовать простому принципу, заключающемуся в том, что если переменная, которой присвоено значение в операции с номером i, не встречается ни в одной операции между i и j, то операция присваивания 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 обрабатыва-

ют 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 возможно 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(27ta): 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 Т, 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 С(К.О).

Алгоритм свертки триад последовательно просматривает триады линейного участка и для каждой триады делает следующее:

1. если операнд есть переменная, которая содержится в таблице Т, то операнд заменяется соответствующим значением константы;
2. если операнд есть ссылка на особую триаду типа С (К. 0), то операнд заменяется значением константы К;
3. если все операнды триады являются константами, то триада может быть свернута. Тогда данная триада выполняется, и вместо нее помещается особая триада вида С(К.О), где К — 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 А := В, тогда:
   - если В — константа, то А со значением константы заносится в таблицу Т (есл; там уже было старое значение для А, то это старое значение исключается);
   - если В — не константа, то А вообще исключается из таблицы Т, если онс там есть.

Рассмотрим пример выполнения алгоритма.

Пусть фрагмент исходной программы (записанной на языке типа Pascal) имеет вид:

```
I := 1 + 1:
I := 3:
J := 6*1 + I:
```

Ее внутреннее представление в форме триад будет иметь вид:

1. + (1.1)
2. := (1.^A1)
3. := (1.3)
4. * (6.1)
5. + Г4.1)
6. := (J.^A5)

Процесс выполнения алгоритма свертки показан в табл. 5.4.

Таблица 5.4. Пример работы алгоритма свертки

Триада	Шаг 1	Шаг 2	ШагЗ	Шаг 4	Шаг 5	Шаг 6
1	С (2, 0)	С (2, 0)	С (2, 0)	С (2, 0)	С (2, 0)	С (2, 0)
2	:= (1,^A1)	:= (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)	С (18, 0)	С (18, 0)	С (18, 0)
5	+ (^a4,I)	+ (^a4,I)	+ (^A4,I)	+ (^A4,I)	С (21, 0)	С (21, 0)
6	:= С. ^a5)	а ^a5)	С- ^a5)	а ^a5)	:= С- ^a5)	а 21)
Т	(,)	(I, 2)	(I. 3)	(I, 3)	а 3)	(I, 3) а 2i)

Если исключить особые триады типа С(К,0) (которые не порождают объектного кода), то в результате выполнения свертки получим следующую последовательность триад:

1. := (1.2)
2. := (1.3)
3. := (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 результирующей программы. Поэтому для нелинейных участков программы (ветвлений и циклов) алгоритм будет более сложным, чем последовательный просмртр 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 C 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 C j .0), то он заменяется на ссылку на триаду с номером j (^{A}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. O);
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.^{A}2)
4. * (C.B)
5. + (D,^{A}4)
6. := (A,^{Л}5)
7. * (C.B)
8. + (D.^{A}7)
9. := (C,^{Л}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. 5.5.

Таблица 5.5. 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, ^{Л}1)	0	0	0	0	2	2) + ф, ^{л}1)
3) := (D, ^{л}2)	0	0	0	3	3	3) := (D, ^{л}2)
4) * (C, B)	0	0	0	3	1	4) SAME (1, 0)
5) + (D, ^{л}4)	0	0	0	3	4	5) + (D,M)
6) := (A, ^{A}5)	6	0	0	3	5	6) (A, ^{л}5)
7) * (C, B)	6	0	0	3	1	7) SAME (1, 0)
8) + (D, ^{л}7)	6	0	0	3	4	8) SAME (5, 0)
9) (C, ^{л}8)	6	0	9	3	5	9) := (C, ^{л}5)

Теперь, если исключить триады особого вида SAME (j. O), то в результате выполнения алгоритма получим следующую последовательность триад:

1. * (C,B)
2. + (D.^{A}1)
3. := (D.^{A}2)
4. + (D.^{A}l)

5. := (A.[A]4)

6. := (C.[Л]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 изменение 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 One 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, как правило, говорят 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» (true); 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, то 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 результирующего 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» главы 5). 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 результирующей 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, 21].

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 оказываются 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 1:=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.

[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.

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.

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, 21, 52].

Например, цикл

```
S := 10;
for i:=1 to N do A[i] := i*S:
```

может быть заменен последовательностью операций

```
S := 10;
T := S; 1 := 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 -. 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 индуктивных переменных — i и S. Если их 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 достаточно 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, 21].

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 ACT.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 ACT.i] := 0;
```

Развертывание циклов можно выполнить для циклов, кратность выполнения которых известна уже на этапе компиляции. Тогда цикл кратностью N можно заменить линейной последовательностью из 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 возможно 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 системы 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, 31, 50, 52]. 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 назначения для 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. Результат 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. Тогда 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 проанализировать 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 + FНа 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 + ВиО + 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 уже 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 контекстно-свободным языком;
   - синтаксис любого языка программирования может быть описан с помощью контекстно-свободной грамматики;
   - любой язык программирования является контекстно-зависимым языком;
   - семантика любого языка программирования может быть описана с помощью контекстно-свободной грамматики?
2. Какие задачи в компиляторе решает семантический анализ? 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 контекстно-зависимых языков. При этом не будет требоваться дополнительный семантический анализ цепочки. Почему такой подход не применяется в компиляторах на практике?

4. На языке С описаны переменные:

```
int i.j.k; /* целочисленные переменные */
doiihlp x.v.z: /* вещественные переменные */
```

Объясните, есть ли различия в объектном коде, порождаемом компилятором для перечисленных ниже пар операторов, и если есть, то в чем они заключаются:

```
O i = X/ У: i = (int)(x / y):
O i = X/ У: i = ((int)x) / ((int)y);
O i = j / к: i = ((double)j) / (double)k);
O z - X/ y; z=((int)x) / ((int)y):
O z = j / к: z = (double)Cj / k);
O z = j / к: z = ((double)j) / ( (double)k)
```

Объясните, какие функции по преобразованию типов должен будет включить в объектный код компилятор в каждом из перечисленных случаев. В каких случаях выполняется неявное преобразование типов?

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. Почему распределение памяти не может быть выполнено до выполнения семантического анализа? Как работает процесс распределения памяти с таблицей идентификаторов?

7. Может ли компилятор для языков С и Pascal использовать статическую схему организации дисплея памяти процедуры (функции)? А динамическую схему?

8. 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? Требуются ли какие-либо изменения в стековой организации дисплея памяти в таком случае?

9. От чего зависит состав информации, хранящейся в таблице RTTI? Регламентируется ли состав информации в этой таблице синтаксисом или семантикой исходного языка программирования? Зависит ли информация в таблице RTTI от архитектуры целевой вычислительной системы?

10. Почему в большинстве компиляторов помимо генерации результирующего объектного кода выполняется еще и оптимизация его? Можно ли построить компилятор, исключив фазу оптимизации кода?

11. От чего зависит эффективность объектного кода, построенного компилятором? Влияют ли на эффективность результирующего кода синтаксис и семантика исходного языка программирования?

12. В чем заключается основной принцип СУ-перевода? Является ли СУ-перевод наиболее эффективным методом порождения результирующего кода?
13. Сравните между собой основные способы внутреннего представления программы. На каких этапах компиляции лучше всего использовать каждый из этих способов? Какие (или какой) из способов внутреннего представления программы обязательно должен уметь обрабатывать компилятор?
14. Почему имеются трудности с оптимизацией выражений, представленных в форме обратной польской записи?
15. От чего зависит код, порождаемый процедурой СУ-перевода для каждого узла дерева операций? Должна ли процедура СУ-перевода обращаться к таблице идентификаторов? Должна ли процедура СУ-перевода учитывать тип операндов? Как зависят действия этой процедуры от типа порождаемого ею внутреннего представления программы?
16. 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?
17. Почему линейные участки легче поддаются оптимизации, чем все другие части внутреннего представления программы и объектного кода?
18. Почему можно утверждать, что исключение из исходного языка операций с указателями и адресами значительно усиливает возможности компилятора по оптимизации кода?
19. Массив A содержит 10 элементов (индексы от 0 до 9). Будет ли условный оператор языка Pascal if Ci<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?
20. 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:

    O 9

    O И

    O 21

    O O

    O -2?

    Как изменится значение переменной при выполнении этого оператора, если предположить, что при порождении объектного кода компилятор не выполняет оптимизацию логических выражений?
21. Какие варианты распараллеливания выполнения операций может использовать компилятор при порождении объектного кода, необходимого для вычисления выражения A * B * C + A * D / E - B / E - F ?

Задачи

1. **Даны две грамматики (см. задачи к предыдущей главе):**

   Gl({ (,) A&.-.a}, {S.R.T.G.E }.P1. S)
   P1:
   S -> TR
   R —> X | | ATR
   T -> FG
   G -> X | &E | &FG
   E —> ~E j (S) | a

   G2({(.),\&.~,a}.{S .T.EJ.P2.S)
   P2:
   S -> S'T I т
   T —> T&E I E
   E -> ~E | (S) | a

   Постройте для каждой из них цепочку вывода для цепочки символов: a^Aa&~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. **Функции описаны на языке Pascal следующим образом:**

   ```
   procedure A(i : integer; j: integer);
   var t; float;
   begin

   end;

   procedure B:
   var a.b.c: integer;
   begin

   end:
   ```

   Функция А вызывает функцию В, которая затем рекурсивно вызывает функцию А. Отобразите содержимое стека: после выполнения одного такого вызова; двух таких вызовов. Объясните, как будет происходить возврат в вызывающую программу, если после двух вызовов рекурсия заканчивается.

3. **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 1 : integer of
        1: (A : TestRecord);
        2: (B : TestArray);
        3: (C : integer);
end;
```

а также переменные:

```
var
    c1.c2 ; Char;
    11.12 : integer;
    al : TestArray;
    r1.r2.r3 : TestRecord:
    ul : 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?

4. Вычислите следующие выражения, записанные в форме обратной польской записи:

   O 1 2 * 3 5 4 * + +

   O 1 2 3 5 4 * * + +

   0 1 2 + 3 * 5 + 4 -

   O 12 + 3 * 5 4 + -

   Напишите для каждого из них выражение в форме инфиксной (обычной) записи.

5. Дана грамматика G({(,) Л&.-.а}.{S.T.E} .P.S):

   P:
   S -> S^{A}T I т
   T -> T&E I E
   E -> -E | (S) | a

   Постройте схему СУ-компиляции выражений входных цепочек языка, заданного этой грамматикой, во внутреннее представление в форме обратной польской записи.

6. Для грамматики, приведенной выше в задаче № 4, постройте схему СУ-перевода во внутреннее представление программы в форме триад. Выполните перевод в форму триад цепочки символов a^{A}a&-a&(a^{A}-~a).

7. Для грамматики, приведенной выше в задаче № 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^{A}~~a).