COMPUTER MONOGRAPHS
General Editor Stanley Gill
Associate Editor: J. J. Florentin

AUTOMATIC SYNTACTIC ANALYSIS

J. M. FOSTER

MACDONALD • LONDON AND
AMERICAN ELSEVIER INC. • NEW YORK
1970

МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ЭВМ

Дж.Фостер

АВТОМАТИЧЕСКИЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ

Перевод с английского
В. В. Мартынюка

Под редакцией
Э. З. Любимского

ИЗДАТЕЛЬСТВО
«МИР»
Москва 1975

УДК. 681.142.2

Книга посвящена систематическому рассмотрению методов синтаксического анализа, применяемых при компиляции программ для ЭВМ. Она написана на высоком научном уровне, однако от читателя не требуется предварительных знаний о формальных грамматиках и работе компиляторов, а предполагается только знакомство с основами программирования. Описываемые алгоритмы приведены на несколько модифицированном Алголе.

Последовательное изложение проблем синтаксического анализа языков программирования представляет большой интерес для разработчиков систем программирования. Книга будет полезна также студентам и аспирантам, занимающимся программированием.

Редакция литературы по математическим наукам

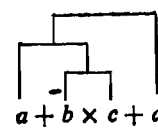
Ф $\frac{20204 - 030}{041\,(01) - 75}$ 30 — 75

1 ВВЕДЕНИЕ

Часто при вычислениях нам приходится иметь дело с объектами, обладающими структурой. Грамматическая структура английского предложения состоит из подлежащего, сказуемого, группы дополнения и прочих элементов. Структурными элементами программы для вычислительной машины являются подпрограмма, процедура, выражение. Блок-схеме соответствует структура из многоугольников и линий. Массив данных имеет структуру, которая определяется соответствующими индексами. Перечень таких примеров можно было бы продолжить. У нас может возникнуть желание описывать подобные структуры таким образом, чтобы суметь потом оперировать с соответствующими классами, как мы оперируем с классом грамматически правильных английских предложений, с классом синтаксически правильных алгольных программ или с классом правильно сформированных массивов или файлов определенного вида. Такие описания пригодятся для того, чтобы мы могли определять, принадлежит ли заданный объект определенному классу, или чтобы мы могли получать примеры элементов класса, или чтобы мы могли оперировать с целыми классами, например при поиске пересечения двух классов. Структура сама по себе тоже может интересовать нас. Например, грамматическая структура предложения помогает нам интерпретировать его смысл, раскрывая связи между частями. Аналогично структура алгольной программы показывает нам нужные связи между соответствующими частями и тем самым помогает компилировать эту программу. Так, структура алгольного выражения

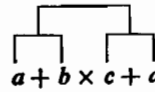
$$a + b \times c + d$$

может быть изображена в виде



который показывает, какие части арифметического выражения связаны знаком плюс, а какие — знаком умножения. Если бы

соответствующая структура имела вид



то интерпретация выражения была бы совершенно другой.

Поэтому нам нужны формальные описания структур объектов. Поскольку раньше всего были изучены грамматические описания предложений, то обычно пользуются терминологией, заимствованной из лингвистики, хотя целью рассмотрений могут оказаться и нелингвистические применения грамматик. В этой книге мы будем иметь дело только с одномерными объектами, а именно с последовательностями (предложениями) из неделимых элементов (слов). Под структурой (грамматическим разбором) предложения будут пониматься порядок и структура его частей. Мы будем пользоваться так называемыми контекстно-свободными грамматиками. Они не являются единственным средством описания структур предложений, но обычно именно на них основывается применение грамматик для компиляции [1, 7, 10, 13, 15].

Грамматики могут применяться для двух различных целей. Во-первых, мы могли бы захотеть рассматривать грамматику как формальный набор правил для порождения всех корректных предложений какого-то языка. В таком случае дело сводится к составлению грамматик для порождения языков и к преобразованиям грамматик. Например, авторы сообщения об Алголе придумали такую грамматику, которая могла бы формально порождать правильные (грамматически) алгольные программы. Во-вторых, кто-то мог бы задать нам набор грамматических правил и некоторое предложение и попросить нас либо разобрать это предложение в соответствии с этими правилами, либо выяснить, существует ли хотя бы один способ такого разбора. Этот второй подход привел к постановке задач отыскания алгоритмов, которые будут производить грамматический разбор эффективно и заведомо гарантированы от зацикливания. Именно такие задачи мы будем рассматривать в данной книге. Подобные алгоритмы называются алгоритмами грамматического разбора или синтаксическими анализаторами.

Грамматики, используемые для описания языков программирования, гораздо более просты, чем те грамматики, которые нужны для естественных языков. Поэтому мы никоим образом не будем затрагивать последних, и всюду далее подразумевается, что рассматриваемые грамматики непригодны для естественных языков [8]. Языки программирования разрабатываются с таким расчетом, чтобы было легко читать и писать на них (а также компилировать с них), поэтому они обычно имеют

простую структуру. Впрочем, они все же достаточно сложны и, как мы увидим, их применение сопряжено с определенными трудностями. С другой стороны, попытки изучать естественные языки с помощью контекстно-свободных грамматик приводят к появлению грамматик, подразумевающих много двусмысленностей; из этого следует, что такие грамматики не могут обеспечить исчерпывающее описание естественного языка. Обычно необходимо найти все варианты грамматического разбора заданных предложений. Это приводит к тому, что процесс грамматического разбора для естественного языка оказывается медленным; между тем для языков программирования можно использовать более быстрые способы грамматического разбора.

Задача грамматического разбора по существу состоит в отыскании подходящих мест для того, чтобы вставить пропущенные скобки. Рассмотрим выражение

$$a + b + c \times d \times e + f$$

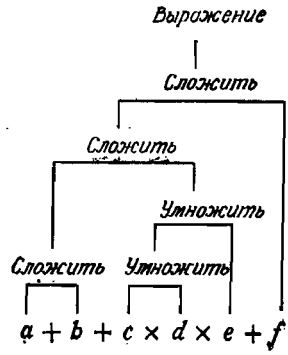
В скобочной записи оно приобретает вид

$$(((a + b) + ((c \times d) \times e)) + f)$$

и задача грамматического разбора становится тривиальной, так как скобки описывают структуру. Однако такое систематическое внесение скобок в алгольное выражение или в английское предложение сделало бы текст неудобочитаемым, и поэтому обычно скобки отсутствуют, а ставятся они только тогда, когда без них нельзя обойтись. У нас может появиться желание различать разные способы образования групп символов. Если переписать предыдущее выражение, употребляя круглые скобки для групп, соответствующих знаку плюс, и квадратные скобки для групп, соответствующих знаку умножения, то оно будет выглядеть следующим образом:

$$(((a + b) + [[c \times d] \times e]) + f)$$

Возможна и такая запись:



Описываемые в этой книге методы грамматического разбора предназначены для компиляции программ для вычислительных машин, однако при этом затрагиваются и некоторые проблемы синтаксического анализа естественных языков. От читателя не требуется предварительных знаний о формальных грамматиках и работе компиляторов, но предполагается некоторое знакомство с основами программирования для вычислительных машин. Алгоритмы грамматического разбора описываются на несколько модифицированном варианте языка Алгол, понимание которого не вызовет затруднений у читателя, знакомого с языком Алгол-60. Чтобы избежать громоздких программ, применяются методы обработки списков. Краткий обзор основных сведений об этих методах приводится в приложении 1.

В гл. 2 приводятся некоторые элементарные определения и примеры грамматик. В гл. 3 рассматривается общая проблематика грамматического разбора. Глава 4 посвящена методам грамматического разбора, применимым к любым контекстно-свободным грамматикам. В гл. 5 рассматриваются некоторые ограниченные классы грамматик, для которых можно обеспечить высокую скорость синтаксического анализа. В гл. 6 речь идет о преобразованиях грамматик и, в частности, о получении эквивалентных грамматик, более удобных для синтаксического анализа, чем исходные грамматики. Из гл. 7, а также из первой части гл. 6 можно получить краткое представление о том, как разбор структуры программы может использоваться для компиляции.

Впервые синтаксический анализ применялся в компиляторах Брукера и Морриса [3, 4, 5] и Айронса [21, 22]. Более подробную библиографию содержит отличная обзорная статья Фельдмана и Гриса [10].

Не всякую часть компилятора можно написать, основываясь на применении контекстно-свободных грамматик. На практике синтаксический анализ обычно используется на первых этапах компиляции, когда выявляется структура программы и производится некоторая предварительная обработка этой программы. В частности, невозможно сформулировать в терминах контекстно-свободных грамматик такое ограничение, как «все переменные должны быть описаны прежде, чем они используются»; для этого потребуется специальная последующая обработка структуры.

2 КОНТЕКСТНО-СВОБОДНЫЕ ГРАММАТИКИ

2.1. ТЕРМИНОЛОГИЯ

Мы будем пользоваться следующими терминами и обозначениями. Построенные надлежащим образом последовательности будут называться *предложениями*. Они будут состоять из упорядоченных объектов, называемых *словами*. Мы не будем касаться структуры слов, а предположим, что умеем как-то распознавать слова. В примере на стр. 7 символы a, b, c, d, e, f, $+$, $\times$ были словами. Слова будут записываться строчными буквами или будут такими символами, как $+$ и $\times$. Для описательных целей мы будем пользоваться некоторыми отличными от слов символами, которые будем называть *классами*. Их называют также нетерминальными символами. В предыдущем примере *Сложить*, *Умножить* и *Выражение* были классами. Классы будут всегда обозначаться прописными буквами. Предложение — это особый класс, который обычно обозначается символом S. Всякому классу будет соответствовать набор возможных структурных вариантов, каждый из которых состоит из упорядоченных элементов, являющихся либо словами, либо классами. Эти варианты называются *правилами подстановки*. Таким образом, вся грамматика состоит из набора слов (словаря), набора классов, названия особого класса, т. е. предложения, и из набора правил подстановки.

В качестве простейшего примера рассмотрим грамматику со словарем a, b, c, d, с классами S и T, с символом предложения S и с правилами подстановки, записываемыми следующим образом:

$$S \rightarrow a \mid bTTd$$
$$T \rightarrow bc \mid a$$

Такая запись означает, что имеются два структурных варианта для S: либо a, либо сначала b, затем T, затем T, затем d. Для класса T также имеются два структурных варианта. Тогда существуют пять правильных предложений:

a
$bbcbcd$
$bbcad$
$babcd$
$baad$

Будем говорить, что эти предложения могут быть *выведены* из S; возможность вывода предложения $babcd$ указывается так:

$$S \rightarrow babcd$$

Часть предложения, которая выводится из какого-то класса, называется *фразой*. Таким образом, в предложении $babcd$ имеется фраза a, выводимая из T, и фраза bc, также выводимая из T.

Правила подстановки могут рассматриваться как наборы правил для переписывания классов, и процесс вывода $babcd$ из S может рассматриваться следующим образом. Начинаем с символа предложения

$$S$$

и заменяем его на один из соответствующих ему вариантов

$$bTTd$$

В полученном результате выбираем один из классов и переписываем строку, заменяя этот класс на один из соответствующих ему вариантов

$$bTbcd$$

Продолжаем такой процесс до тех пор, пока больше не останется ни одного класса, в таком случае последовательность называется *терминальной*. В нашем примере терминальная последовательность получается после следующего переписывания:

$$babcd$$

Будем говорить также, что $bTbcd$ выводится из S:

$$S \rightarrow bTbcd$$

и что a выводится из T:

$$T \rightarrow a$$

Заметим, что специфическое свойство таких правил подстановки состоит в том, что подстановки, применяемые для конкретных классов в конкретных местах, не зависят ни от окружающего контекста, ни от последовательности применявшихся ранее операций подстановки. Класс T может быть заменен без всяких ограничений как на a, так и на bc. Этим объясняется название «контекстно-свободные», которое применяется к таким грамматикам. Точный смысл этого названия состоит в том, что один «сосед» в последовательности не может оказывать влияния на другого отдаленного «соседа»; это требование накладывает сильное ограничение на языки, которые могут быть описаны такими грамматиками.

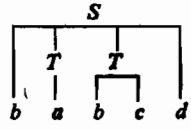
В приведенном выше примере мы могли бы выполнять последовательность операций подстановки двумя способами:

$$S$$
$$bTTd$$
$$bTbcd$$
$$babcd$$

или

$$S$$
$$bTTd$$
$$baTd$$
$$babcd$$

Эти два способа обработки не будут рассматриваться как различные грамматические разборы строки *babcd* и не будут свидетельствовать о двусмыслеиности грамматики. Чтобы разбор не представлялся двусмысленным, мы можем либо принять дополнительное правило, что в частично переписанной последовательности всегда применяется подстановка для самого левого класса, либо рассматривать этот разбор как структуру

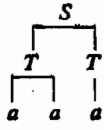


а не прослеживать последовательность операций подстановки.

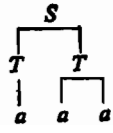
Тем не менее легко придумать грамматики с двусмысленным грамматическим разбором. Пример

$$S \to TT$$
$$T \to a \mid aa$$

дает нам два различных грамматических разбора для предложения *aaa*, а именно



и



Поскольку в языках программирования необходимо избегать двусмысленностей, вопрос о том, является ли грамматика двусмысленной, приобретает большое значение.

В качестве слов можно рассматривать любые объекты при условии, чтобы можно было различать слова. Например, если задана грамматика, описывающая набор предложений

идентификатор
идентификатор + *идентификатор*
идентификатор + *идентификатор* + *идентификатор*

то мы вольны интерпретировать идентификатор не как конкретный набор букв *и, д, е, н, т, и, ф, и, к, а, т, о, р*, а как общее представление для множества различных объектов при условии, что эти объекты не будут смешиваться со знаком +. Такое слово будет называться *представляющим*. Мы будем пользоваться двумя представляющими словами; первое из них, *идентификатор* (сокращенно *ид*), обозначает множество последовательностей из букв и цифр, начинающихся с буквы, второе *число* употребляется в обычном смысле. Для приведенной выше грамматики правильным является, например, следующее предложение:

$$abc + d2 + xy3t$$

Мы будем рассматривать грамматики, удовлетворяющие следующим правилам. Не должно существовать выводов

$$X \to X$$

где X — любой класс. Такие выводы ничего не добавляют к множеству описываемых предложений, но создают возможность бесконечных возвратов при грамматическом разборе (X является X, который является X ...). Каждый класс должен появляться при некотором выводе из S. Если какой-то класс никогда не появляется, то он не приносит пользы для описания предложений и поэтому может быть исключен.

Для всякого класса должен найтись по крайней мере один терминальный вывод. Это означает, что не должно существовать классов, подобных классу Y, для которого единственное правило подстановки имеет вид

$$Y \to Ya$$

и никогда не породит терминальную строку.

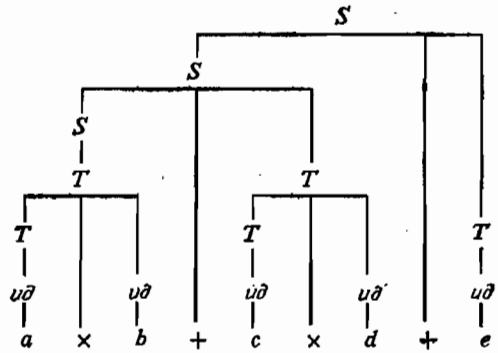
2.2. ПРИМЕРЫ ГРАММАТИК И ИХ СВОЙСТВ

В этом параграфе мы рассмотрим некоторые типичные грамматики и опишем их свойства, имеющие отношение к грамматическому разбору.

Грамматики, описывающие конечные наборы допустимых предложений, представляют сравнительно мало интереса. Заслуживают внимания те случаи, когда множества предложений бесконечны. Для того чтобы грамматически описывать такие бесконечные множества, необходимо применять рекурсию. Рекурсия означает, что в некоторой выводимой из класса X последовательности элементов (слов и классов) должно найтись вхождение X. Типичным примером является описание множества арифметических выражений, образуемых из идентификаторов и знаков $+$ и $\times$. Для этого часто используется следующая грамматика, в которой рекурсивны оба класса S и T:

$$S \to T \mid S + T$$
$$T \to \textit{идентификатор} \mid T \times \textit{идентификатор}$$

В этой грамматике типичный грамматический разбор предложения имеет вид



Класс X, для которого существует вывод

$$X \to X\alpha$$

где α — непустая последовательность элементов, называется *рекурсивным слева*; в рассмотренном примере оба класса S и T являются рекурсивными слева. Аналогично класс X, для которого существует вывод

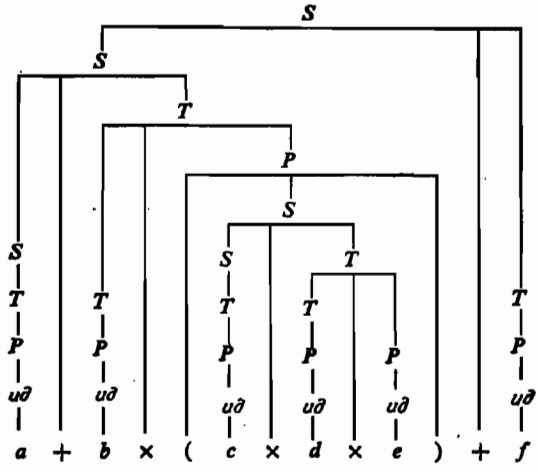
$$X \to \beta X$$

где β — непустая последовательность элементов, будет называться *рекурсивным справа*.

Другая типичная грамматика описывает выражения, включающие идентификаторы, знаки $+$, $\times$ и круглые скобки, и имеет вид

$$S \to T \mid S + T$$
$$T \to P \mid T \times P$$
$$P \to \textit{ид} \mid (S)$$

Здесь все классы S, T и P являются рекурсивиыми. Типичный грамматический разбор предложения выглядит так:



Класс X, для которого существует вывод

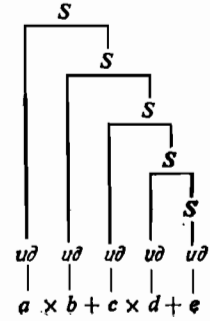
$$X \to \alpha X \beta,$$

где α и β — непустые последовательности элементов, называется *самовставляемым*. Классы S, T и P являются самовставляемыми и, кроме того, классы S и T — рекурсивные слева.

Может оказаться, что две различные грамматики описывают одно и то же миожество предложений. Например, описанное ранее миожество выражений, включающих идентификаторы и знаки $+$ и $\times$, можно было бы описать и такой грамматикой:

$$S \to ид \mid ид + S \mid ид \times S$$

Однако при использовании этой грамматики типичный грамматический разбор выглядит несколько иначе:



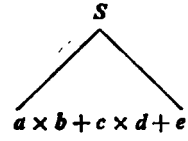
Если наша задача состоит в компилировании этого выражения, то удобен разбор, соответствующий первой грамматике, так как он обеспечивает указание аргументов для операций, а вторая грамматика почти бесполезна. Отсюда видно, что может возникнуть необходимость предпочесть одну грамматику другой с учетом более подходящей структуры грамматического разбора. В гл. 6 указывается способ частичного решения этой проблемы путем преобразования грамматик.

При всем сказанном класс контекстно-свободных грамматик слишком широк, и в его рамках могут описываться слишком запутанные грамматики. Например, рассмотрим грамматику

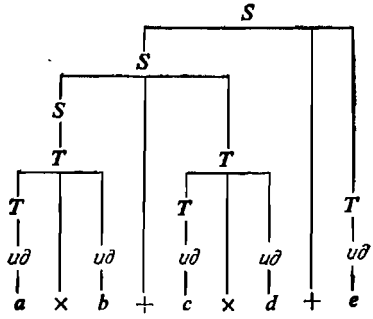
$$S \to a \mid aSa$$

Она описывает множество предложений, состоящих из четного числа слов a, окружающих специально выделенное слово a в середине предложения. Как показано в следующей главе, этот пример неудобен для грамматического разбора; в то же время маловероятно, чтобы он понадобился кому-нибудь на практике. Предлагались различные подмножества класса контекстно-свободных грамматик, но до сих пор ни одно из них не оказалось вполне удовлетворительным. В настоящее время дело все еще обстоит так, что либо методы грамматического разбора слишком общи и неэффективны, либо они применимы к недостаточно широкому классу языков.

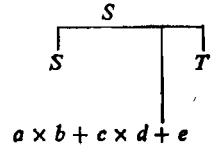
Грамматический разбор можно осуществить следующим способом, основанным на схеме, которая была предложена Эмерелом [34]. Если заданы конкретное предложение и грамматика, строим треугольник



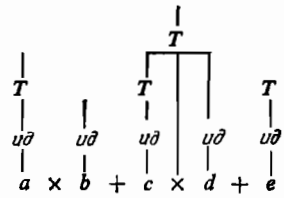
и пытаемся заполнить внутреннюю часть треугольника непротиворечивым деревом вывода, которое представляет грамматический разбор:



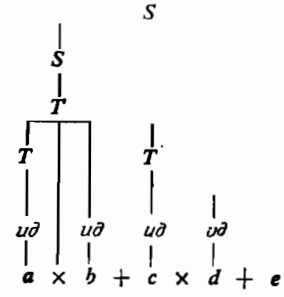
В принципе не имеет значения то, как мы пытаемся заполнить внутреннюю часть треугольника. Мы можем заполнять треугольник от вершины вниз



или снизу к вершине

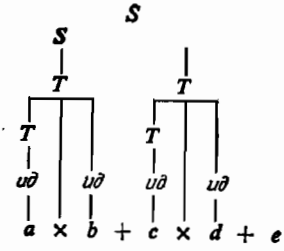


или же от левого угла



или в любом другом порядке. Однако порядок заполнения треугольника может в большой степени влиять на эффективность процесса.

Каким бы ни был этот порядок, отдельные шаги состоят из попыток подыскать правило подстановки, подходящее для рассматриваемого участка строки. Например, для предыдущей схемы возможная процедура состоит в том, чтобы продвигаться по схеме

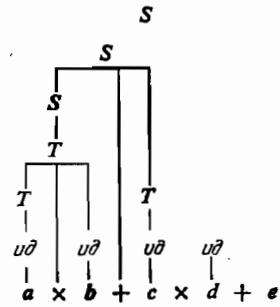


применяя правило подстановки

$$T \rightarrow T \times ид$$

Этот конкретный шаг подходит для начальной локальной ситуации и окажется правильным также и на последующих

этапах грамматического разбора. В некоторой локальной ситуации мы можем избрать шаг



применяя правило подстановки

$$S \rightarrow S + T$$

Некоторое время спустя мы обнаружили бы невозможность дальнейшего грамматического разбора по этому правилу; это означало бы, что мы должны вернуться назад и попытаться применить другой локальный шаг. Такой грамматический разбор, вообще говоря, не эффективен, так как приходится делать пробные попытки, которые позднее оказываются ошибочными.

Рассмотрим проблему выбора локального шага вне зависимости от того порядка, в котором мы пытаемся заполнять треугольник. Мы отбираем какое-то правило подстановки, удовлетворяющее некоторым априорным тестам. Например, выбираются только такие правила подстановки, которые подходят для текущей локальной ситуации. Априорные тесты могут быть и гораздо более сложными. Может оказаться, что мы отвергаем правило подстановки потому, что оно неизбежно привело бы к превышению допустимого количества слов в терминальной последовательности. Мы могли бы отвергнуть правило и потому, что оно требует использования слова, которое в нашем случае не встречается. Если рассматриваемая грамматика содержит правило подстановки

$$S \rightarrow S - T$$

то было бы бесполезно пытаться применить это правило, если в предложении нет знака минус. Для того чтобы узнать о таком обстоятельстве, нам не потребовалось бы изучать детальную структуру предложения. Другой возможный априорный критерий отклонения правил подстановки основывается на рассмотрении первого слова из выводимой по этому правилу фразы.

Если нужное слово не может быть получено в качестве первого выводимого слова, то следует отвергнуть данное правило подстановки. Например, фраза из класса T должна начинаться с идентификатора.

После того как выбрано правило подстановки, удовлетворяющее всем априорным критериям, это правило применяется и мы переходим к следующему шагу. Если позднее мы обнаружим, что полученная структура не годится для грамматического разбора нашего предложения, то нам придется вернуться, выбрать другое правило подстановки и предпринять новую попытку. Если необходимо найти все способы грамматического разбора, то мы должны испробовать все подходящие априори правила подстановки. Если известно, что язык не содержит двусмысленностей или если достаточно найти один способ грамматического разбора, то после отыскания первого подходящего грамматического разбора работа может быть прекращена.

Скорость разбора зависит от умения избегать неудачных попыток, при которых выполняется работа, оказывающаяся впоследствии бесполезной. Если порядок выбора шагов и априорные критерии таковы, что на каждом шаге применимо только одно правило подстановки, то такой метод грамматического разбора уже не может быть заметно улучшен (если только проверка априорных критериев не является слишком медленной). Однако предположим, что на каждом шаге имеются несколько возможностей выбора. Тогда общее число всех вариантов разбора растет с экспоненциальной скоростью. Поэтому исключительную важность приобретает выбор априорных критериев и порядка действий.

Для иллюстрации рассмотрим грамматику

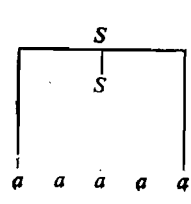
$$S \rightarrow a \mid aSa$$

Если производить для этой грамматики разбор сверху вниз

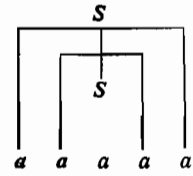
$$S$$

$$aaaaa$$

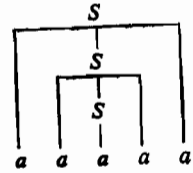
то на первом шаге имеем



на втором шаге



и на третьем шаге



Мы руководствовались здесь следующим априорным критерием. Если осталось одно незатронутое слово, применяем подстановку

$$S \to a$$

в противном случае применяем

$$S \to aSa$$

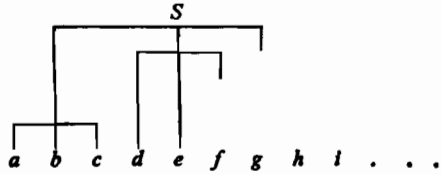
При этом на каждом этапе возможен только один способ обработки. Однако если начать разбор от левого угла и, руководствуясь априорным критерием, выбирать только такие правила подстановки, которые начинаются с первого незатронутого слова (иногда этот критерий бывает полезным, но в данном случае он не помогает), то нам придется ходить по ложным путям и эффективность будет потеряна.

Мы можем выбирать не только порядок заполнения треугольника и априорные критерии, но также и порядок перебора правил подстановки, удовлетворяющих нашим априорным критериям. Последнее может принести пользу только в том случае, если известно, что грамматика не содержит двусмысленностей или же если нас устраивает любой грамматический разбор; в противном же случае придется испробовать все допустимые правила подстановки. Однако если годится любой грамматический разбор, то разумный выбор конкретного порядка перебора правил подстановки может способствовать повышению вероятности быстрого завершения разбора.

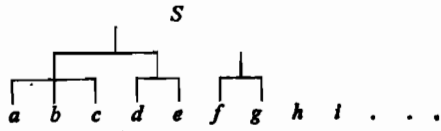
4 УНИВЕРСАЛЬНЫЕ МЕТОДЫ ГРАММАТИЧЕСКОГО РАЗБОРА

В этой главе рассматриваются некоторые методы получения всех вариантов грамматического разбора предложения, принадлежащего произвольной контекстно-свободной грамматике.

В естественных языках предложения обычно бывают довольно короткими, и их можно легко размещать в оперативной памяти машины. При этом применимы методы разбора, требующие одновременного присутствия в памяти всего предложения. Но это требование обычно невыполнимо применительно к предложениям (программам), которые должны анализироваться компилятором. Такие предложения могут состоять из тысяч слов. В связи с этим возникает необходимость пытаться в процессе разбора заполнять треугольник таким способом, который требовал бы присутствия в памяти только части предложения. Очевидный способ разбора предложения по частям состоит в том, чтобы перебирать слова по одному слева направо в их естественной последовательности. Такой способ сводится к двум основным вариантам заполнения треугольника разбора: сверху вниз и слева направо



или снизу вверх и слева направо



При этом основное различие между конкретными методами проявляется в априорных критериях, применяющихся для сокращения перебора правил подстановки. Алгоритмы разбора сверху вниз и снизу вверх несколько отличаются по своим свойствам.

Хотя общий путь грамматического разбора проходит слева направо и либо вниз, либо вверх, может потребоваться несколько попыток разбора конкретной последовательности слов, так как непригодность того или иного варианта разбора может проявиться не сразу или же сама грамматика может оказаться двусмысленной. Все эти попытки разбора могут проводиться параллельно, с тем чтобы последовательность обрабатывалась строго слева направо, или же последовательно, с возвратами к начальной точке предложения после каждого неудавшегося или завершенного разбора. Описываемые ниже алгоритмы являются в этом смысле параллельными.

4.1. РАЗБОР СВЕРХУ ВНИЗ

При разборе сверху вниз мы начинаем с символа предложения и последовательно производим подстановки для отдельных классов, пытаясь добиться соответствия применительно ко всему предложению. Сущность алгоритма может быть проиллюстрирована на следующем примере. Рассмотрим грамматику

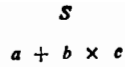
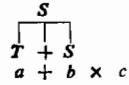
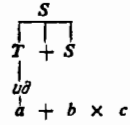
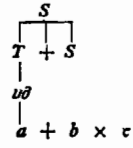
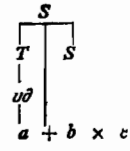
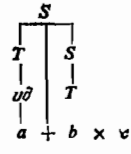
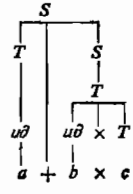
$$S \rightarrow T \mid T + S$$
$$T \rightarrow ид \mid ид \times T$$

где *ид* — представление для идентификаторов. Начнем с разбора предложения

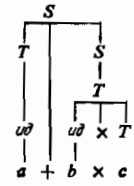
$$a + b \times c$$

выбирая подстановки, которые окажутся подходящими, и демонстрируя одновременное построение дерева грамматического разбора. Затем мы укажем полную последовательность шагов в процессе анализа, исключив из рассмотрения построение деревьев разбора.

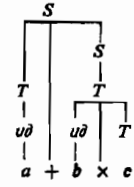
Начинаем с символа S и заменяем его на соответствующие варианты; в данном случае правильной окажется замена на $T + S$. На последующих шагах, если первый символ оказывается словом, то он сравнивается с первым символом предложения, и в случае совпадения этот символ исключается и из предложения, и из последовательности элементов. Эта последовательность представляет возможную структуру оставшейся части предложения. Если же первый символ обозначает класс, то он заменяется на один из вариантов, соответствующих этому классу. Поскольку мы правильно подбираем подстановки, сравниваемые слова будут совпадать, но в общем случае они могут оказаться различными, и это означало бы неудачу разбора. Ниже приводятся последовательные шаги всего анализа с указанием остатка предложения, остатка разбора и текущего дерева разбора для каждого этапа.

$a + b \times c$	S	
$a + b \times c$	$T + S$	
$a + b \times c$	$ud + S$	
$+ b \times c$	$+ S$	
$b \times c$	S	
$b \times c$	T	
$b \times c$	$ud \times T$	

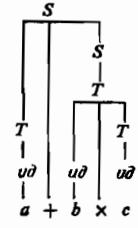
× c × T

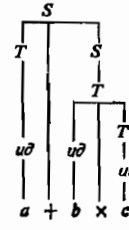

S
T
S
T
ud
ud × T
a + b × c

c T


S
T
S
T
ud
ud T
a + b × c

c ud


S
S
T
T
T
ud
ud ud
a + b × c


S
T
S
T
T
ud
ud ud
ud
a + b × c

Теперь продемонстрируем перебор всех вариантов с указанием для каждого этапа остатка предложения и остатка структуры, но без указания полученного разбора.

$a+b\times c$	S
$a+b\times c$	T
	$T+S$
$a+b\times c$	*ид*
	ид $+S$
	ид $\times T$
	ид $\times T+S$
$+b\times c$	—
	$+S$
	$\times T$
	$\times T+S$
$b\times c$	S
$b\times c$	T
	$T+S$
$b\times c$	*ид*
	ид $\times T$
	ид $+S$
	ид $\times T+S$
$\times c$	—
	$\times T$
	$+S$
	$\times T+S$
c	T
	$T+S$
c	*ид*
	ид $\times T$
	ид $+S$
	ид $\times T+S$
—	—
	$\times T$
	$+S$
	$\times T+S$

На последнем этапе оба остатка пусты, что соответствует представлению правильного разбора.

Алгоритм разбора сверху вниз приведен в приложении 2.

4.2. АПРИОРНЫЕ КРИТЕРИИ ДЛЯ ГРАММАТИЧЕСКОГО РАЗБОРА СВЕРХУ ВНИЗ

Метод разбора сверху вниз имеет один существенный недостаток. Рассмотрим грамматику

$$S \to T \mid S + T$$
$$T \to ид \mid T \times ид$$

и попытаемся выполнить соответствующий ей разбор предложения, рассмотренного в разд. 4.1. Последовательные подстановки будут давать

$$S$$

$$T$$
$$S + T$$

$$ид$$
$$T \times ид$$
$$T + T$$
$$S + T + T$$

и процесс никогда не завершится. Итак, этот метод будет работать только для грамматик, не содержащих рекурсивных слева классов, т. е. не допускающих выводов

$$T \to T\alpha$$

Грейбах [35] показал, что всегда можно преобразовать любую грамматику в эквивалентную ей грамматику без рекурсии слева. Эквивалентной называется грамматика, которая определяет те же предложения, содержит такие же двусмысленности [1]), но порождает для данного предложения другой грамматический разбор. В гл. 6 обсуждаются такие эквивалентные преобразования грамматик, а также проблемы, связанные с получением для преобразованной грамматики правильного разбора.

Существует априорный критерий, который исключает возможность зацикливания, подобного рассмотренному выше. Каждый класс должен порождать терминальную последовательность слов, содержащую по крайней мере одно слово. Тогда если число элементов в остатке структуры больше числа слов в остатке предложения, то такой разбор не может быть правильным. Это

[1]) Если они есть. — *Прим. перев.*

правило приносит практическую пользу только тогда, когда длина предложения мала. Поэтому оно может оказаться применимым при обработке предложений естественного языка, но не годится для компиляции. Ниже мы будем предполагать, что грамматика не содержит рекурсивных слева классов.

Можно предложить полезный априорный критерий, основанный на рассмотрении текущего слова в предложении. Предположим, что b — текущее слово и что мы собираемся заменить крайний слева класс X по одному из соответствующих ему правил подстановки. Нам нужно только отобрать те правила, которые могут породить фразы, начинающиеся с b. Если предварительно был выполнен специальный анализ грамматики и получена матрица соответствия слов и классов, указывающая, какие классы могут начинаться с каких слов, то эту матрицу можно использовать для отбрасывания тех правил подстановки, которые не могут породить данное начальное слово.

Построить такую матрицу нетрудно. Предположим, что в начале мы имеем матрицу соответствия слов и классов классам; эта матрица указывает те слова и классы, с которых начинаются правила подстановки, соответствующие классам. Например, для грамматики

$$S \to a \mid Tb$$
$$T \to c \mid Td$$

получаем

	S	T	a	b	c	d
S	0	1	1	0	0	0
T	0	1	0	0	1	0

Это означает, что S начинается с T или a и что T начинается с T или c.

К каждой строке, соответствующей какому-либо классу, добавляются (по операции «или») строки, соответствующие классам, уже представленным в данной строке; такой процесс продолжается до тех пор, пока вся матрица не перестанет изменяться [1]). После этого матрица содержит указания всех слов, с которых могут начинаться классы:

	S	T	a	b	c	d
S	0	1	1	0	1	0
T	0	1	0	0	1	0

[1]) Эта процедура представляет собой обобщение операции построения транзитивного замыкания. Если исключить из рассмотрения столбцы, соответствующие словам, то для оставшейся квадратной матрицы строится обычное транзитивное замыкание. — *Прим. перев.*

Другой способ получения аналогичного результата состоит в том, чтобы для грамматики (не содержащей рекурсивных слева классов) последовательно заменять классы, встречающиеся в левых частях правил подстановки, на соответствующие этим классам варианты подстановки до тех пор, пока все правила подстановки не будут начинаться со слов. Например, грамматика

$$S \to T \mid T + S$$
$$T \to \text{ид} \mid \text{ид} \times T$$

порождает

$$S \to \text{ид} \mid \text{ид} \times T \mid \text{ид} + S \mid \text{ид} \times T + S$$
$$T \to \text{ид} \mid \text{ид} \times T$$

Затем формируем для каждой комбинации из класса и слова те подстановки для данного класса, которые начинаются с данного слова. Априорный критерий состоит в том, что выбираются подстановки, соответствующие классу, с которого начинается остаток разбора, и слову, с которого начинается остаток предложения. Это дает такой же результат, как применение предыдущего априорного критерия в случае отсутствия рекурсии слева. Преобразование правил подстановки выполняется предварительно, а не во время грамматического разбора. Поэтому такой метод немного быстрее, но требует несколько большего расхода памяти.

Преобразованная грамматика, в которой все правила подстановки начинаются со слов, называется грамматикой в нормальной форме, а такой метод анализа обычно называют прогнозирующим анализом. Поскольку полученная грамматика может отличаться от исходной, необходимо выполнить дополнительные действия по приведению разбора в соответствие с исходной грамматикой.

Можно руководствоваться другим критерием экономичности грамматического разбора. Этот критерий, по-видимому, более эффективен для грамматик, допускающих много двусмысленностей. Если оказывается, что при нескольких вариантах разбора нужно выводить одну и ту же последовательность слов из одной и той же последовательности классов и слов, то описанные ранее методы потребовали бы, чтобы такой разбор повторялся соответственно несколько раз. Это объясняется тем, что различные варианты разбора отделяются на ранней стадии, когда еще не удается опознать повторяющуюся работу. Если существует реальная возможность таких повторений, то желательно принять специальные меры для того, чтобы всякий вариант анализа выполнялся только однократно.

Может оказаться также, что последовательность слов не соответствует остатку разбора. Не менее желательно, чтобы мы не занимались многократно установлением этого факта. Позволяющий избежать повторений при анализе метод, ориентированный на естественные языки, описан в статье [25].

4.3. ГРАММАТИЧЕСКИЙ РАЗБОР СНИЗУ ВВЕРХ

В отличие от метода разбора сверху вниз, при котором путем последовательных подстановок символ предложения «расширяется» и порождает все предложение, метод разбора снизу вверх основывается на том, что исходным объектом является все предложение, а правила подстановки применяются «в обратном направлении» с тем, чтобы «ужать» предложение и свести его к символу предложения. Например, в грамматике

$$S \rightarrow T \mid T + S$$
$$T \rightarrow \text{ид} \mid \text{ид} \times T$$

мы можем анализировать предложение, выполняя следующие последовательные шаги:

$a + b \times c$	
$\text{ид} + \text{ид} \times \text{ид}$	
$\text{ид} + \text{ид} \times T$	$T \rightarrow \text{ид}$
$T + \text{ид} \times T$	$T \rightarrow \text{ид}$
$T + T$	$T \rightarrow \text{ид} \times T$
$T + S$	$S \rightarrow T$
S	$S \rightarrow T + S$

В принципе, подстановки могли бы выполняться в любом порядке, но так как рассматривается схема, при которой предложение обрабатывается слева направо, то на каждом этапе подстановка должна производиться как можно левее. Мы снова разбираем то же предложение в той же грамматике, продвигаясь слева направо и применяя подходящие правила подстановки; на стр. 30 показано построение дерева грамматического разбора.

4.4. АПРИОРНЫЕ КРИТЕРИИ ДЛЯ АНАЛИЗА СНИЗУ ВВЕРХ

В чистом виде метод разбора снизу вверх не очень эффективен, так как возможно большое число неудачных попыток. Однако можно воспользоваться эффективным априорным критерием.

$a + b \times c$	$a + b \times c$
$ud + b \times c$	$\begin{array}{l} ud \\ \vert \\ a + b \times c \end{array}$
$T + b \times c$	$\begin{array}{l} T \\ \vert \\ ud \\ \vert \\ a + b \times c \end{array}$
$T + ud \times c$	$\begin{array}{l} T \\ \vert \\ ud \quad ud \\ \vert \quad \vert \\ a + b \times c \end{array}$
$T + ud \times ud$	$\begin{array}{l} T \\ \vert \\ ud \quad ud \quad ud \\ \vert \quad \vert \quad \vert \\ a + b \times c \end{array}$
$T + ud \times T$	$\begin{array}{l} T \quad\quad T \\ \vert \quad\quad \vert \\ ud \quad ud \quad ud \\ \vert \quad \vert \quad \vert \\ a + b \times c \end{array}$
$T + T$	$\begin{array}{l} \quad\quad T \\ \quad\quad \vert\!\diagdown\!\vert\!\diagup\!\vert \\ T \quad\quad\quad T \\ \vert \quad\quad\quad \vert \\ ud \quad ud \quad ud \\ \vert \quad \vert \quad \vert \\ a + b \times c \end{array}$
$T + S$	$\begin{array}{l} \quad\quad S \\ \quad\quad \vert \\ \quad\quad T \\ \quad\quad \vert\!\diagdown\!\vert\!\diagup\!\vert \\ T \quad\quad\quad T \\ \vert \quad\quad\quad \vert \\ ud \quad ud \quad ud \\ \vert \quad \vert \quad \vert \\ a + b \times c \end{array}$
S	$\begin{array}{l} \quad\quad\quad\quad S \\ \quad\quad\quad\quad \vert\!\diagdown \\ \quad\quad\quad\quad\quad S \\ \quad\quad\quad\quad\quad \vert \\ \quad\quad\quad\quad\quad T \\ \quad\quad\quad\quad\quad \vert\!\diagdown\!\vert\!\diagup\!\vert \\ T \quad\quad\quad\quad\quad T \\ \vert \quad\quad\quad\quad\quad \vert \\ ud \quad ud \quad\quad ud \\ \vert \quad \vert \quad\quad \vert \\ a + b \times c \end{array}$

Рассмотрим предыдущий пример на этапе

$$ид + b \times c$$

Имеет смысл применять только те правила подстановки, которые начинаются с *ид*. Каждое такое правило подстановки является частью определения промежуточного класса. Если с этого класса не может начинаться предложение, то ни к чему применять соответствующее правило подстановки. В данном случае применимы два правила подстановки

$$T \rightarrow ид$$
$$T \rightarrow ид \times T$$

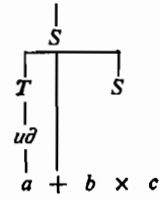
и, поскольку S может начинаться с T, следует попытаться применить оба. Применение второго правила подстановки оказывается неудачным, остается конструкция

$$T + b \times c$$

Опять применимы два правила подстановки, на этот раз

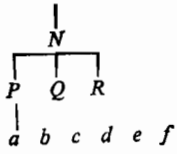
$$S \rightarrow T$$
$$S \rightarrow T + S$$

и следует испробовать оба. Применение первого приводит к неудаче. Теперь мы знаем, что разбор имеет следующий вид:



и поэтому остаток должен начинаться с S. Следовательно, к началу фразы $b \times c$ применимы только правила подстановки, описывающие такие классы, с которых может начинаться S.

В данном примере этот критерий не сокращает перебор правил подстановки, но в более сложной грамматике мы могли бы получить существенное сокращение перебора. В нашем примере фраза $b + c$ должна разбираться как S, т. е. оказывается, что она сводится к символу предложения. В общем же случае остаток сводится в результате разбора к любой последовательности промежуточных классов



и далее мы применяем только правила подстановки, описывающие такие классы, с которых может начинаться Q.

Построение матрицы, указывающей, с каких классов могут начинаться примеры других классов, производится точно так же, как описано для метода анализа сверху вниз.

Рассмотрим работу этого алгоритма при разборе предложения $a + b \times c$ применительно к грамматике

$$S \to T \mid S + T$$
$$T \to \text{ид} \mid T \times \text{ид}$$

и покажем, что рекурсия слева не вызывает осложнений. Приводятся все варианты, но построение дерева разбора не рассматривается.

ПРЕДЛОЖЕНИЕ	ЦЕПЬ	ПРАВИЛО ПОДСТАНОВКИ
$a + b \times c$	$(S),S$	—
$a + b \times c$	$(\text{ид}),T\ \ (S),S$	$T \to \text{ид}$

Слева записывается предложение, справа — примененное правило подстановки и в середине некоторая последовательность пар. Первым элементом каждой пары является остаток подстановки, а вторым элементом — название класса, которому соответствует данное правило подстановки. Если первый элемент первой пары соответствует первому слову в предложении, то мы удаляем его и из предложения, и из записи подстановки.

ПРЕДЛОЖЕНИЕ	ЦЕПЬ	ПРАВИЛО ПОДСТАНОВКИ
$+ b \times c$	ПУСТО, $T\ (S), S$	—

Если первый элемент в последовательности разбора оказывается пустым, то он удаляется и используются правила подстановки, которые, во-первых, начинаются с соответствующего класса и, во вторых, описывают классы, с которых может начинаться первый класс из следующей пары. В нашем случае применяются правила подстановки, начинающиеся с T и описывающие классы, с которых может начинаться S. Символ T удаляется из записи подстановки:

ПРЕДЛОЖЕНИЕ	ЦЕПЬ	ПРАВИЛО ПОДСТАНОВКИ
$+ b \times c$	$(x, \text{ид}), T\quad (S), S$	$T \to T \times \text{ид}$
	ПУСТО, $S(S), S$	$T \to T$

Если первый символ оказывается словом, не идентичным первому слову из остатка предложения, то разбор не удался. Таким образом, первый из приведенных выше вариантов не го-

дится. Развивая второй вариант, мы получаем

ПРЕДЛОЖЕНИЕ	ЦЕЛЬ	ПРАВИЛА ПОДСТАНОВКИ
$+ b \times c$	—	—
	$(+, T), S\ \ (S), S$	$S \rightarrow S + T$

Первый из этих вариантов означал бы завершение разбора и поэтому не пригоден.

ПРЕДЛОЖЕНИЕ	ЦЕЛЬ	ПРАВИЛА ПОДСТАНОВКИ
$b \times c$	$(T), S\ \ (S), S$	—
$b \times c$	$(\textit{ид}), T\ \ (T), S\ \ (S), S$	$T \rightarrow \textit{ид}$
$\times c$	ПУСТО, $T\ \ (T), S\ \ (S), S$	—
$\times c$	—	—
	$(+, T), S\ \ (S), S$	$S \rightarrow S + T$
	$(\times, \textit{ид}), T\ \ (T), S\ \ (S), S$	$T \rightarrow T \times \textit{ид}$
c	$(\textit{ид}), T\ \ (T), S\ \ (S), S$	—
—	ПУСТО, $T\ \ (T), S\ \ (S), S$	—
—	$(\overline{+}, T), S\ \ (S), S$	$\overline{S} \rightarrow S + T$
	$(\times, \textit{ид}), T\ \ (T), S\ \ S, (S)$	$T \rightarrow T \times \textit{ид}$

Первая из трех последних подстановок приводит к правильному разбору, остальные оказываются непригодными.

4.5. ОБЩАЯ ПРОГРАММА ГРАММАТИЧЕСКОГО РАЗБОРА

Ангер [32] описал некоторый алгоритм грамматического разбора, основанный на методе анализа сверху вниз с применением ряда априорных критериев. Для работы этого алгоритма требуется, чтобы в памяти находилось все предложение.

В отличие от изложенного выше алгоритма разбора сверху вниз, для которого задаются левая граница фразы и название класса, но неизвестна правая граница, алгоритм Ангера всегда знает название класса и обе границы фразы, которую нужно вывести из этого класса. Очевидно, что это справедливо в начальный момент, а последующие шаги обеспечивают сохранение этого свойства.

Предположим, что последовательность слов

$$abcdefg$$

должна быть сопоставлена с правилом подстановки

$$p \rightarrow aQdeR$$

Согласование возможно, только если bc выводимо из Q и fg выводимо из R. Алгоритм состоит в следующем. Нам задаются последовательность слов α и правило подстановки β, и мы хотим показать их совместимость. Правая часть правила подстановки может содержать некоторые слова. Проверяем последовательность α, чтобы убедиться, что эти слова встречаются в правильном порядке. Если это не так, то данный разбор не проходит. Если же это условие выполнено, то по очереди выделяем различные способы разбора (если все они должны быть перечислены). Фрагменты последовательности α, заключенные между опорными словами, должны быть выведены из тех классов, которые встречаются в β между соответствующими словами. При этом мы снова имеем дело с сегментами, которые должны быть выведены из конкретных классов, и можно рекурсивно обратиться к тому же алгоритму сопоставления. Два класса, расположенные рядом в записи правила подстановки, могут породить значительное число различных вариантов.

Далее в алгоритме Ангера используются априорные критерии для отбраковки правил подстановки. Если в последовательности α найдены слова и если выбран конкретный способ расщепления α, то к полученным частям последовательности применяются четыре дополнительных теста, прежде чем пытаться проводить дальнейший разбор с использованием полного алгоритма.

Во-первых, проверяется, что сегмент имеет достаточную длину, чтобы его можно было вывести из данного класса. Таблица минимальных достаточных длин формируется заранее, до начала грамматического разбора.

Второй и третий тесты напоминают априорные критерии для анализа слева направо. Для слов на левом и правом концах сегмента проверяется, могут ли с них начинаться и кончаться фразы, выводимые из данного класса. Соответствующая информация может точно так же храниться в таблице.

Четвертый тест состоит в проверке отсутствия каких-либо слов, которые не могут встречаться в фразах, выводимых из данного класса. Эта информация тоже может храниться в таблицах.

Ангер утверждает, что все эти ускоренные проверки нужны для обеспечения достаточной эффективности алгоритма разбора и что данный алгоритм применительно к ряду грамматик дал хорошие результаты по сравнению с другими методами.

Можно добавлять и другие априорные критерии. Нет нужды проводить разбор фразы по общему алгоритму, если можно написать специальную программу разбора, предназначенную для рассматриваемого класса. Эта задача упрощается тем, что известны обе границы фразы.

4.6. СРАВНЕНИЯ

Общее сравнение различных методов разбора трудно провести по двум причинам. Во-первых, для некоторых грамматик лучшие результаты дают методы разбора сверху вниз, а для других грамматик оказываются предпочтительнее методы разбора снизу вверх, так что получается, что эффективность метода зависит от конкретной грамматики. Во-вторых, если найдена некая грамматика, для которой данный метод оказывается неэффективным, то обычно удается найти преобразованный вариант этой грамматики, для которого тот же метод вполне эффективен. Практика показала, что для применяемых в компиляторах простых грамматик можно подобрать преобразованные варианты, удобные для любого метода. Тем не менее существенным недостатком метода анализа сверху вниз и слева направо является необходимость устранения всех классов, рекурсивных слева.

Гриффитс и Петрик [18] провели сравнение различных алгоритмов разбора на базе гипотетической машины (основанной на машине Тьюринга) и определили число шагов, затрачиваемых этой машиной на разбор различными методами для различных грамматик. Они показали также, что для любой грамматики всегда найдется эквивалентная грамматика, для которой метод разбора сверху вниз оказывается не намного медленнее, чем метод разбора снизу вверх для исходной грамматики.

Большиинство универсальных алгоритмов основано на методах разбора сверху вниз. В то же время большинство алгоритмов специального назначения, ориентированных на ограниченные классы грамматик, в каком-то смысле основываются на разборе снизу вверх.

Если говорить о ранних системах, то компилятор компиляторов Брукера и Морриса основывался на разборе сверху вниз, а в системе PSYCHO Айронса проводился разбор снизу вверх.

5 СПЕЦИАЛЬНЫЕ МЕТОДЫ ГРАММАТИЧЕСКОГО РАЗБОРА

Маловероятно, чтобы методы разбора, порождающие все варианты анализа любого предложения в любой контекстно-свободной грамматике, оказались достаточно быстрыми для того, чтобы их можно было применять в компиляторах. Поэтому для быстрой компиляции обычно применяются алгоритмы разбора, которые порождают только один вариант анализа и пригодны только для ограниченного класса грамматик. Такие алгоритмы могут основываться на разборе сверху вниз или снизу вверх и обычно работают слева направо. Алгоритмы разбора и класс грамматик, к которым они применимы, обычно выбираются таким образом, чтобы существовали априорные критерии, согласно которым на каждом шагу оказывалось бы подходящим только одно правило подстановки, т. е. чтобы можно было получать разбор непосредственно, без возвращения назад. Последнее характерно не для всякого специального метода; в частности, система PSYCHO Айронса применима лишь к некоторым грамматикам, но допускает возвраты.

Первый метод, описываемый в этой главе, может основываться как на разборе сверху вниз, так и на разборе снизу вверх; остальные методы основаны на разборе снизу вверх.

Рассмотрим грамматический анализ сверху вниз и слева направо с тем же априорным критерием, что и в предыдущей главе: правило подстановки применяется только в том случае, если очередное слово предложения может служить началом какого-то примера этой подстановки. Если грамматика такова, что для любого класса ни одно слово не может служить началом примера более чем одной подстановки, то этот априорный критерий означает, что на каждом шаге можно применить только одно правило подстановки. Например, для грамматики

$$S \to aX$$
$$X \to b \mid cX$$

предложения

$$ab$$
$$acb$$
$$\ldots$$
$$ac \ldots\ldots cb$$

можно разобрать сразу, руководствуясь при сопоставлениях класса X тем, что, когда остаток предложения начинается с b, нужно применять первое правило подстановки, а когда остаток начинается с c — второе правило подстановки.

Легко выяснить, обладает ли та или иная грамматика этим свойством. Для этого нужно построить матрицу возможных начальных слов для каждого класса и проверить, пересекаются ли наборы слов, с которых начинаются подстановки для какого-либо класса. Впрочем, маловероятно, чтобы заданная грамматика сразу обладала этим свойством, и поэтому скорей всего потребуется применить специальное преобразование грамматики (см. гл. 6).

Если у грамматики есть это свойство, то алгоритм разбора, описанный в приложении 2, может быть существенно ускорен. Однако еще более быстрый вариант можно получить, если оттранслировать саму данную грамматику в программу соответствующего разбора. Основанный на этом принципе компилятор описан автором в статье [14].

Будем предполагать, как в программе из приложения 2, что переменная *симв* содержит очередное слово из последовательности. Каждый класс можно оттранслировать в рекурсивную процедуру распознавания фраз, выводимых из этого класса. Предыдущая грамматика транслируется в следующую процедуру:

```
procedure S;
begin if симв ≠ 'a' then отказ;
      симв:= следующий;
      X
end;
procedure X;
begin if симв = 'b' then завершение
      else if симв = 'c' then begin симв:= следующий;
                                    X
                               end
      else отказ
end;
симв:= следующий;
S
```

Первое слово заносится в *симв* и вызывается процедура S. Если это слово не есть a, то разбор не получится. Если же это a, то следующее слово заносится в *симв* и вызывается X. Процедура X работает аналогичным образом. Это быстрый процесс, и по такому принципу можно было бы написать

распознаватель, работающий методом подгонки, без знания грамматики.

До сих пор мы имели дело исключительно с грамматиками, в которых правила подстановки определяются только в терминах слов и классов. Иногда оказывается полезным введение пустой подстановки

$$T \rightarrow \varnothing$$

означающей, что класс T может быть заменен на пустую последовательность.

С применением пустой подстановки можно записать грамматику арифметических выражений с идентификаторами и знаками $+$ и $\times$, определяющую тот же язык, что и ранее, но при этом и удовлетворяющую априорному критерию, который требуется для данного метода:

$$S \rightarrow TX$$
$$X \rightarrow \varnothing \mid + TX$$
$$T \rightarrow \textit{ид}\, Y$$
$$Y \rightarrow \varnothing \mid \times \textit{ид}\, Y$$

При использовании пустой подстановки возникает необходимость усилить априорный критерий. Если какому-то классу соответствует пустая подстановка, то наборы слов, с которых начинаются непустые подстановки, должны отличаться друг от друга, а также и от набора слов, которые могут следовать за данным классом в рассматриваемом языке.

Можно повысить эффективность трансляции правил подстановки в программу, если учитывать, что для класса, используемого только в одном месте, процедура может быть записана непосредственно в соответствующем месте. Кроме того, мы можем операторы процедуры для класса в конце правила подстановки заменять на операторы перехода:

```
procedure S;
begin па: if not ид (симв) then отказ;
          симв:=следующий;
          if симв='×' then begin симв:=следующий;
                                 go to па
                           end;
          if симв='+' then begin симв:=следующий;
                                 go to па
                           end
end;
симв:=следующий;
S
```

Этот метод обсуждается далее в гл. 6, где описывается способ построения разбора для исходной грамматики.

5.1. МЕТОДЫ РАЗБОРА СНИЗУ ВВЕРХ

Рассмотрим процесс грамматического разбора снизу вверх и слева направо, при котором всегда удается правильно выбрать правило подстановки для очередного применения. На промежуточном этапе этого процесса левая часть последовательности уже частично сведена в некую последовательность классов и слов, а правая часть исходной последовательности остается неизменной. Это состояние можно описать в виде

$$I_1 I_2 \ldots I_n W_j W_{j+1} \ldots W_t$$

где I_k — либо слово, либо класс, а W_h — слово.

Если очередное применяемое правило подстановки определяется по интервалу от I_1 до I_n и от W_j до W_{j+t-1}, то принято говорить, что данный язык принадлежит классу $LR(t)$. Флойд описал алгоритм разбора для языков из класса $LR(1)$.

Если же очередное правило подстановки определяется по последним m элементам I и по первым s словам W, то говорят, что язык ограничен контекстом (m, s), или принадлежит классу $BC(m, s)$. Иногда $BC(m, s)$ определяется в терминах последних m слов из I. Описываемые ниже алгоритмы разбора предназначаются для языков из $BC(1, 1)$.

Если априорный критерий выбора очередной подстановки включает рассмотрение только двух элементов на стыке I и W, то следует ожидать, что он требует мало времени, и тогда такой метод должен обеспечивать быстрый грамматический разбор.

5.2. ГРАММАТИКИ С ОПЕРАТОРНЫМ ПРЕДШЕСТВОВАНИЕМ

Операторной грамматикой называется грамматика, среди правил подстановки которой нет правил вида

$$A \to \alpha BC\beta$$

где α и β — любые последовательности, возможно пустые. Многие практически полезные грамматики являются операторными грамматиками или могут быть легко преобразованы в такую форму. Например, хотя грамматика Алгола описывается не как операторная грамматика, ее можно сделать операторной, внеся в нее тривиальные изменения.

Определим следующие три возможных отношения между словами, принадлежащими словарю операторной грамматики,

$=$, $<$ и $>$. В общем случае между любой парой слов могут выполняться три, два, одно или ни одного из этих отношений.

Если существует правило подстановки

$$A \to \alpha p q \beta$$

или

$$A \to \alpha p B q \beta$$

где α и β — любые последовательности, возможно пустые, то мы будем говорить, что

$$p = q$$

Если существует подстановка

$$A \to \alpha B q \beta$$

и вывод

$$B \to \tau p$$

или

$$B \to \tau p C$$

то будем говорить, что

$$p > q$$

Если существует подстановка

$$A \to \alpha p B \beta$$

и вывод

$$B \to q \tau$$

или

$$B \to C q \tau$$

то будем говорить, что

$$p < q$$

Если для каждой пары слов выполняется не более чем одно из этих отношений, то данная грамматика называется грамматикой с операторным предшествованием.

Очевидно, что эту информацию полезно учитывать в процессе грамматического разбора. В самом деле, предположим, что упорядоченная пара (p, q) — два соседних слова в частично сведенном предложении, т. е. предположим, что они разделяются только символами классов, но не терминальными символами. В таком случае они принадлежат одной и той же простой фразе тогда и только тогда, когда $p = q$. Простая фраза — это фраза, не содержащая никакой другой фразы [1]). Если $p > q$, то p принадлежит некой фразе, которой не принадлежит q, а классы

---

[1]) Напоминаем, что фраза — это часть последовательности, которая выводится из какого-то класса. — *Прим. перев.*

между p и q также принадлежат этой фразе. Аналогично если $p < q$, то q принадлежит некой фразе, которой не принадлежит p. Поэтому становится легко определять фразовую структуру предложения. Само по себе это не обеспечивает полного построения дерева разбора, так как используемые подстановки могут не быть непосредственно выводимыми, но обычно их совсем легко найти, когда структура известна. Правила подстановки вида

$$A \to B$$

не обнаруживаются в такой фразовой структуре; рассматриваемая структура состоит из подстановок, каждая из которых содержит хотя бы одно слово.

Рассмотрим грамматику

$$S \to T \mid S + T$$
$$T \to P \mid T \times P$$
$$P \to \text{ид} \mid (S)$$

Это грамматика для выражений со знаками $+$, $\times$ и с круглыми скобками.

Выполняются следующие отношения

	$+$	$\times$	(	)	ид
$+$	$>$	$<$	$<$	$>$	$<$
$\times$	$>$	$>$	$<$	$>$	$<$
(	$<$	$<$	$<$	$=$	$<$
)	$>$	$>$	$=$	$>$	
ид	$>$	$>$		$>$	

Произведем теперь разбор предложения

$$a + b \times (c + d)$$

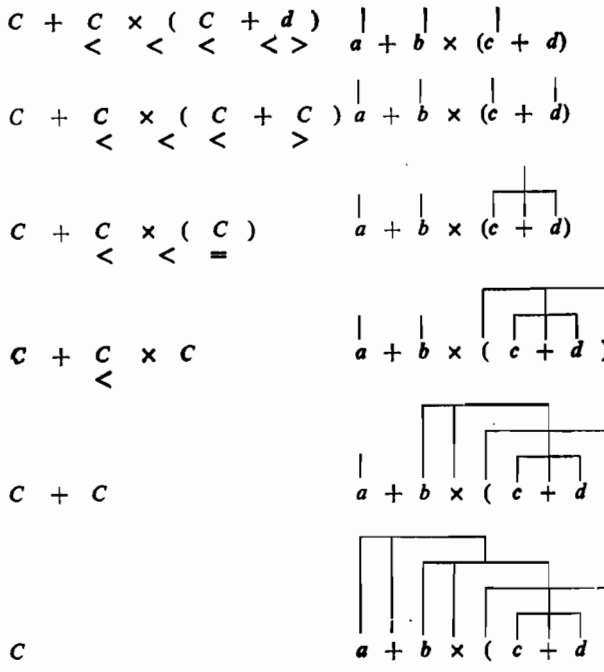
используя символ C для обозначения неизвестных классов и показывая построение дерева фраз:

$$\begin{array}{ccccccccccc} a & + & b & \times & (& c & + & d &) \\ & > & & < & > & & < & > & \end{array} \qquad a + b \times (c + d)$$

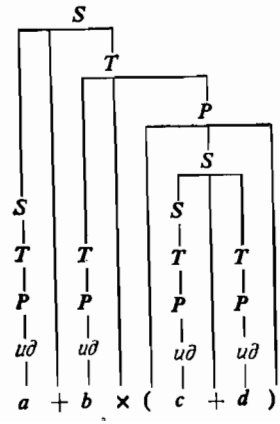
$$\begin{array}{ccccccccccc} C & + & b & \times & (& c & + & d &) \\ & < & > & & < & > & & < & > \end{array} \qquad \begin{array}{c} | \\ a + b \times (c + d) \end{array}$$

$$\begin{array}{ccccccccccc} C & + & C & \times & (& c & + & d &) \\ & < & & < & > & & < & > & \end{array} \qquad \begin{array}{cc} | & | \\ a + b & \times (c + d) \end{array}$$

Заметим, что первое отношение выполняется между + и игнорируется.



Тот факт, что анализ в действительности имеет вид



следует установить, принимая во внимание эффект подстановок типа

$$S \to T$$

и подыскивая подходящие классы для вершин дерева.

Это можно описать в форме разбора слева направо следующим образом. Формируется магазинный список I из слов и классов; в начале разбора этот список пуст. Процесс начинается с занесения первого слова предложения в магазинный список. На каждом промежуточном этапе состояние определяется списком I и остатком предложения:

$$I_1 I_2 \ldots I_n \qquad W_j W_{j+1} \ldots W_h$$

Рассматриваем последнее слово в I; пусть это слово p. Если $p = W_j$ или $p < W_j$, то добавляем W_j к магазинному списку и переходим к следующему слову. Если между p и W_j не выполняется ни одно из отношений $=$, $<$, $>$, то предложение неправильно построено. Если $p > W_j$, то отыскиваем наибольшее множество слов из I, удовлетворяющих отношениям

$$I_a < I_b = I_c = \ldots = p$$

Все слова и классы от I_a до конца магазинного списка, за исключением I_a, соответствуют некоторой подстановке и могут быть свернуты в какой-то класс. Слово W_j не удаляется из последовательности, и тот же процесс повторяется. Единственная трудность состоит в том, что конкретный класс не определяется этим процессом, а должен быть установлен другими средствами. Обычно это не представляет труда, так как фразы часто содержат некоторые отличительные слова. Так, для предыдущей грамматики, если найдена фраза, содержащая $+$, то она может возникнуть только из правила подстановки

$$S \to S + T$$

Заметим, что при этом процессе исключаются из рассмотрения классы, попадающие в магазинный список, и учитываются только отношения между словами.

Если отвлечься от формирования дерева разбора, то можно легко записать этот алгоритм:

```
I:=ПУСТО;
n3: симв:=следующий;
n1:if I=ПУСТО ∨ равно(hd(I), симв) ∨ меньше(hd(I), симв)
   then I:=cons(симв,I)
   else if больше(hd(I),симв)
   then begin W:=hd(I);
        n2:I:=tl(I);
           if I=ПУСТО ∨ меньше(hd(I),W)
           then go to n1
           else go to n2
   end
   else отказ;
   go to n3;
```

Информацию об отношениях между словами можно хранить в виде матрицы, но часто оказывается возможным определить для каждого слова два веса, f и g, таким образом, чтобы

$f(p) = g(q)$ выполнялось при $p = q$
$f(p) < g(q)$ выполнялось при $p < q$
$f(p) > g(q)$ выполнялось при $p > q$

Для предыдущего примера мы можем выбрать

	f	g
+	3	2
×	5	4
(	1	6
)	6	1
ид	5	6

В таком случае выражение

меньше(*hd* (*I*),*симв*)

в нашем алгоритме может быть заменено на

$f(hd(I)) < g(симв)$

и т. д.

Флойд [11] предложил алгоритм исследования грамматики с целью выяснения, является ли она грамматикой с операторным предшествованием, алгоритмы построения матрицы отношений, а также вычисления f и g, если это возможно. Он также доказал теоремы, обосиовывающие эти алгоритмы.

Чтобы показать, что для грамматики с операторным предшествованием не всегда можно подобрать f и g, рассмотрим грамматику

$$S \to aX \mid Yd$$
$$X \to bcb$$
$$Y \to cda$$

для которой, в частности, справедливы следующие отношения

$$a < b$$
$$c = b$$
$$a > d$$
$$c = d$$

Грамматика с операторным предшествованием может оказаться неопределенной. Процесс грамматического разбора только определяет дерево, но не идентифицирует вершины.

Хотя этот способ грамматического разбора очень быстр и весьма широко применяется в компиляторах, особенно для разбора арифметических выражений, множество грамматик, к которым он применим, все же слишком ограничено. Основная трудность состоит в том, что отношения между словами должны выполняться всегда, в любом контексте. Так в Алголе

else $>$:

потому что в описаниях массивов «:» разделяет арифметические выражения. Однако в то же время

else $<$:

потому что в условных операторах «:» указывает метку. Поэтому Алгол не является грамматикой с операторным предшествованием. Впрочем, контексты этих двух применений разделителя «:» совершенно различны, так что это противоречие является кажущимся. Разумеется, в процессе предварительного просмотра программы можно было бы заменить «:» в описаниях массивов на некий новый символ.

5.3. ГРАММАТИКИ С ПРЕДШЕСТВОВАНИЕМ

Один тип грамматик с предшествованием, выходящий за рамки операторных грамматик, применялся Виртом и Вебером [33]. В их методе отношения $=$, $<$ и $>$ определяются между парами элементов как слов, так и классов. При этом, если существует правило подстановки

$$A \to \alpha l_1 l_2 \beta$$

мы будем говорить, что $I_1 = I_2$. Если существует правило подстановки

$$A \to \alpha B I_2 \beta$$

и вывод

$$B \to \sigma I_1$$

или если существует правило подстановки

$$A \to \alpha B C \beta$$

и выводы

$$B \to \sigma I_1$$
$$C \to I_2 \delta$$

то будем говорить, что $I_1 > I_2$.

Если же существует правило подстановки

$$A \to \alpha I_1 B \beta$$

и вывод

$$B \to I_2 \sigma$$

то будем говорить, что $I_1 < I_2$. Если между каждой парой элементов выполняется не более чем одно из этих отношений и если никакие два правила подстановки не имеют одинаковых правых частей, то такая грамматика является грамматикой с предшествованием.

Предположим, что в частично сведенном предложении элемент I_1 непосредственно предшествует элементу I_2. Они принадлежат одной и той же простой фразе тогда и только тогда, когда $I_1 = I_2$. Если $I_1 < I_2$, то I_2 принадлежит какой-то простой фразе, не содержащей I_1. А если $I_1 > I_2$, то I_1 принадлежит какой-то простой фразе, не содержащей I_2. Как и для грамматик с операторным предшествованием, для рассматриваемых грамматик легко определять фразовую структуру предложений. Однако в этом случае удается построить полный разбор, так как правила подстановки оказываются единственными. Можно организовать матрицу отношений и использовать ее в алгоритме разбора аналогично тому, как это делается для грамматик с операторным предшествованием. Иногда удается построить и функции f и g, обеспечивающие представление отношений. Вирт и Вебер предлагают алгоритмы для выявления отношений.

Заметим, что обычно оказывается необходимым преобразовать грамматику. Например, в грамматике

$$S \to T \mid S + T$$
$$T \to P \mid T \times P$$
$$P \to \text{ид} \mid (S)$$

мы имеем $+ = T$, так как $S \to S + T$, и в то же время $+ < T$, так как $S \to S + T$ и $T \to T \times P$. Имеется также противоречие в том, что одновременно выполняются отношения $(= S$ и $(< S$. Эта грамматика может быть преобразована путем добавления дополнительных классов

$$S \to U$$
$$U \to T \mid U + T$$
$$T \to V$$
$$V \to P \mid V \times P$$
$$P \to \text{ид} \mid (S)$$

В результате получаем грамматику с предшествованием.

На примере этой модифицированной грамматики на следующей странице показан разбор слева направо с использованием данного метода; построение дерева анализа исключается из рассмотрения.

Рассмотренный способ оказывается вполне удовлетворительным, хотя он и требует использования специального метода отыскания классов, соответствующих той или иной подстановке.

5.4. МАТРИЦЫ ПЕРЕХОДОВ

Этот метод грамматического разбора, описанный в работе [31], весьма напоминает метод разбора для грамматик с операторным предшествованием. Разбор производится тоже снизу вверх. Используются два магазинных списка. В один из них заносятся операторы, такие, как $+$ и $\times$. В другой заносятся операнды. Предполагается, что операторы и операнды различимы между собой. Оператор с вершины магазинного списка и очередной оператор из предложения служат для выбора из матрицы программы для выполнения.

Такая программа обычно либо добавляет новый оператор к операторному магазинному списку, либо исключает операторы из операторного магазинного списка и операнды из списка операндов, формируя из них новый операнд, который добавляется к списку операндов.

Все это очень близко к методу операторного предшествования, причем два магазинных списка служат для того, чтобы не иметь дела с классами. Методы обработки грамматик с целью получения матрицы переходов на подпрограммы описаны в статье [9].

a + b × (c + d)
 >

P + b × (c + d)
 >

V + b × (c + d)
 >

T + b × (c + d)
 >

U + b × (c + d)
 = < >

U + P × (c + d)
 = < >

U + V × (c + d)
 = < = << >

U + V × (P + d)
 = < = << >

U + V × (V + d)
 = < = << >

U + V × (T + d)
 = < = << >

U + V × (U + d)
 = < = << = < >

U + V × (U + P)
 = < = << = < >

U + V × (U + V)
 = < = << = < >

U + V × (U + T)
 = < = << = = >

U + V × (U)
 = < = << >

U + V × (S)
 = < = <= =

U + V × P
 = < = =

U + V
 = <

U + T
 = =

U

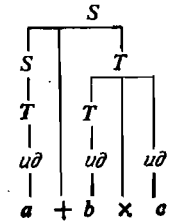
S

Нам уже известны различные причины, по которым может оказаться желательным преобразование заданной грамматики в другой эквивалентный вид. Например, может возникнуть необходимость исключить левую рекурсию, или привести грамматику в нормальную форму, или преобразовать ее в грамматику с предшествованием. Однако применение исходной грамматики, вероятно, определяется в терминах структуры разбора, порождаемой именно этой грамматикой, а не в терминах структуры разбора, порождаемой измененной грамматикой. Поэтому необходимы методы возвращения к разбору в рамках исходной грамматики. Описываемый ниже метод такого возвращения использовался в одной системе построения компиляторов [14].

Введем третий тип элементов и будем называть элементы этого типа *действиями*. Для обозначения действий будем употреблять жирный шрифт. Действия похожи на слова тем, что они являются терминальными и к ним неприменимы дальнейшие подстановки, но их следует интерпретировать несколько иначе. Предложение, порожденное грамматикой с действиями, будет содержать смесь слов и действий, однако грамматическому разбору будет подлежать предложение, состоящее только из слов. Во время процесса сопоставления при разборе действия исключаются из рассмотрения. Например, грамматика

$$S \to T \mid S + T$$
$$T \to \textit{ид} \mid T \times \textit{ид}$$

порождает разбор

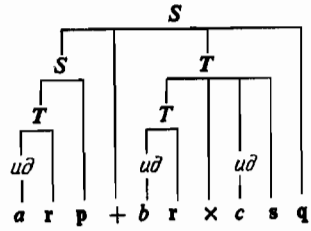


В соответствующей грамматике с действиями

$$S \to T\mathbf{p} \mid S + T\mathbf{q}$$

$$T \to \textit{ид}\,\mathbf{r} \mid T \times \textit{ид}\,\mathbf{s}$$

соответствующий разбор имеет вид



Поскольку различные действия отнесены в конец каждого правила подстановки, то при заданной последовательности слов и действий

$$a\mathbf{r}\mathbf{p} + b\mathbf{r} \times c\mathbf{s}\mathbf{q}$$

не представляет труда построить дерево грамматического разбора.

Предложение просматривается слева направо, и каждый раз, когда встречается действие, соответствующее правило подстановки применяется к элементам, находящимся слева.

Однако совершенно необязательно использовать в разных правилах подстановки разные действия или помещать действия в конец правил подстановки. Если вместо того, чтобы преобразовать некую грамматику, мы преобразуем ту же грамматику, но с действиями в конце правил подстановки, то получим новую грамматику, в которой те же действия будут занимать другие места. Например, приведенная выше грамматика может быть преобразована в грамматику

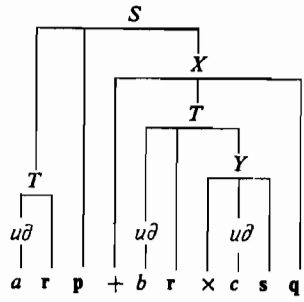
$$S \to T\mathbf{p} \mid T\mathbf{p}X$$

$$X \to + T\mathbf{q} \mid + T\mathbf{q}X$$

$$T \to \textit{ид}\,\mathbf{r} \mid \textit{ид}\,\mathbf{r}Y$$

$$Y \to \times \textit{ид}\,\mathbf{s} \mid \times \textit{ид}\,\mathbf{s}Y$$

Используя эту грамматику для разбора того же самого предложения, мы получаем



т. е. совсем другую структуру разбора, но действия встречаются на тех же местах, что и прежде. Поэтому, оперируя действиями в преобразованной грамматике, мы можем восстановить исходное дерево грамматического разбора.

Заметим, что в этом случае мы преобразовали грамматику с левой рекурсией в грамматику без левой рекурсии.

В гл. 7 будет показано, что процесс компиляции часто может быть описан с помощью действий в некоторой грамматике, причем эти действия выполняются над стеком результатов. При этом не возникает необходимости строить дерево, а действия в преобразованной грамматике применяются непосредственно для компиляции.

6.1. ПРЕОБРАЗОВАНИЯ ДЛЯ ИСКЛЮЧЕНИЯ ЛЕВОЙ РЕКУРСИИ

Грейбах показал, что для любой заданной грамматики всегда найдется эквивалентная грамматика без левой рекурсии. В литературе описаны различные преобразования, обеспечивающие удаление левой рекурсии [14, 19, 26, 35]. Ниже излагается такое преобразование, опубликованное ранее в работе [14].

Будем обозначать пустую фразу через $\varnothing$, а недопустимую фразу через ε. Если в предложении встречается символ $\varnothing$, он может быть исключен; если же встречается символ ε, то предложение некорректно. Разумеется, можно установить соответствие между символами $\varnothing$ и ε, с одной стороны, и символами 0 и 1, с другой. Мы будем использовать символ δ_{rs} вместо $\varnothing$, если $r = s$, и вместо ε, если $r \neq s$. Рассмотрим сначала очень простое рекурсивное слева описание

$$X \to a \mid Xb$$

определяющее предложения, которые начинаются с символа a и далее содержат любое количество символов b. Грамматика

$$X \to aY$$
$$Y \to \varnothing \mid bY$$

определяет те же предложения. Здесь Y — вспомогательный класс, введенный при преобразовании. В общем случае производится аналогичное построение.

Предположим, что имеется n классов L_i и что все они образуют цикл левой рекурсии. Это означает, что для любых L_i, L_j существует вывод $L_i \to L_j\alpha$. Если $i = j$, то заведомо $\alpha \neq \varnothing$, но при $i \neq j$ допускается $\alpha = \varnothing$. Будем пользоваться сокращенным обозначением

$$L_i \to A_i \mid L_jB_{ij}$$

которое означает

$$L_1 \to A_1 \mid L_1B_{11} \mid L_2B_{21} \mid \ldots \mid L_nB_{n1}$$
$$L_2 \to A_2 \mid L_1B_{12} \mid L_2B_{22} \mid \ldots \mid L_nB_{n2}$$
$$\ldots\ldots\ldots\ldots\ldots\ldots$$
$$L_n \to A_n \mid L_1B_{1n} \mid L_2B_{2n} \mid \ldots \mid L_nB_{nn}$$

Собираем вместе все правила подстановки для L_i и по мере необходимости определяем новые вспомогательные классы, чтобы привести правила подстановки в вид

$$L_i \to A_i \mid L_jB_{ji}$$

где A_i не может начинаться с какого-либо класса L_j и не существует вывода $A_i \to L_j\alpha$. Может потребоваться, чтобы правила подстановки для A_i или B_{ji} содержали $\varnothing$ или ε.

После этого эквивалентное множество подстановок для L_i без рекурсивных слева классов задается так:

$$L_i \to A_iX_{ji}$$
$$X_{rs} \to \delta_{rs} \mid B_{rt}X_{ts}$$

где X_{rs} — вспомогательные классы, число которых равно n^2.

Во-первых, заметим, что любая фраза, выводимая из L_i в первой грамматике, выводима из L_i и во второй грамматике и наоборот. В самом деле, фразы, выводимые из L_i в первой грамматике, имеют вид

$A_pB_{pq}B_{qr} \ldots B_{zi}$ (без повторения сокращенных обозначений)

и то же самое справедливо для второй грамматики. Более того, если все B_{ij} и A_i различны между собой, то обе грамматики не

содержат двусмысленностей. Если же среди них есть повторения и при этом существуют d выводов некой фразы в одной грамматике, то существуют d различных способов ее вывода и в другой грамматике.

Во-вторых, не существуют выводы

$$X_{ij} \to X_{ij}$$

так как такой вывод смог бы возникнуть, только если бы существовал вывод

$$B_{ip}B_{pq}B_{qr}\ldots B_{zi} \to \varnothing$$

Но это невозможно, так как при этом существовал бы вывод $L_i \to L_i$ в первой грамматике. Вывод $L_i \to L_i$ во второй грамматике повлек бы за собой аналогичный вывод в первой, что исключено.

В-третьих, вторая грамматика может содержать правила подстановки, содержащие ε. Можно исключить эти правила подстановки, не оставив никаких классов, для которых были бы невозможны терминальные выводы [1]). В первой грамматике существует вывод

$$L_i \to L_j\alpha$$

для каждой пары i, j. Поэтому для любых i, j найдется последовательность индексов p, q, ..., z, такая, что для строки

$$B_{jp}B_{pq}\ldots B_{zi}$$

существует терминальный вывод.

Поэтому для любого класса X_{rs} найдется вывод

$$X_{rs} \to B_{rh}B_{hk}\ldots B_{ms}X_{ss}$$

а поскольку $X_{ss} \to \varnothing$, то для X_{rs} существует корректный терминальный вывод.

Следовательно, данные грамматики эквивалентны. Используя введенные действия, можно восстановить разбор в соответствии с первой грамматикой, так как

$$L_i \to A_i\mathbf{p}_i \mid L_iB_{ji}\mathbf{q}_{ji}$$

преобразуется в

$$L_i \to A_j\mathbf{p}_jX_{ji}$$
$$X_{rs} \to \delta_{rs} \mid B_{ri}\mathbf{q}_{ri}X_{is}$$

Путем такого последовательного удаления рекурсивных слева правил можно изменять грамматику до тех пор, пока она не освободится от левой рекурсии.

---

[1]) То есть выводы терминальных последовательностей. — *Прим. перев.*

6.2. ДРУГИЕ ПРЕОБРАЗОВАНИЯ

Оба метода разбора сверху вниз и снизу вверх, по-видимому, наиболее эффективны тогда, когда правила подстановки для каждого класса начинаются, по мере возможности, с различных и непересекающихся наборов слов. Очевидно, что в таком случае априорные критерии, оперирующие с набором возможных начальных слов для правил подстановки, будут отсекать больше правил подстановки. Предельным является описанный в гл. 5 случай, когда каждому классу соответствуют только свои индивидуальные слова, начинающие соответствующие правила подстановки. В этом случае методы разбора сверху вниз и снизу вверх сливаются в один метод и анализ производится исключительно быстро. Поэтому целесообразно рассмотреть преобразования, приводящие к такому результату.

Мы можем применять к правилам подстановки дистрибутивные преобразования совершенно аналогично дистрибутивным преобразованиям выражений, содержащих знаки $+$ и $\times$. Грамматика

$$X \to YZ$$
$$Y \to A \mid B$$

эквивалентна грамматике

$$X \to AZ \mid BZ$$

а грамматика

$$X \to CD \mid CE$$

эквивалентна грамматике

$$X \to CY$$
$$Y \to D \mid E$$

Последовательным применением таких преобразований к заданной грамматике обычно можно сократить совпадения начальных элементов в правилах подстановки, а нередко удается совсем исключить такие совпадения [14]. В самом деле, если для некоего класса имеются два правила подстановки, начинающиеся с одинаковых элементов

$$X \to AB_1B_2 \ldots \mid AC_1C_2 \ldots$$

то мы можем ввести новый класс и записать

$$X \to AY$$
$$Y \to B_1B_2 \ldots \mid C_1C_2 \ldots$$

Если же для некоего класса имеются два правила подстановки, в которых нет явного совпадения, но которые могут на-

чинаться с одного и того же слова, например

$$X \to AB_1B_2 \ldots \mid CD_1D_2 \ldots$$
$$A \to P \mid Q$$
$$C \to P \mid R$$

то мы можем сначала записать

$$X \to PB_1B_2 \ldots \mid QB_1B_2 \ldots \mid PD_1D_2 \ldots \mid RD_1D_2 \ldots$$

а затем

$$X \to PY \mid QB_1B_2 \ldots \mid RD_1D_2 \ldots$$
$$Y \to B_1B_2 \ldots \mid D_1D_2 \ldots$$

К сожалению, существуют два затруднения, которые препятствуют тому, чтобы этот метод всегда исключал такие совпадения. Первое состоит в необходимости использовать пустую фразу $\varnothing$, а тем самым включать в рассмотрение слова, следующие за начальным классом. Имея исходную грамматику

$$X \to A \mid AB$$
$$B \to XC$$
$$C \to B \mid D$$

мы получаем из нее

$$X \to AZ$$
$$Z \to \varnothing \mid B$$
$$Y \to XC$$
$$C \to B \mid D$$

и возникает трудность в связи с правилом

$$Z \to \varnothing \mid B$$

так как за классом Z может следовать B.

Второе затруднение возникает из-за того, что процесс преобразования может оказаться бесконечным циклом. Рассмотрим грамматику

$$A \to X \mid Y$$
$$X \to B \mid LX$$
$$Y \to C \mid LY$$

Сначала мы получаем из нее

$$A \to B \mid C \mid LX \mid LY$$
$$X \to B \mid LX$$
$$Y \to C \mid LY$$

а затем

$$A \to B \mid C \mid LZ$$
$$Z \to X \mid Y$$
$$X \to B \mid LX$$
$$Y \to C \mid LY$$

Теперь мы находимся в таком же положении относительно классов Z, X и Y, в каком были первоначально относительно A, X и Y, и поэтому процесс никогда не закончится. В данном случае мы могли бы заметить аналогию и записать

$$A \to B \mid C \mid LA$$
$$X \to B \mid LX$$
$$Y \to C \mid LY$$

Другой пример неудачного преобразования возникает при исходной грамматике

$$S \to a \mid aSa$$

Для получения грамматик с операторным предшествованием или грамматик с предшествованием используются сравнительно простые преобразования. Обычно они сводятся к исключению нерекурсивных классов или к введению новых классов, как показано в гл. 5.

7 ИСПОЛЬЗОВАНИЕ ГРАММАТИЧЕСКОГО АНАЛИЗА ДЛЯ КОМПИЛЯЦИИ

В этой книге речь идет в основном о грамматическом анализе предложений, а не о сопутствующей этому анализу последовательной компиляции. Последнему вопросу посвящен целый ряд статей [4, 6, 10, 20, 21, 28, 30, 31]. Можно дождаться завершения построения полного дерева грамматического анализа, а затем обрабатывать его. Другой возможный вариант состоит в том, чтобы выполнять действия компиляции непосредственно в процессе грамматического анализа. Краткое описание такого способа приводится ниже.

Рассмотрим некую грамматику с действиями в концах правил подстановки. В процессе разбора слева направо у нас формируется магазинный список. Рассматриваемые действия либо добавляют нечто к этому магазинному списку, либо извлекают из него некоторые элементы и помещают результат снова в магазинный список. Предположим сначала для простоты, что мы хотим преобразовывать выражения вида

$$a + b + c + d$$

в такую форму [1]:

ВЫБ, a
СЛ, b
СЛ, c
СЛ, d

Будем использовать грамматику

$$S \rightarrow \textit{ид}\,\mathbf{p} \mid S + \textit{ид}\,\mathbf{q}$$

Предполагается, что интерпретация *ид* добавляет имя идентификатора к магазинному списку. Действие **p** выполняется после первого идентификатора следующим образом [2]:

p означает x: == *извлечь*; *поместить* (*сочленить* ('ВЫБ', x))

[1]) ВЫБ — занесение содержимого ячейки памяти в сумматор; СЛ — прибавление к сумматору содержимого ячейки памяти. — *Прим. перев.*

[2]) Операция *извлечь* означает выборку из магазинного списка, а *поместить* — занесение в магазинный список; операция *сочленить* применяется к двум или более операндам и сцепляет эти операнды в одну строку. — *Прим. перев.*

Действие **q** выполняется, когда двум верхним элементам магазинного списка соответствуют S и *ид*:

q означает x : = *извлечь*; y : = *извлечь*;
поместить (*сочленить* (y, *сочленить* ('СЛ', x)))

Будем использовать эту грамматику в преобразованном виде:

$$S \rightarrow \textit{ид}\mathbf{p}X$$
$$X \rightarrow \varnothing \mid + \textit{ид}\mathbf{q}X$$

Ниже перечисляются последовательные шаги:

Предложение	Используемое действие	Магазинный список
$a+b+c+d$	*и* —	—
$+b+c+d$	*д*	('a')
$+b+c+d$	**p**	('ВЫБ, a')
$+c+d$	*ид*	('b', 'ВЫБ, a')
$+c+d$	**q**	('ВЫБ, a СЛ, b')
$+d$	*ид*	('c', 'ВЫБ, a СЛ, b')
$+d$	**q**	('ВЫБ, a СЛ, b СЛ, c')
—	*ид*	('d', 'ВЫБ, a СЛ, b СЛ, c')
—	**q**	('ВЫБ, a СЛ, b СЛ, c СЛ, d')

А если бы мы захотели, чтобы из выражения

$$a+b+c+d$$

получалось

ВЫБ, d
СЛ, c
СЛ, b
СЛ, a

то мы воспользовались бы грамматикой

$$S \rightarrow \textit{ид}\mathbf{p} \mid \textit{ид} + S\mathbf{r}$$

в которой *ид* и **p** действуют так же, как в предыдущем примере. Действие же **r** над двумя верхними элементами магазинного списка, соответствующими классам *ид* и S, выполняется следующим образом:

r означает x: = *извлечь*; y: = *извлечь*;
поместить (*сочленить* (x, *сочленить* ('СЛ', y)))

Теперь последовательность шагов имеет следующий вид:

Предложение	Используемое действие	Магазинный список
$a+b+c+d$	—	—
$+b+c+d$	*ид*	(‘*a*’)
$+c+d$	*ид*	(‘*b*’, ‘*a*’)
$+d$	*ид*	(‘*c*’, ‘*b*’, ‘*a*’)
—	*ид*	(‘*d*’, ‘*c*’, ‘*b*’ ‘*a*’)
—	p	(‘ВЫБ, *d*’, ‘*c*’, ‘*b*’, ‘*a*’)
—	r	(‘ВЫБ, *d* СЛ, *c*’ ‘*b*’, ‘*a*’)
—	r	(‘ВЫБ, *d* СЛ, *c* СЛ, *b*’, ‘*a*’)
—	r	(‘ВЫБ, *d* СЛ, *c* СЛ, *b* СЛ, *a*’)

В качестве более сложного примера рассмотрим грамматику

$$S \to \textit{ид} := E$$
$$E \to T \mid E + T \mid E - T$$
$$T \to \textit{ид} \mid T \times \textit{ид} \mid T/\textit{ид}$$

Мы можем ввести компилирующие действия

$$S \to \textit{ид} := E;\ \mathbf{p}$$
$$E \to T \mid E + T\mathbf{q} \mid E - T\mathbf{r}$$
$$T \to \textit{ид}\mathbf{s} \mid T \times \textit{ид}\mathbf{t} \mid T/\textit{ид}\mathbf{u}$$

где [1])

p означает *поместить* (*сочленить* (*извлечь*, *сочленить* (‘ЗАП’, *извлечь*)))

q означает *поместить* (*сочленить* (*извлечь*, *сочленить* (‘ЗАП, *W*1’, *сочленить* (*извлечь*, ‘СЛ, *W*1’))))

r означает *поместить* (*сочленить* (*извлечь*, *сочленить* (‘ЗАП, *W*1’, *сочленить* (*извлечь*, ‘ВЫЧ, *W*1’))))

s означает *поместить* (*сочленить* (‘ВЫБ’, *извлечь*))

t означает *z*: = *извлечь*; *поместить* (*сочленить* (*извлечь*, *сочленить* (‘УМ’, *z*)))

u означает *z*: = *извлечь*; *поместить* (*сочленить* (*извлечь*, *сочленить* (‘ДЕЛ’, *z*)))

---

[1]) ЗАП — запись содержимого сумматора в ячейку памяти; ДЕЛ — деление; УМ — умножение содержимого сумматора на содержимое ячейки памяти; ВЫЧ — вычитание. — *Прим. перев.*

Например, это обеспечит трансляцию формулы

$$g:=a+b\times c/d+e\times f$$

в фрагмент программы:

ВЫБ, *e*
УМ, *f*
ЗАП, *W*1
ВЫБ, *b*
УМ, *c*
ДЕЛ, *d*
СЛ, *W*1
ЗАП, *W*1
ВЫБ, *a*
СЛ, *W*1
ЗАП, *g*

Преобразуем эту грамматику и получим новую грамматику:

$S \to$ *ид*: = E **p**
$E \to TX$
$X \to \varnothing \mid + T\,\mathbf{q}X \mid - T\,\mathbf{r}X$
$T \to$ *ид* **s** Y
$Y \to \varnothing \mid \times$ *ид* **t** $Y \mid /$ *ид* **u** Y

Эта новая грамматика может быть оттранслирована методом, описанным в начале гл. 5. В результате получаем программу

```
procedure S;
begin if not ид (симв) then отказ;
    симв: = следующий;
    if симв ≠ ':' then отказ;
    симв: = следующий;
    if симв ≠ '=' then отказ;
    симв: = следующий;
    E;
    if симв ≠ ';' then отказ;
    p; comment вызов действия для p;
end;
procedure E;
```

```
begin T;
   na: if симв = '+' then begin симв:= следующий;
                                T;
                                q;
                                go to na
                          end
     else if симв = '—' then begin симв:= следующий;
                                   T;
                                   r;
                                   go to na
                             end
end;
procedure T;
begin if not ид (симв) then отказ;
      s;
   nb: if симв = '×' then begin симв:= следующий;
                                if not ид (симв) then отказ;
                                t;
                                go to nb
                          end
     else if симв = '/' then begin симв:= следующий;
                                   if not ид (симв) then отказ;
                                   u;
                                   go to nb
                             end
end;
симв:= следующий;
S
```

Эта программа выполняет требуемую функцию, и процесс трансляции данной грамматики в программу полностью автоматизируется.

Реальные компиляторы выполняют гораздо более сложные задачи, но на этапе анализа в них обычно используются такие же методы.

7.1. НЕДОПУСТИМЫЕ ПРЕДЛОЖЕНИЯ

Если программа грамматического разбора сталкивается с предложением, которое не принадлежит данной грамматике, то она, конечно, обнаружит ошибку. Однако желательно

указывать такую ошибку наглядно, и хотелось бы продолжать разбор, чтобы найти последующие ошибки. Автору неизвестны публикации, в которых предлагались бы какие-нибудь вполне удовлетворительные методы, обеспечивающие такие возможности в общем случае.

Если грамматический анализ осуществляется последовательным просмотром предложения, по одному слову слева направо, то можно указывать место в предложении, где произошел заключительный отказ. Однако человек не всегда согласится с такой локализацией места ошибки. Например, пропущенная закрывающая скобка в арифметическом выражении не проявится как ошибка, пока не закончится все выражение. Тем не менее обычно оказывается достаточным указать место, где при анализе произошел отказ, и, по возможности, слово, наличие которого позволило бы благополучно провести анализ. Если же алгоритм грамматического анализа основан на том, что каждый вариант разбора исследуется, пока нет отказов, а в случае отказа происходит возврат и выбирается другой вариант разбора, то указание ошибок может оказаться более затруднительным. В этом случае можно было бы указывать место, где произошел отказ при самом продолжительном варианте разбора.

Более значительные трудности вызывает проблема возобновления разбора после обнаружения ошибки, с тем чтобы попытаться довести его до завершения. Маловероятно, чтобы для этого пригодились универсальные общие методы. В большинстве компиляторов производится переход к некоей опознаваемой позиции вроде конца оператора и разбор проводится далее с этого места. Обычно выбор таких позиций производится применительно к конкретной ситуации.

Если алгоритм грамматического анализа способен выполнять одновременно несколько вариантов разбора, иногда можно сделать несколько различных предположений об ошибках и, прокорректировав их соответствующим образом, продолжать различные варианты разбора. Например, можно предположить, что в точке обнаружения ошибки пропущено слово, или вставлено лишнее слово, или неправильное слово стоит на месте нужного. В литературе описаны такие методы, но они применимы только в простых случаях, а в более сложных ситуациях они могут оказаться непригодными.

ПРИЛОЖЕНИЕ 1

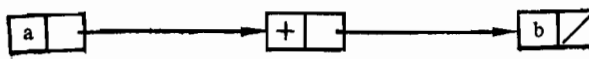
ЭЛЕМЕНТАРНАЯ ОБРАБОТКА СПИСКОВ

В программах грамматического анализа, приводимых в этой книге, используются методы обработки списков. Это объясняется тем, что нельзя заранее предсказать, сколько памяти потребуется для выполнения этих программ, а также тем, что грамматики и грамматический разбор очень удобны для представления в виде списков. Разумеется, можно было бы преобразовать эти программы, использовав, например, массивы для представления грамматик. Ниже дается краткий обзор методов обработки списков, достаточный для того, чтобы понимать наши программы.

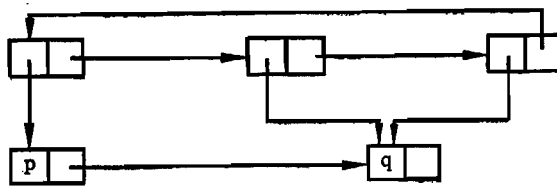
В памяти вычислительной машины отводится определенная область для хранения списков, и эта область разделяется на ячейки, в каждую из которых могут поместиться два элемента. Элементы, которые заносятся в эти ячейки, представляют собой либо адреса других ячеек, либо части данных, называемые атомами. Название «атомы» выбрано потому, что структура этих атомов не принимается во внимание системой обработки списков. Мы могли бы выписать содержимое этих ячеек и указать их явные адреса, но такой способ записи оказался бы гораздо менее удобным для практического применения. Вместо этого мы будем использовать любой из следующих двух способов обозначения. В первом способе ячейка, содержащая два элемента, представляется прямоугольником

а элементы ячейки, являющиеся адресами других ячеек, представляются стрелками, направленными в эти ячейки,

Атомы представляются путем записи их внутри прямоугольников



Косая черта представляет специальный атом, называемый ПУСТО. Структуры любой сложности могут быть построены по следующему образцу:

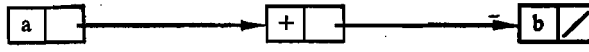


Заметим, что эти структуры могут содержать циклы, а на одну ячейку может быть направлено любое число указателей.

Принято соглашение называть левую часть прямоугольника началом, а правую часть — остатком. Другой способ обозначения состоит в том, чтобы представлять в виде

$$(a, +, b)$$

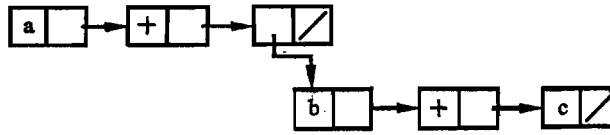
список



Каждый элемент списка помещается в начало ячейки, а остатки используются для того, чтобы связать между собой ячейки, образующие список. Аналогично запись

$$(a, +, (b, +, c))$$

представляет список



Программы написаны на Алголе, в который включены дополнительно переменные, принимающие списковые значения. Для выполнения элементарных операций обработки списков используются три функции. Если x — список, то $hd(x)$ представляет собой значение начального элемеита этого списка. Аналогично $tl(x)$ задает остаток списка x. Значением функции cons (x, y) является ячейка, содержащая x в своей начальной части и y в остатке. Заметим, что для построения списка

$$(a, b, c)$$

мы пишем

$$cons(a, cons(b, cons(c, \text{ПУСТО})))$$

Заметим также, что

$$hd\,(cons\,(x,\ y)) = x$$
$$tl\,(cons\,(x,\ y)) = y$$

но неверно, что

$$cons\,(hd\,(x),\ tl\,(x)) = x$$

потому что функция *cons* образует новую ячейку. Значение $cons\,(hd\,(x),\ tl\,(x))$ представляет собой копию ячейки x, но это не сама ячейка x.

В качестве примера рассмотрим программу, которая переворачивает список m и помещает результат на место n:

```
n: = ПУСТО;
r: if m ≠ ПУСТО then begin n: = cons (hd (m), n);
                           m: = tl (m);
                           go to r
                     end
```

ПРИЛОЖЕНИЕ 2

АЛГОРИТМ ГРАММАТИЧЕСКОГО РАЗБОРА СВЕРХУ ВНИЗ

Функции *hd*, *tl* и *cons* описаны в приложении 1. Функция *следующий* считывает следующее слово из предложения; *терминальный* — логическая функция, которая является истиной, если ее аргумент — слово, и ложью, если ее аргумент — класс. Функция *априори* принимает значение истина, если ее аргумент, являющийся правилом подстановки, удовлетворяет априорным критериям.

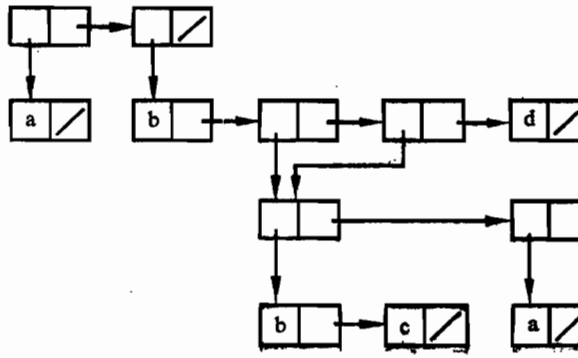
В машинной памяти слова представляются некоторым стандартным способом, мы будем соответственно заключать их в кавычки. Классы представляются списками соответствующих вариантов подстановок; каждое правило подстановки представляется последовательным списком составляющих компонент. Специальному классу, который определяет предложения, соответствует перемеинная *предложение*. Так, например, для грамматики

$$S \rightarrow a \mid bTTd$$
$$T \rightarrow bc \mid a$$

переменная *предложение* содержит список

(('*a*'), ('*b*', (('*b*', '*c*'), ('*a*')), (('*b*', '*c*'), ('*a*')), ('*d*'))

Разумеется, двум вхождениям *T* будут соответствовать указатели на один и тот же подсписок:



Наиболее интересны грамматики, включающие рекурсивные классы; для представления таких классов используются циклические списки.

В описываемой программе переменная *la* служит для хранения списка всех текущих частично проведенных вариантов разбора; в переменной *симв* хранится текущее слово. В результате сопоставления текущего слова и частичных разборов некоторые вариаиты разбора отпадают и забываются, некоторые переносятся в список *lb*, состоящий из вариантов частичного разбора, подготовленных для сопоставления со следующим словом, а остальные порождают более развитые варианты разбора, которые снова помещаются в *la* и далее обрабатываются аналогичным образом. Когда список *la* окажется пустым, в него будет перенесен список *lb* и весь процесс повторится. Варианты разбора, которые завершаются в конце предложения, являются правильными грамматическими разборами данного предложеиия.

Каждый частичный разбор представляется списком концов правил подстановки. Этот список указывает структуру, которую должен иметь остаток предложения для того, чтобы соответствовать этому варианту разбора. Например, если мы анализируем предложение *babcd* в грамматике

$$S \rightarrow a \mid bTTd$$

$$T \rightarrow bc \mid a$$

и уже обработали *bab*, то останется частичный разбор

$$((\text{'}c\text{'}),\ (\text{'}d\text{'}))$$

означающий, что остаток предложения должен состоять из слова *c*, а следом за ним *d*. Запись

$$(\text{'}c\text{'})$$

остается после подстановки $T \rightarrow bc$, а запись

$$(\text{'}d\text{'})$$

остается после подстановки $S \rightarrow bTTd$.

```
        la: = cons (cons (cons (предложение, ПУСТО), ПУСТО),
        ПУСТО);
    z4 : lb: = ПУСТО; симв: = следующий;
    zz : if la ≠ ПУСТО then
        begin lc: = hd (la); la: = tl (la);
            z1 : if lc = ПУСТО then begin if последнее слово
                                          then успех end
                                           else go to zz;
            ld: = hd (lc); lc: = tl (lc);
            if ld = ПУСТО then go to z1;
            le: = hd (ld); ld: = tl (ld);
            if терминальный (le) then begin if le = симв
                                        then lb: = cons (cons (ld,
                                                lc), lb);
                                                go to zz
                                        end;
            z3 : if le ≠ ПУСТО then
                begin if not априори (hd (le)) then go to z3;
                    la: = cons (cons (hd (le), cons (ld, le)), la);
                    le: = tl (le);
                go to z3
                end;
                go to z2
        end;
        if lb = ПУСТО then отказ;
        la: = lb;
        go to z4;
```

СПИСОК ЛИТЕРАТУРЫ

Принятые сокращения:

ACM Association for Computing Machinery
AFIPS American Federation of Information Processing Societies
CACM Communications of the Association for Computing Machinery
IEEE Institute of Electrical and Electronic Engineers
IFIP International Federation for Information Processing
JACM Journal of the Association for Computing Machinery
SJCC Spring Joint Computer Conference

1. Bar Hillel Y., Language and information, Addison-Wesley, 1964.
2. Brooker R. A., *CACM*, **10** (April 1967), 223.
3. Brooker R. A., Morris D., An assembly program for a phrase structure language, *Computer Journal*, **3** (1960), 168.
4. Brooker R. A., Morris D., A general translation program for phrase structure languages, *JACM*, **9** (Jan. 1962), 1.
5. Brooker R. A., Morris D., Rohl J. S., Experience with the compiler-compiler, *Computer Journal*, **9** (1967), 345.
6. Cheatham T. E., Satley K., Syntax directed compiling, Proc. AFIPS, 1964 (SJCC), 31.
7. Chomsky N., Formal properties of grammars, Handbook of Mathematical Psychology, 2, Wiley, 1963, p. 323.
8. Chomsky N., Aspects of the theory of syntax, MIT Press, 1965.
9. Eichel J., Paul M., Bauer F. L., Samelson K., A syntax controlled generator of formal language processors, *CACM*, **6** (Aug. 1963), 451.
10. Feldman J., Gries D., Translator writing systems, *CACM*, **11** (Feb. 1968), 77.
11. Floyd R. W., Syntactic analysis and operator precedence, *JACM*, **10** (July 1963), 316.
12. Floyd R. W., Bounded context syntactic analysis, *CACM*, **7** (Feb. 1964), 62.
13. Floyd R. W., The syntax of programming languages — a survey, *IEEE Trans.*, **EC13**, **4** (Aug. 1964), 346.
14. Foster J. M., A syntax improving program, *Computer Journal*, **11** (May 1968), 31.
15. Ginsburg S., The mathematical theory of context-free languages, McGraw-Hill Inc., 1966.
16. Ginsburg S., Greibach S., Deterministic context-free languages, *Information and Control*, **9** (1960), 620.
17. Greibach S., Formal Parsing Systems, *CACM*, **7** (Aug. 1964), 499.
18. Griffiths T. V., Petrick S. R., On the relative efficiency of context-free grammar recognisers, *CACM*, **8** (May 1965), 289.
19. Griffiths T. V., Petrick S. R., Top-down versus bottom-up analysis, IFIP Congress 68, B80.
20. Ingerman P. Z., A syntax oriented translator, Academic Press, 1966.

Irons E. T., A syntax directed compiler for Algol 60, *CACM*, **4** (Jan. 1961), 51.
Irons E. T., The structure of the syntax-directed compiler, *Annual Review in Automatic Programming*, **3** (1963), 207.
Irons E. T., An error correcting parse algorithm, *CACM*, **6** (Nov. 1965), 669.
Kuno S., Oettinger A. G., Multiple path syntactic analyser, IFIP Congress 62, p. 306.
Kuno S., The predictive analyser and a path elimination technique, *CACM*, **8** (July 1965), 453.
Kurki-Suonio R., On top-to-bottom recognition and left-recursion, *CACM*, **9** (July 1966), 527.
Landweber P. S., Decision problems of phrase structure grammars, *IEEE Trans.*, **EC13** (Aug. 1964), 354.
Ledley R. S., Wilson J. B., Automatic programming language translation trough syntactical analysis, *CACM*, **5** (March 1962), 145.
Narasimhan R., Syntax directed translation of classes of pictures, *CACM*, **9** (March 1966), 166.
Reynolds J. C., An introduction to the COGENT programming system, ACM 20th National Conference, 1965, p. 422.
Samelson K., Bauer F. L., Sequential formula translation, *CACM*, **3** (Feb. 1960), 76.
Unger S. H., A global parser for context-free phrase structure grammars, *CACM*, **11** (April 1968), 240.
Wirth N., Weber H., EULER — a generalisation of Algol, and its formal definition, Part 1, *CACM*, **9** (Jan. 1966), 13; Part 2, *CACM*, **9** (Feb. 1966), 89.
Amarel S. Готовится к печати.

ОГЛАВЛЕНИЕ

УВАЖАЕМЫЙ ЧИТАТЕЛЬ!

Ваши замечания о содержании книги, ее оформлении, качестве перевода и другие просим присылать по адресу: 129820, Москва, И-110, ГСП, 1-й Рижский пер., д. 2, издательство «Мир».

Дж. Фостер

АВТОМАТИЧЕСКИЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ

Редактор Л. Н. Бабынина Художник В. М. Новоселова
Художественный редактор В. И. Шаповалов Технический редактор Н. И. Манохина
Корректор Т. С. Лаврова

Сдано в набор 17/X 1974 г. Подписано к печати 24/III 1975 г. Бумага тип. № 1
60×90 1/16 = 2,25 бум. л. 4.50 усл. печ. л. Уч.-изд. л. 3.02. Изд. № 1/7554. Цена 23 к. Зак. 392

ИЗДАТЕЛЬСТВО «МИР»
Москва, 1-й Рижский пер., 2

Ордена Трудового Красного Знамени Ленинградская типография № 2
имени Евгении Соколовой Союзполиграфпрома при Государственном комитете
Совета Министров СССР по делам издательств, полиграфии и книжной торговли
198052, Ленинград, Л-52, Измайловский пр., 29.