Министерство образования и науки Российской Федерации
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

А.А. МАЛЯВКО

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

ФОРМАЛЬНЫЕ ЯЗЫКИ И МЕТОДЫ ТРАНСЛЯЦИИ

ЧАСТЬ 2

Синтаксический анализ

Утверждено
Редакционно-издательским советом университета
в качестве учебного пособия

НОВОСИБИРСК
2011

УДК 004.43(075.8)
М 219

Рецензенты:

А.В. Гунько, канд. техн. наук, доцент,
Е.Л. Романов, канд. техн. наук, доцент

Работа подготовлена на кафедре
вычислительной техники для студентов IV курса АВТФ

Малявко А.А.

М 219 Системное программное обеспечение. Формальные языки и методы трансляции: учеб. пособие. В 3 ч. / А.А. Малявко. – Новосибирск: Изд-во НГТУ, 2011. – Ч. 2. Синтаксический анализ. – 160 с.

ISBN 978-5-7782-1668-6

Во второй части учебного пособия изложены основные свойства формальных грамматик и их связь с задачей автоматного синтаксического анализа, нисходящие и восходящие методы синтаксического акцепта, т. е. восстановления дерева грамматического разбора, теоретические основы и методы проверки пригодности формальных грамматик для реализации этих методов, способы преобразования грамматик в конечные автоматы со стековой памятью (так называемые распознаватели), а также основные способы расширения акцепторов до синтаксических анализаторов, решающих задачи нейтрализации ошибок и преобразования входного текста в промежуточную форму представления – постфиксную запись.

Пособие рекомендуется студентам старших курсов и аспирантам, а также преподавателям смежных дисциплин. Оно может быть полезно студентам и аспирантам ряда других технических специальностей, связанных с разработкой и использованием программного обеспечения.

УДК 004.43(075.8)

ISBN 978-5-7782-1668-6

© Малявко А.А., 2011
© Новосибирский государственный технический университет, 2011

1. ВВЕДЕНИЕ В СИНТАКСИЧЕСКИЙ АНАЛИЗ

Синтаксический анализ логически является следующим после лексического этапом процесса трансляции. Синтаксический анализатор обрабатывает формируемую лексическим анализатором последовательность лексем, т. е. внутренних эквивалентов слов текста транслируемой программы, и выполняет следующие действия:

1) проверяет синтаксическую правильность последовательности слов программы, выявляет границы правильных предложений;
2) в случае обнаружения синтаксических ошибок формирует диагностические сообщения для автора транслируемой программы;
3) формирует очередное промежуточное представление программы – постфиксную форму записи (ПФЗ), для чего преобразует каждое правильное предложение в ПФЗ;
4) пополняет информационные таблицы транслятора именами новых объектов, создаваемых при формировании ПФЗ.

Первые две функции выполняются той частью синтаксического анализатора, которая называется синтаксическим акцептором, обычно реализуемым в виде конечного автомата, но уже со стековой (или магазинной) памятью. Преобразование последовательности лексем в постфиксную форму записи и операции с информационными таблицами транслятора реализуются в виде совокупности действий, расширяющих функциональность конечного автомата. Структура синтаксического анализатора приведена на рис. 1.

Задача проверки правильности входной последовательности слов (лексем) транслируемой программы является первичной, все остальные задачи решаются на ее основе. Для того чтобы понять, как решается эта задача, рассмотрим формальное описание синтаксиса языков

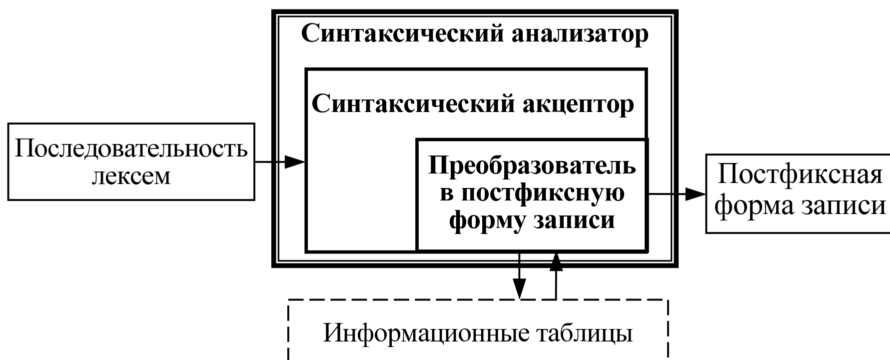


Рис. 1. Структура синтаксического анализатора

программирования и методы преобразования такого описания в конечный автомат, распознающий правильные предложения.

1.1. Формальные грамматики, основные понятия

Формальной грамматикой $\boldsymbol{G}$ называется совокупность, состоящая:

- из алфавита терминальных символов $\boldsymbol{A}_t$;
- алфавита нетерминальных символов $\boldsymbol{A}_n$;
- начального нетерминального символа S;
- системы правил подстановки $\boldsymbol{P}$

$$\boldsymbol{G} = \{ \boldsymbol{A}_t, \boldsymbol{A}_n, S, \boldsymbol{P} \}.$$

Алфавит терминальных символов $\boldsymbol{A}_t$ есть конечное множество всех слов языка, порождаемого данной грамматикой. Понятие «терминальный» в данном случае обозначает неразложимость, элементарность таких символов с точки зрения синтаксических правил. Например, любой идентификатор при синтаксическом анализе считается простейшим символом, даже если он является цепочкой из нескольких литер. Более того, под одиночным терминальным символом, как правило, понимается вся группа слов, таких как идентификаторы или константы. В самом деле, с точки зрения синтаксиса, как совокупности правил образования предложений из слов, совершенно безразлично, какой именно идентификатор находится в левой части оператора присваивания или какие именно константы содержатся в выражении.

Наряду со словосочетанием «терминальный символ» часто будет использоваться просто слово «терминал».

В дальнейшем терминальные символы будем обозначать либо одиночными литерами, представляющими собой отдельные слова в языках программирования (+, *, =, ;, {, …), либо терминами, называющими слово или группу слов (идентификатор, константа, id, const, …), либо просто малыми буквами латинского алфавита, обозначающими произвольный терминал (a, b, c, …).

Например: $\boldsymbol{A_t}$={идентификатор, константа, +, –, *, /, =}

Алфавит нетерминальных символов $\boldsymbol{A_n}$ есть конечное множество названий синтаксических конструкций, например: <предложение>, <выражение>, <список аргументов>, <условный оператор>, <тело функции>. Нетерминальные символы используются только в метаязыке, на котором описывается язык программирования, никакой нетерминальный символ как слово языка не может появиться в тексте правильной программы. Наряду со словосочетанием «нетерминальный символ» в дальнейшем будет использоваться просто «нетерминал». Нетерминальные символы будем обозначать либо словосочетаниями в угловых скобках, как в начале этого абзаца, либо словами, содержащими только прописные буквы (например FUNCTION, ARGUMENTS_LIST), либо большими буквами латинского алфавита в курсивном начертании *A*, *B*, *C*, ..., *Z*.

Начальный нетерминальный символ *S* есть один из нетерминальных символов. Этим символом обычно обозначается наиболее общая синтаксическая конструкция, например: <правильная программа>.

Символы (как терминальные, так и нетерминальные) могут образовывать цепочки, которые будут обозначаться малыми буквами греческого алфавита α, β, γ, ... ω. Особое значение будет иметь пустая цепочка символов, для которой будет использоваться обозначение $\boldsymbol{\varepsilon}$.

Система правил подстановки ***P*** (иногда называемая системой порождающих правил или продукций) есть конечное множество пар цепочек вида α : β, причем цепочка α (левая часть правила) должна содержать хотя бы один нетерминальный символ.

Каждая такая пара цепочек называется правилом подстановки (порождающим правилом, продукцией) и определяет возможный способ замены левой части правила на его правую часть. Правила подстановки обычно нумеруются, причем в левой части правила с номером 1 должен находиться одиночный начальный нетерминал грамматики.

Приведем пример грамматики языка, в котором правильными предложениями являются только двоичные числа-перевертыши, т. е. последовательности двоичных цифр, выглядящие одинаково как с начала, так и с конца. Алфавит терминальных символов этого языка состоит из двух цифр: $A_t = \{\ 0,\ 1\ \}$. Алфавит нетерминальных символов содержит единственный символ (он же начальный нетерминал): $A_n = \{\ S\ \}$. Система порождающих правил ***P*** грамматики этого языка (назовем ее G_1) может выглядеть так:

P : {

1. *S : 0 S 0*
2. *S : 1 S 1*
3. *S : 0*
4. *S : 1*
5. *S : ε*

}

1.2. Связь между языками и грамматиками

Пусть дана грамматика ***G*** с системой порождающих правил ***P***.

Говорят, что цепочка ψ *непосредственно выводится* из цепочки σ (σ → ψ), если:

1) σ = ω α δ (ω и δ произвольные, возможно пустые цепочки);

2) ψ = ω β δ (ω и δ – те же самые цепочки);

3) в системе порождающих правил ***P*** есть правило α : β.

Тогда, подставив β вместо α в цепочке σ = ω α δ, получим ω β δ, т. е. цепочку ψ. Обратная подстановка не подразумевается, т. е. из возможности непосредственного вывода σ → ψ не следует возможность непосредственного вывода ψ → σ.

Например, в грамматике G_1 цепочка *0 1 S 1 0* непосредственно выводится из цепочки *0 S 0*, т. е. *0 S 0 → 0 1 S 1 0*.

Говорят, что цепочка ψ *выводится* из цепочки σ (σ ⇒ ψ), если существует последовательность непосредственных выводов:

$$\sigma \rightarrow \psi_1 \rightarrow \psi_2 \rightarrow \psi_3 \ldots \psi_n \rightarrow \psi$$

Например, в грамматике G_1 цепочка *0 1 0 1 0* выводится из цепочки *0 S 0*, т. е. *0 S 0 ⇒ 0 1 0 1 0*, поскольку *0 S 0 → 0 1 S 1 0 →0 1 0 1 0*.

Правильным предложением языка ***L***, определяемого грамматикой ***G***, называется цепочка, состоящая только из терминальных символов и выводимая из начального нетерминала *S*.

Любая цепочка терминальных символов, для которой не существует вывода из начального нетерминала *S* грамматики ***G***, является неправильным предложением языка ***L***. Цепочка, содержащая символы, не входящие в алфавит терминальных символов $\boldsymbol{A}_t$, в частности, нетерминальные символы, вообще не является предложением языка ***L***.

Таким образом, грамматика ***G*** разбивает бесконечное множество всех возможных цепочек, содержащих только терминальные символы (слова языка) на два непересекающихся подмножества:

- подмножество правильных предложений, которое и является языком ***L***, порождаемым грамматикой ***G***;
- подмножество неправильных предложений.

Например, цепочки *10101* и *1111111111* являются правильными предложениями языка, определяемого грамматикой $\boldsymbol{G}_1$, а такие цепочки, как *10100* и *0111111111101*, – нет. Цепочка *10101* выводится из нетерминала *S* (т. е. $S \Rightarrow 10101$) следующим образом:

$$S \rightarrow 1S1 \rightarrow 10S01 \rightarrow 10T01 \rightarrow 10101$$

Для цепочки *10100* невозможно найти последовательность непосредственных выводов, приводящую к ней из начального нетерминала этой грамматики. Следовательно, она не является правильным предложением рассматриваемого языка.

Изобразим вывод цепочки *10101* из начального нетерминала *S* грамматики $\boldsymbol{G}_1$ в графическом виде (рис. 2).

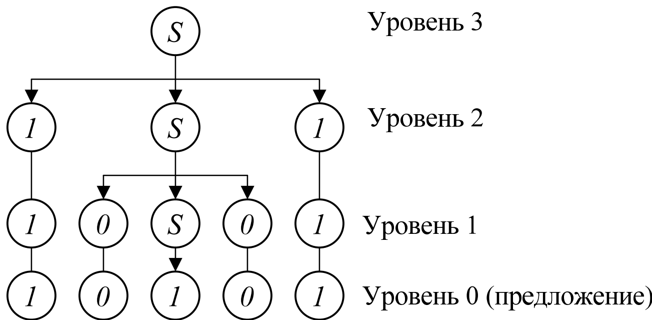


Рис. 2. Полное представление дерева грамматического разбора

В этом представлении линиями со стрелками показаны непосредственные выводы, т. е. подстановки правых частей порождающих правил

вместо соответствующих левых частей, а без стрелок – простой перенос символов с одного уровня дерева на другой. Такой граф, отражающий структуру вывода предложения из начального нетерминального символа, называется деревом его грамматического разбора.

Обычно в литературе под деревом грамматического разбора понимается граф, из которого удалены вершины, поставленные в соответствие символам, просто переносимым с одного уровня дерева на другой, как показано на рис. 3.

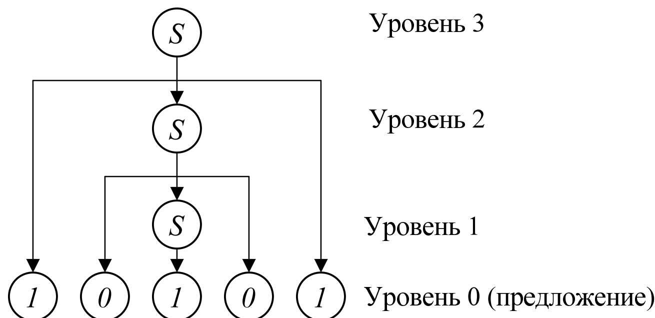


Рис. 3. Традиционное представление дерева грамматического разбора

Основная задача синтаксического акцепта – проверка правильности последовательности слов транслируемой программы – теперь формулируется так: предложение языка (т. е. последовательность терминалов порождающей язык грамматики) является правильным, если может быть установлен факт существования дерева его грамматического разбора, и неправильным – в противном случае.

Само дерево разбора при этом можно в принципе и не строить. Однако, как будет показано ниже, полное или частичное построение дерева разбора позволяет решать остальные задачи синтаксического анализа, а также задачи семантического анализа и генерации кода в процессе проверки правильности входного предложения.

Сложность задачи восстановления дерева разбора предложений языка существенно зависит от свойств грамматики, определяющей этот язык. Выше отмечалось, что любая грамматика $\boldsymbol{G}$ определяет единственный язык $\boldsymbol{L}$. Однако существуют языки, для определения которых можно построить более чем одну грамматику. Различающимися считаются грамматики, для которых деревья грамматического разбора хотя бы одного правильного предложения данного языка будут иметь разную структуру.

Приведем пример языка, который определяется более чем одной грамматикой: язык скобочных арифметических выражений со знаками операций различного приоритета. Здесь и далее формальные грамматики будем определять только как системы порождающих правил, помня о соглашениях об обозначении терминальных и нетерминальных символов и считая, что начальный нетерминал – это всегда символ, являющийся левой частью первого правила.

Грамматика G_{a1}		Грамматика G_{a2}	
1	$S : S + T$	1	$S : U R$
2	$S : T$	2	$R : + S$
3	$T : T * V$	3	$R :$
4	$T : V$	4	$U : V W$
5	$V : (S)$	5	$W : * U$
6	$V : ident$	6	$W :$
7	$V : const$	7	$V : (S)$
		8	$V : ident$
		9	$V : const$

Эти две грамматики различаются не только количеством правил и обозначениями нетерминалов. Пусть дано предложение *(a+b)*c*, где *a*, *b* и *c* – идентификаторы (или константы). Построим выводы этого предложения из начальных нетерминалов грамматик G_{a1} и G_{a2}, заменяя на каждом шаге самый левый нетерминал в очередной цепочке символов на правую часть одного из правил для этого нетерминала:

Для G_{a1}:

$$S \to T \to T*V \to V*V \to (S)*V \to (S+T)*V \to (T+T)*V \to$$
$$(V+T)*V \to (a+T)*V \to (a+V)*V \to (a+b)*V \to (a+b)*c$$

Для G_{a2}:

$$S \to UR \to VWR \to (S)WR \to (UR)WR \to (VWR)WR \to$$
$$(aWR)WR \to (aR)WR \to (a+S)WR \to (a+UR)WR \to (a+VWR)WR \to$$
$$(a+bWR)WR \to (a+bR)WR \to (a+b)WR \to (a+b)*UR \to$$
$$(a+b)*VWR \to (a+b)*cWR \to (a+b)*cR \to (a+b)*c$$

Даже не преобразуя эти два вывода в графически представленные деревья грамматического разбора, легко можно заметить, что эти деревья существенно отличаются друг от друга. Таким образом, существу-

ют эквивалентные, но различающиеся по своим свойствам грамматики, определяющие один и тот же язык. Свойства грамматик оказывают большое влияние на возможность использования данной грамматики для синтаксического анализа и для автоматизации построения синтаксического анализатора.

1.3. Классификация формальных грамматик

Все грамматики можно разделить на четыре основных класса по Хомскому, впервые сформулировавшему приведенные ниже признаки деления на классы.

1. Класс 0 – *общие*. К этому классу относятся грамматики, на порождающие правила которых не накладывается дополнительных ограничений, кроме сформулированных выше при определении понятия грамматик.
2. Класс 1 – *контекстно-зависимые*. К этому классу относятся грамматики, у которых длина цепочки левой части каждого правила не больше длины цепочки правой части (естественно, за исключением правил, содержащих пустую цепочку в правой части).
3. Класс 2 – *контекстно-свободные*. К этому классу относятся грамматики, у которых левая часть каждого порождающего правила состоит из одного нетерминального символа.
4. Класс 3 – *регулярные*. Грамматики этого класса содержат правила, левая часть каждого из которых – один нетерминал, а правая часть содержит не более одного нетерминала. Заметим, что обычно в литературе приводится несколько другое определение этого ограничения, а именно: в правой части может находиться либо пустая цепочка, либо один терминал, либо один терминал и один нетерминал.

Кратко охарактеризуем следствия, вытекающие из этих на первый взгляд сугубо численных ограничений.

Регулярные грамматики позволяют описывать последовательности терминальных символов и вложенные друг в друга конструкции, состоящие из таких последовательностей. Регулярные грамматики эквивалентны аппарату регулярных выражений и могут использоваться наравне с ними для определения лексики языка, однако формальное описание лексики в терминах регулярных выражений является значительно более компактным.

Контекстно-свободные грамматики позволяют определять синтаксис формальных языков, в том числе – языков программирования. Под синтаксисом понимается совокупность правил построения предложе-

ний языка из слов. Независимость от контекста означает возможность подстановки правой части любого из правил для замены нетерминала из левой части правила в любой цепочке, содержащей этот нетерминал. Отсюда непосредственно следует невозможность включения смысловых ограничений на построение синтаксически правильных конструкций в рамках контекстно-свободных грамматик.

Например, следующий фрагмент программы на языке C/C++, содержащий два предложения:

```
...
char * PointerToChar;
...
PointerToChar = 3.14159;
...
```

синтаксически абсолютно правилен.

Объявление указательной переменной с идентификатором *PointerToChar* соответствует всем правилам языка. Идентификатор *PointerToChar* может быть употреблен в левой части оператора присваивания. Вещественная константа может образовывать арифметическое выражение, находящееся в правой части оператора присваивания. Однако смысл этих двух предложений не согласован, в программе есть семантическая ошибка: переменной указательного типа не может присваиваться вещественное значение. В рамках контекстно-свободной грамматики невозможно определить язык таким образом, чтобы подобную последовательность операторов нельзя было вывести из начального нетерминала грамматики.

Контекстно-зависимые грамматики *в принципе* позволяют определять семантику языка совместно с его синтаксисом. Другими словами, *в принципе* возможно построить контекстно-зависимую грамматику, из начального нетерминала которой нельзя вывести приведенный выше фрагмент программы, но любой из нижеследующих вывести можно:

```
...
char * PointerToChar; ... PointerToChar = "3.14159";
...
```

и

```
...
float NumberPi; ... NumberPi = 3.14159;
...
```

Слова «в принципе» выделены курсивом по той причине, что такие грамматики построить можно только теоретически. Полная контекстно-зависимая грамматика, определяющая совместно синтаксис и семантику таких языков, как C/C++, будет необозримой и бесполезной с практической точки зрения. Задачи семантического анализа решаются, как правило, без полного формального описания семантики.

Общие грамматики предоставляют еще более широкие возможности для описания формальных языков, чем контекстно-зависимые, но на практике не используются в силу тех же причин.

1.4. Свойства контекстно-свободных грамматик

Свойства грамматик определяются свойствами ее системы порождающих правил и свойствами нетерминальных символов. Они могут служить основанием для разбиения всего класса на более узкие подклассы. Выделим некоторые наиболее важные свойства.

1.4.1. Рекурсия

Нетерминальный символ X называется *рекурсивным*, если из него могут быть выведены цепочки, содержащие X, т. е.

$$X => \alpha X \beta,$$

где α и β – произвольные, возможно пустые (но не обе одновременно) цепочки символов. Если хотя бы в одном таком выводе цепочка α пуста, то X называется *леворекурсивным*. Если хотя бы в одном таком выводе цепочка β пуста, то X называется *праворекурсивным*. X может быть одновременно и право- и леворекурсивным.

Рекурсия может быть прямой и косвенной. Нетерминальный символ X называется *прямрекурсивным*, если существует порождающее правило $X : \alpha X \beta$, где α и β – произвольные, возможно пустые цепочки (но не обе сразу, иначе такое правило является тавтологией и его можно безболезненно удалить из системы правил подстановки). Если α – пустая цепочка (либо из α можно вывести пустую цепочку), то X называется *прямолеворекурсивным*, если же β – пустая цепочка (либо из β можно вывести пустую цепочку), то X называется *прямоправорекурсивным*.

Некоторые нетерминалы могут быть одновременно и прямолево- и прямоправорекурсивными.

Нетерминальный символ X называется _косвенно-рекурсивным_, если в системе порождающих правил существует такая совокупность правил:

$$X : \alpha_1 \, Y_1 \, \beta_1$$

$$Y_1 : \alpha_2 \, Y_2 \, \beta_2$$

$$\ldots$$

$$Y_n : \alpha_{n+1} \, X \, \beta_{n+1},$$

где α_1, α_2, ... α_{n+1}, β_1, β_2, ... β_{n+1} – произвольные, возможно пустые цепочки символов. Косвенная рекурсия аналогично прямой может быть правой, левой и общей.

Грамматика называется _рекурсивной_, если рекурсивен хотя бы один ее нетерминальный символ, и _нерекурсивной_ в противном случае. Нерекурсивные грамматики с конечным количеством порождающих правил определяют так называемые _конечные_ языки, в которых количество правильных предложений ограничено. Такие языки, а следовательно, и грамматики, не представляют интереса с точки зрения построения трансляторов.

1.4.2. Однозначность

Грамматика называется _однозначной_, если любое правильное предложение порождаемого ею языка имеет единственное дерево грамматического разбора, и _неоднозначной_ в противном случае. Грамматики $\boldsymbol{G_{a1}}$ и $\boldsymbol{G_{a2}}$ являются однозначными, но для того же самого языка скобочных арифметических выражений можно привести пример неоднозначной грамматики:

Грамматика $\boldsymbol{G_{a3}}$	
1	$S : S + S$
2	$S : T$
3	$T : T * T$
4	$T : V$
5	$V : (S)$
6	$V : ident$
7	$V : const$

Оказывается, что для предложения $a+b+c$ в грамматике $\boldsymbol{G_{a3}}$ можно построить два различных вывода:

$$S \to S+S \to T+S \to V+S \to a+S \to a+S+S \to a+T+S \to$$
$$a+V+S \to a+b+S \to a+b+T \to a+b+V \to a+b+c$$

и

$$S \to S+S \to S+S+S \to S+S+T \to S+S+V \to S+S+c \to S+T+c \to$$
$$S+V+c \to S+b+c \to T+b+c \to V+b+c \to a+b+c$$

Структуры деревьев грамматического разбора, соответствующие этим выводам, различаются (рис. 4).

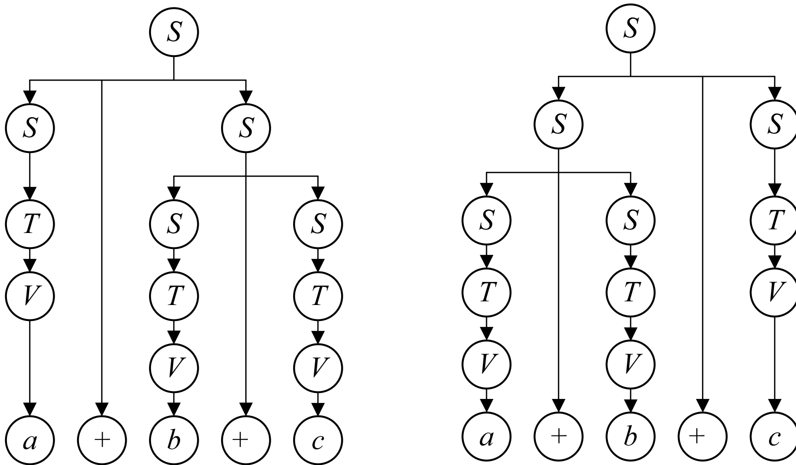


Рис. 4. Два разных дерева грамматического разбора одного предложения в неоднозначной грамматике

Согласно первому выводу выражение $a+b+c$ рассматривается как $a+(b+c)$, а согласно второму – как $(a+b)+c$.

Свойство однозначности грамматики существенно влияет на возможность построения синтаксического акцептора на ее основе.

В грамматике $\boldsymbol{G}_{a1}$ для этого предложения можно построить как минимум две разные последовательности непосредственных выводов:

$$S \to S+T \to S+T+T \to T+T+T \to V+T+T \to a+T+T \to a+V+T \to$$
$$a+b+T \to a+b+V \to a+b+c$$

и

$$S \to S+T \to S+V \to S+c \to S+T+c \to S+V+c \to S+b+c \to$$
$$T+b+c \to V+b+c \to a+b+c$$

В грамматике G_{a2}, естественно, также можно по-разному выбирать нетерминалы для подстановки и получить такие два вывода:

$S \to UR \to VWR \to aWR \to aR \to a{+}S \to a{+}UR \to a{+}VWR \to$

$a{+}bWR \to a{+}bR \to a{+}b{+}S \to a{+}b{+}UR \to a{+}b{+}VWR \to$

$a{+}b{+}cWR \to a{+}b{+}cR \to a{+}b{+}c$

и

$S \to UR \to U{+}S \to U{+}UR \to U{+}U{+}S \to U{+}U{+}UR \to$

$U{+}U{+}U \to U{+}U{+}VW \to U{+}U{+}V \to U{+}U{+}c \to U{+}VW{+}c \to$

$U{+}V{+}c \to U{+}b{+}c \to VW{+}b{+}c \to V{+}b{+}c \to a{+}b{+}c$

Однако, как легко можно видеть, деревья разбора, полученные из этих выводов, имеют одну и ту же структуру, т. е. эти выводы не являются различными, а просто различаются способы их записи (см. рис. 5).

Для грамматики G_{a1}: Для грамматики G_{a2}:

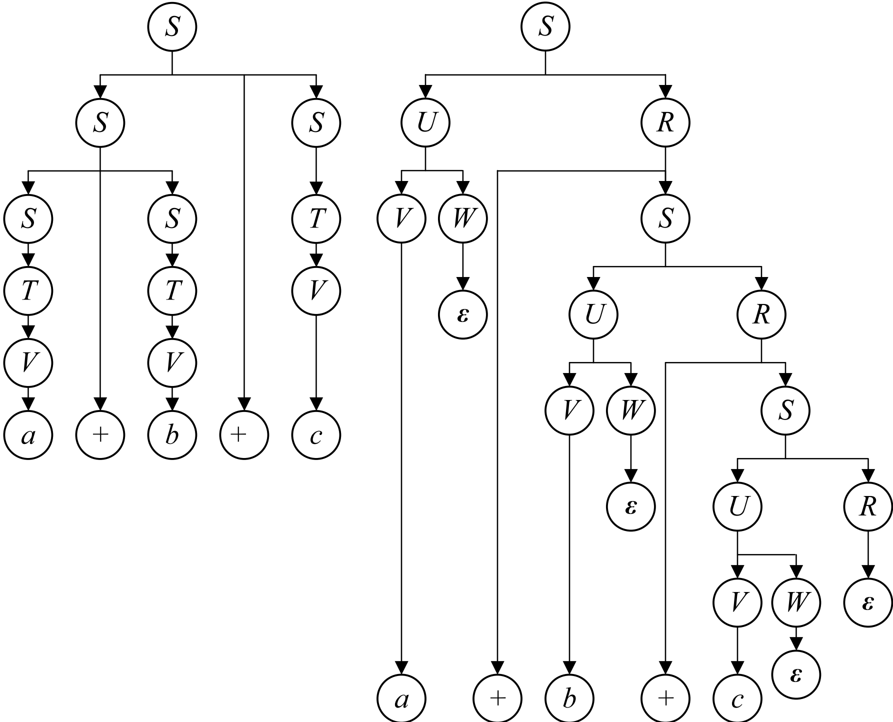


Рис. 5. Деревья разбора для однозначных грамматик

В грамматике G_{a1} выражение $a+b+c$ при любом разборе трактуется как $(a+b)+c$, но в грамматике G_{a2} – всегда как $a+(b+c)$.

Оказывается, что для решения задач синтаксического анализа (и для автоматизации построения синтаксических анализаторов) могут быть использованы только однозначные грамматики. Существующие методы преобразования грамматик в конечные автоматы не могут быть применены к неоднозначным грамматикам. К сожалению, определение того, однозначна ли заданная грамматика, является в общем случае алгоритмически неразрешимой проблемой, т. е. не существует универсального алгоритма, способного для любой заданной контекстно-свободной грамматики определить, однозначна ли она.

Методы построения синтаксических анализаторов путем преобразования формальной грамматики в конечный автомат не приводят к желаемому результату для неоднозначных грамматик. Однако и многие однозначные грамматики также не допускают подобного преобразования, но по другим причинам. Примером может служить простая с виду регулярная грамматика G_1, для которой синтаксический анализатор не может быть построен описываемыми далее способами.

1.5. Свойства символов грамматики

1.5.1. Аннулируемость

Нетерминальный символ называется *аннулируемым*, если из него может быть выведена пустая цепочка символов. В противном случае нетерминал называется *неаннулируемым*. В литературе по формальным языкам такие символы обычно называются (не)аннулирующими. Процедура определения свойства аннулируемости для всех нетерминалов грамматики состоит в следующем.

Шаг 1. Образуем вспомогательные:

- множество N из всех нетерминальных символов грамматики;
- пустое множество A.

Шаг 2. Просмотрим все правила грамматики. Если некоторое очередное правило имеет вид

$$X : \varepsilon ,$$

то:

- вычеркнем все правила для нетерминала X из системы правил;
- вычеркнем нетерминал X из правых частей всех оставшихся правил;

- перенесем нетерминал *X* из множества *N* в множество *A*;
- вернемся к началу шага 2.

После завершения этой процедуры в множестве *N* останутся все неаннулируемые нетерминалы. Все аннулируемые нетерминалы теперь содержатся в множестве *A*.

1.5.2. Недостижимость

Символ (терминальный или нетерминальный) называется *недостижимым*, если он не появляется ни в одной цепочке символов, выводимой из начального нетерминала грамматики. Для определения недостижимых нетерминалов применяется следующая процедура.

Шаг 1. Образуем вспомогательные:

- множество *M*, состоящее в начальный момент из единственного начального нетерминала;
- множество *T*, в начальный момент пустое.

Шаг 2. Просмотрим все правила для каждого из нетерминалов, входящих в множество *M*, и для каждого из них:

- каждый нетерминал, встречающийся в правой части просматриваемого правила, добавим к множеству *M*, если его там нет;
- каждый терминал, встречающийся в правой части просматриваемого правила, добавим к множеству *T*, если его там нет;
- вычеркнем все просмотренные правила из системы порождающих правил;
- вернемся к началу шага 2, если в множество *M* был добавлен хотя бы один символ.

После завершения этой процедуры в множестве *M* содержатся все достижимые нетерминалы, в множестве *T* – все достижимые терминалы. Остальные символы из соответствующих алфавитов являются недостижимыми. Недостижимые символы можно (и нужно) удалить из соответствующих алфавитов, а из системы порождающих правил – удалить все правила, у которых недостижимый нетерминал находится в левой части. После этого в системе порождающих правил останутся только те правила, которые содержат достижимые символы. Язык, определяемый грамматикой, от этого не изменится.

1.5.3. Бесплодность

Нетерминальный символ называется *бесплодным* (бесполезным), если из него не может быть выведена ни одна цепочка, состоящая только из терминальных символов. Для определения свойства бесплодности нетерминалов применяется следующая процедура.

Шаг 1. Образуем вспомогательное множество U, содержащее в начальный момент все нетерминальные символы грамматики.

Шаг 2. Просмотрим все правила грамматики. Если правая часть просматриваемого правила не содержит нетерминальных символов (в частности, является пустой цепочкой), то:

- нетерминал, находящийся в левой части этого правила, удалим из множества U;
- вычеркнем из системы порождающих правил все те правила, у которых этот нетерминал находится в левой части;
- из правых частей всех оставшихся правил вычеркнем этот нетерминальный символ;
- вернемся к началу шага 2, если из множества U был удален хотя бы один символ.

После завершения этой процедуры в множестве U останутся только бесплодные нетерминалы (если таковые были в этой грамматике).

Все правила, содержащие бесплодный нетерминал как в левой, так и в правой части, следует удалить из системы порождающих правил, а из алфавита нетерминальных символов нужно удалить все бесплодные нетерминалы. Удаление бесплодных символов и правил для них никак не повлияет на определяемый грамматикой язык, поскольку язык есть множество всех цепочек терминальных символов, выводимых из начального нетерминала грамматики.

Недостижимые и бесплодные нетерминалы могут появиться либо в результате ошибок при разработке системы порождающих правил, либо в результате эквивалентных преобразований грамматики с целью изменения ее свойств или свойств ее символов. Некоторые эквивалентные преобразования грамматик рассматриваются далее.

Кроме свойств отдельных символов большое значение имеют некоторые отношения между символами, определяемые всей совокупностью правил грамматики. Важнейшими отношениями являются отношение предшествования и отношение следования. Эти отношения мы будем определять в виде множеств, сопоставляемых с символами грамматики и содержащих опять-таки символы этой грамматики.

1.6. Множества предшественников символов грамматики

Предшественником некоторого символа X называется символ, с которого начинается цепочка, выводимая из X. Считается, что любой символ является предшественником самого себя, т. е. учитываются выводы длины 0.

Термин «предшественник» в русском языке имеет совсем другой смысл: нечто, появляющееся **до** того, чему оно предшествует. Тем не менее в литературе по формальным языкам и грамматикам (и в настоящем пособии) он используется именно в таком смысле, который определен в предыдущем абзаце.

Вычисление множеств предшественников нетерминальных символов производится согласно следующей процедуре (с иллюстрацией на примере грамматики $\boldsymbol{G}_{a2}$).

Шаг 0. Прежде всего модифицируем эту грамматику, добавив к системе правил особое правило с номером 0, в котором специальным псевдотерминальным символом ► обозначен конец файла, содержащего правильное предложение:

Грамматика $\boldsymbol{G}_{a2}$	
0	$Z : S$ ►
1	$S : U\ R$
2	$R : +\ S$
3	$R :$
4	$U : V\ W$
5	$W : *\ U$
6	$W :$
7	$V : (\ S\ )$
8	V : *ident*
9	V : *const*

Это правило явно или неявно обычно добавляется любым преобразователем грамматики в синтаксический акцептор, и делается это по двум причинам:

– во-первых, добавляемое правило просто является формулировкой очевидного для языков программирования утверждения, что правильное предложение (выводимое из начального нетерминала S) содержится в отдельном файле (новый начальный нетерминал Z); следовательно, после последнего символа правильного предложения находится вполне определенный символ «конец файла»;

– во-вторых, в исходной грамматике для начального нетерминала может существовать несколько разных правил, а это создает определенные трудности для некоторых алгоритмов анализа грамматики и построения автомата.

Шаг 1. Построим заготовку матрицы отношения предшествования, сопоставив каждому символу грамматики по одной строке и по одному столбцу в одном и том же порядке и заполнив единичками главную диагональ. Отметки на главной диагонали фиксируют тот факт, что каждый символ является предшественником самого себя.

Шаг 2. Просмотрим все правила грамматики и, если правая часть не является пустой цепочкой, поставим отметку 2 в клетку матрицы, стоящую на пересечении строки, помеченной нетерминалом из левой части, и столбца, помеченного первым символом правой части. В результате будет получена матрица отношения непосредственного предшествования в правилах грамматики:

	S	*U*	*R*	*V*	*W*	+	*	(	)	*i*	*c*	►
S	1	2										
U		1		2								
R			1			2						
V				1				2		2	2	
W					1		2					
+						1						
*							1					
(								1				
)									1			
i										1		
c											1	
►												1

Шаг 3. Выполним транзитивное замыкание отношения предшествования согласно простому соображению: если *Z* есть предшественник символа *Y*, а *Y* есть предшественник символа *X*, то *Z* будет также предшественником символа *X*. Применительно к матричному представлению отношения предшествования это означает выполнение следующих действий:

- просматриваются все строки матрицы сверху вниз;
- в каждой строке просматриваются все клетки, кроме находящихся на главной диагонали;

- если клетка столбца, помеченного символом *Y*, не пуста, то к просматриваемой строке логически добавляется содержимое строки, помеченной этим же символом;
- если хотя бы одна клетка матрицы изменилась, то после последней строки просмотр начинается заново.

Заметим, что просмотр матрицы на третьем шаге можно ограничить только строками, помеченными нетерминальными символами.

После завершения этой процедуры получена матрица полного отношения предшествования (для выделения результатов разных шагов результат логического объединения содержимого строк на третьем шаге отмечен цифрой 3).

	S	*U*	*R*	*V*	*W*	+	*	(	)	*i*	*c*	▶
S	1	2		3				3		3	3	
U		1		2				3		3	3	
R			1			2						
V				1				2		2	2	
W					1		2					
+						1						
*							1					
(								1				
)									1			
i										1		
c											1	
▶												1

Матрица полного отношения предшествования содержит в каждой строке отметки как для терминальных, так и для нетерминальных символов. В дальнейшем интерес будут представлять только терминалы-предшественники нетерминальных символов. Выпишем соответствующие множества для всех нетерминалов грамматики:

$$M_{\text{пр}}(S) = \{(, i, ▶\}$$

$$M_{\text{пр}}(U) = \{(, i, ▶\}$$

$$M_{\text{пр}}(R) = \{+\}$$

$$M_{\text{пр}}(V) = \{(, i, ▶\}$$

$$M_{\text{пр}}(W) = \{*\}$$

1.7. Множества предшественников цепочек символов

При построении синтаксических анализаторов часто приходится оперировать не отдельными символами, а цепочками символов. После вычисления множеств предшественников для отдельных символов легко можно сформулировать способ определения множеств предшественников для цепочек символов (смысл таких множеств тот же самый – в множество предшественников цепочки β входят те и только те терминальные символы, с которых начинаются цепочки, выводимые из β).

Пусть цепочка β имеет вид β = *ABCD*..., где *A*, *B*, *C*, *D*, ... – произвольные (не обязательно нетерминальные) символы грамматики, для которых известны множества предшественников.

Тогда множество предшественников цепочки β можно вычислить по следующей формуле:

$$M_{\text{пр}}(\beta) = M_{\text{пр}}(A) \cup \begin{cases} \emptyset, \text{ если } A - \text{ неаннулируемый символ} \\ M_{\text{пр}}(B) \cup \begin{cases} \emptyset, \text{ если } B - \text{ неаннулируемый символ} \\ \ldots \end{cases} \end{cases}$$

Здесь $\emptyset$ – обозначение пустого множества.

Неаннулируемым символом является либо терминал, либо неаннулируемый нетерминал.

1.8. Множества последователей символов грамматики

Символ *Y* называется *последователем* символа *X*, если хотя бы в одной цепочке η, выводимой из начального нетерминала грамматики, символ *Y* непосредственно следует за *X*:

$$S => \eta = \ldots XY \ldots$$

Интерес будут представлять множества последователей только нетерминальных символов, однако для их вычисления придется определять множества последователей всех символов грамматики. Вначале введем понятие непосредственного последователя.

Символ *Y* является *непосредственным последователем* символа *X*, если:

1) в грамматике есть правило вида

$$N : \varphi\, X\, Y\, \psi,$$

где φ и ψ – произвольные, возможно пустые цепочки;

2) в грамматике есть правило вида

$$N : \varphi X \sigma Y \psi ,$$

где φ и ψ – произвольные, возможно пустые цепочки, а σ – цепочка, состоящая только из аннулируемых нетерминалов.

Непосредственными последователями, к сожалению, полные множества последователей не исчерпываются. Если символ Y является последователем (в том числе – непосредственным) символа Z, и:

3) в грамматике есть правило вида

$$Z : \varphi X \sigma ,$$

где φ – произвольная, возможно пустая цепочка, а σ – цепочка, состоящая только из аннулируемых нетерминалов или пустая, то Y является последователем символа X.

И, наконец, символ Y является последователем символа X, если Y есть (непосредственный) последователь символа Z, и:

4) в грамматике есть совокупность правил вида

$$\begin{array}{lcl} Z & : & \varphi_1 Z_1 \sigma_1 \\ Z_1 & : & \varphi_2 Z_2 \sigma_2 \\ \ldots & & \\ Z_{n-1} & : & \varphi_n Z_n \sigma_n \\ Z_n & : & \varphi_0 X \sigma_0 , \end{array}$$

где $\varphi_0, \varphi_1, \ldots, \varphi_n$ – произвольные, возможно пустые цепочки, а $\sigma_0, \sigma_1, \ldots, \sigma_n$ – цепочки, состоящие только из аннулируемых нетерминалов, или пустые.

Условия 1 и 2 определяют отношение «Y есть непосредственный последователь X», а условия 3 и 4 – в том виде, как они записаны – отношение «X есть последний (замыкающий) символ в цепочке, выводимой из Z». Отношение «Y есть последователь X», очевидно, есть произведение этих двух отношений.

Процедура вычисления множеств последователей по системе правил грамматики состоит из следующей последовательности шагов.

1. Определение отношения «непосредственное следование» между символами грамматики в матричной форме.

2. Определение матрицы отношения «замыкающие в цепочках».

3. Перемножение этих матриц и получение матрицы отношения «Y есть последователь X».

1.8.1. Определение отношения «непосредственное следование»

Шаг 1. Построим пустую заготовку матрицы отношения «*Y* есть непосредственный последователь *X*», строки и столбцы которой помечены в одинаковом порядке всеми символами грамматики. Согласно условию 1 просмотрим правые части всех правил и для каждой пары следующих друг за другом символов поставим 1 в клетку на пересечении строки, помеченной первым символом, и столбца, помеченного вторым символом пары.

	S	*U*	*R*	*V*	*W*	+	*	(	)	*i*	*c*	▶
S									1			1
U			1			2						
R												
V					1		2					
W												
+	1							2		2	2	
*		1						2		2	2	
(	1							2		2	2	
)												
i												
c												
▶												

Шаг 2. Выполним транзитивное замыкание построенной матрицы по аннулируемым нетерминалам для выявления всех непосредственных последователей согласно условию 2. Смысл этого действия состоит в том, что все непосредственные последователи каждого аннулируемого нетерминала *N* являются непосредственными последователями всех тех символов, по отношению к которым *N* есть непосредственный последователь.

Выпишем рабочую копию всех правил грамматики. Просмотрим посимвольно правую часть каждого правила. Каждый встретившийся аннулируемый нетерминал будем вычеркивать и, если это был не первый и не последний символ в правиле, для каждой вновь образовавшейся пары следующих друг за другом символов пометим соответствующую клетку матрицы точно так же, как на шаге 1, но с использованием отметки 2.

Для грамматики $\boldsymbol{G}_{a2}$ этот шаг не приведет к изменению матрицы, поскольку вычеркивание аннулируемых нетерминалов R и W оставляет в правых частях соответствующих правил по одному символу. Однако в общем случае при выполнении этого шага матрица отношения «Y есть непосредственный последователь X» может измениться.

Шаг 3. Заметим, что если непосредственным последователем какого-либо символа X является нетерминал Y, то все предшественники символа Y, очевидно, являются последователями символа X. Поэтому просмотрим все столбцы матрицы, помеченные нетерминалами. Если в клетке какой-либо строки находится непустая отметка, то добавим отметку 3 в те клетки этой строки, которые находятся в столбцах, помеченных терминалами – предшественниками символа, помечающего данный столбец. Для грамматики $\boldsymbol{G}_{a2}$ получим:

	S	*U*	*R*	*V*	*W*	+	*	(	)	*i*	*c*	►
S									1			1
U			1			3						
R												
V					1		3					
W												
+	1							3		3	3	
*		1						3		3	3	
(	1							3		3	3	
)												
i												
c												
►												

1.8.2. Определение отношения «замыкающие в цепочках»

Шаг 1. Построим пустую заготовку матрицы отношения «X есть последний символ в цепочке, выводимой из Z» (т. е. замыкающий в цепочке). Отметим все клетки главной диагонали, имея в виду, что каждый символ есть последний символ в цепочке, выводимой из него применением вывода длины 0. Затем просмотрим все правила грамматики за исключением добавленного нулевого. Если правая часть правила не пуста, отметим клетку матрицы, находящуюся на пересечении строки, помеченной последним символом правой части, и столбца, по-

меченного нетерминалом из левой части. Если последний символ правой части есть аннулируемый нетерминал, то вычеркнем его, с оставшейся цепочкой правой части повторим действия по отметке клетки матрицы, и так до тех пор, пока правая часть не станет пустой или ее последний символ не окажется терминалом или неаннулируемым нетерминалом.

	S	*U*	*R*	*V*	*W*	+	*	(	)	*i*	*c*	►
S	1		1									
U	1	1			1							
R	1		1									
V		1		1								
W		1			1							
+						1						
*							1					
(								1				
)				1					1			
i				1						1		
c				1							1	
►												1

Шаг 2. Выполним транзитивное замыкание матрицы, построенной на шаге 3 для получения отношения согласно условию 4. Смысл этого действия состоит в следующем: если символ *V* может быть последним в цепочке, выводимой из символа *U*, то все символы, замыкающие цепочки, выводимые из *V*, могут быть последними в цепочках, выводимых из *U*. Транзитивное замыкание этой матрицы выполняется по столбцам:

- просматриваются все столбцы матрицы в определенном порядке (например, слева направо);
- в каждой строке просматриваются все клетки;
- если клетка строки, помеченной символом *Z*, не пуста, то к просматриваемому столбцу логически добавляется содержимое столбца, помеченного символом *Z*;
- если хотя бы одна клетка матрицы изменилась, то после последнего столбца просмотр начинается заново.

Для рассматриваемого примера в результате получим:

	S	*U*	*R*	*V*	*W*	+	*	*(*	*)*	*i*	*c*	►
S	1		1									
U	1	1	2		1							
R	1		1									
V	2	1	2	1	2							
W	2	1	2		1							
+						1						
*							1					
(								1				
)	2	2	2	1	2				1			
i	2	2	2	1	2					1		
c	2	2	2	1	2						1	
►												1

1.8.3. Определение отношения «Y есть последователь символа X»

Шаг 1. Вычислим матрицу отношения «*Y* есть последователь символа *X*», перемножив (по правилам логического умножения и сложения; при этом непустые отметки считаются значениями «истина», а пустые – значениями «ложь») матрицу, полученную на шаге 4, и матрицу, полученную на шаге 2. В результате получим:

	S	*U*	*R*	*V*	*W*	+	*	*(*	*)*	*i*	*c*	►
S									1			1
U			1			1			1			1
R									1			1
V			1		1	1	1		1			1
W			1			1			1			1
+	1											
*		1										
(	1											
)			1		1				1			1
i			1		1				1			1
c			1		1				1			1
►												

Шаг 2. Заметим, что если символ Y есть последователь символа X, то каждый предшественник символа Y также есть последователь символа X. Выпишем из строк матрицы, полученной на предыдущем шаге, множества последователей нетерминальных символов, после чего заменим в них каждый нетерминал множеством его предшественников, объединяя множества по общепринятым правилам:

$$M_{\text{посл}}(S) = \{\,),\ \blacktriangleright\} = \{\,),\ \blacktriangleright\}$$

$$M_{\text{посл}}(U) = \{\,R, +,),\ \blacktriangleright\} = \{\,+,),\ \blacktriangleright\}$$

$$M_{\text{посл}}(R) = \{\,),\ \blacktriangleright\} = \{\,),\ \blacktriangleright\}$$

$$M_{\text{посл}}(V) = \{\,R, W, +, *,),\ \blacktriangleright\} = \{\,+, *,),\ \blacktriangleright\}$$

$$M_{\text{посл}}(W) = \{\,R, +,),\ \blacktriangleright\} = \{\,+,),\ \blacktriangleright\}$$

Множества предшественников и последователей символов грамматики будут самым существенным образом использоваться в процессе преобразования формальных грамматик в синтаксические акцепторы.

1.9. Эквивалентные преобразования грамматик

Необходимость эквивалентных преобразований грамматики возникает в тех случаях, когда требуется изменить ее свойства в целом или свойства отдельных правил или символов. При преобразованиях грамматик должно выполняться естественное требование: исходная грамматика и грамматика, полученная в результате преобразования, должны порождать один и тот же язык.

В результате преобразований могут изменяться система порождающих правил и алфавиты терминальных и нетерминальных символов. Начальный нетерминал грамматики может быть в принципе переименован (во всех правилах как в левых, так и в правых частях), но не может быть заменен другим нетерминалом. Такая замена простонапросто приведет к изменению множества правильных предложений, т. е. к изменению языка, порождаемого грамматикой.

Для иллюстрации методов эквивалентных преобразований в основном будем использовать уже известные грамматики $\boldsymbol{G_{a1}}$ и $\boldsymbol{G_{a2}}$.

1.9.1. Ликвидация нетерминального символа

Не пряморекурсивный нетерминальный символ, не являющийся начальным нетерминалом, может быть удален из алфавита нетерминалов, если:

– вместо всех вхождений этого символа в правые части правил грамматики подставить каждую цепочку символов, являющуюся правой частью каждого правила, имеющего этот нетерминал в левой части; при этом каждое правило, в правой части которого присутствовало k вхождений удаляемого нетерминала превращается в k*n правил, если для удаляемого нетерминала в грамматике было n правил;

– удалить все правила, имеющие этот нетерминал из системы порождающих правил, поскольку теперь он является недостижимым.

Удалим, например, нетерминал R из грамматики $\boldsymbol{G}_{a2}$, для чего правило 1 преобразуем в два правила, подставив в правую часть вместо символа R цепочку $+ S$ и пустую цепочку, т. е. все возможные подстановки нетерминала R. Ниже показана система порождающих правил грамматики $\boldsymbol{G}_{a2}$ до преобразования (слева) и после него (справа):

Грамматика $\boldsymbol{G}_{a2}$	
1	*S* : *U R*
2	*R* : + *S*
3	*R* :
4	*U* : *V W*
5	*W* : * *U*
6	*W* :
7	*V* : (*S*)
8	*V* : *ident*
9	*V* : *const*

Грамматика $\boldsymbol{G}_{a2\text{-}1}$	
1	*S* : *U* + *S*
2	*S* : *U*
3	*U* : *V W*
4	*W* : * *U*
5	*W* :
6	*V* : (*S*)
7	*V* : *ident*
8	*V* : *const*

Если теперь удалить из грамматики $\boldsymbol{G}_{a2\text{-}1}$ и нетерминал W, то будет получена грамматика $\boldsymbol{G}_{a2\text{-}2}$, отличающаяся от грамматики $\boldsymbol{G}_{a1}$ только названиями нетерминальных символов и правыми частями первого и третьего правил. Именно этими различиями определяется разница в ассоциативности знака операции сложения при выводе цепочек вида $a + b + c$, упоминавшаяся в разделе 1.4.2:

Грамматика $G_{a2\text{-}2}$	
1	$S : U + S$
2	$S : U$
3	$U : V * U$
4	$U : V$
5	$V : (S)$
6	$V : ident$
7	$V : const$

Легко можно убедиться (и доказать), что ликвидация нетерминала, выполняемая путем подстановки вместо него всех правых частей его правил, совершенно не изменяет язык, порождаемый грамматикой.

1.9.2. Образование нового нетерминального символа, факторизация

Любую цепочку символов, встречающуюся в правой части какого-либо одного или нескольких разных правил, можно заменить новым нетерминальным символом, превратив ее в правую часть правила для этого нетерминала. Например, в грамматике $G_{a2\text{-}2}$ можно преобразовать первое правило к виду $S : U\ X$, если добавить новый нетерминал X и правило для него $X : + S$.

Особенно полезным подобное преобразование может быть при так называемой факторизации, т. е. выделении из правых частей нескольких правил общей подцепочки (общего фактора). Например, грамматику $G_{a2\text{-}2}$ можно преобразовать обратно в грамматику $G_{a2\text{-}1}$, выделив из правил 3 и 4 общий левый фактор – цепочку, состоящую из одного символа V, заменяя остатки правил новым нетерминалом W (при этом вместо двух правил остается одно $U : V\ W$) и добавляя два правила для нетерминала W:

$$W : * U$$

$$W :$$

Обычно таким путем избавляются от существования в грамматиках правил с общим левым фактором для одного и того же нетерминала. Наличие подобных правил придает грамматике нежелательные свойства, которые мы будем рассматривать в последующем.

1.9.3. Преобразование косвенной рекурсии в прямую

Иногда (особенно в случае левой рекурсии) требуется из заданной грамматики получить эквивалентную ей, но не содержащую косвенно-рекурсивных нетерминалов. Перед таким преобразованием необходимо выявить косвенно-рекурсивные подмножества нетерминальных символов. Для этого выполняется следующая процедура, которую мы будем иллюстрировать с использованием условной грамматики $\boldsymbol{G}_x$.

Грамматика $\boldsymbol{G}_x$	
1	*X* : *N M*
2	*N* : *i Y*
3	*M* : ; *X*
4	*M* :
5	*Y* : , *Z*
6	*Y* :
7	*Z* : * *N*

Шаг 1. Строится пустая заготовка матрицы отношения рекурсивности, содержащая ровно столько строк и столбцов, сколько нетерминалов содержит алфавит нетерминальных символов. Строки и столбцы этой матрицы помечаются нетерминалами в одинаковой последовательности.

Затем просматриваются правила грамматики. Если просматриваемое правило для нетерминала *N* содержит в правой части нетерминал *M*, то в клетку строки, помеченной символом *N*, и столбца, помеченного символом *M*, заносится значение 1.

	X	*N*	*M*	*Y*	*Z*
X		1	1		
N				1	
M	1				
Y					1
Z		1			

Шаг 2. Выполняется (по обычным правилам) транзитивное замыкание матрицы с занесением в клетки отметок 2, 3, 4, … (соответствующих порядковому номеру просмотра матрицы). В итоге получим:

	X	*N*	*M*	*Y*	*Z*
X	2	1	1	2	3
N		3		1	2
M	1	2	2	3	4
Y		2		3	1
Z		1		2	3

Отметки на главной диагонали матрицы обозначают наименьшую глубину рекурсивного цикла для каждого символа (в данном случае ни один нетерминал не является пряморекурсивным). Нетерминалы *N*, *Y* и *Z* образуют отдельный замкнутый рекурсивный цикл с глубиной 3 (глубиной рекурсивного цикла называется минимальное количество непосредственных выводов, приводящих от нетерминала к цепочке, содержащей этот же нетерминал). Нетерминалы *X* и *M* также образуют рекурсивный цикл с глубиной 2.

Теперь допустим, что требуется превратить нетерминал *N* из косвеннорекурсивного в пряморекурсивный. Для этого нужно последовательно ликвидировать так, как было описано в разделе 1.9.1, два нетерминала *Y* и *Z*, находящиеся в одном рекурсивном цикле с нетерминалом *N*. В результате удаления символа *Z* получим грамматку $\boldsymbol{G}_{x\text{-}1}$ (слева), а после удаления нетерминала *Y* грамматика приобретет вид, показанный справа:

Грамматика $\boldsymbol{G}_{x\text{-}1}$	
1	*X* : *N M*
2	*N* : *i Y*
3	*M* : ; *X*
4	*M* :
5	*Y* : , * *N*
6	*Y* :

Грамматика $\boldsymbol{G}_{x\text{-}2}$	
1	*X* : *N M*
2	*N* : *i* , * *N*
3	*N* : *i*
4	*M* : ; *X*
5	*M* :

При этих преобразованиях изменились свойства символа *N*, но не изменились свойства символов *X* и *M*. Если теперь начальный нетерминал *X* превратить в пряморекурсивный (путем удаления нетерминала *M*), то грамматика будет выглядеть следующим образом (будучи эквивалентной исходной грамматике $\boldsymbol{G}_x$):

Грамматика $G_{x\text{-}3}$	
1	$X : N ; X$
2	$X : N$
3	$N : i , * N$
4	$N : i$

Очевидно, что это преобразование (как и все остальные) изменяет не только свойства отдельных символов, но и некоторые свойства грамматики в целом.

1.9.4. Изменение стороны рекурсии

Рекурсивные нетерминальные символы грамматики могут обладать свойством левой, правой и общей рекурсии, причем в любом сочетании. Это свойство может влиять на применимость грамматики в качестве основы при построении синтаксического анализатора.

Рассмотрим один из способов преобразования, предназначенный для устранения свойства левой рекурсивности нетерминала.

Пусть нетерминал N является леворекурсивным. В том случае, если он косвенно леворекурсивен, вначале необходимо применить предыдущее преобразование для превращения этого символа в прямолеворекурсивный.

Вначале рассмотрим простейший случай, когда этот нетерминал в системе порождающих правил имеет в точности два правила вида:

$$N : N\,\beta$$

$$N : \gamma,$$

причем цепочка γ не имеет символа N в своем множестве предшественников.

Заметим, что случай отсутствия правила вида $N : \gamma$ нет смысла рассматривать, поскольку тогда нетерминал N будет бесплодным и его следует просто удалить из грамматики вместе с правилом $N : N\,\beta$.

Из нетерминала N выводятся цепочки:

$$\gamma$$

$$\gamma\,\beta$$

$$\gamma\,\beta\,\beta$$

$$\gamma\,\beta\,\beta\,\beta$$

$$\ldots$$

Добавим новый нетерминал M и заменим два правила для символа N на такую совокупность правил:

$$N : \gamma\ M$$

$$M : \beta\ M$$

$$M : \boldsymbol{\varepsilon}$$

Теперь из нетерминала N выводятся в точности такие же цепочки, как и ранее, но он уже не является прямолеворекурсивным. Новый нетерминал M обладает свойством правой рекурсии (а если из β может быть выведена пустая цепочка, то и левой рекурсии тоже, но это уже будет предметом следующего этапа преобразования грамматики).

Теперь этот способ смены стороны рекурсии можно обобщить на случай, когда для нетерминала N в исходной грамматике есть более чем два правила:

$$\begin{array}{ll} N : N\,\beta_1 & \qquad N : \gamma_1 \\ N : N\,\beta_2 & \qquad N : \gamma_2 \\ \ldots & \qquad \ldots \\ N : N\,\beta_k & \qquad N : \gamma_r \end{array}$$

Добавляется новый нетерминал M. Каждое правило вида $N : N\,\beta_1$ заменяется на правило $M : \beta_1\ M$. Каждое правило вида $N : \gamma_1$ заменяется правилом $N : \gamma_1\ M$. Добавляется одно правило $M : \boldsymbol{\varepsilon}$. В результате получается:

$$\begin{array}{ll} N : \gamma_1\ M & \qquad M : \beta_1\ M \\ N : \gamma_2\ M & \qquad M : \beta_2\ M \\ \ldots & \qquad \ldots \\ N : \gamma_r\ M & \qquad M : \beta_k\ M \\ & \qquad M : \boldsymbol{\varepsilon} \end{array}$$

Легко можно показать, что из нетерминала N согласно этой совокупности правил выводится в точности такое же множество цепочек, что и согласно исходной совокупности правил.

Этим преобразованием можно устранить свойство левой рекурсии всех нетерминалов грамматики, а следовательно, – свойство леворекурсивности грамматики в целом.

Существуют и другие, более сложные методы эквивалентных преобразований грамматик, однако мы их рассматривать не будем.

1.10. Постановка задачи синтаксического акцепта на основе формальной грамматики языка

Пусть даны формальная грамматика $\boldsymbol{G} = \{ \boldsymbol{A}_t, \boldsymbol{A}_n, S, \boldsymbol{P} \}$, порождающая некоторый язык $\boldsymbol{L}$, и цепочка ω, состоящая из терминальных символов этой грамматики, т. е. предложение. Задача синтаксического акцепта состоит в том, чтобы выяснить, правильно ли это предложение. Поскольку любое правильное предложение языка $\boldsymbol{L}$ выводится из начального нетерминала S грамматики $\boldsymbol{G}$, эта задача, по существу, сводится к задаче выяснения, существует ли дерево грамматического разбора цепочки ω:

Ясно, что выяснить, существует ли дерево, можно путем восстановления последовательности непосредственных выводов цепочки ω из начального нетерминала S.

Дерево грамматического разбора имеет два измерения, поэтому задачу его восстановления можно решать, двигаясь:

- сверху вниз по вертикали и
  - слева направо по горизонтали,
  - справа налево по горизонтали;
- снизу вверх по вертикали и
  - слева направо по горизонтали,
  - справа налево по горизонтали.

Направление движения по горизонтали сопряжено с направлением чтения и обработки входной цепочки терминалов. Естественным является направление слева направо, т. е. от первого символа цепочки к последнему. Обратное направление движения использовать в принци-

пе можно, но основанные на нем методы не имеют принципиального отличия от тех методов, которые используют естественное направление обработки, и поэтому обычно не рассматриваются.

Методы, осуществляющие восстановление дерева разбора в направлении сверху вниз по вертикали обычно называются нисходящими, а в направлении снизу вверх – восходящими.

При нисходящем восстановлении дерева разбора на каждом шаге должно быть принято решение о замене самого левого нетерминала текущего уровня на правую часть одного из тех правил грамматики, в которых этот нетерминал находится в левой части.

При восходящем восстановлении дерева на каждом шаге должно быть принято решение о том, какую из правых частей правил грамматики, находящихся внутри цепочки символов, образующей текущий уровень, нужно заменить на нетерминал из левой части этого правила.

Если решение, принятое на каком-либо шаге, впоследствии может быть пересмотрено (или если, как в алгоритме Эрли, параллельно или квазипараллельно просматриваются все возможные варианты, получаемые при принятии различных решений), то говорят, что метод синтаксического акцепта является возвратным или недетерминированным.

Сложность (зависимость затрат времени на восстановление дерева от длины предложения n) недетерминированных методов оценивается от $O(n^3)$ до $O(2^n)$ и является неприемлемой для применения в трансляторах с практической точки зрения. К счастью, для некоторых классов грамматик существуют детерминированные (безвозвратные или безоткатные) как нисходящие, так и восходящие методы восстановления дерева разбора, сложность которых оценивается как $O(n)$. Именно такие методы и будут рассмотрены далее.

Процесс восстановления структуры дерева мы будем стремиться организовать в виде последовательности шагов, на каждом из которых используется только один (очередной, на первом шаге – самый первый) терминальный символ из входной цепочки и формируется цепочка символов, образующая очередной уровень дерева. Забегая вперед, заметим, что один и тот же входной символ может использоваться при выполнении нескольких последовательно следующих друг за другом шагов. Переход к следующему символу из входной цепочки будет осуществляться только по мере необходимости, т. е. в тот момент, когда текущий символ становится ненужным для продолжения процесса восстановления дерева.

В результате выполнения некоторой последовательности шагов восстановления дерева формируется информация, необходимая для принятия решений на последующих шагах. Такую информацию мы предполагаем хранить в специально организованной памяти (стеке).

Восстановление дерева разбора, естественно, предполагает использование системы порождающих правил грамматики, определяющей данный язык. Все сказанное означает, что грамматика будет преобразовываться в конечный автомат со стековой памятью.

Для каждой группы методов (как нисходящих, так и восходящих) будут преследоваться три основные цели.

1. Формирование идей методов и их необходимую детализацию до полного понимания существа процессов синтаксического акцепта.
2. Определение условий, которым должна удовлетворять формальная грамматика для реализации рассматриваемого метода на ее основе.
3. Разработка алгоритмов преобразования системы порождающих правил формальной грамматики в конечный автомат со стековой памятью, способный распознавать правильность предложений языка.

Отметим, что ни одна реализация рассматриваемых в данном учебном пособии методов синтаксического анализа в действительности не восстанавливает полное дерево разбора в каком бы то ни было представлении. Каждый из этих методов всего только выясняет, может или не может входная цепочка символов быть выведена из начального нетерминала грамматики. Восстановить дерево разбора, как таковое, в принципе возможно, если проследить полную историю работы автомата для данной цепочки.

2. НИСХОДЯЩИЕ МЕТОДЫ СИНТАКСИЧЕСКОГО АКЦЕПТА

Нисходящими называются такие методы синтаксического акцепта, при которых восстановление дерева протекает сверху (от начального нетерминала ***S***) вниз(к цепочке ω).

Каждый шаг процесса восстановления дерева состоит в применении одного непосредственного вывода, т. е. в замене одного нетерминального символа цепочкой символов, являющейся правой частью какого-либо правила для этого нетерминала. Главное прагматическое требование к этому процессу – необходимость организации безоткатного однонаправленного движения вниз по дереву. Отсюда следует, что выбор правила на каждом шаге должен осуществляться таким образом, чтобы гарантировать восстановление дерева для любого правильного предложения и обнаружение невозможности это сделать для любого неправильного предложения. Оказывается, дать такие гарантии можно отнюдь не для любой грамматики и, более того, – не для любого языка. Существуют языки, для которых любая порождающая грамматика не позволяет организовать безвозвратное нисходящее восстановление дерева грамматического разбора (в чистом виде, т. е. без применения специальных мер). Существуют и такие языки, для которых одни порождающие грамматики пригодны для нисходящего восстановления дерева, а другие – нет.

Вначале будет сформулирована основная идея группы нисходящих методов, затем определены критерии выбора правила для выполнения каждого очередного непосредственного вывода и на этой основе – условий, которым должна удовлетворять грамматика для организации восстановления дерева разбора сверху вниз. В общей форме будут определены алгоритм нисходящего синтаксического акцепта, способы процедурной реализации синтаксического акцептора и соответственно преобразования порождающей грамматики в текст программы так называемого «рекурсивного спуска». Затем будут рассмотрены автоматная реализация нисходящего синтаксического акцептора, устройство конечного автомата со стековой памятью для хранения состояний и методы преобразования по-

рождающей грамматики в управляющую таблицу такого автомата. В качестве альтернативы приводится такая автоматная реализация, при которой автомат имеет единственное состояние, а в его стеке хранятся не состояния, а символы порождающей грамматики.

2.1. Основная идея нисходящего восстановления дерева грамматического разбора

Основная идея группы нисходящих методов восстановления дерева грамматического разбора была разработана на основе изучения довольно узкого подкласса контекстно-свободных грамматик – так называемых s-грамматик. К s-грамматикам относятся грамматики, порождающие правила которых удовлетворяют трем следующим условиям.

1. Нет правил с пустой правой частью, соответственно нет аннулируемых нетерминалов.
2. Правая часть каждого правила начинается с терминала.
3. Если для некоторого нетерминала есть несколько правил, то их правые части начинаются с различных терминальных символов.

Такие грамматики интересны исключительно с теоретической точки зрения, поскольку класс языков, порождаемых такими грамматиками, слишком узок.

Приведем пример s-грамматики:

Грамматика $\boldsymbol{G_s}$	
	Правила
1	*A* : (*A*) *X*
2	*A* : *ident X*
3	*X* : + *Y*
4	*X* : * *Y*
5	*Y* : (*A*)
6	*Y* : *ident*

Эта грамматика порождает язык, являющийся правильным подмножеством языка всех арифметических выражений, порождаемого грамматиками $\boldsymbol{G_{a1}}$ и $\boldsymbol{G_{a2}}$ (читателям в качестве упражнения предлагается попытаться построить s-грамматику, полностью эквивалентную грамматикам $\boldsymbol{G_{a1}}$ и $\boldsymbol{G_{a2}}$.). Например, предложение *(a + b) * c* является правильным в языках, порождаемых всеми тремя грамматиками, однако правильное арифметическое выражение *a + b * c* не может быть выведено из начального нетерминала грамматики $\boldsymbol{G_s}$.

Несмотря на ограниченную практическую применимость s-грамматик, они послужили теоретическим обоснованием для формирования идей нисходящего синтаксического акцепта.

Рассмотрим способ восстановления дерева грамматического разбора последовательности терминалов *(a + b) * c* из начального нетерминала *A* грамматики $\boldsymbol{G}_s$, т. е. способ поиска вывода $A \Rightarrow (a + b) * c$. Начальный нетерминал является корнем (начальным уровнем) дерева грамматического разбора. Для этого нетерминала в грамматике есть два правила:

A : *(A) X*

A : *ident X*

Для принятия решения о том, правой частью какого из этих двух правил нужно заменить начальный нетерминал на первом шаге, нет никаких других оснований, кроме анализа символов предложения.

Очевидно, что первым непосредственным выводом не может быть применение правила *A* : *ident X*, поскольку из цепочки, начинающейся с терминала *ident*, не может быть выведена цепочка, начинающаяся с открывающей скобки. Именно в силу того что первым символом анализируемого предложения является открывающая скобка, на первом шаге восстановления дерева его разбора из нетерминала *A* должно быть применено правило номер 1 грамматики, т. е. построен первый уровень восстанавливаемого дерева (рис. 6.) $A \rightarrow (A) X$:

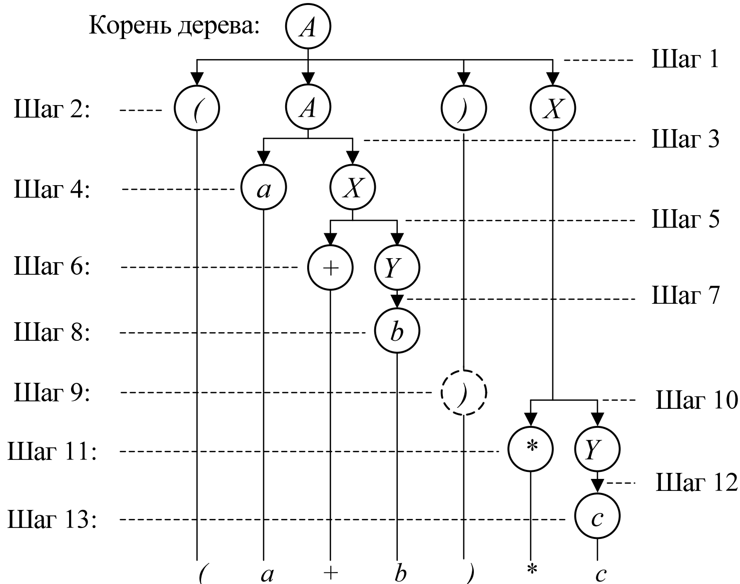


Рис. 6. Построение дерева разбора в s-грамматике

На рис. 6 показано традиционное представление дерева разбора (за исключением вершины в пунктире, соответствующей символу), которая нужна для пояснения действий, выполняемых на шаге 9). Однако в процессе восстановления этого дерева фактически используются полные цепочки символов, являющиеся его уровнями (в рамки заключены текущие символы уровней для каждого очередного шага):

0	$\boxed{A}$	$A \to (\,A\,)\,X$
3	$(\,\boxed{A}\,)\,X$	$A \to a\,X$
5	$(\,a\,\boxed{X}\,)\,X$	$X \to +\,Y$
7	$(\,a + \boxed{Y}\,)\,X$	$Y \to b$
9	$(\,a + b\,\boxed{)}\,X$	
10	$(\,a + b\,)\,\boxed{X}$	$X \to *\,Y$
12	$(\,a + b\,)\,*\,\boxed{Y}$	$Y \to c$
13	$(\,a + b\,)\,*\,c$	Останов, дерево восстановлено

На шаге 2 выясняется, что текущим символом очередного уровня дерева разбора (т. е. цепочки *(A) X*) является терминал (и проверяется, совпадает ли этот терминал с текущим символом, прочитанным из предложения. Поскольку эти два символа совпадают, выполняется переход к следующему символу в текущем уровне дерева разбора и чтение следующего терминала из предложения. Теперь текущим символом предложения является идентификатор *a*, а текущим символом уровня дерева разбора – нетерминал *A*. Если предложение правильно, то из цепочки *A) X* должен выводиться остаток предложения, т. е. цепочка *a + b) * c –*.

При выполнении третьего шага восстановления дерева нужно принять решение о выборе правила для замены нетерминала *A*, являющегося самым первым символом текущего уровня. В силу того что текущим терминалом из предложения сейчас является идентификатор *a*, на этом шаге для замены нетерминала *A* не может быть использовано правило номер 1 (из цепочки *(A) X) X*, получившейся бы при применении этого правила, невозможно вывести цепочку *a + b) * c)*. Нетерминал *A* на этом шаге следует заменить на *ident X* именно потому, что правая часть этого правила начинается с того же терминала, что и остаток предложения. Применение этого непосредственного вывода позволит получить следующий уровень дерева разбора:

$$A \to (\,A\,)\,X \to (\,ident\;X\,)\,X.$$

Подставим символ *a* вместо общего названия группы слов *ident*:

$$A \rightarrow (A) X \rightarrow (a X) X.$$

На шаге 4 (аналогично шагу 2) проверяется совпадение терминала из уровня дерева с первым символом остатка предложения и чтение следующих символов из каждой цепочки. Шаги, показанные на рис. 6. слева, кажутся тривиальными, но достаточно обратить внимание на шаг 9, чтобы понять их важность и необходимость: здесь проверяется, есть ли в анализируемом предложении закрывающая скобка, парная ранее встреченной открывающей.

Если бы предложение выглядело, например, так: *(a + b * c*, то ошибка была бы обнаружена именно на шаге 9. Заметим, что останов по ошибке возможен и при выполнении любого из шагов, показанных на рис. 6 справа, если с текущего символа из предложения не начинается ни одна правая часть правила для текущего нетерминала из очередного уровня дерева разбора. Пусть, например, входным предложением является цепочка терминалов *(a + *) * c*. В этом случае останов по ошибке произойдет на шаге 7, поскольку для нетерминала *Y* в грамматике нет правила, начинающегося с символа *.

На следующем, пятом шаге нужно найти правило для замены нетерминала *X*, зная что из цепочки *X*) *X* должна быть выведена цепочка *+ b) * c*. Из двух правил для нетерминала *X*:

$$X : + Y$$

$$X : * Y$$

на этом шаге, очевидно, должно быть выбрано первое правило:

$$A \rightarrow (A) X \rightarrow (a X) X \rightarrow (a + Y) X$$

В противном случае будет получена цепочка *(a * Y) X*, из которой, очевидно, не может быть выведено предложение *a + b) * c*.

Шестой шаг аналогичен второму. Текущим символом уровня дерева разбора является терминал +, поэтому выполняется сравнение этого символа с текущим терминалом из предложения и переход к очередным символам (нетерминалу *Y* из дерева и терминалу *b* из предложения), поскольку сравниваемые символы совпадают.

Последующие шаги восстановления дерева разбора данного предложения выполняются точно таким же образом. После шага 13 оказывается, что ни в предложении, ни на текущем уровне дерева нет текущих символов. Процесс восстановления дерева закончен, предложение является правильным.

Общее описание алгоритма любого шага восстановления дерева разбора для s-грамматик можно сформулировать следующим образом.

1. Построить начальный уровень дерева в виде цепочки, содержащей единственный начальный нетерминал S; установить этот символ текущим, прочитать и сделать текущим первый символ t из предложения.

2. Определить тип текущего символа s из текущего уровня дерева разбора.

3. Если текущий символ s – нетерминал, то выполнить поиск правила, имеющего этот нетерминал, в левой части, и правую часть, начинающуюся с текущего терминала t из предложения, и:
   - 3.1) остановиться по ошибке, если нет такого правила;
   - 3.2) если правило найдено, то сформировать новый уровень дерева разбора, заменив символ s правой частью найденного правила и установить текущим первый символ правой части.

4. если текущий символ s – это терминал, то сравнить его с текущим символом t из предложения, и:
   - 4.1) остановиться по ошибке, если символы s и t различны;
   - 4.2) Если символы s и t совпадают, то проверить, существуют ли следующие символы предложения и текущего уровня, и:
     - 4.2.1) остановиться по ошибке, если в какой-либо цепочке следующий символ есть, а в другой – нет;
     - 4.2.2) остановиться по концу процесса восстановления дерева правильного предложения, если обе цепочки исчерпались одновременно;
     - 4.2.3) если же ни та, ни другая цепочка не закончилась, то установить текущими следующие символы предложения и текущего уровня дерева и вернуться к определению типа текущего символа s (пункт 2 алгоритма).

На каждом шаге j восстановления дерева разбора правильного предложения (рис. 7) этим алгоритмом гарантируется, что из остатка уровня дерева, начинающегося с символа s, выводится остаток предложения, начинающийся с терминала t.

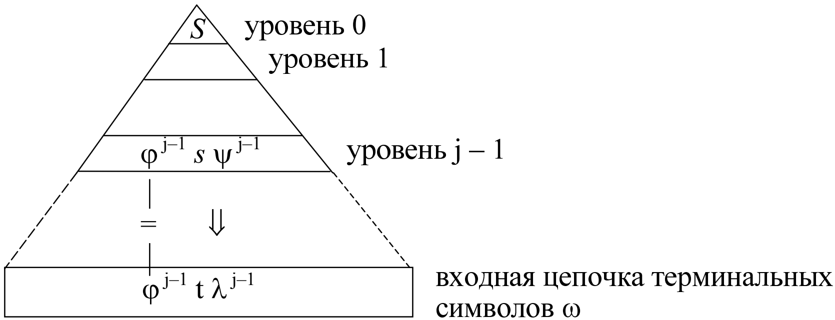


Рис. 7. Связь остатка уровня дерева и остатка предложения

Выпишем это утверждение отдельно:

$$\varphi^{j-1}\, s\, \psi^{j-1} \Rightarrow \varphi^{j-1}\, t\, \lambda^{j-1}.$$

В силу того что цепочки φ^{j-1} слева и справа от знака операции вывода одинаковы, их можно отбросить:

$$s\, \psi^{j-1} \Rightarrow t\, \lambda^{j-1}.$$

Если символ s – это терминал, то согласно п. 4 алгоритма он должен совпадать с терминалом t:

$$s\, \psi^{j-1} = t\, \psi^{j-1} \Rightarrow t\, \lambda^{j-1}.$$

Если же символ s – это нетерминал, то согласно п. 3 алгоритма он будет заменен правой частью правила, начинающейся с терминала t (того правила, чье множество выбора содержит терминал t):

$$s\, \psi^{j-1} \rightarrow \beta_i\, \psi^{j-1} = t\, \gamma_i\, \psi^{j-1} \Rightarrow t\, \lambda^{j-1},$$

где $\beta_i = t\, \gamma_i$ – правая часть i-го правила грамматики.

2.2. Множества выбора правил грамматик

Детерминированность (безоткатность) процесса восстановления дерева разбора алгоритмом из предыдущего пункта обеспечивается ограничениями, наложенными на порождающие правила s-грамматики, существенно используемыми в момент выбора правила для замены нетерминала на текущем уровне.

Можно считать, что с каждым правилом s-грамматики связано множество терминальных символов, содержащее ровно по одному терминальному символу – самому первому символу, с которого начинается правая часть этого правила. Будем называть эти множества множествами выбора. Выпишем эти множества в отдельной колонке.

Грамматика $\boldsymbol{G_s}$		
	Правила	Множества выбора правил
1	*A* : (*A*) *X*	(
2	*A* : *ident X*	*ident*
3	*X* : + *Y*	+
4	*X* : * *Y*	*
5	*Y* : (*A*)	(
6	*Y* : *ident*	*ident*

Сформулируем теперь п. 3 алгоритма нисходящего восстановления дерева разбора следующим образом:

3. Если текущий символ текущего уровня дерева – это нетерминал, то выполнить поиск правила, имеющего этот нетерминал в левой части, множество выбора которого содержит текущий терминал *t* из предложения, и:

…

Существо действий, выполняемых в этом пункте, не изменилось. Для детерминированности процесса нисходящего восстановления дерева принципиально важно то, что множества выбора правил с одним и тем же нетерминалом в левых частях попарно не пересекаются (не содержат общих терминалов), и совершенно не важна мощность этих множеств.

Однако теперь в результате явного выделения множеств выбора возникла возможность использовать грамматики, для правил которых мощности множеств выбора могут отличаться от единицы. Для s-грамматик мощности множеств равны единице, потому что правая часть каждого правила начинается с терминала и нет правил с пустой правой частью.

Снимем второе ограничение на правила s-грамматик. Грамматики, у которых в правой части любого правила находится цепочка, начинающаяся с терминала, либо пустая цепочка, называются q-грамматиками.

Приведем пример q-грамматики.

Грамматика G_q		
	Правило	Множества выбора правил
1	A : (A) X	(
2	A : *ident* X	*ident*
3	A : *const* X	*const*
4	X : + A	+
5	X : * A	*
6	X : ε	), ►

Эта грамматика получена из грамматики G_s путем:

– добавления правила A : *const* X, правая часть которого начинается с терминального символа;

– добавления правила с пустой правой частью X : ε;

– замены нетерминала Y нетерминалом A в правилах для нетерминала X с непустой правой частью (теперь это возможно, однако до добавления правила X : ε такая замена приводила бы к тому, что все нетерминалы были бы бесплодными и грамматика не порождала бы никакого языка);

– удаления нетерминала Y и правил для него, поскольку теперь это недостижимый нетерминал.

Грамматика G_q эквивалентна грамматикам G_{a1} и G_{a2}, т. е. порождает тот же язык скобочных арифметических выражений, но свойства ее существенно отличаются от свойств этих грамматик. Основное отличие состоит в том, что при восстановлении деревьев разбора не могут учитываться приоритеты знаков арифметических операций. Например, предложение $a + b * c$ в грамматиках G_{a1} и G_{a2} эквивалентно предложению $a + (b * c)$, однако в грамматике G_q оно рассматривается как $(a + b) * c$.

Некоторые q-грамматики оказываются пригодными для нисходящего восстановления дерева грамматического разбора. Это грамматики, у которых множества выбора правил с одинаковыми нетерминалами в левой части не пересекаются. Как видно из колонки «Множества выбора», грамматика G_q именно такова, однако пока не определено, каким способом было вычислено множество выбора правила X : ε.

Этот способ можно определить на основе требования выводимости остатка правильного предложения из остатка текущего уровня дерева разбора, сформулированного в конце разд. 2.2.1:

$$s\,\psi^{j-1} \Rightarrow t\,\lambda^{j-1}.$$

Подставим для определенности вместо обозначения произвольного текущего символа s нетерминал X:

$$X\,\psi^{j-1} \Rightarrow t\,\lambda^{j-1}.$$

Если для данного предложения на этом шаге восстановления дерева разбора должно быть применено правило $X : \boldsymbol{\varepsilon}$, то условие выводимости будет выглядеть так:

$$\psi^{j-1} \Rightarrow t\,\lambda^{j-1}.$$

В силу того что цепочка $\varphi^{j-1}\,X\,\psi^{j-1}$, из которой взят остаток $X\,\psi^{j-1}$, выведена из начального нетерминала грамматики, любой терминал, с которого начинается цепочка, выводимая из ψ^{j-1}, обязан быть последователем нетерминала X (см. разд. 1.8, определение отношения следования).

Таким образом, множество выбора любого правила с пустой правой частью совпадает с множеством последователей нетерминала из левой части этого правила. Множество последователей нетерминала X в грамматике $\boldsymbol{G_q}$ состоит из единственного символа *)*.

Попытаемся восстановить дерево грамматического разбора того же самого предложения *(a + b) * c* из начального нетерминала грамматики $\boldsymbol{G_q}$. При этом интерес представляют только те шаги, на которых принимается решение о выборе правила для замены нетерминала:

0 $\boxed{A}$	$A \rightarrow (\,A\,)\,X$
1 $(\,\boxed{A}\,)\,X$	$A \rightarrow a\,X$
2 $(\,a\,\boxed{X}\,)\,X$	$X \rightarrow +\,A$
3 $(\,a + \boxed{A}\,)\,X$	$A \rightarrow b\,X$
4 $(\,a + b\,\boxed{X}\,)\,X$	$X \rightarrow \boldsymbol{\varepsilon}$
5 $(\,a + b\,\boxed{)}\,X$	
6 $(\,a + b\,)\,\boxed{X}$	$X \rightarrow *\,A$
7 $(\,a + b\,)\,*\,\boxed{A}$	$A \rightarrow c\,X$
8 $(\,a + b\,)\,*\,c\,\boxed{X}$	$X \rightarrow ?$

Как видно из этой истории, нетерминал X на шаге 2 заменяется правой часть правила 4 (X : + A), на шаге 4 – правой частью правила 6 (X : ε), а на шаге 6 – правой частью правила 5 (X : * A). Все эти правила для замены определены на основе множеств выбора точно так же, как и правила для замены нетерминала A.

Однако на шаге 8 возникает проблема: текущего входного символа нет, предложение исчерпано, поэтому нет формальных оснований для выбора какого-бы то ни было правила для замены нетерминала X. С другой стороны, неформальный взгляд на сложившуюся к шагу 8 ситуацию приводит к выводу, что правильным действием будет выбор правила 6 (X : ε). Для того чтобы такой выбор стал обоснованным и формально, нужно добавить к грамматике специальное правило:

$$Z : A \blacktriangleright,$$

трактуемое как утверждение: после правильного предложения следует конец файла (псевдотерминал ►).

В результате этого нетерминалы A и X обретут символ ► в качестве своих последователей, поэтому он появится в множестве выбора правила 6:

Грамматика ***G*** *?*		
	Правило	Множества выбора правил
0	*Z* : *A* ►	*(*, *ident*, *const*
1	*A* : *(A) X*	*(*
2	*A* : *ident X*	*ident*
3	*A* : *const X*	*const*
4	*X* : + *A*	+
5	*X* : * *A*	*
6	*X* : **ε**	*)*, ►

Применение правила 6 (X : ε) на последнем шаге восстановления дерева становится формально обоснованным, поскольку предложение теперь надо рассматривать так: *(a + b) * c* ►.

Однако полученная в результате добавления этого правила грамматика ***G*** *?* теперь не удовлетворяет ограничениям, наложенным на q-грамматики. Множество выбора добавленного правила также не может быть вычислено по ранее сформулированным правилам. Расширим эти правила.

Заметим, что способ вычисления множества выбора для правил с непустой правой частью, начинающихся с терминального символа, фактически основан на том, что из цепочки вида $t\ \alpha$ могут быть выведены только цепочки, начинающиеся с терминала t. А это утверждение, в свою очередь, является следствием (или эквивалентом) того, что терминал t является предшественником цепочки $t\ \alpha$.

Распространяя эти рассуждения на случай, когда цепочка символов правой части правила $Z : A$ ▶ начинается с нетерминала, можно получить множество выбора правила номер 0 как множество предшественников цепочки A ▶, равное {*(*, *ident*, *const*}.

Грамматика $\boldsymbol{G}_?$ с полностью определенными множествами выбора получает окончательный вид:

Грамматика $\boldsymbol{G}_?$		
	Правило	Множества выбора правил
0	$Z : A$ ▶	*(*, *ident*, *const*
1	$A : (A)\ X$	*(*
2	$A : ident\ X$	*ident*
3	$A : const\ X$	*const*
4	$X : + A$	+
5	$X : * A$	*
6	$X : \varepsilon$	*)*, ▶

Вывод предложения $(a + b) * c$ ▶ из начального нетерминала Z грамматики $\boldsymbol{G}_?$ теперь будет выглядеть так:

0	$\boxed{Z}$ ▶	$Z \to A$ ▶
1	$\boxed{A}$ ▶	$A \to (A)\ X$
2	$(\boxed{A})\ X$ ▶	$A \to a\ X$
3	$(a \boxed{X})\ X$ ▶	$X \to +\ A$
4	$(a + \boxed{A})\ X$ ▶	$A \to b\ X$
5	$(a + b \boxed{X})\ X$ ▶	$X \to \varepsilon$
6	$(a + b \boxed{)}\ X$ ▶	
7	$(a + b) \boxed{X}$ ▶	$X \to *\ A$
8	$(a + b) * \boxed{A}$ ▶	$A \to c\ X$
9	$(a + b) * c \boxed{X}$ ▶	$X \to \varepsilon$
10	$(a + b) * c$ ▶	Останов, дерево восстановлено

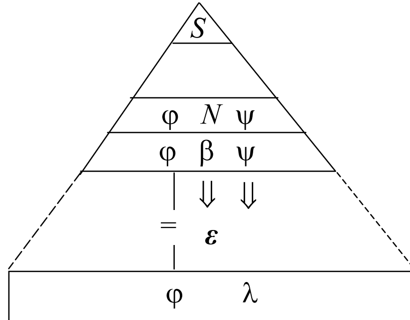
Способ вычисления множества выбора добавочного правила может быть распространен и на другие правила, правая часть которых начинается с нетерминала.

При этом требует уточнения один специальный случай, когда правая часть правила $N : \beta$ представляет собой цепочку, состоящую только из аннулируемых нетерминалов.

Из очевидных соображений следует, что множество выбора такого правила является объединением множества предшественников цепочки β с множеством последователей нетерминала N из левой части.

В самом деле, если при выводе некоторого предложения используется это правило и из цепочки β выводится непустая последовательность терминалов, то эта последовательность начинается с одного из предшественников β. Поэтому все предшественники цепочки β должны входить в множество выбора правила $N : \beta$.

Если же при выводе правильного предложения из цепочки β была выведена пустая цепочка символов, то цепочка терминалов λ, выведенная из остатка уровня дерева ψ, обязана начинаться с любого из символов-последователей нетерминала N:



Итак, окончательно зафиксируем три способа вычисления множества выбора ($M_{выб}$) любого правила грамматики $N : \alpha_j$ в зависимости от того, какова цепочка символов α_j, составляющая его правую часть.

1. $M_{выб}(N : \alpha_j) = M_{пр}(\alpha_j)$, если α_j содержит хотя бы один терминал или неаннулируемый нетерминал.

2. $M_{выб}(N : \alpha_j) = M_{посл}(N)$, если α_j есть пустая цепочка.

3. $M_{выб}(N : \alpha_j) = M_{пр}(\alpha_j) \cup M_{посл}(N)$, если α_j не пуста, но состоит только из аннулируемых нетерминалов.

Здесь $M_{\text{пр}}(\alpha_j)$ – множество предшественников цепочки α_j;
$M_{\text{посл}}(N)$ – множество последователей нетерминала N.

Понятие множеств выбора существенным образом используется в определении класса грамматик, включающего s- и q-грамматики.

2.3. LL(1)-грамматики

LL(1)-грамматикой называется такая контекстно-свободная грамматика, у которой множества выбора правил с одинаковым нетерминалом в левой части попарно не пересекаются.

Любая такая грамматика может быть использована для организации нисходящего детерминированного восстановления дерева грамматического разбора предложений порождаемого ею языка. Другими словами, на основе такой грамматики может быть построен детерминированный нисходящий синтаксический акцептор, проверяющий правильность предложений языка, порождаемого грамматикой.

Принято считать, что символы в названии класса **LL(1)**-грамматик обозначают:

– первая буква **L** (сокращение слова left – левый) – чтение слов анализируемого предложения производится слева направо;

– вторая буква **L** (сокращение слова leftmost – самый левый) – на каждом шаге принимается решение для замены самого левого нетерминала из текущего уровня восстанавливаемого дерева;

– цифра **1** в скобках обозначает количество символов из начала остатка предложения, необходимых для принятия решения о выборе правила на каждом шаге детерминированного нисходящего восстановления дерева грамматического разбора.

Любая s-грамматика одновременно является **LL(1)**-грамматикой. Некоторые (но не все) q-грамматики являются **LL(1)**-грамматиками. Существуют грамматики, не являющиеся ни s-грамматиками, ни q-грамматиками, но относящиеся к классу **LL(1)**. Это грамматики, в которых есть порождающие правила, правая часть которых начинается с нетерминального символа, но такая, что множества выбора правил с одинаковым нетерминалом в левой части попарно не пересекаются.

Вычислим множества выбора для правил известных нам грамматик $\boldsymbol{G}_{a1}$ и $\boldsymbol{G}_{a2}$, используя свойства символов и множества предшественников и последователей, определенные в разд. 2.1, и проверим, относятся ли эти грамматики к классу **LL(1)**:

Грамматика G_{a1}				Грамматика G_{a2}			
	Правило	Способ	Множество		Правило	Способ	Множество
0	*Z* : *S* ▶	$M_{пр}$(*S* ▶)	(, *ident*, *const*	0	*Z* : *S* ▶	$M_{пр}$(*S* ▶)	(, *ident*, *const*
1	*S* : *S* + *T*	$M_{пр}$(*S* + *T*)	(, *ident*, *const*	1	*S* : *U R*	$M_{пр}$(*U R*)	(, *ident*, *const*
2	*S* : *T*	$M_{пр}$(*T*)	(, *ident*, *const*	2	*R* : + *S*	$M_{пр}$(+ *S*)	+
3	*T* : *T* * *V*	$M_{пр}$(*T* * *V*)	(, *ident*, *const*	3	*R* :	$M_{посл}$ (*R*)	(, ▶
4	*T* : *V*	$M_{пр}$(*V*)	(, *ident*, *const*	4	*U* : *V W*	$M_{пр}$(*V W*)	(, *ident*, *const*
5	*V* : *(S)*	$M_{пр}$(*(S)*)	(	5	*W* : * *U*	$M_{пр}$(+ *S*)	*
6	*V* : *ident*	$M_{пр}$(i)	*ident*	6	*W* :	$M_{посл}$ (*W*)	+, (, ▶
7	*V* : *const*	$M_{пр}$(*c*)	*const*	7	*V* : *(S)*	$M_{пр}$(*(S)*)	(
				8	*V* : *ident*	$M_{пр}$(*ident*)	*ident*
				9	*V* : *const*	$M_{пр}$(*const*)	*const*

Вначале рассмотрим свойства грамматики G_{a2} . Заметим, что в ее системе порождающих правил для каждого нетерминала либо есть единственное правило, либо множества выбора нескольких правил, содержащих один и тот же нетерминал в левой части (правила 2 и 3 для *R*, правила 5 и 6 для *W*, правила 7, 8 и 9 для *V*) не пересекаются. Поэтому при восстановлении дерева разбора на любом шаге по одному текущему входному символу можно выбрать единственное правило для замены любого нетерминала – то, чье множество выбора содержит текущий входной терминал.

Например, если самым левым нетерминалом на данном шаге оказался символ *W*, а первым терминалом остатка входной цепочки является знак операции сложения + (или закрывающая скобка *)* или конец файла ▶), то заменять символ *W* следует на правую часть правила 6, т. е. на пустую цепочку. Если же первый терминал остатка входной цепочки есть знак операции умножения *, то вместо *W* необходимо подставить цепочку * *U*, т. е. правую часть правила 5. Если же первый символ остатка предложения есть идентификатор *ident*, константа *const* или открывающая скобка *(*, то никакое правило не может быть использовано для замены нетерминала *W*. Это свидетельствует о том, что входная цепочка не может быть выведена из начального нетерминала этой грамматики и, следовательно, не является правильным предложением. Например, при попытке восстановления дерева разбора цепочки *(x+ y) z* ▶ может возникнуть именно эта ситуация. Очередной уровень дерева разбора, выведенный из нетерминала *S*, будет выглядеть так:

$$S \Rightarrow (\, x + y\,)\ \boxed{W}\ R,$$

т. е. самым левым нетерминалом на этом уровне будет символ W. Первым символом остатка входной цепочки является идентификатор z. Поскольку идентификатор не входит ни в одно множество выбора правил, имеющих нетерминал W в левой части, постольку не существует способа вывода остатка входной цепочки, имеющего вид z ▶, из цепочки $W\ R$, а следовательно, и дерева грамматического всей входной цепочки $(\ x + y\ )\ z$ ▶ из начального нетерминала S.

Таким образом, грамматика $\boldsymbol{G}_{a2}$ принадлежит к классу **LL(1)**-грамматик.

Обратимся теперь к грамматике $\boldsymbol{G}_{a1}$. Множества выбора правил 1 и 2 для нетерминала S одинаковы (а следовательно, пересекаются). Одинаковы также множества выбора правил 3 и 4 для нетерминала T. Поэтому попытка восстановления дерева разбора правильной цепочки (например: $(\ x\ +\ y\ )\ *\ z$ ▶) может завести нас в тупик вследствие неверного принятия решения уже на первом шаге, если вместо подстановки по правилу 2

$$S \rightarrow T$$

будет выбрана подстановка по правилу 1

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

И та и другая подстановки в данный момент применимы, поскольку множества выбора и правила 1, и правила 2 содержат терминал $($. Если будет выбрано правило 1, то впоследствии никаким способом уже нельзя будет вывести цепочку $(\ x + y\ )\ *\ z$ ▶ из цепочки $S + T$.

Возможность принятия неверного решения при использовании грамматики $\boldsymbol{G}_{a1}$ возникает каждый раз, когда на текущем уровне восстанавливаемого дерева самым левым оказывается либо нетерминал S, либо нетерминал T.

Грамматика $\boldsymbol{G}_{a1}$ не относится к классу **LL(1)**-грамматик. Это является следствием леворекурсивности нетерминалов S и T. Любая левая рекурсия (в том числе – косвенная) влечет за собой пересечение множеств выбора правил, имеющих в левой части леворекурсивный нетерминал. Поясним это на простом примере прямой левой рекурсии. Пусть в системе порождающих правил грамматики есть такие правила для нетерминала N:

$$N : N\beta$$

$$N : \gamma\ .$$

Заметим, что первое правило – причина левой рекурсии, а из правой части второго правила должна быть выводима цепочка терминалов или пустая цепочка (если это не так или второго правила просто нет, то нетерминал N бесплоден и должен быть удален из грамматики со всеми своими правилами).

Каково бы ни было множество выбора второго правила, оно целиком содержится и в множестве выбора первого правила, поскольку

$$M_{\text{выб}}(N\beta) = M_{\text{пр}}(N\beta),$$

а множество предшественников цепочки $N\beta$ по определению включает в себя множество предшественников нетерминала N, которое содержит, в свою очередь, множество предшественников цепочки γ. Следовательно, множества выбора этих правил пересекаются по множеству предшественников цепочки γ.

Итак, любая леворекурсивная грамматика не принадлежит классу **LL(1)**-грамматик. Левую рекурсию всегда можно преобразовать в правую или в общую, как было показано в разд. 1.9.4. Однако не только свойство левой рекурсии может быть причиной непригодности грамматики для построения нисходящего синтаксического акцептора. К сожалению, в общем случае задача нахождения **LL(1)**-грамматики, эквивалентной заданной грамматике, алгоритмически неразрешима.

Алгоритм нисходящего восстановления дерева грамматического разбора, сформулированный в разд. 2.1, в принципе может быть использован с любой **LL(1)**-грамматикой, но применение его на практике потребует уточнения ряда деталей, в том числе – способа выполнения третьего шага, т. е. способа поиска правила для замены самого левого нетерминала цепочкой из левой части. Такая детализация может привести к радикальному изменению внешнего вида алгоритма при сохранении его сути.

Реализация общего алгоритма для конкретной грамматики обычно сводится к построению специального алгоритма, определяемого совокупностью порождающих правил, или к преобразованию грамматики в управляющую таблицу конечного автомата. Методы построения специальных алгоритмов или управляющих таблиц по грамматике легко формализуются. Следовательно, если заданная грамматика принадлежит классу **LL(1)**-грамматик, то построение нисходящего синтаксического акцептора предложений порождаемого ею языка может быть автоматизировано.

Рассмотрим несколько вариантов реализации общего алгоритма и методы соответствующего преобразования грамматики.

2.4. Процедурная реализация рекурсивного спуска

LL(1)-грамматика может быть легко преобразована в программу синтаксического акцептора, реализующую так называемый рекурсивный спуск. Такая реализация нисходящего восстановления дерева разбора называется процедурной в отличие от автоматных, которые будут рассмотрены далее.

Рекурсивный спуск может быть реализован только на таком языке программирования, который допускает рекурсивный вызов функций, что в сочетании с нисходящим существом процесса восстановления дерева разбора нашло отражение в названии.

Для иллюстрации способа построения такого синтаксического акцептора на примере грамматики $\boldsymbol{G}_{a2}$ будем использовать С-подобный язык программирования.

Будем считать, что имеется глобальная переменная *CurrentLexem*, предназначенная для хранения очередного терминального символа входной цепочки. Имеется также функция чтения очередного символа из этой цепочки *GetLexem()*, по существу – лексический анализатор.

В операциях сравнения текущей лексемы для элементов множеств выбора правил будем использовать обозначение #имя_терминала#, например: #ident#. Такой конструкции в С-подобных языках нет. Все остальные операторы, используемые в приводимых функциях, полностью соответствуют правилам синтаксиса языка С.

Собственно рекурсивный синтаксический акцептор – это функция *RecurciveDescent()*, возвращающая логическое значение (*TRUE*, если входное предложение правильно, и *FALSE* – в противном случае):

```
bool RecurciveDescent() {
    CurrentLexem = GetLexem();  // прочитать первый терминал
    if ( S() == FALSE )             // вызвать функцию S() для проверки
                                 // всего предложения до EOF (End Of File)
        return FALSE;              // останов по ошибке, если
                                    // предложение неверно
    if ( CurrentLexem != #EOF# )  // после правильного предложения не
                                 // должно быть ничего, кроме конца файла
        return FALSE;              // если это не так – останов по ошибке
    return TRUE;                    // предложение правильное, останов
}
```

Далее каждому нетерминалу грамматики ставится в соответствие функция, выполняющая действия по проверке правильности терминальных цепочек, выводимых из этого нетерминала, и определяемые совокупностью правил для него и текущим (очередным) терминалом.

Напишем эти функции, руководствуясь правилами грамматики для каждого нетерминала и знанием множеств выбора правил. Операции над терминалами будем записывать по правилам языка C/C++ для данных типа *char*, однако следует помнить, что при реализации терминальные символы представляются лексемами, т. е. структурами (см. ч. I, гл. 1).

Функция для нетерминала *S*:

```
bool S() {
   if ( U() == FALSE )          // вызываем функцию разбора
            return FALSE;       // нетерминала U
                                // и останавливаемся, если она
                                // остановилась по ошибке
   if ( R() == FALSE )          // аналогично для нетерминала R
            return FALSE;
   return TRUE;                 // возврат
}
```

Заметим, что в грамматике существует единственное правило с нетерминалом *S* в левой части и правая часть этого правила *S* : *U R* содержит только нетерминальные символы. Соответственно проверка того, что начало некоторой цепочки выводится из нетерминала *S*, сводится к вызову функции *U()*, проверке результата ее работы, затем к вызову функции *R()* и проверке возвращенного ею результата. Только в том случае, если и функция *U()*, и функция *R()* не обнаружили ошибок, функция *S()* возвращает значение *TRUE*. Заметим также, что в приведенном выше тексте функции *S()* нет проверки принадлежности входного символа к множеству выбора правила *S* : *U R*. Такая проверка могла бы быть предусмотрена вставкой оператора:

```
if ( (CurrentLexem != #ident#) && (CurrentLexem != #const#)
     && (CurrentLexem != #(#) )
        return FALSE;
```

перед первым оператором функции *S()*. Однако, как будет показано далее, эта проверка фактически нужна только в функции *V()*, где будет осуществляться выбор одного из правил и куда акцептор обязательно

попадет после вызова функции *U()* из функции *S()*. Заметим, что вставка этой проверки в функцию *S()*, с одной стороны, позволяет раньше (в последовательности вызовов функций) выявить возможную ошибку в предложении, но, с другой стороны, замедляет работу акцептора, поскольку производится при каждом вызове функции *S()* и дублирует действия, необходимые для выбора правила в функции *V()*.

Для нетерминала *U*:

```
bool U() {
    if ( V() == FALSE )         // вызываем функцию разбора
        return FALSE;           // нетерминала V и
                                // останавливаемся, если она
                                // остановилась по ошибке
    return W();                 // возвращаем то, что вернет функция W()
}
```

Функция *U()* очень похожа на функцию *S()*, поскольку для нетерминала *U*, как и для *S*, в грамматике есть единственное правило *U* : *V W*, правая часть которого содержит только нетерминалы. Однако здесь мы использовали несколько другую, более короткую и эффективную технику обработки результата вызова функции *W()*, соответствующей последнему нетерминалу в правой части правила.

Для нетерминала *R*:

```
bool R() {
    if ( CurrentLexem == #+# ) {    // очередной терминал
                                    // входит в множество
                                    // выбора правила R : +S
        CurrentLexem = GetLexem(); // прочитаем следующий
        return S();                 // и вызовем функцию S()
                                    // с возвратом ее результата
    }
    if ( ( CurrentLexem == #)# ) ||    // очередной терминал
        ( CurrentLexem == #EOF# ) )   // входит в множество
                                    // выбора правила R : ε
            return TRUE;            // применяем это правило, т. е. не
                                    // делаем ничего и возвращаемся
    return FALSE;                   // очередной терминал
                                    // не входит ни в одно
                                    // множество выбора,
                                    // поэтому останов по ошибке
}
```

Для нетерминала *W*:

```
bool W() {
    if ( CurrentLexem == #*# ) {        // очередной терминал
                                        // входит в множество
                                        // выбора правила W : * U
        CurrentLexem = GetLexem();      // прочитаем следующий
        return U();                     // и вызовем функцию U()
                                        // с возвратом ее результата
    }
    if ( ( CurrentLexem == #+# ) ||     // очередной терминал входит
        ( CurrentLexem == #)# ) ||      // в множество выбора
        ( CurrentLexem == #EOF# ) )     // правила W : ε, применяем это
            return TRUE;                // правило, т. е. не
                                        // делаем ничего и возвращаемся
    return FALSE;                       // очередной терминал не входит
                                        // ни в одно множество выбора,
                                        // поэтому останов по ошибке
}
```

И, наконец, для нетерминала *V*:

```
bool V() {
    switch (CurrentLexem) {             // здесь использована другая
                                        // техника проверки
                                        // очередного терминала
    case #(# :                          // в этом случае должно
                                        // применяться правило V : ( S )
        CurrentLexem = GetLexem();      // прочитаем следующий терминал
        if ( S() == FALSE )             // вызовем S(), и если она
                                        // обнаружит ошибку,
            return FALSE;               // остановимся
                                        // если ошибки не было,
        if ( CurrentLexem != #)# )      // то очередным терминалом
                                        // должна быть скобка )
            return FALSE;               // если нет – остановимся
                                        // если да, действуем так же, как
                                        //в случае правил V : i и V : c
    case #ident# :                      // это правило V : i
    case #const# :                      // это правило V : c
        CurrentLexem = GetLexem();      // прочитаем следующий терминал
        break;                          // и на выход
```

```
    default :                          // очередной терминал
                                       // не входит ни в одно
                                       // множество выбора
      return FALSE;                    // остановимся по ошибке
    }
    return TRUE;                       // возвратимся для
                                       // продолжения разбора
}
```

Проиллюстрируем процесс восстановления дерева разбора этим акцептором на примере правильного предложения *(x + y) * z* ▶.

В приведенной таблице первая строка содержит номер такта, вторая строка – очередной символ входного предложения, третья – название активной (вызванной на данном такте) функции, остальные строки – названия вызванных к данному моменту функций (по существу – это условное представление стека возвратов).

Для сокращения объема истории восстановления дерева символом *Z* обозначена функция *RecursiveDescent()*.

История восстановления дерева

Такт	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Вход	(	(	(	(	x	x	x	+	+	+	+	+	y	y	y
Функция	Z	S	U	V	S	U	V	U	W	U	S	R	S	U	V
Стек		Z	S	U	V	S	U	S	U	S	V	S	R	S	U
			Z	S	U	V	S	V	S	V	U	V	S	R	S
				Z	S	U	V	U	V	U	S	U	V	S	R
					Z	S	U	S	U	S	Z	S	U	V	S
						Z	S	Z	S	Z		Z	S	U	V
							Z		Z				Z	S	U
														Z	S
															Z

Такт	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Вход	)	)	)	)	)	)	)	*	*	z	z	▶	▶	▶	▶	▶
Функция	U	W	U	S	R	S	V	U	W	U	V	U	W	U	S	Z
Стек	S	U	S	R	S	V	U	S	U	W	U	W	U	S	Z	
	R	S	R	S	V	U	S	Z	S	U	W	U	S	Z		
	S	R	S	V	U	S	Z		Z	S	U	S	Z			
	V	S	V	U	S	Z				Z	S	Z				
	U	V	U	S	Z						Z					
	S	U	S	Z												
	Z	S	Z													
		Z														

Любое правильное предложение языка, порождаемого грамматикой $\boldsymbol{G}_{a2}$, будет принято разработанным нами акцептором, любое неправильное – отвергнуто.

Заметим, что при процедурной реализации нисходящего синтаксического акцепта уровни восстанавливаемого дерева разбора как цепочки символов в явном виде нигде не формируются и не хранятся. Неявно же эти цепочки для каждого шага работы акцептора полностью определены текущим содержанием стека возвратов. В нем хранятся точки возврата, из которых должно продолжаться выполнение функций акцептора, с каждой из таких точек можно сопоставить ожидаемые терминальные или нетерминальные символы.

Для рассматриваемого примера, начиная с такта 4 и вплоть до такта 22, в течение времени разбора цепочки, выводимой из нетерминала *S*, в стеке находилась точка возврата в функцию *V()*, определяющая ожидание закрывающей скобки *)* в соответствии с правилом *V* : *(S)*. С некоторыми точками возврата можно сопоставить только ожидание пустой цепочки ε. Такие точки соответствуют, например, возврату из функции *W()* в функцию *U()* (см. такт 9).

Преобразование **LL(1)**-грамматики в процедурную реализацию рекурсивного спуска может быть выполнено как вручную, так и автоматизированно. Такое преобразование, в частности, может выполнять учебный пакет автоматизации проектирования трансляторов «Вебтранслаб».

Кроме процедурной могут использоваться и автоматные реализации нисходящего восстановления дерева грамматического разбора. Известно несколько вариантов автоматной реализации. Любая автоматная реализация предполагает наличие стековой (магазинной) памяти, функционально аналогичной стеку возвратов в процедурной реализации.

Мы рассмотрим способы преобразования **LL(1)**-грамматики:

– в конечный автомат с несколькими состояниями (в определенном смысле являющийся эквивалентом процедурной реализации рекурсивного спуска);

– конечный автомат с одним состоянием.

2.5. Автоматная реализация рекурсивного спуска Автомат с несколькими состояниями

Функционирование конечного автомата со стековой (магазинной) памятью и несколькими состояниями определяется управляющей таблицей. Как обычно, предполагается, что автомат при запуске оказывается в особом начальном состоянии, на каждом такте по входному символу и текущему состоянию определяет и выполняет операции над входным потоком символов, стековой памятью и собственным состоянием.

Для выявления характера этих операций и структуры управляющей таблицы рассмотрим еще раз, но несколько с другой точки зрения, существо процесса нисходящего синтаксического акцепта.

Процесс восстановления дерева методом рекурсивного спуска можно интерпретировать как управляемое входной цепочкой движение по порождающим правилам грамматики. Для такого рассмотрения удобно считать, что каждое правило завершается обозначением пустой цепочки ε. В этом случае обработка правил с пустой правой частью ничем не будет отличаться от обработки остальных правил. Управляющая таблица автомата при этом будет обладать некоторой избыточностью, впоследствии легко удаляемой.

Начиная с нетерминала S в правой части добавочного правила $Z : S$ ▶ движение осуществляется следующим образом:

1. По правым частям правил посимвольно слева направо.

Обработка любого нетерминала состоит в переключении на первое из всех правил для этого нетерминала. Более точно – на нетерминал из левой части первого правила с сохранением в стеке точки возврата в текущую правую часть правила. До переключения может осуществляться проверка принадлежности текущего входного символа к множеству предшественников данного нетерминала (однако отметим, что такая проверка обычно только замедляет процесс восстановления дерева и поэтому реализуется не каждым построителем автомата по грамматике).

Обработка терминального символа (из правила) состоит в проверке его совпадения с текущим входным символом и при положительном результате проверки завершается чтением следующего символа предложения. Отрицательный результат проверки есть основание для останова по обнаружению ошибки.

Обработка пустой цепочки ε, завершающей каждое правило, состоит в возврате по верхушке стека. Возврат в добавочное правило для обработки псевдотерминала ▶ рассматривается как успешное окончание процесса восстановления дерева при условии, что текущим входным символом является признак конца входной цепочки ▶.

2. По левым частям правил сверху вниз.

При этом движении используются только правила, имеющие в левой части один и тот же нетерминал. Для каждого правила прежде всего проверяется, содержит ли его множество выбора текущий входной символ.

При отрицательном результате проверки осуществляется переход к левой части следующего правила, тем самым обеспечивается поиск подходящего правила для замены нетерминала.

При положительном результате проверки выполняется переключение на обработку первого символа из правой части данного правила, т. е. подстановка правой части вместо нетерминала из левой части.

Если такого правила нет вообще (ни одно из множеств выбора правил для данного нетерминала не содержит текущего входного символа), то восстановить дерево невозможно, следует остановиться по обнаружении ошибки во входном предложении.

Таким образом, мы выяснили, что каждому символу каждого правила грамматики (в том числе нетерминалам, находящимся в левых частях правил, и обозначениям пустой цепочки, замыкающим каждое правило) должно быть поставлено в соответствие в точности такое же одно состояние автомата. С каждым состоянием должно быть связано множество выбора и два адреса перехода (один используется при положительном результате проверки принадлежности текущего входного символа множеству выбора, второй – при отрицательном). Под адресом перехода понимается номер состояния.

Далее будет показано, что при соблюдении определенных правил нумерации состояний и введении операции управления остановом по ошибке можно обойтись только одним адресом перехода.

С каждым состоянием должны быть также связаны операции управления стековой памятью (занесение адреса возврата, снятие адреса с верхушки стека и переключение в состояние возврата) и операция управления чтением следующего входного символа. Все операции управления могут задаваться булевскими значениями *true* / *false*, которые далее будем называть флажками. Введем обозначения для флажков управления операциями:

– флажок **a** управляет чтением следующего входного символа;

– флажок **s** управляет занесением адреса точки возврата (вычисляемого как номер текущего состояния плюс 1) в стек;

– флажок **r** обеспечивает переключение автомата в состояние, номер которого снимается с верхушки стека возвратов;

– флажок **e** запрещает останов по ошибке в случае, когда состояние соответствует нетерминалу из левой части и есть еще хотя бы одно правило для такого нетерминала.

С учетом сказанного каждая клетка управляющей таблицы автомата должна содержать следующие поля:

Множество выбора состояния	Флажки				Адрес перехода
	a	**s**	**r**	**e**	

Из соображений простоты реализации ясно, что вместо множеств выбора как массивов терминальных символов в каждой клетке управляющей таблицы следует хранить указатели на эти массивы, размещаемые вне управляющей таблицы.

При практических применениях автоматной реализации рекурсивного спуска в состав клетки управляющей таблицы обычно включаются дополнительное поле, указывающее на действие, сопровождающее синтаксический акцепт (например, для нейтрализации ошибок) или относящееся к задачам семантического анализа и формирования объектного кода.

Для построения управляющей таблицы автомата по заданной **LL(1)**-грамматике (в качестве иллюстрации будем использовать грамматику $\boldsymbol{G}_{a2}$) необходимо выполнить следующую процедуру.

1. Определение и нумерация множества состояний. Для этого перенумеруем все символы системы порождающих правил грамматики, исключая символ Z в левой части добавочного правила, но включая обозначения пустых цепочек так, чтобы:

– символ S в добавочном правиле $Z : S$▶ получил номер 0;

– символы, следующие друг за другом в правых частях правил, имели последовательно возрастающие номера; при соблюдении этого требования адрес возврата, помещаемый в стек при обработке нетерминального символа в правой части правила, вычисляется как номер текущего состояния плюс единица, и для его хранения не требуется отдельного поля в клетке управляющей таблицы;

– одинаковые нетерминалы в левых частях правил имели последовательно возрастающие номера; при соблюдении этого требования легко обеспечивается перебор правил при обработке нетерминалов из левых частей правил.

Приведем результаты выполнения шага 1 для грамматики $\boldsymbol{G}_{a2}$, на примере которой будем иллюстрировать формирование автомата:

Грамматика $\boldsymbol{G}_{a2}$	
0	$Z : S_0 \blacktriangleright_1$
1	$S_2 : U_{11} R_{12}\, \varepsilon_{13}$
2	$R_3 : +_{14} S_{15}\, \varepsilon_{16}$
3	$R_4 : \varepsilon_{17}$
4	$U_5 : V_{18} W_{19}\, \varepsilon_{20}$
5	$W_6 : *_{21} U_{22}\, \varepsilon_{23}$
6	$W_7 : \varepsilon_{24}$
7	$V_8 : (_{25} S_{26})_{27}\, \varepsilon_{28}$
8	$V_9 : i_{29}\, \varepsilon_{30}$
9	$V_{10} : c_{31}\, \varepsilon_{32}$

2. Сформируем множества выбора для каждого состояния управляющей таблицы. Способ образования множества выбора состояния зависит от того, какому символу и из какой части правила оно поставлено в соответствие.

Если состояние соответствует нетерминалу N из левой части правила $N : \alpha$, то его множество выбора есть множество выбора данного правила:

– множество предшественников цепочки α, если она содержит хотя бы один терминал или неаннулируемый нетерминал;

– множество последователей N, если цепочка α пуста;

– объединение этих двух множеств, если цепочка α не пуста, но состоит только из аннулируемых нетерминалов).

Если состояние соответствует нетерминальному символу из правой части правила, то его множество выбора есть объединение множеств выбора всех правил грамматики для этого нетерминала.

Если состояние соответствует терминальному символу (такие символы могут появляться только в правых частях правил), то его множество выбора содержит только этот терминальный символ.

Для состояний, соответствующих обозначениям пустой цепочки, множества выбора есть множество последователей нетерминала из левой части данного правила.

3. Формирование значений флажков управления операциями.

Флажок **a** устанавливается (имеет значение *true*) только в состояниях, соответствующих терминальным символам (которые, естественно, могут находиться только в правых частях правил).

Флажок **s** устанавливается в состояниях, соответствующих нетерминальным символам, находящимся в правых частях правил.

Флажок **r** устанавливается в состояниях, соответствующих обозначениям пустой цепочки символов в конце правой части каждого правила.

Флажок **e** устанавливается в состояниях, соответствующих нетерминальным символам, находящимся в левой части правил, за исключением последнего правила для каждого нетерминала.

4. Адрес перехода образуется следующим образом.

В клетках состояний, соответствующих нетерминалам из левых частей правил, адрес перехода есть номер состояния, соответствующего первому символу правой части данного правила.

В клетках состояний, соответствующих символам из правых частей правил, адрес перехода формируется только в том случае, если для этого состояния не установлен флажок **r** (в том случае если флажок **r** установлен, переход осуществляется по адресу, снимаемому со стека возвратов). Если флажок в данном состоянии **r** установлен, в поле адреса перехода будем заносить значение 0. Особое значение адреса перехода (Stop) формируется для состояния 1. Переход по этому адресу означает останов автомата по окончании восстановления дерева разбора правильного предложения при условии, что стек пуст. В противном случае (стек не пуст) операция Stop означает останов по ошибке.

Для состояний, соответствующих терминальным символам, адрес перехода есть номер состояния, соответствующего следующему символу правила (при используемом способе нумерации состояний вычисляется как номер текущего состояния плюс единица).

Для состояний, соответствующих нетерминальным символам в правых частях правил, адрес перехода есть номер состояния, приписанного первому такому (одноименному) нетерминалу, но находящемуся в левой части правил.

Приведем результаты применения процедуры преобразования грамматики в управляющую таблицу автомата для грамматики $\boldsymbol{G}_{a2}$ (для флажков управления значению *true* сопоставлено 1, значению *false* – пустая клетка).

Управляющая таблица автомата

Номер состояния	Множество выбора	Флажки				Переход
		a	**s**	**r**	**e**	
0	(i c		1			2
1	▶					Stop
2	(i c					11
3	+				1	14
4	) ▶					17
5	(i c					18
6	*				1	21
7	+) ▶					24
8	(				1	25
9	i				1	29
10	c					31
11	(i c		1			5
12	+) ▶		1			3
13	i c* +() ▶			1		0
14	+	1				15
15	(i c		1			2
16	i c* +() ▶			1		0
17	i c* +() ▶			1		0
18	(i c		1			8
19	* +) ▶		1			6
20	i c* +() ▶			1		0
21	*	1				22
22	(i c		1			5
23	i c* +() ▶			1		0
24	i c* +() ▶			1		0
25	(	1				26
26	(i c		1			2
27	)	1				28
28	i c* +() ▶			1		0
29	i	1				30
30	i c* +() ▶			1		0
31	c	1				32
32	i c* +() ▶			1		0

Такой автомат будем называть неоптимизированным, в силу того что добавление обозначений пустой цепочки в конец правой части правил 1, 2, 4, 5, 7, 8 и 9 в управляющей таблице привело к образова-

нию состояний, зарезервированных для возможного включения действий в грамматику (см. раздел 4). Эти состояния с номерами 13, 16, 20, 23, 28, 30 и 32 являются избыточными при решении задачи чистого синтаксического акцепта, т. е. без учета задач нейтрализации ошибок, семантического анализа и генерации кода.

Если для этих состояний при расширении синтаксического акцептора до анализатора так и не будут определены действия, то они легко могут быть удалены из таблицы. Перед удалением каждого избыточного состояния следует установить флажок **r** в предыдущем состоянии, но только в том случае, если в нем не установлен флажок **s**. Если же в состоянии, номер которого на 1 меньше, чем номер удаляемого состояния, ранее был установлен флажок **s**, то нужно сбросить его и не устанавливать флажок **r**.

Состояния 17 и 24 также могут быть удалены из таблицы, но способ их удаления другой. В эти состояния автомат может попасть только из состояний 4 и 7 соответственно. Поэтому перед удалением состояния 17 флажок **r** должен быть установлен (перенесен из состояния 17) в состояние 4, а перед удалением состояния 24 – в состояние 7. Отметим, что пакет «Вебтранслаб» удаляет избыточные состояния, соответствующие обозначениям ε, добавленным в концы непустых правил, и не удаляет состояния 17 и 24.

После удаления избыточных состояний (и соответственно перенумерации оставшихся состояний) необходимо должным образом модифицировать адреса переходов в оставшихся состояниях.

Приведем результат удаления избыточных состояний для рассматриваемого автомата (см. таблицу). Такой автомат будем далее называть оптимизированным.

Отметим, что удаление избыточных состояний целесообразно не столько с точки зрения минимизации памяти, занимаемой управляющей таблицей, сколько в целях оптимизации затрат времени на восстановление дерева разбора. Каждое из состояний будет многократно использоваться в процессе работы автомата, на что будет тратиться время компьютера.

Выявление того, какие состояния действительно являются избыточными, следует выполнять только тогда, когда в синтаксический акцептор добавлены действия прочих этапов трансляции, а именно нейтрализации ошибок, преобразования в постфиксную форму записи, семантического анализа и генерации объектного кода.

Оптимизированная управляющая таблица автомата

Номер состояния	Множество выбора	Флажки				Переход
		a	**s**	**r**	**e**	
0	*(i c*		1			2
1	▶					Stop
2	*(i c*					11
3	+				1	13
4	*)* ▶			1		0
5	*(i c*					15
6	*				1	17
7	+ *)* ▶			1		0
8	*(*				1	19
9	*i*				1	22
10	*c*					23
11	*(i c*		1			5
12	+ *)* ▶					3
13	+	1				14
14	*(i c*					2
15	*(i c*		1			8
16	* + *)* ▶					6
17	*	1				18
18	*(i c*					5
19	*(*	1				20
20	*(i c*		1			2
21	*)*	1		1		0
22	*i*	1		1		0
23	*c*	1		1		0

Программная модель автомата с несколькими состояниями и стековой памятью должна реализовывать следующий алгоритм.

1. Запуск и инициализация. Очистить стек, прочитать первый символ входной цепочки, установить в качестве текущего состояние 0 и перейти к шагу **2**.

2. Проверить, принадлежит ли очередной символ множеству выбора текущего состояния. Если да, то перейти к шагу **3**, иначе – к шагу **6**.

3. Если в клетке текущего состояния установлен флажок **a**, то прочитать следующий символ входной цепочки.

4. Если в клетке текущего состояния установлен флажок **s**, то поместить в стек номер текущего состояния, увеличенный на единицу.

5. Определение номера следующего состояния. Для этого прежде всего, проверяется значение флажка **r** текущего состояния.

5.1. Если флажок **r** установлен, то:

5.1.1) если стек не пуст – снять с верхушки стека номер состояния, установить его в качестве текущего и перейти к шагу **2**;

5.1.2) если стек пуст – перейти к шагу **7**.

5.2. Если флажок **r** не установлен, то:

5.2.1) если текущим является состояние **1**:

5.2.1.1) если стек пуст, то перейти к шагу **8**;

5.2.1.2) если стек не пуст, перейти к шагу **7**;

5.2.2) если текущим является любое другое состояние, то взять номер состояния из поля адреса перехода клетки текущего состояния. Установить в качестве текущего состояние с этим номером и вернуться к шагу **2**.

6. Если в клетке текущего состояния установлен флажок **e**, то установить в качестве текущего следующее состояние (его номер вычисляется, как номер текущего состояния плюс единица) и вернуться к шагу **2**, иначе – перейти к шагу **7**.

7. Останов по ошибке.

8. Останов по окончании разбора правильного предложения.

Приведем примеры историй работы неоптимизированного и оптимизированного автоматов по восстановлению дерева разбора той же входной цепочки символов *(x + y) * z* ▶, которая рассматривалась для процедурной реализации рекурсивного спуска. Сравнение характеристик различных реализаций нисходящих методов с использованием историй работы будет проведено в раздел 2.7.

Неоптимизированный автомат реализует для этой цепочки такую историю:

Такт	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Вход	*(*	*(*	*(*	*(*	*(*	*(*	*(*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	+	+	+	+	+	+	+
Сост	0	2	11	5	18	8	25	26	2	11	5	18	8	9	29	30	19	6	7	24	20	12
Стек		1	1	12	12	19	19	19	27	27	12	12	19	19	19	19	12	20	20	20	12	27
				1	1	12	12	12	19	19	27	27	12	12	12	12	27	12	12	12	27	19
						1	1	1	12	12	19	19	27	27	27	27	19	27	27	27	19	12
									1	1	12	12	19	19	19	19	12	19	19	19	12	1
											1	1	12	12	12	12	1	12	12	12	1	
													1	1	1	1		1	1	1		

22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
+	+	*y*	*y*	*y*	*y*	*y*	*y*	*y*	*y*	)	)	)	)	)	)	)	)	)	)	)	)	)	)
3	14	15	2	11	5	18	8	9	29	30	19	6	7	24	20	12	3	4	17	13	16	13	27
13	13	13	16	16	12	12	19	19	19	19	12	20	20	20	12	16	13	13	13	16	13	27	19
27	27	27	13	13	16	16	12	12	12	12	16	12	12	12	16	13	16	16	16	13	27	19	12
19	19	19	27	27	13	13	16	16	16	16	13	16	16	16	13	27	13	13	13	27	19	12	1
12	12	12	19	19	27	27	13	13	13	13	27	13	13	13	27	19	27	27	27	19	12	1	
1	1	1	12	12	19	19	27	27	27	27	19	27	27	27	19	12	19	19	19	12	1		
			1	1	12	12	19	19	19	19	12	19	19	19	12	1	12	12	12	1			
					1	1	12	12	12	12	1	12	12	12	1		1	1	1				
							1	1	1	1		1	1	1									

46	47	48	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70
*	*	*	*	*z*	*z*	*z*	*z*	*z*	*z*	▶	▶	▶	▶	▶	▶	▶	▶	▶	▶	▶	▶	▶	▶
28	19	6	21	22	5	18	8	9	29	30	19	6	7	24	20	23	20	12	3	4	17	13	1
19	12	20	20	20	23	23	19	19	19	19	23	20	20	20	23	20	12	1	13	13	13	1	
12	1	12	12	12	20	20	23	23	23	23	20	23	23	23	20	12	1		1	1	1		
1		1	1	1	12	12	20	20	20	20	12	20	20	20	12	1							
					1	1	12	12	12	12	1	12	12	12	1								
							1	1	1	1		1	1	1									

Оптимизированный автомат реализует такую историю:

Такт	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Вход	(	(	(	(	(	(	(	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	+	+	+	+	+
Сост.	0	2	11	5	15	8	19	20	2	11	5	15	8	9	22	16	6	7	12	3
Стек		1	1	12	12	16	16	16	21	21	12	12	16	16	16	12	12	12	21	21
				1	1	12	12	12	16	16	21	21	12	12	12	21	21	21	16	16
						1	1	1	12	12	16	16	21	21	21	16	16	16	12	12
									1	1	12	12	16	16	16	12	12	12	1	1
											1	1	12	12	12	1	1	1		
													1	1	1					

20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41
+	*y*	*y*	*y*	*y*	*y*	*y*	*y*	*y*	)	)	)	)	)	)	)	*	*	*	*z*	*z*	*Z*
13	14	2	11	5	15	8	9	22	16	6	7	12	3	4	21	16	6	17	18	5	15
21	21	21	21	12	12	16	16	16	12	12	12	21	21	21	16	12	12	12	12	12	12
16	16	16	16	21	21	12	12	12	21	21	21	16	16	16	12	1	1	1	1	1	1
12	12	12	12	16	16	21	21	21	16	16	16	12	12	12	1						
1	1	1	1	12	12	16	16	16	12	12	12	1	1	1							
				1	1	12	12	12	1	1	1										
						1	1	1													

42	43	44	45	46	47	48	49	50	51
z	*z*	*z*	▶	▶	▶	▶	▶	▶	▶
8	9	22	16	6	7	12	3	4	1
16	16	16	12	12	12	1	1	1	
12	12	12	1	1	1				
1	1	1							

Результаты работы этих автоматов одинаковы, но затраты времени, измеряемые в количестве тактов, различны. Для приводимого короткого примера разница не очень велика, но при синтаксическом анализе больших текстов программ (десятки и сотни тысяч слов, а это типичный случай после включения всех заголовочных файлов) она может оказаться существенной.

2.6. Нисходящий автомат с одним состоянием

LL(1)-грамматику легко можно преобразовать в конечный автомат с единственным состоянием и стековой памятью, управляемый:

– текущим входным символом;

– символом, находящимся на верхушке стека.

Поведение такого автомата определяется управляющей таблицей, столбцы которой соответствуют входным символам, строки – символам, которые могут находиться в стеке, а в клетках указана некоторая последовательность операций над стеком, входным потоком и состоянием автомата.

Введем условные обозначения для операций автомата.

Обычные для стековой памяти операции обозначим так:

$\downarrow X$ – занесение символа X в стек;

↑ – снятие одного символа с верхушки стека. Заметим, что попытка выполнения этой операции при пустом стеке должна приводить к останову автомата по обнаружению ошибки во входной цепочке.

Над входным потоком определена единственная операция, которую будем обозначать так:

→ – чтение следующего символа из входной цепочки.

Для управления состоянием будем использовать единственный знак операции Stop, предназначенный для останова автомата по успешному окончанию восстановления дерева разбора. Для обозначения операции останова по обнаружению ошибок во входной цепочке будем использовать обычное соглашение: клетка управляющей таблицы пуста.

Теперь опишем процедуру преобразования системы порождающих правил грамматики в управляющую таблицу автомата.

1. Построить заготовку таблицы, имеющую ровно столько столбцов, сколько символов есть в терминальном алфавите грамматики (включая псевдотерминал ▶), и столько строк, сколько символов есть в нетерминальном алфавите (далее мы увидим, что в процессе преобразования возможно появление дополнительных строк). Озаглавим столбцы терминалами грамматики (порядок следования столбцов не имеет значения), строки – нетерминалами (опять же в произвольном порядке).

2. Имея в виду, что автомат предназначен для разбора цепочки, выводимой из правой части добавочного правила грамматики Z : S ▶, будем считать, что перед запуском в его стеке должна оказаться правая часть этого правила, причем самым нижним символом в стеке будет псевдотерминал ▶, а верхним соответственно – начальный нетерминал S. Поскольку рано или поздно псевдотерминал ▶ может стать верхним символом в стеке, добавим к таблице строку и озаглавим ее этим символом.

3. Для каждой строки таблицы, начиная с первой, сформируем знаки операций в ее клетках следующим образом:

3.1) если строка озаглавлена нетерминальным символом (пусть это будет символ N), то последовательно в произвольном порядке переберем все правила грамматики, имеющие этот нетерминал в левой части;

3.1.1) если очередное правило имеет вид N : $M\ \alpha$, где M – нетерминальный символ, а α – цепочка символов $s_1\ s_2\ ...\ s_k$ (возможно, пустая), то во все клетки данной строки, находящиеся на пересечении со столбцами, помеченными терминалами из множества выбора данного правила, занесем такую последовательность знаков операций:

$$\uparrow\downarrow s_k \ldots \downarrow s_2 \downarrow s_1 \downarrow M$$

Если среди символов $s_1\ s_2 \ldots s_k$ встречаются терминалы, то добавим к таблице новые строки, озаглавленные этими терминалами, но только при условии, что таких строк еще нет.

Основанием для формирования этой последовательности знаков операций является следующее рассуждение. Начало текущего остатка входной цепочки (если она является правильной в языке, порождаемом данной грамматикой), выводимое из N (символ N следует удалить из стека операцией $\uparrow$, поскольку правило для его замены уже определено), должно выводиться из правой части правила $N : M\ \alpha$, поскольку текущий входной терминал принадлежит множеству выбора этого правила. Первые символы этого начала должны выводиться из нетерминала M, поэтому символ M должен оказаться на верхушке стека (его нужно занести в стек последним). То что последует на входе автомата после цепочки, выводимой из M, должно выводиться из символа s_1, поэтому он должен оказаться в стеке непосредственно под символом M и т. д.

Для правильного функционирования автомата принципиально важна именно такая последовательность выполнения операций;

3.1.2) если очередное правило имеет вид $N : t\ \alpha$, где t – терминальный символ, а α – возможно, пустая цепочка символов $s_1\ s_2 \ldots s_k$, то в клетку, находящуюся на пересечении со столбцом, помеченным терминалом t (очевидно, что множество выбора данного правила содержит единственный символ t), занесем такую последовательность знаков операций:

$$\uparrow\downarrow s_k \ldots \downarrow s_2 \downarrow s_1 \rightarrow$$

Если среди символов $s_1\ s_2 \ldots s_k$ встречаются терминалы, то добавим к таблице новые строки, озаглавленные этими терминалами, но только при условии, что таких строк еще нет.

Эта последовательность знаков операций завершается чтением следующего входного символа вместо записи первого символа правой части правила в стек. Причина очевидна: терминал, с которого начинается цепочка правой части правила, совпадает с текущим входным символом. Если его заносить в стек, то только для того, чтобы удалить на следующем такте;

3.1.3) если очередное правило имеет вид $N : \varepsilon$, то во все клетки данной строки, находящиеся на пересечении со столбцами, помеченными терминалами из множества выбора данного правила, занесем

единственный знак операции ↑. Очевидно, что удаление символа *N* с верхушки стека без выполнения каких-либо других действий соответствует применению этого правила;

3.2) если текущая строка озаглавлена терминалом *t*, то в клетку, находящуюся на пересечении с одноименным столбцом (т. е. также озаглавленным терминалом *t*), занесем последовательность знаков операций ↑→. Терминал *t* мог быть занесен в стек при выполнении последовательности операций, сформированных согласно **3.1.1** и **3.1.2**. Если он появился на верхушке стека, то входным символом обязан быть именно этот терминал, иначе входная цепочка неверна. Последовательность знаков операций ↑→ обеспечивает переход к следующим символам из стека и из входной цепочки;

3.3) и, наконец, если текущая строка озаглавлена псевдотерминалом ▶, то в клетку, находящуюся на пересечении с одноименным столбцом, занесем знак операции **Stop**.

Применим эту процедуру к грамматике $\boldsymbol{G}_{a2}$ и получим такую управляющую таблицу:

	i	*c*	+	*	*(*	*)*	▶
S	↑↓ *R* ↓ *U*	↑↓ *R* ↓ *U*			↑↓ *R* ↓ *U*		
U	↑↓ *W* ↓ *V*	↑↓ *W* ↓ *V*			↑↓ *W* ↓ *V*		
R			↑↓ *S* →			↑	↑
W			↑	↑↓ *U* →		↑	↑
V	↑→	↑→			↑↓ *)* ↓ *S* →		
▶							**Stop**
)						↑→	

Приведем пример проверки правильности той же самой входной цепочки *(x + y) * z* ▶ этим автоматом:

Такт	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	19	20
Вход	*(*	*(*	*(*	*x*	*x*	*x*	+	+	*y*	*y*	*y*	*)*	*)*	*)*	*	*z*	*z*	▶	▶	▶
Стек	*S*	*U*	*V*	*S*	*U*	*V*	*W*	*R*	*S*	*U*	*V*	*W*	*R*	*)*	*W*	*U*	*V*	*W*	*R*	▶
	▶	*R*	*W*	*)*	*R*	*W*	*R*	*)*	*)*	*R*	*W*	*R*	*)*	*W*	*R*	*R*	*W*	*R*	▶	
		▶	*R*	*W*	*)*	*R*	*)*	*W*	*W*	*)*	*R*	*)*	*W*	*R*	▶	▶	*R*	▶		
			▶	*R*	*W*	*)*	*W*	*R*	*R*	*W*	*)*	*W*	*R*	▶			▶			
				▶	*R*	*W*	*R*	▶	▶	*R*	*W*	*R*	▶							
					▶	*R*	▶			▶	*R*	▶								
						▶					▶									

При запуске автомата (перед тактом 0) в стек заносятся символы ▶ и S, текущим входным символом устанавливается первый символ входной цепочки. Далее автомат, управляемый текущим входным символом и символом, находящимся на верхушке стека, реализует следующую историю работы, приводящую к останову на такте 20 по успешному восстановлению дерева грамматического разбора.

2.7. Оценки сравнительных характеристик различных реализаций нисходящего синтаксического акцепта

В настоящем разделе были рассмотрены теоретические основы и некоторые из известных нисходящих методов решения задачи синтаксического акцепта – проверки принадлежности произвольной цепочки терминальных символов (слов, выявляемых лексическим анализатором) к языку, порождаемому заданной формальной грамматикой. Синтаксический акцептор строится путем преобразования системы порождающих правил грамматики, которая должна обладать определенными необходимыми свойствами – относиться к классу **LL(k)**-грамматик при конечном значении **k**. Рассматривались методы преобразования грамматики в синтаксический акцептор для **k**, равного 1. Их нетрудно обобщить на случай **k**>1, однако особой практической ценности это не имеет.

Синтаксический акцептор, формирующий для любой входной цепочки логическое значение (истина/ложь) в качестве результата проверки правильности, является необходимой основой для построения синтаксического анализатора. В задачи синтаксического анализа кроме проверки правильности входят:

– формирование диагностических сообщений с максимально возможной точностью локализации места обнаружения синтаксической ошибки для неправильных входных цепочек. Заметим сразу, что место (позиция неправильного символа во входной цепочке), где ошибка обнаружена, и место, где ошибка должна быть исправлена, далеко не всегда совпадают;

– нейтрализация синтаксических ошибок, состоящая в организации поиска последующих действительно существующих ошибок путем попыток «исправления» входной цепочки. Заметим, что слово «исправления» взято в кавычки, потому что на самом деле исправить ошибочную входную цепочку синтаксический анализатор не в состоянии, он может только попытаться найти такой вариант ее модифика-

ции, чтобы продолжить поиск других возможных ошибок. Заметим также, что задача нейтрализация ошибок решается не во всех трансляторах;

– преобразование правильной входной цепочки символов в представление на некотором промежуточном языке, синтаксис которого должен быть максимально приближен к синтаксису выходного языка транслятора и поэтому может существенно отличаться от синтаксиса входного языка.

Синтаксический анализатор обычно реализуется путем расширения функциональности синтаксического акцептора, поэтому методы решения его задач в известной мере зависят от того, какой именно синтаксический акцептор был взят за основу. Эти методы рассматриваются в разделе 4, а здесь они были охарактеризованы только для того, чтобы отметить практически одинаковую пригодность любого метода нисходящего восстановления дерева разбора в качестве основы реализации синтаксического анализатора.

Применительно только к задачам проверки правильности цепочек рассмотренные нами варианты нисходящего акцепта можно сравнивать по скорости работы и по требуемому для реализации объему памяти.

1. Оценить затраты времени на синтаксический акцепт можно, например, по среднему количеству тактов на один входной символ для автоматных реализаций и по количеству вызовов функций (в определенном смысле эквивалентных тактам) для процедурной реализации. Сведем в таблицу имеющиеся данные для рассматривавшегося нами примера входной цепочки, имевшей длину 8 символов:

Способ реализации		**Кол-во тактов**	**Среднее кол-во**
1	Процедурная реализация	31	3.875
2	Автомат с несколькими состояниями	71	8.875
3	Оптимизированный автомат	52	6.5
4	Автомат с одним состоянием	21	2.625

Из таблицы, казалось бы, следует, что варианты реализации нисходящего акцептора делятся на две группы: быструю со средними затратами времени порядка 3…4 тактов на один символ и медленную с затратами времени порядка 7…9 тактов на символ. На самом деле эта разница объясняется тем, что процедурная реализация и автомат с одним состоянием при обработке каждого символа выполняют не одну, а несколько операций.

Такие автоматы называются расширенными в отличие от простых, выполняющих единственную операцию над стеком за один такт.

Реально же различие между простыми и расширенными автоматами по скорости работы оказывается практически несущественным именно потому, что общий объем выполняемых операций практически одинаков.

2. По требуемому объему памяти затруднительно сравнивать процедурную и автоматные реализации. Поэтому ограничимся только сравнением количества клеток в управляющих таблицах для автоматных реализаций, сопоставляя эти данные с численными характеристиками алфавитов и порождающих правил грамматики:

Способ реализации		**Кол-во клеток**	**Кол-во полей / операций**
2	Автомат с несколькими состояниями	33	6 (полей)
3	Оптимизированный автомат	23	6 (полей)
4	Автомат с одним состоянием	49	До 4 (операций)

Количество клеток в управляющих таблицах автоматов с несколькими состояниями определяется общим количеством символов в системе порождающих правил грамматики. Если на основе акцептора будет реализовываться синтаксический анализатор, то многие состояния, удаленные при оптимизации, придется восстановить (см. раздел 4), поэтому различия между этими вариантами нельзя считать существенными.

Количество клеток в управляющей таблице автомата с одним состоянием определяется произведением количества терминальных символов (с учетом псевдотерминала, обозначающего конец файла) на количество нетерминалов плюс тех терминалов, которые встречаются в правых частях порождающих правил не на первой позиции.

Различие объемов линейных таблиц автоматов с несколькими состояниями и двумерной таблицы автомата с одним состоянием, не очень заметное для простейшей грамматики $\boldsymbol{G}_{a2}$, становится существенным для грамматик реальных языков. Так, например, для языков класса Pascal или C/C++ (при типичных количествах: терминалов – 100…150, нетерминалов – примерно столько же, правил – 600…800 при среднем количестве символов в правиле порядка трех) количество клеток линейной таблицы составит величину порядка 1500…2500, а двумерной – порядка 15000…25000.

Ясно, что такое различие (примерно на порядок) может оказывать существенное влияние на выбор способа реализации нисходящего синтаксического акцептора. Однако существуют методы уплотнения разреженных таблиц (управляющая таблица автомата с единственным состоянием именно такой и является, поскольку содержит много пустых клеток), позволяющие существенно уменьшить объем памяти, требуемый для их хранения.

Таким образом, и по затратам времени, и по требуемому объему памяти рассмотренные методы реализации нисходящего синтаксического акцепта оказываются практически эквивалентными. Выбор в пользу одного из них обычно осуществляется на основе таких неформализуемых критериев, как удобство решения прочих задач трансляции – преобразование в постфиксную запись, и/или последовательность тетрад, семантический анализ и генерация объектного кода.

3. ВОСХОДЯЩИЕ МЕТОДЫ СИНТАКСИЧЕСКОГО АКЦЕПТА

Восходящими называются такие методы синтаксического акцепта, при которых цепочка терминальных символов входного предложения шаг за шагом «сворачивается» в цепочки, содержащие, кроме терминальных еще и нетерминальные символы, являющиеся очередными уровнями дерева грамматического разбора. Этот процесс продолжается до тех пор, пока все правильное предложение не окажется свернутым в цепочку, содержащую единственный нетерминальный символ – начальный нетерминал грамматики, либо до тех пор, пока не будет обнаружена невозможность восстановления дерева для неправильных предложений.

Как и для группы нисходящих методов, главным прагматическим требованием к организации процесса сворачивания цепочек является детерминированность (однонаправленность) движения по дереву снизу вверх. Опять-таки не любая грамматика обладает такими свойствами, чтобы по ее правилам можно было осуществлять детерминированное восходящее восстановление дерева разбора.

3.1. Основная идея восходящего восстановления дерева грамматического разбора

Для иллюстрации основной идеи будем использовать грамматику языка арифметических выражений $\boldsymbol{G}_{a1}$ с добавленным специальным правилом вида $Z : S$ ►, не содержащую аннулируемых нетерминалов.

Грамматика G_{a1}	
0	$Z : S$ ►
1	$S : S + T$
2	$S : T$
3	$T : T * V$
4	$T : V$
5	$V : (S)$
6	V : *ident*
7	V : *const*

Наличие правил вида $N : \varepsilon$ создает некоторые проблемы на начальном этапе формулирования основной идеи восходящего синтаксического акцепта. Впоследствии мы увидим, что наличие аннулируемых нетерминалов в грамматике на самом деле не препятствует восходящему восстановлению дерева грамматического разбора.

Временно, до тех пор пока не будут определены условия останова по ошибке, будем считать, что входное предложение является заведомо правильным.

Основная идея группы восходящих методов синтаксического акцепта возникла примерно из следующих рассуждений.

Пусть дана цепочка терминалов *a* ►, содержащая правильное предложения языка, порождаемого этой грамматикой.

Первый терминальный символ этого предложения является правой частью правила 6 грамматики (V : *ident*).

Правую часть какого-либо правила грамматики, находящуюся в предложении (в действительности – в цепочке символов, образующей текущий уровень дерева разбора), принято называть основой.

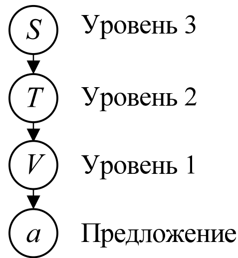
Заменив основу *a* нетерминалом из левой части правила, получим цепочку символов V ►.

Замену основы нетерминалом из левой части правила принято называть операцией свертки.

В полученной в результате свертки цепочке символов, образующей самый нижний уровень дерева разбора, теперь можно снова обнаружить основу, но уже другого правила – $T : V$. Выполнив свертку этой основы по правилу 4, получим следующий уровень дерева: цепочку символов T ►. В этой цепочке, в свою очередь, содержится основа T правила с номером 2 ($S : T$).

Применяя свертку к этой основе, получаем цепочку символов S ►.

Поскольку входная цепочка символов прочитана полностью, делаем вывод, что дерево разбора предложения *a* из начального нетерминала грамматики S восстановлено:



Таким образом, основная идея восходящего метода на данный момент представляется очень простой. Процесс восстановления дерева грамматического разбора есть последовательность шагов преобразования цепочек символов, на каждом из которых производится:

– обнаружение в текущем уровне дерева основы – цепочки символов, являющейся правой частью какого-либо правила грамматики;

– свертка основы, т. е. замена ее нетерминалом из левой части соответствующего правила и получение очередного уровня дерева.

Однако если попытаться применить основную идею к несколько более сложному правильному предложению (например, к цепочке $a + b * c$ ►), то сразу возникают затруднения.

Во входном предложении можно обнаружить сразу несколько основ. Следует ли как-то упорядочить свертки этих основ или их можно выполнять в произвольной последовательности?

Ситуация может оказаться более сложной: в цепочках символов, являющихся уровнями дерева разбора, могут быть обнаружены перекрывающиеся основы. В самом деле, правые части правил 2 и 4 грамматики $\boldsymbol{G}_{a1}$ полностью содержатся в правых частях правил 1 и 3 соответственно. Следовательно, если на очередном уровне дерева разбора обнаруживается основа одного из правил 1 или 3, то возникает конфликтная ситуация, поскольку свертка по правилу 1 (основа $S + T$) исключает возможность выполнения свертки по внутренней основе T правила 3. И наоборот, свертка внутренней основы T приводит к исчезновению основы $S + T$. Какое именно правило нужно выбирать для свертки в подобных случаях?

Не исключено, что могут возникнуть и другие трудности.

Для того чтобы ответить на эти вопросы, нужно сформулировать основную идею в виде алгоритма, точно определяющего порядок выполнения действий для любой ситуации, которая может возникнуть в процессе восстановления дерева грамматического разбора.

При этом должны соблюдаться все ранее определенные основные требования к синтаксическому акцептору:

– детерминированность (безвозвратность) процесса восстановления дерева разбора;

– гарантированность успешного восстановления дерева для любого правильного предложения;

– обнаружение ошибки и отказ восстановления дерева для любого неправильного предложения.

3.2. Стековый автомат для реализации основной идеи восходящего анализа

Восходящий синтаксический акцептор будем реализовывать в виде конечного автомата со стековой памятью (рис. 8). Автомат имеет единственное (пока) состояние и стек, над которым можно выполнять обычные действия:

– добавлять символ (терминальные символы считываются из предложения, нетерминальные символы образуются в результате выполнения свертки);

– удалять произвольное количество символов, образующих основу, с верхушки.

Кроме того, будем считать, что автомат может просматривать произвольные символы, находящиеся в стеке. Ясно, что это требуется для обнаружения основы.

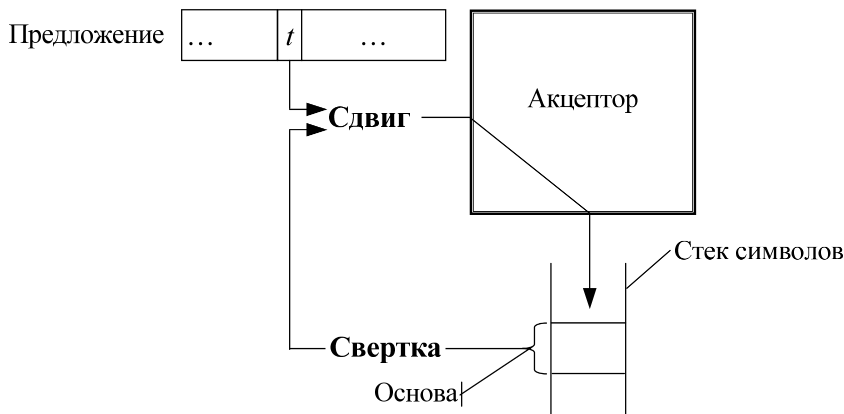


Рис. 8. Стековый автомат восходящего акцептора

Алгоритм реализации основной идеи восходящего анализа теперь можно определить как следующую совокупность шагов.

1. Инициализация. Очистить стек.

2. Сдвиг. Если на входе находится псевдотерминал ► (end of file), то перейти к шагу 5. Иначе очередной символ с входа автомата переместить на верхушку стека. Операцию перемещения символа с входа автомата в стек далее будем сокращенно обозначать буквой **S** (от английского слова Shift).

3. Поиск правила.

3.1. Взять первое правило грамматики в качестве текущего.

3.2. Сравнить (двигаясь по правилу справа налево, а по стеку сверху вниз) цепочку правой части текущего правила с цепочкой, находящейся на вершине стека. Если правая часть текущего правила обнаружена на верхушке стека, то перейти к шагу 4, иначе – к пункту 3.3.

3.3. Если просмотрены не все правила грамматики, то взять в качестве текущего следующее правило и вернуться к пункту 3.2, иначе вернуться к шагу 2.

4. Свертка. Удалить из стека ровно столько символов, сколько их в правой части найденного правила, поместить нетерминал из левой части этого правила на вход автомата перед очередным терминалом из предложения, и вернуться к шагу 2. Операцию свертки будем обозначать буквой **R** (от английского Reduce) с параметром, равным номеру правила.

5. Если в стеке находится единственный символ *S* (начальный нетерминал грамматики), то перейти к шагу 6, иначе – к шагу 7.

6. Останов по окончании восстановления дерева разбора. Входная цепочка, безусловно, является правильным предложением.

7. Останов по невозможности восстановления дерева разбора. Входная цепочка, ***вероятно*** (ниже объясняется, почему это так), не является правильным предложением.

Попытаемся использовать этот автомат для восстановления дерева разбора правильного арифметического выражения

$$x+y*z►$$

Последовательность действий автомата сведем в таблицу.

Такт	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Вход	*x*	+	*V*	+	*T*	+	*S*	+	*y*	*	*V*	*	*T*	*
Опер.		**S**	**R6**	**S**	**R4**	**S**	**R2**	**S**	**S**	**S**	**R6**	**S**	**R4**	**S**
Стек		*x*		*V*		*T*		*S*	+	*y*	+	*V*	+	*T*
									S	+	*S*	+	*S*	+
										S		*S*		*S*

Каждый столбец таблицы содержит: номер такта, текущий символ на входе автомата, обозначение выполненной на этом такте операции и образовавшееся в результате состояние стека.

Такт 0 – инициализация автомата. На такте 1 выполнен сдвиг терминала x в стек. На такте 2 в стеке обнаружена основа правила 6, и выполнена свертка по этому правилу. Нетерминал V образовался на входе автомата перед текущим терминалом +. На такте 3 этот нетерминал заносится в стек, в результате чего на верхушке стека образуется основа правила 4. На такте 4 эта основа снимается со стека путем выполнения операции свертки **R4**, на входе автомата возникает нетерминал T. Вплоть до такта 13 на каждом шаге автомат выполняет предписанные алгоритмом действия, никаких конфликтных ситуаций не обнаруживается.

После такта 13 на верхушке стека образуется сразу 2 основы: правая часть правила 1 ($S + T$) и правая часть правила 2 (T).

Эта ситуация называется конфликтом «свертка/свертка», возникающим в процессе восстановления дерева разбора.

Согласно пункту 3.2. алгоритма автомат первой обнаруживает основу правила 1 и выполняет свертку по нему. Продолжение истории работы автомата выглядит так:

Такт	14	15	16	17	18	19	20	21	22	23	24
Вход	S	*	z	▶	V	▶	T	▶	S	▶	▶
Опер.	**R1**	**S**	**S**	**S**	**R6**	**S**	**R4**	**S**	**R2**	**S**	**Error**
Стек		S	*	z	*	V	*	T	*	S	S
			S	*	S	*	S	*	S	*	+
				S		S		S		S	S

На такте 24 автомат останавливается по ошибке, поскольку входным символом является ▶ (end of file), в стеке нет ни одной основы, и содержимое стека не является единственным начальным нетерминалом грамматики.

Однако входное предложение, безусловно, является правильным арифметическим выражением и дерево его разбора существует. Автомат, руководствуясь изложенным выше алгоритмом (см. пункт 7 алгоритма), просто не сумел найти способ восстановления этого дерева.

Возможно, причина неправильного поведения автомата состоит в том, что им было выбрано не то правило на шаге 13. Посмотрим, что будет, если на этом шаге выбрать не правило 1, а правило 2:

Такт	14	15	16	17	18	19	20	21	22	23	24
Вход	S	*	z	▶	V	▶	T	▶	S	▶	▶
Опер.	**R2**	**S**	**S**	**S**	**R6**	**S**	**R4**	**S**	**R2**	**S**	**Error**
Стек		S	*	z	*	V	*	T	*	S	S
		+	S	*	S	*	S	*	S	*	*
		S	+	S	+	S	+	S	+	S	S
			S	+	S	+	S	+	S	+	+
				S		S		S		S	S

Оказывается, и это продолжение процесса восстановления дерева этим автоматом тоже приводит к останову по ошибке.

Если проанализировать ситуацию, которая возникла на такте 13, то можно сделать такой вывод: кроме конфликта «свертка/свертка» на этом такте имеет место еще и конфликт: «сдвиг/свертка». Если на такте 14 выполнить сдвиг вместо любой свертки, несмотря на наличие двух основ на верхушке стека, то автомат далее отработает такую историю:

Такт	14	15	16	17	18	19	20	21	22
Вход	z	▶	V	▶	T	▶	S	▶	▶
Опер.	**S**	**S**	**R6**	**S**	**R3**	**S**	**R1**	**S**	**Stop**
Стек	*	z	*	V	+	T		S	S
	T	*	T	*	S	+			
	+	T	+	T		S			
	S	+	S	+					
		S		S					

На такте 20 опять возникает конфликт «свертка/свертка», но выбор правила 1 приводит к тому, что на такте 22 автомат останавливается, успешно восстановив дерево грамматического разбора.

Можно сказать, что алгоритм восходящего стекового автомата с одним состоянием не обеспечивает детерминированности принятия решения на каждом шаге процесса восстановления дерева разбора. Причины этого:

– не обнаруживаются конфликты типа «свертка/свертка», следовательно, на некоторых шагах может быть обнаружена не та основа, по которой действительно требуется выполнять свертку;

– даже если модифицировать п. 3.2 алгоритма и перебирать все правила, а не останавливаться на первом, основа которого найдена в

стеке, на данный момент неизвестен способ разрешения таких конфликтов, т. е. выбора одной основы для свертки из нескольких;

– не обнаруживаются конфликты типа «сдвиг/свертка»;

– соответственно нет способа разрешения таких конфликтов.

Можно попытаться усовершенствовать этот алгоритм путем добавления действий по обнаружению и разрешению возникающих конфликтов во время работы автомата.

Более перспективен другой путь – предотвратить возникновение конфликтов в процессе работы автомата за счет введения нескольких состояний и анализа конфликтных ситуаций до начала восстановления дерева разбора, а именно, при построении его управляющей таблицы.

3.3. Восходящий синтаксический акцептор с несколькими состояниями

Будем считать, что автомат такого акцептора задается управляющей таблицей, строки которой соответствуют состояниям, а столбцы – символам грамматики. В клетках управляющей таблицы располагаются обозначения операций, которые должен уметь выполнять автомат.

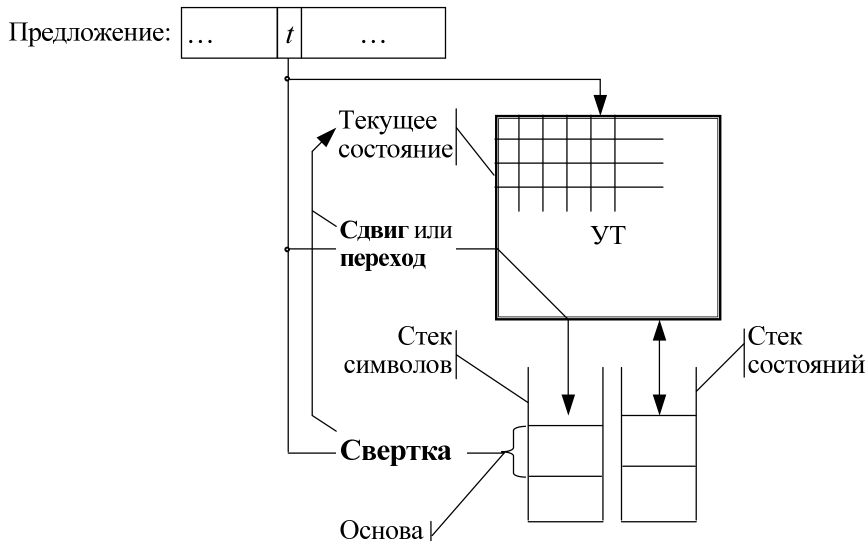


Рис. 9. Восходящий синтаксический акцептор как автомат с несколькими состояниями и двумя стеками

Для выяснения перечня требуемых операций восходящего акцептора вернемся к существу задач, решаемых при реализации основной идеи, ориентируясь на то, что начало текущего уровня восстанавливаемого дерева хранится в стеке автомата.

1. Проверка наличия (или отсутствия) на верхушке стека основы – цепочки, являющейся правой частью какого-либо правила грамматики.

Для решения этой задачи свяжем занесение произвольного символа (эта операция была названа сдвигом) в стек с переходом автомата из одного состояния в другое. Очевидно, что тогда (по аналогии с конечными автоматами без памяти, используемыми при лексическом анализе) состояние, в котором автомат оказывается после занесения в стек последнего символа правой части некоторого правила, можно рассматривать как финальное для этого правила. Если автомат оказывается в финальном состоянии для некоторого правила, то должна быть выполнена операция свертки по этому правилу.

В стек может заноситься:

– терминальный символ, читаемый из входной цепочки. Занесение такого символа в стек сопровождается переходом к следующему символу входной цепочки. Будем называть такую операцию сдвигом и обозначать латинской буквой **S** от английского слова Shift;

– нетерминальный символ, образуемый в результате свертки. При выполнении такой операции позиция обрабатываемого символа во входной цепочке не изменяется. Такую операцию в отличие от сдвига будем называть переходом (напомним, что любое занесение символа в стек сопровождается изменением состояния автомата) и обозначать латинской буквой **G** от слова Go.

Знаки операций сдвига и перехода должны иметь один параметр – номер состояния, в котором должен оказаться автомат после выполнения операции: **Sn** или **Gn**.

Основа в стеке может появиться только в результате выполнения операции сдвига или перехода. Факт ее образования – переключение автомата в финальное состояние, сопоставленное с конкретным правилом грамматики.

2. Свертка найденной основы (рис. 9), т. е. удаление всей правой части правила, накопленной в стеке, с его верхушки и помещение нетерминала из левой части этого правила на вход автомата. Такую операцию будем обозначать латинской буквой **R** от английского слова Reduce. Знак операции свертки должен иметь как минимум два параметра, определяемых видом порождающего правила: количество символов, которые долж-

ны быть удалены из стека автомата, и нетерминальный символ, который должен быть вставлен в цепочку текущего уровня восстанавливаемого дерева вместо правой части правила. Полное обозначение операции свертки на данный момент выглядит как **R**k.

Теперь зададимся вопросом: в каком состоянии должен оказаться автомат после выполнения операции свертки? Для нахождения ответа рассмотрим фрагмент истории работы, непосредственно предшествующий этой операции и начинающийся с обработки первого символа правой части правила $N : \beta$. Цепочка β есть последовательность символов w_1 , w_2 , ... w_k , каждый из которых может быть и терминалом, и нетерминалом. Пусть перед обработкой символа w_1 автомат находился в состоянии C_1 (в это состояние его привела обработка цепочки символов φ, являющейся началом текущего уровня дерева и уже находящейся в стеке). Обработка символа w_1 состоит в выполнении операции сдвига (если w_1 – терминал) или перехода (если w_1 – нетерминал) и привела к занесению символа в стек и к переключению автомата в состояние C_2. Обработка всей цепочки β_i привела к помещению последовательности символов w_1 , w_2 , ... w_k в стек и к переключению автомата в состояние C_{k+1}:

w_1	w_2	...	w_k	
C_1	C_2	...	C_k	C_{k+1}

Достижение автоматом состояния C_{k+1} означает, что на верхушке стека символов находится основа правила $N : \beta_i$, следовательно, в этом состоянии может быть выполнена свертку по этому правилу. Решение о том, выполнять свертку или нет, зависит от текущего входного терминала, в силу того что возможны конфликты типа «сдвиг/свертка». Рассмотрение способов принятия этого решения приводится далее, а пока сосредоточимся на выяснении того, в каком состоянии автомат должен оказаться после свертки.

Свертка цепочки β в нетерминал N в конечном итоге должна привести к преобразованию текущего (к моменту C_{k+1}) уровня дерева φ β к виду φ N. Операция свертки удаляет цепочку β из стека. Соответственно после выполнения свертки автомат должен оказаться точно в том же самом состоянии C_1, в котором он находился перед первым символом этой цепочки, а вместо этой цепочки на входе должен образоваться нетерминал N. Для обработки этого символа в состоянии C_1 (в соответствующей клетке управляющей таблицы автомата) должна

быть предусмотрена операция перехода **Gx**, завершающая формирование начала текущего уровня в требуемом виде φ *N*.

Рассмотрим этот момент подробнее. Казалось бы логичным выполнять занесение в стек нетерминала, заменяющего правую часть правила, непосредственно при выполнении операции свертки. Однако занесение в стек любого символа должно сопровождаться переключением состояния автомата в новое состояние. Но номер нужного состояния не зависит от того, какое правило применяется для выполнения свертки и какой нетерминал образуется в ее результате. Номер нового состояния определяется тем, в правой части какого правила находится нетерминал, формируемый операцией свертки. Действительно, рассмотрим два правильных выражения $x + y + z$ и $(x + y) * z$. При восстановлении их деревьев разбора подцепочка $x + y$ в конечном итоге должна оказаться свернутой в нетерминал *S*. Дальнейшие переходы восходящего анализатора для этих входных цепочек должны быть различными, поскольку первая построена согласно правилу $S : S + T$ (нетерминал, полученный в результате свертки, является первым символом его правой части), а начало второй – согласно правилу $V : (S)$ грамматики $\boldsymbol{G}_{a1}$. Именно поэтому операция свертки, завершающая формирование нетерминала *S* из подцепочки $x + y$, не может и не должна осуществлять переход автомата в новое состояние по этому нетерминалу.

Вернемся к вопросу определения номера состояния после свертки. Как видно из приведенной истории работы, удаление k символов из стека можно рассматривать, как возврат на k шагов по последовательности состояний автомата. Для реализации такого возврата достаточно ввести дополнительный стек состояний.

При запуске автомата в этот стек должен быть помещен номер начального состояния. Любая операция сдвига или перехода должна заносить номер состояния, в которое переключается автомат, на верхушку стека. Тогда при выполнении свертки по правилу с k символами в правой части достаточно снять (удалить) ровно k номеров со стека состояний и переключить автомат в состояние, номер которого оказался на верхушке. Тем самым обеспечивается восстановление именно того состояния, в котором автомат находился перед первым символом правой части применяемого правила.

А теперь мы обнаруживаем, что стек символов вообще не нужен. Действительно, символы, помещаемые в этот стек, больше нигде и никогда не используются. Поэтому оставим в рассматриваемой автоматной модели только стек состояний и окончательно сформулируем спо-

собы выполнения операций восходящего акцептора. Поскольку символы грамматики автоматом теперь используются только в качестве заголовков столбцов управляющей таблицы, будем считать, что они (символы) перенумерованы последовательностью чисел, начинающейся с нуля, и заменим символы их номерами в заголовках. Порядок нумерации может быть произвольным, но для определенности будем считать, что наименьшие номера (0, 1, …) имеют нетерминалы, а терминалам присвоены следующие по порядку номера.

Окончательный смысл операций восходящего акцептора состоит в следующем (рис. 10).

1. Операция Shift (сдвиг), обозначаемая как **Sx**. Перейти к следующему символу во входной цепочке, занести номер x в стек состояний, переключить автомат в состояние **x**.

2. Операция Go (переход), обозначаемая как **Gx**. Занести номер **x** в стек состояний, переключить автомат в состояние **x**.

3. Операция Reduce (свертка), обозначаемая как **Rk,n**. Снять k номеров состояний со стека, переключить автомат в состояние, оказавшееся на верхушке стека, установить текущим столбец таблицы с номером n (номер n присвоен нетерминальному символу, образующеемуся в результате выполнения свертки).

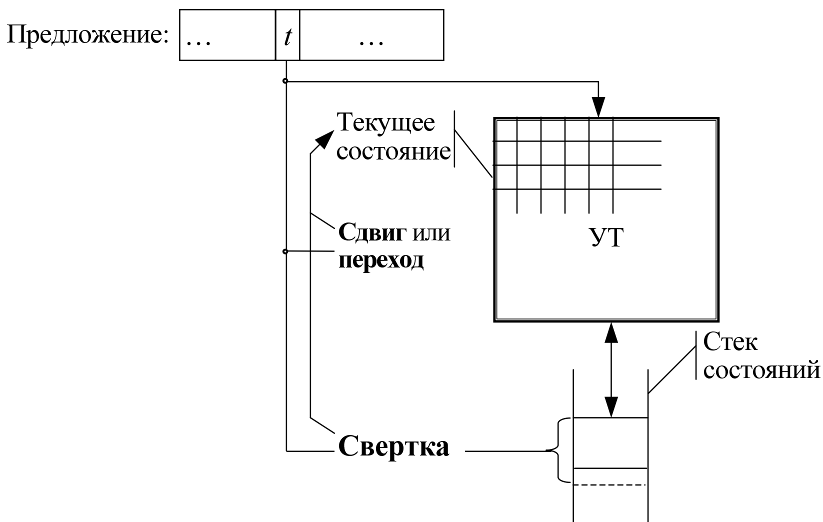


Рис. 10. Восходящий акцептор как автомат с несколькими состояниями и стеком состояний

Для полного определения автомата требуются еще две операции:

4. Операция **Stop**, не имеющая сокращенного обозначения. Останов по окончании восстановления дерева разбора правильного предложения.

5. Операция останова по обнаружении ошибки. Будем считать, что пустая клетка, не содержащая какого-либо из перечисленных знаков операций, содержит операцию останова по ошибке.

Теперь, полностью определив смысл и обозначения операций восходящего акцептора и структуру его управляющей таблицы, займемся методами ее построения на основе заданной грамматики.

3.4. Понятие конфигурации. Связь между конфигурациями, состояниями и операциями восходящего акцептора

Введем понятие конфигурации. Под конфигурацией понимается правило грамматики, в котором в произвольной точке правой части помещен дополнительный метасимвол – маркер. Маркером отмечается возможное разбиение правой части правила в процессе восходящего синтаксического анализа на цепочки: «прочитанную» (находящуюся слева от маркера и, условно, – в стеке символов) и еще не обработанную, находящуюся справа от маркера. Ясно, что если правая часть правила содержит ровно k символов (k = 0, 1, 2, …), то из него может быть образовано в точности k+1 различных конфигураций. Например, из правила $S : S + T$ можно образовать следующие конфигурации (в качестве маркера используется символ ▼):

$$S : \blacktriangledown\ S + T$$

$$S : S\ \blacktriangledown\ + T$$

$$S : S +\ \blacktriangledown\ T$$

$$S : S + T\ \blacktriangledown$$

Для любой контекстно-свободной грамматики с конечным количеством правил множество всех возможных различных конфигураций также конечно.

Между конфигурациями, состояниями восходящего автомата и операциями, которые должны выполняться в этих состояниях, существуют определенные связи (соотношения), на основании которых может

быть определен метод преобразования грамматики в управляющую таблицу.

Рассмотрим связи между конфигурациями и операциями:

Если в некоторой конфигурации маркер находится перед терминалом (N : α ▼t β), то в состоянии номер m, которому соответствует эта конфигурация, должна выполняться операция Shift, осуществляющая переключение автомата в такое состояние, которому соответствует конфигурация с маркером, размещенным после этого терминала (N : α t▼ β),. Знак этой операции должен находиться в той клетке управляющей таблицы, которая расположена на пересечении строки состояния m со столбцом, помеченным терминальным символом t (или номером, приписанным этому терминалу).

Если маркер в конфигурации N : α ▼M β находится перед нетерминалом, то в соответствующем ей состоянии m должна выполняться операция Go, осуществляющая переключение автомата в такое состояние, которому соответствует конфигурация с маркером, размещенным после этого нетерминала, т. е. N : α M▼ β. Знак этой операции должен находиться в той клетке управляющей таблицы, которая расположена на пересечении строки состояния m со столбцом, помеченным нетерминальным символом M (или его номером).

И, наконец, если маркер в конфигурации находится после последнего символа правила, то в соответствующем этой конфигурации состоянии должна выполняться операция Reduce **Rk,n**, удаляющая из стека **k** номеров состояний (ровно столько символов в правой части соответствующего правила), переключающая автомат в состояние, номер которого оказался на верхушке стека после этого удаления, и готовящая номер столбца **n** управляющей таблицы для последующей операции Go.

Теперь рассмотрим связи между конфигурациями и состояниями автомата. Допустим, что каким-либо образом некоторому состоянию поставлена в соответствие определенная конфигурация. Если эта конфигурация имеет вид

$$N : \alpha \blacktriangledown M \beta ,$$

где M – нетерминальный символ, и в грамматике есть правила:

$$M : \gamma_1$$
$$M : \gamma_2$$
$$\ldots$$
$$M : \gamma_n ,$$

то все конфигурации вида $M : \blacktriangledown \gamma_1 , M : \blacktriangledown \gamma_2 , \ldots M : \blacktriangledown \gamma_n$ также должны быть поставлены в соответствие этому состоянию автомата.

Действительно, если в некоторой цепочке терминалов, выводимой из начального нетерминала грамматики, обнаружена подцепочка, выводимая из N и начинающаяся с цепочки, выводимой из α, то автомат в этот момент находится в состоянии, отмеченном маркером в конфигурации $N : \alpha \blacktriangledown M \beta$.

Поскольку входная цепочка правильна, ее продолжение (а автомат в этот момент находится перед первым символом этого продолжения) обязано выводиться из цепочки $M \beta$. Следовательно, оно выводится из одной из цепочек вида $\gamma_i \beta$.

В силу того что автомат должен восстановить этот пока еще неизвестный вывод, данному его состоянию должны быть сопоставлены все конфигурации, образуемые путем помещения маркера перед первыми символами порождающих правил для нетерминала M. Это позволит по символам, перед которыми находятся маркеры во вновь образованных конфигурациях, определить для данного состояния знаки операций в управляющей таблице, необходимые для восстановления нужного вывода.

Конфигурации вида $M : \blacktriangledown \gamma_i$ будем называть дочерними по отношению к исходной (родительской) конфигурации вида $N : \alpha \blacktriangledown M \beta$.

Заметим, что если в какой-либо из вновь образованных конфигураций вида $M : \blacktriangledown \gamma_i$ маркер находится перед нетерминалом L (цепочка γ_i начинается с этого нетерминального символа), то данному состоянию автомата должны быть поставлены в соответствие и все конфигурации вида $L : \blacktriangledown \lambda_j$ (дочерние по отношению к $M : \blacktriangledown \gamma_i$).

Таким образом, состоянию автомата может соответствовать не одна конфигурация, а некоторое подмножество полного множества конфигураций заданной грамматики.

Пусть дано подмножество (одна или несколько конфигураций), сопоставленных состоянию автомата. Применение ранее описанного способа добавления к подмножеству конфигураций, дочерних по отношению к одной или нескольким родительским вплоть до момента, когда это подмножество перестанет изменяться (расширяться), называется его замыканием. При выполнении замыкания новые дочерние конфигурации добавляются к подмножеству только в том случае, если их в нем еще нет.

Пусть известно замыкание подмножества конфигураций, соответствующих некоторому состоянию автомата. Разобьем его на более

мелкие подмножества, содержащие конфигурации, в которых маркер находится перед одним и тем же символом грамматики. В особое подмножество выделим конфигурации, в которых маркер находится после последнего символа правила.

Каждая из конфигураций последнего подмножества (если оно не пусто), имеющая вид $X : \chi$ ▼, указывает на необходимость выполнения автоматом, оказавшимся в данном состоянии, операции свертки вида $\mathbf{Rl,n_x}$. Здесь **l** – длина цепочки χ, а $\mathbf{n_x}$ – номер колонки управляющей таблицы автомата, поставленной в соответствие нетерминальному символу *X*.

Наличие нескольких различных конфигураций в этом подмножестве является причиной возникновения конфликтов типа свертка/свертка, похожих на те, которые рассматривались в разд. 3.2. Отличие состоит в том, что при использовании множеств конфигураций эти конфликты обнаруживаются не в процессе восстановления дерева разбора, а на этапе построения управляющей таблицы автомата. Детально подобные ситуации будут изучены далее.

Вернемся к таким конфигурациям данного состояния, в которых маркер находится перед одним и тем же символом *w*. Очевидно, что если автомат в процессе работы оказался в данном состоянии и его текущим входным символом является *w*, то должна быть выполнена операция сдвига, если *w* – терминал, или перехода, если *w* – нетерминал. Выполнение такой операции переключает автомат в некоторое другое состояние C, номер которого заносится на верхушку стека.

Применительно к подмножеству конфигураций, в которых маркер находится перед символом *w*, такую операцию следует трактовать, как перенос маркера через этот символ. Тем самым образуется подмножество конфигураций, которое должно быть сопоставлено с тем состоянием C, в которое переключается автомат. Состав этого подмножества может впоследствии измениться при его замыкании. Назовем подмножество конфигураций в его начальном составе (в момент образования путем переноса маркера через *w*) базовым или просто базой по отношению к состоянию C.

Таким образом, если известно подмножество конфигураций какого-либо состояния автомата, то могут быть определены базовые подмножества конфигураций, а после выполнения их замыкания – и полные множества конфигураций некоторых других состояний, а именно тех, в которые автомат может переключиться из исходного.

В силу того что база начального состояния автомата всегда известна – ею является конфигурация $Z : ▼ S ▶$, на основе рассмотренных связей между состояниями, операциями и конфигурациями может быть построена процедура преобразования грамматики в управляющую таблицу восходящего синтаксического акцептора.

Рассмотрим эту процедуру, выполняемую в два этапа.

Этап 1. Формирование подмножеств конфигураций, соответствующих состояниям автомата. Определение набора состояний.

Этап 2. Преобразование таблицы конфигураций в управляющую таблицу восходящего акцептора.

3.5. Определение набора состояний восходящего акцептора

Подмножества конфигураций, соответствующих состояниям автомата, будем формировать в табличном представлении. Начальное состояние таблицы подмножеств конфигураций (далее – просто таблицы конфигураций) для любой грамматики (с точностью до наименования начального нетерминала *S*) определяется добавочным правилом и выглядит так:

Состояние	*Образовано*		*База*	*Конфигурация*	*Символ*	*Отм.*
	Из	*Через*				
0			Да	$Z : ▼ S ▶$	*S*	

Поясним назначение колонок таблицы конфигураций:

состояние – содержит номер, присвоенный состоянию автомата, определяемому подмножеством конфигураций;

образовано – из подмножества конфигураций какого состояния путем переноса маркера через какой символ;

база – просто отметка Да, если конфигурация, записанная в строке, является базовой для данного состояния;

конфигурация – пояснения не требуется;

символ – вспомогательная колонка, содержащая тот символ грамматики, перед которым в данной конфигурации находится маркер;

отм. – также вспомогательная колонка. Ее назначение станет ясно из описания процедуры.

При определении процедуры мы будем использовать грамматику $\boldsymbol{G}_{a1}$ для иллюстрации построения таблицы конфигураций. Начальное

состояние таблицы соответствует моменту, когда образована база начального состояния.

Шаг 1. Замыкание. Просматриваются все конфигурации нового состояния автомата, и если в какой-либо из них маркер находится перед нетерминалом, то каждое правило грамматики, имеющее этот нетерминал в левой части, преобразуется в конфигурацию путем помещения маркера перед первым символом правой части. Для каждой вновь образованной конфигурации просматриваются все строки таблицы данного состояния. Если вновь образованной конфигурации нет ни в одной такой строке, то формируется новая строка. Новая конфигурация заносится в соответствующую клетку этой строки, а в клетку колонки, обозначенной *Символ*, заносится первый символ правой части правила (если правило имеет вид $N : \varepsilon$, то эта клетка остается пустой), все остальные клетки новой строки оставляются пустыми. Этот шаг выполняется для данного состояния до тех пор, пока не перестанут добавляться новые строки.

Приведем построенный фрагмент таблицы конфигураций после выполнения этого шага для нулевого состояния:

Состояние	*Образовано*		*База*	*Конфигурация*	*Символ*	*Отм.*
	Из	*Через*				
0			Да	Z : ▼ S ▶	S	
				S : ▼ $S + T$	S	
				S : ▼ T	T	
				T : ▼ $T * V$	T	
				T : ▼ V	V	
				V : ▼ (S)	(	
				V : ▼ i	i	
				V : ▼ c	c	

Шаг 2. Образование новой базы. Просматривается колонка *Отм.* с целью найти такую строку таблицы, которая еще не помечена и в колонке *Символ* содержит какой-либо символ грамматики, отличный от псевдотерминала ▶ (конец файла). Если таких строк нет, то этап построения таблицы конфигураций завершается.

Если такая строка найдена, то зафиксируем номер состояния (далее оно называется исходным), найдем и отметим те строки таблицы, которые принадлежат данному состоянию и содержат тот же самый символ в соответствующей колонке. Выберем конфигурации из соответст-

вующей колонки этих строк и перенесем в каждой из них маркер через один символ. Тем самым будет образовано новое базовое подмножество конфигураций.

Шаг 3. Проверка наличия состояния с таким набором базовых конфигураций, который совпадает с вновь образованной базой. Просматривается таблица конфигураций, и база каждого уже существующего состояния (в том числе – исходного) сравнивается с новой базой, сформированной на предыдущем шаге. Возможны два случая.

3.1. Состояние, имеющее в точности такую же базу, уже существует (подчеркнем, что для этого новая и уже существующая база должны совпадать полностью). В этом случае добавим в колонки *Из* и *Через* этого состояния номер исходного состояния и символ, через который был перенесен маркер для образования базы. Эти данные пригодятся на втором этапе при преобразовании таблицы конфигураций в управляющую таблицу. Возвращаемся к шагу **2**.

3.2. Не найдено состояния, имеющего в точности такую же базу, что и проверяемая. В этом случае образуется новое состояние. В таблицу добавляется столько строк, сколько конфигураций содержится в проверяемой базе. Клетки во вновь добавленных строках заполняются в соответствии с назначением колонок. После завершения формирования нового состояния осуществляется возврат к шагу **1**.

Описание процедуры завершено. Результат ее применения к грамматике G_{a1} приведен на следующей странице. Заметим, что в колонке *Отм*. указан порядковый номер применения второго шага процедуры. Шаги с первого по девятый приводили к образованию таких подмножеств базовых конфигураций, которых к моменту выполнения каждого шага еще не было в таблице. Поэтому были сформированы состояния с номерами 1-9.

На десятом применении второго шага в результате переноса маркера через символ T были получены конфигурации $S : T$ ▼ и $T : T$ ▼ $* V$, в точности совпадающие с базой состояния номер 2. Согласно третьему шагу процедуры новое состояние не образовано, в колонке *Из* состояния 2 отмечено, что его база получена из конфигураций как состояния 0, так и состояния 4. Начиная с одиннадцатого новые состояния были образованы только на 15, 20 и 24 применении второго шага процедуры. Все остальные шаги приводили к образованию уже существующих базовых подмножеств конфигураций.

Таблица конфигураций

Состояние	*Образовано*		*База*	*Конфигурация*	*Символ*	*Отм.*
	Из	*Через*				
0			Да	$Z : \blacktriangledown S \blacktriangleright$	S	1
				$S : \blacktriangledown S + T$	S	1
				$S : \blacktriangledown T$	T	2
				$T : \blacktriangledown T * V$	T	2
				$T : \blacktriangledown V$	V	3
				$V : \blacktriangledown (S)$	$($	4
				$V : \blacktriangledown i$	i	5
				$V : \blacktriangledown c$	c	6
1	0	S	Да	$Z : S \blacktriangledown \blacktriangleright$		
		S	Да	$S : S \blacktriangledown + T$	+	7
2	0, 4	T	Да	$S : T \blacktriangledown$		
		T	Да	$T : T \blacktriangledown * V$	*	8
3	0, 4, 7	V	Да	$T : V \blacktriangledown$		
4	0, 4, 7, 8	$($	Да	$V : (\blacktriangledown S)$	S	9
				$S : \blacktriangledown S + T$	S	9
				$S : \blacktriangledown T$	T	10
				$T : \blacktriangledown T * V$	T	10
				$T : \blacktriangledown V$	V	11
				$V : \blacktriangledown (S)$	$($	12
				$V : \blacktriangledown i$	i	13
				$V : \blacktriangledown c$	c	14
5	0, 4, 7, 8	i	Да	$V : i \blacktriangledown$		
6	0, 4, 7, 8	i	Да	$V : c \blacktriangledown$		
7	1, 9	+	Да	$S : S + \blacktriangledown T$	T	15
				$T : \blacktriangledown T * V$	T	15
				$T : \blacktriangledown V$	V	16
				$V : \blacktriangledown (S)$	$($	17
				$V : \blacktriangledown i$	i	18
				$V : \blacktriangledown c$	c	19
8	2, 10	*	Да	$T : T * \blacktriangledown V$	V	20
				$V : \blacktriangledown (S)$	$($	21
				$V : \blacktriangledown i$	i	22
				$V : \blacktriangledown c$	c	23
9	4	S	Да	$V : (S \blacktriangledown)$	$)$	24
		S	Да	$S : S \blacktriangledown + T$	+	25
10	7	T	Да	$S : S + T \blacktriangledown$		
		T	Да	$T : T \blacktriangledown * V$	*	26
11	8	V	Да	$T : T * V \blacktriangledown$		
12	9	$)$	Да	$V : (S) \blacktriangledown$		

3.6. Преобразование таблицы конфигураций в управляющую таблицу

В результате построения этой таблицы конфигураций было выяснено, что автомат, реализующий восходящее восстановление дерева грамматического разбора предложений языка, порождаемого грамматикой $\boldsymbol{G}_{a1}$, должен иметь ровно 13 состояний (с номерами 0…12). Общее количество символов (как терминальных, так и нетерминальных) вместе с псевдотерминалом ▶ равно 10.

Шаг 1. Строится пустая заготовка таблицы, содержащая 13 строк и 10 столбцов (не считая заголовочных). Строки и столбцы нумеруются, начиная с нуля. Нетерминальные символы грамматики в произвольном порядке помещаются в заголовки столбцов, начиная с нулевого. Затем в произвольном порядке формируются заголовки столбцов, соответствующих терминальным символам. Последняя колонка ставится в соответствие псевдотерминальному символу конца входной цепочки ▶.

Шаг 2. Занесение знаков операций Stop, Shift и Go.

Знак операции Stop заносится в последнюю клетку строки состояния 1. Это соответствует моменту окончания восстановления дерева разбора согласно конфигурации Z : S ▼ ▶.

Просматриваются колонки *Из* и *Через* таблицы конфигураций. Для каждой непустой пары значений из этих колонок формируем знак операции Shift, если символ в колонке *Через* является терминальным, и знак операции Go – в противном случае. Номер состояния, взятый из первой колонки текущей строки таблицы конфигураций, является параметром знака операции. Построенный таким образом знак операции заносится в клетку управляющей таблицы, находящуюся на пересечении строки с номером, взятым из колонки *Из*, и столбца, озаглавленного символом из колонки *Через*.

Например, при просмотре первой строки таблицы конфигураций, соответствующей состоянию номер 1, будет образован один знак операции **G**1, который должен быть занесен в клетку строки номер 0 того столбца управляющей таблицы, который помечен символом *S*. При обработке первой строки состояния 3 таблицы конфигураций будут образованы три одинаковых знака операции **S**3, которые должны быть помещены в клетки столбца, помеченного нетерминалом *V*, находящиеся на его пересечении со строками 0, 4 и 7.

Шаг 3. Занесение знаков операций Reduce. Просматривается содержимое колонки *Конфигурация* в поисках такой конфигурации, в

которой маркер находится после последнего символа правой части порождающего правила. Формируется знак операции **Rk**,**h**, где **k** – количество символов в правой части правила, **h** – номер столбца управляющей таблицы, помеченного нетерминалом из левой части этого правила.

Выполняется попытка занесения сформированного знака операции во все клетки той строки управляющей таблицы, которая имеет номер, взятый из первой колонки таблицы конфигураций. Знак операции свертки не заносится в клетки, столбцы которых помечены нетерминалами. Если клетка уже содержит знак операции Shift или Reduce, то фиксируется конфликт типа сдвиг/свертка или свертка/свертка соответственно. Вопрос о том, что делать при возникновении конфликтов, будет рассматриваться далее.

В качестве примера выполнения шага **3** рассмотрим состояние номер 2, в наборе конфигураций которого имеется конфигурация *S* : *T* ▼. Формируется знак операции **R**1,0, поскольку правая часть правила содержит один символ *T*, а нетерминалу *S* из левой части поставлен в соответствие столбец управляющей таблицы с номером 0. При занесении этого знака в строку номер 2 управляющей таблицы будет обнаружен конфликт типа сдвиг/свертка в клетке столбца, помеченного терминалом *. Возникновение конфликта легко объясняется наличием в наборе конфигураций состояния номер 2 конфигурации *T* : *T* ▼ * *V*.

Применение этой процедуры к построенной нами таблице конфигураций позволит получить следующую управляющую таблицу.

	0	1	2	3	4	5	6	7	8	9
	S	*T*	*V*	+	*	(	)	*i*	*c*	▶
0	**G**1	**G**2	**G**3			**S4**		**S5**	**S6**	
1				**S7**						**Stop**
2				**R**1,0	**S8**	**R**1,0	**R**1,0	**R**1,0	**R**1,0	**R**1,0
3				**R**1,1	**R**1,1	**R**1,1	**R**1,1	**R**1,1	**R**1,1	**R**1,1
4	**G**9	**G**2	**G**3			**S4**		**S5**	**S6**	
5				**R**1,2	**R**1,2	**R**1,2	**R**1,2	**R**1,2	**R**1,2	**R**1,2
6				**R**1,2	**R**1,2	**R**1,2	**R**1,2	**R**1,2	**R**1,2	**R**1,2
7		**G**10	**G**3			**S4**		**S5**	**S6**	
8			**G**11			**S4**		**S5**	**S6**	
9				**S7**			**S12**			
10				**R**3,0	**S8**	**R**3,0	**R**3,0	**R**3,0	**R**3,0	**R**3,0
11				**R**3,1	**R**3,1	**R**3,1	**R**3,1	**R**3,1	**R**3,1	**R**3,1
12				**R**3,2	**R**3,2	**R**3,2	**R**3,2	**R**3,2	**R**3,2	**R**3,2

В этой таблице серым фоном отмечены две клетки, в которых при занесении знаков операций были зафиксированы конфликты типа сдвиг/свертка.

Возникновение конфликтов, казалось бы, свидетельствует о недетерминированности конечного автомата, т. е. о невозможности восстановления дерева разбора некоторых входных цепочек без возвратов к пересмотру ранее принятых решений. Знаки операций сдвига и свертки не могут быть помещены в одну клетку управляющей таблицы в силу противоположности смысла действий, выполняемых этими операциями. То же самое относится и к потенциально возможным конфликтам типа свертка/свертка.

В одной клетке управляющей таблицы может находиться только один знак операции. Поэтому необходимо выяснить, может ли каждый конфликт быть разрешен в пользу одного из знаков операций. Только тогда можно считать завершенным процесс преобразования грамматики в восходящий синтаксический акцептор. Для разрешения конфликтов, очевидно, необходимо исследовать свойства исходной грамматики и порождаемого ею языка.

Однако в действительности для многих грамматик, в том числе для рассматриваемой грамматики $\boldsymbol{G}_{a1}$, возникновение конфликтов обусловлено не их свойствами и даже не свойствами порождаемого языка, а всего только несовершенством способа занесения знаков операций свертки в управляющую таблицу, сформулированного в шаге **3** процедуры.

3.7. Предотвращение конфликтов путем использования множеств последователей нетерминальных символов

Недетерминированность автомата, обнаруженная при попытке занесения знака операции свертки **R**1,0 (или **R**3,0) в клетку состояния номер 2 (или 10), находящуюся на пересечении со столбцом, помеченным терминалом * и содержащую знак операции **S**8, следует понимать как возможность реализации более чем одной истории работы, начиная с момента выполнения одной из этих двух конфликтующих операций.

Рассмотрим две такие истории работы автомата. Пусть для определенности входная цепочка имеет вид $x + y * z$ ▶.

История, в которой в состоянии номер 10 при обработке терминала * выполняется операция свертки **R**3,0, выглядит следующим образом (для наглядности здесь показано содержимое стека символов, реально в автомате не используемого):

Такт	0		1		2		3		4		5		6		7	
Вход	x+y*z▶		+y*z▶		+y*z▶		+y*z▶		+y*z▶		+y*z▶		+y*z▶		+y*z▶	
Сост.	0		5		0		3		0		2		0		1	
Опер.	**S**5		**R**1,2		**G**3		**R**1,1		**G**2		**R**1,0		**G**1		**S**7	
Стеки		0	x	5		0	V	3		0	T	2		0	S	1
				0				0				0				0

8		9		10		11		12		13		14		15	
y*z▶		*z▶		*z▶		*z▶		*z▶		*z ▶		*z ▶		*z ▶	
7		5		7		3		7		10		0		1	
S5		**R**1,2		**G**3		**R**1,1		**G**10		**R**3,0		**G**1		**Error**	
+	7	y	5	+	7	V	3	+	7	T	10		0	S	1
S	1	+	7	S	1	+	7	S	1	+	7				0
	0	S	1		0	S	1		0	S	1				
			0				0				0				

Согласно данной истории автомат отвергает правильную цепочку, т. е. не может восстановить ее дерево разбора.

Вторая история для той же входной цепочки вплоть до такта 12 выглядит точно так же, как первая. Различия начинаются с такта 13. Вместо операции свертки **R**3,0 выполняется конфликтующая с ней операция сдвига **S**8. Продолжение этой истории выглядит так:

13		14		15		16		17		18		19		20		21	
*z▶		z▶		▶		▶		▶		▶		▶		▶		▶	
10		8		6		8		11		7		10		0		1	
S8		**S**6		**R**1,2		**G**11		**R**3,1		**G**10		**R**3,0		**G**1		**Stop**	
T	10	*	8	y	6	*	8	V	11	+	7	T	10		0	S	1
+	7	T	10	*	8	T	10	*	8	S	1	+	7				0
S	1	+	7	T	10	+	7	T	10		0	S	1				
	0	S	1	+	7	S	1	+	7				0				
			0	S	1		0	S	1								
				0					0						0		

Согласно этой истории автомат восстанавливает дерево граммматического разбора для правильной входной цепочки символов.

Причина различий в поведении автоматов состоит в следующем. Выполнение операции свертки в состоянии 10 при входном символе * эквивалентно преобразованию промежуточного уровня восстанавливаемого дерева разбора, имеющего вид

$$\ldots S + T * \ldots$$

в цепочку вида

$$\ldots S * \ldots$$

В этой цепочке сразу после нетерминала *S* следует знак операции умножения *. Однако множество последователей нетерминала *S* содержит только символы +, *)* и ▶. Символа * в этом множестве нет.

Следовательно, в процессе восстановления дерева разбора выполнять любую операцию свертки имеет смысл только в том случае, если в текущем состоянии автомата его входным символом является элемент множества последователей нетерминала, образующегося в результате свертки.

Поэтому при построении управляющей таблицы автомата сформированные на шаге 3 знаки операций свертки следует заносить только в те клетки строки данного состояния, которые находятся на пересечении со столбцами, соответствующими терминальным символам множества последователей нетерминала из левой части правила.

Используя определенные в разд. 1.8 множества последователей нетерминалов, применим модифицированный шаг 3 процедуры преобразования таблицы конфигураций в управляющую таблицу автомата к грамматике $\boldsymbol{G}_{a1}$. Теперь не возникают ранее фиксировавшиеся конфликты типа сдвиг/свертка:

	0	1	2	3	4	5	6	7	8	9
	S	*T*	*V*	+	*	*(*	*)*	*i*	*c*	▶
0	**G**1	**G**2	**G**3			**S**4		**S**5	**S**6	
1				**S**7						**Stop**
2				**R**1,0	**S**8		**R**1,0			**R**1,0
3				**R**1,1	**R**1,1		**R**1,1			**R**1,1
4	**G**9	**G**2	**G**3			**S**4		**S**5	**S**6	
5				**R**1,2	**R**1,2		**R**1,2			**R**1,2
6				**R**1,2	**R**1,2		**R**1,2			**R**1,2
7		**G**10	**G**3			**S**4		**S**5	**S**6	
8			**G**11			**S**4		**S**5	**S**6	
9				**S**7			**S**12			
10				**R**3,0	**S**8		**R**3,0			**R**3,0
11				**R**3,1	**R**3,1		**R**3,1			**R**3,1
12				**R**3,2	**R**3,2		**R**3,2			**R**3,2

Данный автомат является детерминированным.

Для грамматики $\boldsymbol{G}_{a1}$ при построении управляющей таблицы не возникали конфликты типа свертка/свертка. В том случае, если набор конфигураций некоторого состояния содержит две или более конфигураций вида

$$N : \beta\blacktriangledown$$

$$M : \beta\blacktriangledown,$$

занесение знаков операций свертки без использования множеств последователей нетерминалов N и M приведет к возникновению конфликтов типа свертка/свертка в каждой клетке данного состояния. При использовании множеств последователей для занесения знаков операций свертки в таблицу возникновение конфликтов зависит от того, пересекаются ли $M_{\text{посл}}(N)$ и $M_{\text{посл}}(M)$. Если их пересечение пусто, то в данном состоянии конфликты типа свертка/свертка не возникнут.

Необходимо отметить, что использование множеств последователей для занесения знаков операций свертки в управляющую таблицу восходящего синтаксического акцептора автоматически разрешает вопрос о возможности использования в грамматике правил вида $N : \boldsymbol{\varepsilon}$.

Действительно, конфигурация вида $N : \boldsymbol{\varepsilon}\ \blacktriangledown$ (или эквивалентная ей $N : \blacktriangledown$), требующая выполнения свертки пустой цепочки в нетерминал N, появляется в таблице конфигураций только для тех состояний, для которых это диктуется совокупностью всех правил грамматики. Знаки операций свертки по таким правилам появятся в управляющей таблице именно в этих состояниях и только в тех столбцах, которые помечены терминалами, принадлежащими множеству последователей N.

Отметим, что отсутствие лишних знаков операции свертки в клетках столбцов управляющей таблицы, помеченных терминалами, не входящими в множества последователей соответствующих нетерминалов, обеспечивает более быстрое обнаружение ошибок в неправильных входных цепочках. Для того чтобы в этом убедиться, достаточно промоделировать работу автоматов с различными управляющими таблицами при обработке такой входной цепочки:

$$x\ (\ y + z\ )\blacktriangleright$$

Автомат, в котором знаки операций свертки расставлены без учета множеств последователей, обнаружит ошибку, выполнив 7 тактов:

Такт	0		1		2		3		4		5		6		7	
Вход	*x(y+z)*▶		*(y+z)*▶		*(y+z)*▶		*(y+z)*▶		*(y+z)*▶		*(y+z)*▶		*(y+z)*▶		*(y+z)*▶	
Сост.	0		5		0		3		0		2		0		1	
Опер.	**S**5		**R**1,2		**G**3		**R**1,1		**G**2		**R**1,0		**G**1		**Error**	
Стеки		0	*x*	5		0	*V*	3		0	*T*	2		0	*S*	1
				0				0				0				0

Автомат, управляемый таблицей, приведенной в данном пункте, обнаружит ошибку на такте номер 1:

Такт	0		1	
Вход	*x(y+z)*▶		*(y+z)*▶	
Сост.	0		3	
Опер.	**S**5		**Error**	
Стеки		0	*x*	3
				0

Легко видеть, что в первой истории уже самая первая операция свертки, выполняемая на такте 1, не имеет смысла. После идентификатора, сворачиваемого в нетерминал V (далее – в T и затем в S), в правильном предложении не может следовать открывающая скобка. Этот факт обнаруживается во второй истории работы немедленно, что и приводит к останову по ошибке на такте 1.

3.8. LR(0)- и SLR(1)-грамматики и автоматы

Приведем ряд общепринятых определений и обозначений теории восходящего синтаксического акцепта.

Если при преобразовании грамматики ***G*** в управляющую таблицу восходящего синтаксического акцептора не возникает ни одного конфликта типа сдвиг/свертка или свертка/свертка даже без использования множеств последователей нетерминальных символов для расстановки знаков операций свертки, то ***G*** называется **LR(0)**-грамматикой.

Здесь буква **L** является сокращением английского слова Left и означает, что входная цепочка символов обрабатывается слева направо. Буква **R** есть сокращение слова Right и означает, что история работы автомата отражает процесс восстановления так называемого обратного правого дерева грамматического разбора.

Число в скобках обозначает наименьшую длину подцепочек входных символов, необходимых для разрешения или предотвращения всех конфликтов типа сдвиг/свертка или свертка/свертка.

Обозначение **LR(0)**-грамматик свидетельствует о том, что при их использовании для построения управляющей таблицы восходящего синтаксического акцептора конфликты не возникают вообще.

Грамматики, относящиеся к классу **LR(0)** существуют, в чем можно убедиться при построении восходящего акцептора для следующей системы порождающих правил:

Грамматика G_{a0}	
0	*Z* : *S* ▶
1	*S* : *T*
2	*S* : *S* + *T*
3	*T* : (*S*)
4	*T* : *ident*
5	*T* : *const*

Однако синтаксис типичных языков программирования не может быть определен **LR(0)**-грамматикой. Этот класс грамматик слишком узок и может представлять собой только теоретический интерес.

Если при преобразовании грамматики в управляющую таблицу восходящего синтаксического акцептора все конфликты типа сдвиг/свертка или свертка/свертка удается разрешить (или предупредить) с использованием множеств последователей нетерминальных символов, содержащих цепочки терминалов длины 1, то грамматика относится к классу **SLR(1)**-грамматик.

Буква **S** в этом обозначении означает сокращение английского слова Simple – простая.

Преобразование грамматики G_{a1} в управляющую таблицу восходящего акцептора потребовало, как мы убедились, использования множеств последователей нетерминалов, содержащих одиночные терминальные символы. Поскольку при этом все конфликты были предупреждены, данная грамматика относится к классу **SLR(1)**.

К сожалению, для большинства современных языков программирования не существует порождающих грамматик класса **SLR(1)**. Поэтому рассмотрим более широкий класс грамматик, на основе которых может быть организовано восходящее детерминированное восстановление дерева разбора. Этот класс грамматик называют **LR(1)**-грамматиками.

3.9. Ожидаемый правый контекст и LR(1)-автоматы

В качестве примера рассмотрим построение восходящего акцептора на основе грамматики $\boldsymbol{G}_{LR}$, которая определяет предложения такого вида:

[*[*…]]*ident*

или такого вида:

[*[*…]]*ident* = [*[*…]]*ident*,

где […] означает не обязательно присутствующую часть.

Совокупность порождающих правил выглядит так:

Грамматика $\boldsymbol{G}_{LR}$	
0	*Z* : *L* ►
1	*L* : *E* = *R*
2	*L* : *R*
3	*R* : *E*
4	*E* : * *R*
5	*E* : *ident*

Таблица конфигураций, построенная для этой грамматики по уже известной процедуре, выглядит следующим образом:

Таблица конфигураций на основе грамматики G_{LR}

Состояние	*Образовано*		*База*	*Конфигурация*	*Символ*	*Отм.*
	Из	*Через*				
0			Да	*Z* : ▼ *L* ►	*L*	1
				L : ▼ *E* = *R*	*E*	2
				L : ▼ *R*	*R*	3
				E : ▼ * *R*	*	4
				E : ▼ *ident*	*ident*	5
				R : ▼ *E*	*E*	2
1	0	*L*	Да	*Z* : *L* ▼ ►	►	
2	0	*E*	Да	*L* : *E* ▼ = *R*	=	6
			Да	*R* : *E* ▼		
3	0	*R*	Да	*L* : *R* ▼		
4	0, 4, 6	*	Да	*E* : * ▼ *R*	*R*	7
				R : ▼ *E*	*E*	8
				E : ▼ * *R*	*	9
				E : ▼ *ident*	*Ident*	10
5	0, 4, 6	*i*	Да	*R* : *ident* ▼		

О к о н ч а н и е т а б л и ц ы

Состояние	*Образовано*		*База*	*Конфигурация*	*Символ*	*Отм.*
6	2	=	Да	*L* : *E* = ▼ *R*	*R*	11
				R : ▼ *E*	*E*	12
				E : ▼ * *R*	*	13
				E : ▼ *ident*	*ident*	14
7	4	*R*	Да	*E* : * *R* ▼		
8	4, 6	*E*	Да	*R* : *E* ▼		
9	6	*R*	Да	*L* : *E* = *R* ▼		

Вычислим множества последователей нетерминалов грамматики:

$$M_{\text{посл}}(L) = \{▶\},$$

$$M_{\text{посл}}(E) = \{=,\ ▶\},$$

$$M_{\text{посл}}(R) = \{=,\ ▶\},$$

и заметим, что символ = входит в множество последователей нетерминала *R*.

При преобразовании таблицы конфигураций в управляющую таблицу будет обнаружен конфликт, который невозможно разрешить путем использования множеств последователей нетерминала, образующегося в результате свертки (конфликтная клетка выделена серым фоном):

	0	1	2	3	4	5	6
	L	*R*	*E*	*ident*	=	*	▶
0	**G**1	**G**3	**G**2	**S**5		**S**4	
1							**Stop**
2					**S**6		**R**1,1
3							**R**1,0
4		**G**7	**G**8	**S**5		**S**4	
5					**R**1,2		**R**1,2
6		**G**9	**G**8	**S**5		**S**4	
7					**R**2,2		**R**2,2
8					**R**1,1		**R**1,1
9							**R**3,0

В множестве конфигураций состояния 2 конфигурация *L* : *E* ▼ = *R* требует выполнения операции сдвига, если входной символ есть =.

В то же время конфигурация R : E требует выполнение операции свертки, если входной символ принадлежит множеству {=, ▶} последователей нетерминала R.

Изучение правил грамматики, таблицы конфигураций и моделирование поведения автомата позволяют сделать вывод о том, что разрешение этого конфликта в пользу операции свертки было бы неправильным решением, поскольку после ее выполнения из цепочки

$$\ldots E = \ldots ,$$

которую можно привести к виду $E = R$, т. е. к правой части правила 1, возникла бы цепочка символов

$$\ldots R = \ldots ,$$

для которой уже не существует последовательности сверток в единственный начальный нетерминал L.

Действительно, пусть имеется очень простое правильное предложение данного языка:

$x = z$▶

Автомат, в таблице которого конфликт разрешен в пользу операции сдвига, будет обрабатывать его в соответствии с приведенной ниже историей работы и остановится по окончании восстановления дерева грамматического разбора:

Такт	0		1		2		3		4		5		6		7		8		9		10		11	
Вход	x=z▶		=z▶		=z▶		=z▶		z▶		▶		▶		▶		▶		▶		▶		▶	
Сост.	0		5		0		2		6		5		6		8		6		9		0		1	
Опер.	**S**5		**R**1,2		**G**2		**S**6		**S**5		**R**1,2		**G**8		**R**1,1		**G**9		**R**3,0		**G**1		**Stop**	
Стеки		0	x	5		0	E	2	=	6	z	5	=	6	E	8	=	6	R	9		0	L	1
				0				0	E	2	=	6	E	2	=	6	E	2	=	6				0
										0	E	2		0	E	2		0	E	2				
												0				0				0				

Но если бы при построении управляющей таблицы конфликт был разрешен в пользу операции свертки, то история работы автомата для данной входной цепочки выглядела бы так:

Такт	0		1		2		3		4		5	
Вход	$x=z$▶		$=z$▶		$=z$▶		$=z$▶		$=z$▶		$=z$▶	
Сост.	0		5		0		2		0		3	
Опер.	**S5**		**R1,2**		**G2**		**R1,1**		**G3**		**Error**	
Стеки		0	x	5		0	E	2		0	R	3
				0				0				0

Согласно этой истории после шагов 3 и 4 текущий уровень дерева разбора представляет собой цепочку $R = z$▶. Не существует способа вывода такой цепочки из начального нетерминала грамматики. Поэтому, несмотря на то что терминал = принадлежит множеству последователей нетерминала R, этот символ не ожидается на входе автомата после свертки согласно операции **R**1,1, выполняемой в состоянии 2.

Следовательно, конфликт между сдвигом и сверткой, возникающий при построении управляющей таблицы в этом состоянии, следует разрешать в пользу знака операции сдвига. Обратим внимание также и на то, что в состоянии 8 свертка по этому же правилу не конфликтует с операцией сдвига в колонке, помеченной этим же терминалом.

Отсюда возникает идея: при построении таблицы конфигураций нужно связать с каждой конфигурацией, требующей выполнения операции свертки, такие множества терминальных символов, которые могут действительно ожидаться на входе автомата после выполнения операции свертки.

Очевидно, что эти множества должны быть подмножествами полного множества последователей для каждого нетерминала, находящегося в левой части любой такой конфигурации.

Достаточно ясно также, что определять такие множества придется для всех конфигураций в таблице. Это следует из того, что любое состояние восходящего акцептора определяется взаимосвязанным набором конфигураций, перечень которых зависит, во-первых, от того, как (из какого состояния и через какой символ) была сформирована база, а во-вторых, от того, каково множество конфигураций замыкания базы.

Множество символов, допустимых на входе автомата после выполнения операции свертки в любом состоянии (и, естественно, после операции Go, обрабатывающей нетерминал, полученный в результате этой свертки), в литературе принято называть ожидаемым правым контекстом. Конфигурацию, к которой приписан в точности один символ ожидаемого правого контекста, будем называть канонической расширенной конфигурацией:

$$N : \alpha \blacktriangledown \beta, \{ t \}$$

Выясним правила формирования ожидаемого правого контекста.

1. Ожидаемый правый контекст базовой конфигурации нулевого состояния состоит из псевдотерминального символа ▶ конца текста (файла). Действительно, если входное предложение является правильным и восходящему акцептору удалось свернуть его в начальный нетерминал грамматики, то после этой свертки на входе автомата никаких других символов быть не может.

2. Перенос маркера через любой символ (образование базы другого состояния, а при работе автомата – выполнение операции Shift или Go) порождает конфигурацию, правый контекст которой совпадает с правым контекстом исходной конфигурации. Действительно, если взять базовую конфигурацию нулевого состояния и перенести маркер через начальный нетерминал (выполнить операцию Go после свертки правильного предложения в начальный нетерминал грамматики), то на входе автомата должен появиться псевдотерминал конца текста.

3. Осталось выяснить, каким образом формируется ожидаемый правый контекст при выполнении замыкания базового набора конфигураций произвольного состояния. Суть процесса замыкания, как определялось ранее, состоит в следующем.

Если имеется конфигурация вида $N : \alpha \blacktriangledown X \beta$, то каждое правило для нетерминала X превращается в конфигурацию путем установки маркера перед первым символом правой части и добавляется к множеству конфигураций данного состояния, если ее в нем еще нет:

$$X : \gamma_1 \rightarrow X : \blacktriangledown \gamma_1$$

$$X : \gamma_2 \rightarrow X : \blacktriangledown \gamma_2$$

$$\ldots$$

$$X : \gamma_n \rightarrow X : \blacktriangledown \gamma_n$$

При построении таблицы канонических расширенных конфигураций с любой исходной конфигурацией $N : \alpha \blacktriangledown X \beta$ связан правый контекст, содержащий символ t_0, ожидаемый на входе автомата после свертки цепочки $\alpha X \beta$ в нетерминал N:

$$N : \alpha \blacktriangledown X \beta \, , \{t_0\}.$$

Очевидно, что после свертки любой из цепочек γ_i в нетерминал X на входе автомата ожидаются символы, входящие в множество предшественников цепочки β, а если эта цепочка состоит только из анну-

лируемых нетерминалов или вообще пуста – еще и символ t_0. Таким образом, определение множества ожидаемых символов при замыкании любой конфигурации, производится по формуле:

$M_{опк}(X : ▼\gamma_i) = M_{пред}(\beta)$, если β – цепочка, содержащая хотя бы один терминал или неаннулируемый нетерминал;

$M_{опк}(X : ▼\gamma_i) = \{ t_0 \}$, если β – пустая цепочка;

$M_{опк}(X : ▼\gamma_i) = M_{пред}(\beta) \cup \{ t_0 \}$, если β – цепочка, состоящая только из аннулируемых нетерминалов.

Получив множество символов ожидаемого правого контекста $M_{опк}(\alpha ▼ X \beta , \{t_0 \}) = \{ t_1, t_2 \dots t_k\}$, легко можно построить все дочерние по отношению к исходной канонические конфигурации:

$M : ▼\gamma_1 , \{ t_1\}$ $M : ▼\gamma_1 , \{ t_2\}$ … $M : ▼\gamma_1 , \{ t_k\}$

$M : ▼\gamma_2 , \{ t_1\}$ $M : ▼\gamma_2 , \{ t_2\}$ … $M : ▼\gamma_2 , \{ t_k\}$

...

$M : ▼\gamma_n , \{ t_1\}$ $M : ▼\gamma_n , \{ t_2\}$ … $M : ▼\gamma_n , \{ t_k\}$

Теперь окончательно сформулируем правила построения таблицы канонических конфигураций.

1. Базой нулевого состояния является конфигурация вида

$$Z : ▼S ▶ , \{▶\}$$

где S – начальный нетерминал грамматики, ▶ – псевдотерминал «конец файла».

2. При формировании замыкания каждая вновь построенная каноническая конфигурация добавляется к множеству конфигураций данного состояния, если в этом множестве еще нет в точности такой конфигурации (с учетом ожидаемого правого контекста).

3. При формировании базы новое состояние образуется, если нет ни одного состояния с точно таким же базовым набором канонических конфигураций. Еще раз подчеркнем, что канонические конфигурации различаются, если их помеченные правила одинаковы, но различны символы ожидаемого правого контекста.

Построим по грамматике $\boldsymbol{G}_{LR}$ таблицу канонических конфигураций. Для символов ожидаемого правого контекста добавим отдельный столбец в таблицу конфигураций, озаглавленный сокращением ОПК.

Первым действием является образование базовой канонической конфигурации нулевого состояния:

$$Z : \blacktriangledown S \blacktriangleright , \{\blacktriangleright\} .$$

При выполнении замыкания базового набора конфигураций нулевого состояния вначале образуются канонические конфигурации:

$$L : \blacktriangledown\ E = R , \{\blacktriangleright\}$$

$$L : \blacktriangledown\ R , \{\blacktriangleright\} .$$

Затем, поскольку в первой из образованных конфигураций маркер находится перед нетерминалом *E*, формируются две конфигурации:

$$E : \blacktriangledown * R , \{=\}$$

$$E : \blacktriangledown\ ident , \{=\} .$$

Далее, поскольку в конфигурации $L : \blacktriangledown\ R$ маркер находится перед нетерминалом *R*, формируется конфигурация

$$R : \blacktriangledown\ E , \{\blacktriangleright\} .$$

В этой конфигурации маркер находится перед нетерминалом *E*, поэтому образуются две конфигурации, уже присутствующие в множестве конфигураций состояния 0, но отличающиеся от ранее образованных другим правым контекстом:

$$E : \blacktriangledown\ * R , \{\blacktriangleright\}$$

$$E : \blacktriangledown\ ident , \{\blacktriangleright\} .$$

Эти конфигурации добавляются к множеству конфигураций состояния 0. На этом процесс замыкания множества конфигураций состояния 0 заканчивается, поскольку никаких новых конфигураций образовано быть не может.

По обычным правилам формируем базу состояния 1 путем переноса маркера через символ *L* в конфигурации $Z : \blacktriangledown\ L \blacktriangleright, \{\blacktriangleright\}$. Замыкание базы состояния 1 не порождает новых конфигураций.

В базу состояния 2 попадают конфигурации:

$$L : E \blacktriangledown\ = R, \{\blacktriangleright\}$$

$$R : E \blacktriangledown, \{\blacktriangleright\},$$

причем вторая из них предполагает выполнение свертки только в том случае, если текущим входным символом является ▶ (конец файла).

Существенные различия начинаются после построения состояния 9. В отличие от таблицы обычных конфигураций в тот момент, когда из конфигурации R : ▼ E, {►} состояния 4 образуется база вида R : E ▼, {►}, создается состояние 10, поскольку в точности такой же базы в таблице нет. Есть состояние 8, имеющее две базовые конфигурации R : ▼ E, {=} и R : ▼ E, {►}, однако это множество базовых конфигураций отличается от вновь образованного. После возникновения состояния 10 далее будут построены еще три состояния. Результат формирования таблицы канонических конфигураций выглядит так:

Таблица канонических расширенных конфигураций

Состояние	*Образовано*		*База*	*Конфигурация*	*ОПК*	*Символ*	*Отм.*
	Из	*Через*					
0			Да	*Z* : ▼ *L* ►	►	*L*	1
				L : ▼ *E* = *R*	►	*E*	2
				L : ▼ *R*	►	*R*	3
				E : ▼ * *R*	=	*	4
				E : ▼ *ident*	=	*ident*	5
				R : ▼ *E*	►	*E*	2
				E : ▼ * *R*	►	*	4
				E : ▼ *ident*	►	*ident*	5
1	0	*L*	Да	*Z* : *L* ▼ ►	►	►	
2	0	*E*	Да	*L* : *E* ▼ = *R*	►	=	6
			Да	*R* : *E* ▼	►		
3	0	*R*	Да	*L* : *R* ▼	►		
4	0,4	*	Да	*E* : * ▼ *R*	=	*R*	7
			Да	*E* : * ▼ *R*	►	*R*	7
				R : ▼ *E*	=	*E*	8
				R : ▼ *E*	►	*E*	8
				E : ▼ * *R*	=	*	9
				E : ▼ *ident*	=	*ident*	10
				E : ▼ * *R*	►	*	9
				E : ▼ *ident*	►	*ident*	10
5	0,4	*id*	Да	*E* : *ident* ▼	=		
			Да	*E* : *ident* ▼	►		
6	2	=	Да	*L* : *E* = ▼ *R*	►	*R*	11
				R : ▼ *E*	►	*E*	12
				E : ▼ * *R*	►	*	13
				E : ▼ *ident*	►	*ident*	14

Окончание таблицы

Состояние	*Образовано*		*База*	*Конфигурация*	*ОПК*	*Символ*	*Отм.*
	Из	*Через*					
7	4	*R*	Да	*E* : * *R* ▼	=		
			Да	*E* : * *R* ▼	▶		
8	4	*E*	Да	*R* : *E* ▼	=		
			Да	*R* : *E* ▼	▶		
9	6	*R*	Да	*L* : *E* = *R* ▼	▶		
10	6,11	*E*	Да	*R* : *E* ▼	▶		
11	6,11	*	Да	*E* : * ▼ *R*	▶	*R*	15
				R : ▼ *E*	▶	*E*	16
				E : ▼ * *R*	▶	*	17
				E : ▼ *ident*	▶	*ientd*	18
12	6,11	*id*	Да	*E* : *ident* ▼	▶		
13	11	*R*	Да	*E* : * *R* ▼	▶		

Эта таблица обычным образом преобразуется в управляющую таблицу автомата, причем знаки операций свертки заносятся только в те клетки, колонки которых помечены терминалами из ожидаемого правого контекста соответствующих канонических конфигураций:

	0	1	2	3	4	5	6
	L	*R*	*E*	*ident*	=	*	▶
0	**G**1	**G**3	**G**2	**S**5		**S**4	
1							**Stop**
2					**S**6		**R**1,1
3							**R**1,0
4		**G**7	**G**8	**S**5		**S**4	
5					**R**1,2		**R**1,2
6		**G**9	**G**10	**S**12		**S**11	
7					**R**2,2		**R**2,2
8					**R**1,1		**R**1,1
9							**R**3,0
10							**R**1,1
11		**G**13	**G**10	**S**12		**S**11	
12							**R**1,2
13							**R**2,2

В данном случае не возникают ни конфликты типа «сдвиг/свертка», ни конфликты типа «свертка/свертка». Достигнуто это за счет увеличения количества состояний восходящего акцептора. Состояние 2 «расщепилось» на состояния с номерами 2 и 10, конфликт, который невозможно было разрешить путем использования полных множеств последователей нетерминалов, образующихся в результате свертки, не возник вообще.

Грамматики, для которых детерминированный восходящий синтаксический акцептор может быть построен только с использованием канонических конфигураций, называются **LR(1)**-грамматиками.

Однако грамматика $\boldsymbol{G}_{LR}$ на самом деле относится к промежуточному между **SLR(1)**- и **LR(1)**- грамматиками классу грамматик – так называемым **LALR(1)**-грамматикам. Буквы **LA** в названии класса грамматик являются сокращением от английского термина «lookahead», который переводится как «предварительный просмотр». В данном случае речь идет о предварительном (выполняемом при построении управляющей таблицы автомата) просмотре множеств символов ожидаемого правого контекста для разрешения или предупреждения возможных конфликтов «сдвиг/свертка» и «свертка/свертка».

3.10. LALR(1)-грамматики и автоматы

Рассмотренный в предыдущем разделе метод построения восходящего синтаксического акцептора можно модифицировать, используя вместо канонических так называемые расширенные конфигурации. Расширенной называется конфигурация, с которой связано множество символов ожидаемого правого контекста:

$$N : \alpha \blacktriangledown \beta \text{ , } \{t_1, t_2, \dots t_k\}.$$

В отличие от канонических две расширенные конфигурации, имеющие одинаковое маркированное правило и разные множества символов ожидаемого правого контекста, могут быть объединены в одну конфигурацию путем слияния ОПК:

$$N : \alpha \blacktriangledown \beta \text{ , } \{t_1\} \text{ и } N : \alpha \blacktriangledown \beta \text{ , } \{t_2\} \text{ эквивалентны } N : \alpha \blacktriangledown \beta \text{ , } \{t_1, t_2\}.$$

Сформулируем две разных технологии построения таблицы расширенных конфигураций. Согласно одной технологии:

1) строится таблица канонических конфигураций, как в разд. 3.9;

2) просматриваются множества конфигураций каждого состояния. Если в них обнаруживаются пары конфигураций с одинаковым маркированным правилом, то каждая такая пара заменяется одной расширенной конфигурацией с объединенным ожидаемым правым контекстом. Если одна из конфигураций пары являлась базовой, то базовой должна быть и конфигурация, полученная в результате объединения пары конфигураций. Если изменилось множество символов ОПК какой-либо базовой конфигурации (обозначим ее $К_{исх}$), то пересчитываются ожидаемые правые контексты всех ее дочерних конфигураций, т. е. таких, для которых $К_{исх}$ является родительской либо при выполнении замыкании, либо при образовании другой базы. Процедура пересчета должна продолжаться рекурсивно до тех пор, пока фиксируются изменения ожидаемого правого контекста дочерних конфигураций;

3) просматриваются все возможные пары состояний. Если два состояния имеют одинаковые (по маркированным правилам) наборы базовых расширенных конфигураций, то эти состояния сливаются, ожидаемые правые контексты пар всех конфигураций с одинаковым маркированным правилом объединяются. При слиянии состояний одно из них удаляется, но в оставшемся состоянии должна быть сохранена информация о том, откуда образовалось удаляемое состояние, т. е. содержимое колонок «Из» и «Через». Затем, если изменилось множество символов ОПК какой-либо базовой конфигурации (обозначим ее $К_{исх}$), пересчитываются ожидаемые правые контексты всех ее дочерних конфигураций, т. е. таких, для которых $К_{исх}$ является родительской, либо при выполнении замыкания, либо при образовании другой базы. Процедура пересчета должна продолжаться рекурсивно до тех пор, пока фиксируются изменения ожидаемого правого контекста дочерних конфигураций.

Перед слиянием каждой пары состояний можно проверять, не приведет ли это слияние к конфликту, которого не было в исходной таблице канонических конфигураций. Просматриваются пересечения множеств символов ОПК всех возможных пар конфигураций, первая из которых принадлежит одному состоянию, а вторая – другому. Если пересечение не пусто, и конфигурации требуют свертки по разным правилам, то состояния слиянию не подлежат. Далее, если одна из конфигураций требует сдвига по терминалу, принадлежащему ожидаемому правому контексту другой конфигурации, то слияние тоже должно быть заблокировано.

Согласно такой технологии строятся восходящие синтаксические акцепторы в учебном пакете автоматизации проектирования трансляторов «Вебтранслаб».

Вторая технология заключается в следующем:

1) базой нулевого состояния является конфигурация вида

$$Z : \blacktriangledown S \blacktriangleright , \{\blacktriangleright\},$$

где S – начальный нетерминал грамматики, $\blacktriangleright$ – псевдотерминал «конец файла»;

2) при формировании замыкания каждая вновь построенная расширенная конфигурация добавляется к множеству конфигураций данного состояния, если в этом множестве еще нет конфигурации с точно таким же маркированным правилом. Если же в множестве конфигураций состояния уже есть конфигурация с точно таким же маркированным правилом (обозначим ее $К_{исх}$), то ее множество символов ожидаемого правого контекста объединяется с ОПК новой конфигурации. Если ожидаемый правый контекст конфигурации $К_{исх}$ изменился в результате объединения, то пересчитываются ожидаемые правые контексты всех тех конфигураций, для которых конфигурация $К_{исх}$ является родительской как при формировании новой базы, так и при замыкании множеств конфигураций. Процедура пересчета должна продолжаться рекурсивно до тех пор, пока не перестанут изменяться ожидаемые правые контексты дочерних конфигураций.

3) при формировании базы новое состояние образуется, если нет ни одного состояния с точно таким же (по маркированным правилам) базовым набором расширенных конфигураций. Если же существует состояние с точно таким же набором базовых расширенных конфигураций, что и вновь образованная база, то новое состояние не формируется. Вместо этого объединяются множества ОПК одинаковых по маркированным правилам базовых конфигураций. Если при этом изменился ожидаемый правый контекст какой-либо базовой конфигурации, то должны быть пересчитаны ОПК всех ее дочерних конфигураций. Процедура пересчета должна продолжаться рекурсивно до тех пор, пока не перестанут изменяться ожидаемые правые контексты дочерних конфигураций.

Обе технологии требуют выполнения большого объема вычислений при пересчете ожидаемых правых контекстов. Применение любой из этих технологий к грамматике $\boldsymbol{G}_{LR}$ позволит получить таблицу расширенных конфигураций.

Таблица расширенных конфигураций

Состояние	*Образовано*		*База*	*Конфигурация*	*ОПК*	*Символ*	*Отм*
	Из	*Через*					
0			Да	*Z* : ▼ *L* ▶	▶	*L*	1
				L : ▼ *E* = *R*	▶	*E*	2
				L : ▼ *R*	▶	*R*	3
				E : ▼ * *R*	= ▶	*	4
				E : ▼ *i*	= ▶	*i*	5
				R : ▼ *E*	▶	*E*	2
1	0	*L*	Да	*Z* : *L* ▼ ▶	▶	▶	
2	0	*E*	Да	*L* : *E* ▼ = *R*	▶	=	6
			Да	*R* : *E* ▼	▶		
3	0	*R*	Да	*L* : *R* ▼	▶		
4	0, 4, 6	*	Да	*E* : * ▼ *R*	= ▶	*R*	7
				R : ▼ *E*	= ▶	*E*	8
				E : ▼ * *R*	= ▶	*	9
				E : ▼ *i*	= ▶	*i*	10
5	0, 4, 6	*i*	Да	*R* : *i* ▼	= ▶		
6	2	=	Да	*L* : *E* = ▼ *R*	▶	*R*	11
				R : ▼ *E*	▶	*E*	12
				E : ▼ * *R*	▶	*	13
				E : ▼ *i*	▶	*i*	14
7	4	*R*	Да	*E* : * *R* ▼	= ▶		
8	4, 6	*E*	Да	*R* : *E* ▼	= ▶		
9	6	*R*	Да	*L* : *E* = *R* ▼	▶		

Построение управляющей таблицы восходящего синтаксического акцептора с использованием правого контекста вместо множеств последователей при занесении знаков операций свертки не приводит к возникновению конфликтов, поскольку в состоянии 2 операция свертки по правилу *R* : *E* должна выполняться только в том случае, если на входе – псевдотерминал «конец файла».

Управляющая таблица, построенная для грамматики $\boldsymbol{G}_{LR}$ по таблице расширенных конфигураций, полностью совпадает с той, которая была получена в разд. 3.9, и выглядит так:

	0	1	2	3	4	5	6
	L	*R*	*E*	*ident*	=	*	▶
0	**G**1	**G**3	**G**2	**S**5		**S**4	
1							**Stop**
2					**S**6		**R**1,1
3							**R**1,0
4		**G**7	**G**8	**S**5		**S**4	
5					**R**1,2		**R**1,2
6		**G**9	**G**8	**S**5		**S**4	
7					**R**2,2		**R**2,2
8					**R**1,1		**R**1,1
9							**R**3,0

Однако если ранее не было формальных оснований утверждать, что конфликт сдвиг/свертка в состоянии 2 действительно должен быть разрешен в пользу знака операции сдвига, то теперь этот конфликт просто не возникает, в состоянии 2 при входном символе = должна быть выполнена операция сдвига **S**6.

При использовании расширенных конфигураций для предупреждения конфликтов применяется тот же самый способ формирования базовых конфигураций состояний автомата, что и в случае формирования таблицы простых конфигураций. Поэтому размеры управляющей таблицы для любой грамматики будут одинаковы как при использовании расширенных конфигураций, так и без вычисления ожидаемого правого контекста. Грамматики, для которых удается построить детерминированный восходящий синтаксический акцептор с использованием расширенных конфигураций, называются **LALR(1)**-грамматиками.

Класс **LALR(1)**-грамматик более широк, чем класс **SLR(1)**, однако существуют языки (в том числе практически все известные языки программирования), для которых не может быть найдена порождающая **LALR(1)**-грамматика. В то же время доказано, что для любого детерминированного контекстно-свободного языка существует хотя бы одна порождающая **LR(1)**-грамматика. Тем не менее задача нахождения грамматики требуемого класса, даже если известно, что она должна существовать для данного языка, алгоритмически неразрешима.

В том случае, когда грамматика не принадлежит классу **LALR(1)**, следует выполнить тщательный анализ причин возникновения неразрешимых конфликтов. Возможно, в результате анализа будет выявлено, что формально неразрешимые конфликты могут быть разрешены в

пользу одного из конфликтующих знаков операции исходя из семантики языка. Приведем пример такого анализа для упрощенного фрагмента грамматики С-подобного языка, определяющего синтаксис условного оператора, для которого даже при использовании канонических конфигураций при построении восходящего акцептора возникает неразрешимый конфликт:

Грамматика $\boldsymbol{G}_{if}$	
0	*Z* : *S* ▶
1	*S* : *if (E) S X*
2	*S* : *i = E ;*
3	*X* : *else S*
4	*X* :

В этом фрагменте предполагается, что нетерминал *E* определяет логическое или арифметическое выражение.

При построении управляющей таблицы восходящего синтаксического акцептора в клетке состояния, содержащего конфигурации

X : ▼ *else S* , { *else* }

X : ▼ , { *else* },

возникает неразрешимый конфликт типа «сдвиг/свертка».

Существо этого конфликта сводится к ответу на вопрос о том, к какому слову *if* относится слово *else* в таком операторе:

if(a>=b) if (a=b) x=a; else x=(a+b)/2;

Естественной и общепринятой является такая семантика этого оператора, при которой слово *else* относится к ближайшему не закрытому операторными скобками слову *if*, т. е. его эквивалентная запись выглядит так:

if(a>=b) { if (a=b) x=a; else x=(a+b)/2; }

Для того чтобы восходящий синтаксический акцептор восстанавливал дерево разбора исходного оператора в соответствии с этой семантикой, необходимо разрешить конфликт в пользу операции сдвига.

Большинство преобразователей грамматик в восходящий акцептор предоставляет возможность прямо в правиле указывать, как должны быть разрешены возникающие конфликты. Способ использования такой возможности надо искать в документации по преобразователю. Например, преобразователь YACC позволяет это делать путем использования специальной прагмы в правиле 4, принудительно удаляющей

терминал *else* из ожидаемого правого контекста любой конфигурации, в которой маркер находится на месте %delete:

$$X : \%\text{delete } else$$

Преобразователь пакета Вебтранслаб позволяет удалять терминал из ожидаемого правого контекста (или из множества выбора при построении нисходящего синтаксического акцептора) с помощью специального тэга <exclude>:

```
<rule leftpart='X'>
   <exclude>else</exclude>
</rule>
```

3.11. Оценки сравнительных характеристик различных реализаций восходящего синтаксического акцепта

Восходящие синтаксические акцепторы применяются исключительно в автоматной реализации. Процедурная их реализация весьма затруднительна.

Для **LR(0)**-, **SLR(1)**- и **LALR(1)**-грамматик размеры управляющих таблиц автоматов будут одинаковыми независимо от того, применялись ли при их построении простые конфигурации или расширенные. Количество клеток в таблице можно оценить как квадрат суммы количества терминальных и нетерминальных символов грамматики.

В случае **LR(1)**-грамматик (канонические конфигурации) объем памяти, требуемой для управляющей таблицы автомата, может быть значительно (в разы) большим.

Управляющие таблицы восходящих синтаксических акцепторов при больших размерах (десятки и сотни тысяч клеток для грамматик языков программирования) являются слабо заполненными или разреженными. Применяя эффективные методы упаковки таких разреженных таблиц, не рассматриваемые в данном пособии, можно добиться значительного уменьшения объема памяти, требуемого для их хранения, практически без потерь производительности.

Восходящие акцепторы относятся к классу расширенных автоматов. Поэтому оценка затрат времени на восходящий синтаксический акцепт приблизительно такова же, как для нисходящего автомата с одним состоянием, рассмотренного в разд. 2.6.

Применительно к решению задач синтаксического анализа, рассматриваемых в следующем разделе, семантического анализа и генерации объектного кода восходящие автоматы предоставляют практически такие же возможности, как и нисходящие.

4. СОБСТВЕННО СИНТАКСИЧЕСКИЙ АНАЛИЗ

До этого момента рассматривалась только задача синтаксического акцепта, т. е. проверки правильности входной программы в целом. При создании транслятора для реального языка программирования синтаксический акцептор должен быть расширен до синтаксического анализатора для решения задач:

– обработки возможных синтаксических ошибок;

– преобразования правильной входной последовательности слов во внутреннее представление, удобное для последующего семантического анализа, оптимизации, генерации кода или интерпретации.

Решение этих задач обычно осуществляется путем расширения функциональности синтаксического акцептора, что и превращает акцептор в синтаксический анализатор. Семантический анализ и некоторые элементы генерации объектного кода также могут быть реализованы в виде расширения синтаксического акцептора.

Обработка синтаксических ошибок связана с формированием диагностических сообщений для программиста о точке обнаружения и характере ошибки. Кроме того, могут предприниматься меры по нейтрализации синтаксических ошибок, состоящие в организации поиска последующих действительно существующих ошибок путем попыток «исправления» входной цепочки. Заметим, что слово «исправления» взято в кавычки потому, что на самом деле исправить ошибочную входную цепочку синтаксический акцептор не в состоянии, он может только попытаться найти такой вариант ее модификации, чтобы продолжить поиск других возможных ошибок во входной цепочке. Задача нейтрализации ошибок решается не во всех трансляторах.

С точки зрения теории трансляции задача формирования диагностических сообщений интереса не представляет. На практике она сводится к составлению заготовок текстовых сообщений, в которые

должны быть вставлены координаты точки ошибки во входном тексте (например, в виде номера строки и номера символа в строке).

В некоторых трансляторах в диагностические сообщения вставляется перечень слов (терминальных символов), допустимых согласно синтаксису языка в точке обнаружения ошибки. Различные текстовые заготовки связываются с клетками управляющей таблицы синтаксического акцептора. Программная модель конечного автомата разрабатывается таким образом, чтобы при останове акцептора по ошибке можно было извлечь нужную заготовку, подставить в нее координаты ошибочного слова и, возможно, перечень допустимых слов, и отправить сформированный текст диагностического сообщения на заданное устройство вывода. При процедурной реализации акцептора аналогичные действия предусматриваются в тех точках функций, сопоставленных нетерминалам грамматики, где записаны операторы вида

return FALSE;.

Задачу преобразования входной программы в промежуточное внутреннее представление обычно относят к семантическому анализу. Однако, как будет показано в разд. 4.2, решение этой задачи основывается на дереве грамматического разбора, которое строится синтаксическим акцептором. Поэтому в настоящем пособии эта задача рассматривается именно как функция синтаксического анализатора.

4.1. Нейтрализация синтаксических ошибок

Сформулируем более четко постановку задачи нейтрализации синтаксических ошибок. Далее будем ориентироваться на автоматные реализации синтаксического акцептора, помня о том, что процедурная реализация рекурсивного спуска, по сути, эквивалентна автоматной.

Прежде всего, заметим, что начиная с момента обнаружения самой первой ошибки теряет смысл конечная цель процесса трансляции, т. е. формирование объектного кода при компиляции или обработка исходных данных при интерпретации. Поэтому в момент обнаружения ошибки следует заблокировать все процессы преобразований исходного текста и, возможно, ликвидировать результаты таких преобразований, которые были уже накоплены к этому моменту.

Продолжать обработку неправильного входного текста имеет смысл только в целях обнаружения и выдачи диагностических сообщений о других, но действительно существующих ошибках. Это тре-

бует определенной модификации синтаксического акцептора, который остановился в некотором состоянии при накопленном с момента старта содержании стека, имея на входе первый символ неправильного остатка предложения. Если просто перезапустить автомат, ничего не изменяя в его состоянии, то он немедленно остановится по той же самой ошибке. Автомат должен быть перезапущен, но требуется найти обоснованные ответы на следующие вопросы.

1. С какого символа входной цепочки следует продолжать работу?
2. Какое состояние должен иметь автомат в момент перезапуска?
3. Каким должно быть содержание стека в этот момент?

Реализация теоретически обоснованного изменения параметров автомата при перезапуске и называется попыткой нейтрализации. Как будет показано далее, не существует единственно правильного ответа на поставленные вопросы. Различных вариантов попытки нейтрализации ошибок может быть много.

При применении любой процедуры перезапуска автомата должен быть сформулирован критерий принятия решения о том, удалась ли попытка нейтрализации данной ошибки? Другими словами, должен быть известен ответ на вопрос: если после перезапуска автомат снова останавливается по ошибке, то при каких условиях следует попытаться применить некоторый другой вариант перезапуска автомата? При этом немедленно возникает вопрос о том, что делать в случае, когда исчерпаны все предусмотренные варианты?

Естественно, должен быть известен и ответ на такой вопрос: при каких условиях следующий после перезапуска останов автомата по ошибке должен интерпретироваться как обнаружение новой ошибки, для которой следует формировать другое диагностическое сообщение пользователю транслятора?

И, наконец, следует предусмотреть некое пороговое значение количества различных ошибок, по достижении которого процесс обработки текста должен быть прекращен полностью. В самом деле, если по какой-то причине транслятору предъявлен текст вовсе не на его входном языке, то, скорее всего, ошибочным будет признано буквально каждое слово. Более того, формируемая с такими затратами диагностика окажется просто никому не нужной.

Существует несколько методов нейтрализации ошибок. Вначале мы рассмотрим такие два, которые могут быть реализованы путем модификации программной модели любого автомата без изменения структуры его управляющих таблиц. Затем будут рассмотрены методы,

предполагающие «ручное» вмешательство. Под таким вмешательством понимается некоторое расширение порождающей грамматики, на основе которой строится соответствующим образом модифицированный автомат синтаксического акцептора.

4.1.1. Модификация одного символа

Будем исходить из того, что на входе транслятора имеется текст на его входном языке, т. е. ошибки, если они есть, являются одиночными. Под одиночной ошибкой будем понимать такую ситуацию, когда синтаксически неправильная цепочка терминальных символов может быть преобразована в правильную цепочку путем удаления одного символа или добавления одного символа или замены какого-либо символа другим. Приведем пример. Пусть рассматривается язык арифметических выражений и дана цепочка *(x + y * z* ▶, которую нельзя вывести из начального нетерминала никакой грамматики, порождающей данный язык. Эту цепочку можно преобразовать в правильную многими разными способами:

– удалив один символ *(*, можно получить синтаксически правильную цепочку *x + y * z* ▶ ;

– добавляя символ *)* в разные позиции исходной цепочки, можно получить такие правильные цепочки: *(x) + y * z* ▶, *(x + y) * z* ▶ и *(x + y * z)* ▶.

Никакими заменами одиночных символов (псевдотерминал ▶, очевидно, не может участвовать ни в заменах, ни в удалениях или добавлениях) из данной исходной неправильной цепочки нельзя получить правильную.

Заметим, что различные преобразования неправильной цепочки могут давать в результате синтаксически правильные, но по смыслу (семантически) различные цепочки, например *x + y * z* ▶ и *(x + y) * z* ▶. Никаких предположений о том, какая из этих цепочек имелась в виду автором исходного текста, сделать в процессе синтаксического акцепта невозможно. Именно поэтому в трансляторах делается только попытка нейтрализации ошибок, но не их исправления.

Приведем несколько примеров цепочек, в которых ошибка, казалось бы, не является одиночной: *(x + (y * z* ▶, *x + (y * +* ▶, *x + y) ** ▶, *) x + y) * +* ▶. Скоро мы увидим, что первые две цепочки отличаются от последних двух и что понятие одиночной ошибки должно быть уточнено.

Дело в том, что точка внесения изменений в неправильную цепочку при нейтрализации ошибок не может быть произвольной. Синтаксический акцептор на каждом шаге оперирует с единственным очередным символом входной цепочки и не должен возвращаться к ранее обработанным символам. Поэтому выше рассмотренные преобразования удаления, добавления (вставки) и замены символов могут применяться только к тому символу, при обработке которого акцептор остановился по ошибке.

Вернемся к рассмотрению цепочки *(x + y * z* ▶. Ошибка в ней может быть обнаружена только при обработке псевдотерминала ▶. Поэтому из всех приведенных вариантов преобразования в правильную цепочку может быть использован только вариант вставки символа *)* перед ▶, т. е.: *(x + y * z)* ▶, который при перезапуске любого автомата из того же состояния и с тем же содержимым стека приведет к успешному завершению работы. Заметим, что попытка вставить любой другой символ в точку обнаружения ошибки и перезапуск автомата приведут к его останову по ошибке на том же самом псевдотерминале ▶. Повторный останов по ошибке после перезапуска при обработке того же самого входного символа (или следующего при применении удаления или замены) следует трактовать как неудачу попытки нейтрализации ошибки. Напомним, что удалять или заменять псевдотерминал ▶ любыми другими символами нельзя по очевидным причинам.

При обработке цепочки *(x + (y * z* ▶ автомат точно так же остановится по ошибке при обработке псевдотерминала ▶. Однако вставка одиночного символа *)* перед ▶ и перезапуск автомата приведут к тому, что после обработки вставленной закрывающей скобки автомат снова остановится по ошибке на том же самом входном символе ▶. Никакие другие вставки символов во входную цепочку перед псевдотерминалом ▶ к успеху также не приведут. Ошибка не является одиночной, поскольку никакое добавление одного символа в точке обнаружения ошибки не приводит к успеху.

А теперь рассмотрим цепочку *x + y) ** ▶. При ее обработке любой акцептор остановится на входном символе *)*. Опуская прочие возможные варианты, заметим, что удаление этого терминала (перезапуск автомата из того же состояния с тем же содержимым стека) не приведет к останову при обработке символа *, следующего за удаленным. Будем считать, что нейтрализованной одиночной ошибкой являлась лишняя закрывающая скобка (хотя на самом деле, возможно, для исправления ошибки должна быть добавлена открывающая скобка *(* в начало це-

почки, но это решение уже является прерогативой автора входного текста) и продолжим восстановление дерева разбора измененной цепочки $x + y$ * ▶. Через некоторое количество шагов автомат опять остановится, но теперь при обработке псевдотерминала ▶. Поскольку ошибка обнаружена в другой точке входной цепочки (не по символу *, следующему за удаленным терминалом при нейтрализации первой ошибки), ее можно считать новой, т. е. самостоятельной ошибкой и также попытаться нейтрализовать. Легко видеть, что этого можно достигнуть, вставив идентификатор (любой, например x) перед символом ▶, в результате чего будет получена (якобы) правильная цепочка $x + y * x$ ▶. Вполне возможно, однако, что автор текста преследовал цель написать цепочку $(x + y) * z$ ▶.

Из результатов рассмотрения этих примеров можно определить общую процедуру попытки нейтрализации одиночной ошибки. Следует предварительно заметить, что если ошибка обнаруживается по входному символу ▶, то нейтрализовывать ее нет никакого смысла. Акцептором достигнут конец входной цепочки, других одиночных ошибок нет.

1. В момент обнаружения ошибки сформировать диагностическое сообщение, зафиксировать текущее состояние автомата, содержание его стека (перед любым перезапуском состояние и стек восстанавливаются) и позицию очередного символа во входной цепочке (для решения задач нейтрализации ошибок мы будем допускать возможность возврата к уже обработанным символам, но не ранее точки обнаружения ошибки). Принять меры по блокировке процессов преобразования входной цепочки и, возможно, по ликвидации ранее полученных результатов преобразований.

2. Удаление одного символа. Прочитать следующий символ из входной цепочки и перезапустить автомат. Если автомат останавливается при обработке этого символа (возможно, выполнив несколько тактов работы), то перейти к шагу **3**. В противном случае (при чтении очередного входного символа) считать нейтрализацию успешной путем удаления ошибочного символа, и вернуться к обычному режиму функционирования автомата.

3. Замена одного символа.

3.1. Взять первый символ из множества допустимых для данного состояния автомата (способы образования множеств допустимых символов для разных автоматов рассматриваются ниже).

3.2. Восстановить позицию очередного символа во входной цепочке, прочитать его, но установить текущим допустимый символ (тем самым обеспечивается замена ошибочного символа на какой-либо из допустимых). Перезапустить автомат. Если он остановится при обработке текущего символа или следующего, взятого из входной цепочки, то перейти к п. **3.3**. Иначе, если подставленный допустимый и следующий из входной цепочки символы будут обработаны, то считать нейтрализацию успешной путем замены символа, и вернуться к обычному режиму функционирования автомата.

3.3. Если исчерпаны все допустимые символы, то перейти к шагу 4, иначе взять следующий допустимый, и вернуться к п. **3.2**.

4. Вставка одного символа.

4.1. Взять первый допустимый символ, перейти к п. **4.2.**

4.2. Восстановить позицию очередного символа во входной цепочке, но текущим установить допустимый символ (тем самым обеспечивается вставка очередного допустимого символа перед ошибочным). Перезапустить автомат. Если он остановится при обработке текущего символа, или следующего (т. е. ошибочного при первом останове), взятого из входной цепочки, то перейти к п. **4.3**. Иначе, если подставленный допустимый и следующий из входной цепочки символы будут обработаны, то считать нейтрализацию успешной путем вставки символа, и вернуться к обычному режиму функционирования автомата.

4.3. Если исчерпаны все допустимые символы, то перейти к шагу 5, иначе взять следующий допустимый и вернуться к п. **4.2**.

5. Нейтрализация одиночной ошибки не удалась, переключиться в режим переполоха (или паники). Этот метод нейтрализации ошибок рассматривается в разд. 4.1.2.

Для окончательного уяснения принципов нейтрализации одиночных ошибок сформулируем понятие множества допустимых символов. В принципе приведенный выше алгоритм будет работать, даже если для любого состояния допустимым будет все множество терминальных символов грамматики. Однако для уменьшения временны́х затрат целесообразно для каждого состояния автомата определить только такой набор допустимых символов, использование которых для замены или вставки согласно шагам 3 и 4 алгоритма заведомо не обречено на неудачу. Дело в том, что такие наборы (множества) на самом деле уже связаны с состояниями автомата и никакой дополнительной работы по их вычислению выполнять не нужно.

Автомат нисходящего восстановления дерева разбора (см. раздел 2.5) в каждой клетке (состоянии) содержит множество выбора или ссылку на это множество. Именно символы из такого множества и являются допустимыми для большинства состояний. Легко видеть, что при перезапуске из любого состояния подстановка на вход любого символа, не принадлежащего множеству выбора этого состояния, автомат немедленно остановится по ошибке вновь. Нужно отметить, что такой автомат не может останавливаться по ошибке в некоторых состояниях. Таковы, к примеру, состояния 3 и 4. Попасть в состояние номер 3 (для которого останов вообще запрещен установленным флажком **e**) автомат может только в том случае, если текущим входным символом является либо +, либо *)*, либо ▶. Если на входе символ +, то автомат перейдет в состояние 14, иначе – в состояние 4 и далее – в состояние 17. Более того, остановы по ошибкам при использованных методах формирования множеств выбора возможны только в тех состояниях, в которые автомат попадает не ранее, чем будет прочитан очередной символ.

Для нисходящего автомата с одним состоянием (см. раздел 2.6) в момент останова по ошибке следует фиксировать не текущее состояние (оно одно), а как раз множество допустимых символов. Это множество однозначно и полностью определяется по той строке управляющей таблицы, которая помечена символом, находящимся в момент обнаружения ошибки на верхушке стека. Все те терминалы, которыми помечены столбцы таблицы с непустыми клетками в данной строке, и составляют множество допустимых символов для процедуры нейтрализации одиночной ошибки.

Для любого восходящего автомата множество допустимых символов совершенно аналогично определяется как множество терминалов, помечающих столбцы управляющей таблицы с непустыми клетками в строке того состояния, в котором автомат остановился по ошибке.

Вышеописанную процедуру можно модифицировать с целью получения наиболее надежных результатов нейтрализации одиночных ошибок за счет некоторого увеличения временны́х затрат. Суть этой модификации состоит в том, чтобы обеспечить проверку всех возможных вариантов (удаления, замены и вставки) с фиксацией позиции обнаружения следующей ошибки. После этого применяется тот вариант нейтрализации, для которого следующая ошибка (если она обнаруживается) находится на максимальном расстоянии от первичной.

Приведем примеры историй работы различных автоматов, включающие действия по нейтрализации одиночных ошибок для входной цепочки *(x +) * z* ▶ без перебора всех возможных вариантов.

Оптимизированный нисходящий автомат с несколькими состояниями (история неоптимизированного значительно длиннее, но принципиально ничем не отличается) будет работать следующим образом:

Такт	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Вход	*(*	*(*	*(*	*(*	*(*	*(*	*(*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	+	+	+	+	+	+	)	*	(
Сост.	0	2	11	5	15	8	19	20	2	11	5	15	8	9	22	16	6	7	12	3	13	14	14	14
Стек		1	1	12	12	16	16	16	21	21	12	12	16	16	16	12	12	12	21	21	21	21	21	21
				1	1	12	12	12	16	16	21	21	12	12	12	21	21	21	16	16	16	16	16	16
						1	1	1	12	12	16	16	21	21	21	16	16	16	12	12	12	12	12	12
									1	1	12	12	16	16	16	12	12	12	1	1	1	1	1	1
											1	1	12	12	12	1	1	1						
													1	1	1									

24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
(	*(*	*(*	*(*	*(*	*(*	*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*	*	*	*z*	*z*	*z*	*z*	*z*	*z*	▶
2	11	5	15	8	19	20	14	2	11	5	15	8	9	22	16	6	17	18	5	15	8	9	22	16
21	21	12	12	16	16	16	21	21	21	12	12	16	16	16	12	12	12	12	12	12	16	16	16	12
16	16	21	21	12	12	12	16	16	16	21	21	12	12	12	21	21	21	21	21	21	12	12	12	21
12	12	16	16	21	21	21	12	12	12	16	16	21	21	21	16	16	16	16	16	16	21	21	21	16
1	1	12	12	16	16	16	1	1	1	12	12	16	16	16	12	12	12	12	12	12	16	16	16	12
		1	1	12	12	12				1	1	12	12	12	1	1	1	1	1	1	12	12	12	1
				1	1	1						1	1	1							1	1	1	

На такте 21 автомат останавливается по обнаружению ошибки, согласно шагу 2 процедуры пытается удалить ошибочный символ *)* и при чтении следующего символа * вновь останавливается на такте 22. Согласно шагу 3 процедуры автомат пытается заменить ошибочный символ *)* каждым из допустимых для состояния номер 14 символов *(*, *i* и *c*. Подставив *(* вместо ошибочного символа и выполнив такты 23…29, автомат остановится на такте 30 по символу *, следующему за заменяемым. Теперь вместо ошибочного символа *)* делается попытка подставить идентификатор *x* (в качестве терминала *i*), т. е. входная цепочка преобразуется к виду: *(x + x * z* ▶. Начиная с такта 31 (для которого состояние автомата восстанавливается) и вплоть до такта 38 идет обработка идентификатора *x*, подставленного взамен ошибочного

символа и следующего за ним знака операции умножения. На такте 39 нейтрализация первой ошибки считается успешно завершенной, поскольку автомат переходит к обработке символа z. Далее вплоть до такта 47 идет обработка символа z. На такте 48 обнаруживается вторая ошибка во входной цепочке. Нейтрализовывать ее не имеет смысла, поскольку ошибка обнаружена в момент окончания входной цепочки.

В результате обработки этой цепочки будут сформированы два диагностических сообщения. Первое о том, что в точке, помеченной маркером ▼

$(x +$ ▼ $) * z$ ▶,

не ожидается закрывающая скобка, второе – о том, что перед концом файла ожидается закрывающая скобка. Заметим, что если бы применялся полный перебор всех возможных вариантов нейтрализации с применением наилучшего по максимальному расстоянию между ошибками, то вместо замены закрывающей скобки на идентификатор был бы использован вариант вставки идентификатора перед скобкой. При этом второй ошибки вообще не было бы обнаружено, а диагностика могла бы быть сформулирована так: ожидается идентификатор/константа в точке, помеченной маркером.

Нисходящий автомат с одним состоянием будет работать так:

Такт	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Вход	(	(	(	*x*	*x*	*x*	+	+	)	*	*x*	*x*	*x*	*	*z*	*z*	▶	▶	▶
Стек	*S*	*U*	*V*	*S*	*U*	*V*	*W*	*R*	*S*	*S*	*S*	*U*	*V*	*W*	*U*	*V*	*W*	*R*	)
	▶	*R*	*W*	)	*R*	*W*	*R*	)	)	)	)	*R*	*W*	*R*	*R*	*W*	*R*	)	*W*
		▶	*R*	*W*	)	*R*	)	*W*	*W*	*W*	*W*	)	*R*	)	)	*R*	)	*W*	*R*
			▶	*R*	*W*	)	*W*	*R*	*R*	*R*	*R*	*W*	)	*W*	*W*	)	*W*	*R*	▶
				▶	*R*	*W*	*R*	▶	▶	▶	▶	*R*	*W*	*R*	*R*	*W*	*R*	▶	
					▶	*R*	▶					▶	*R*	▶	▶	*R*	▶		
						▶							▶			▶			

Первая ошибка обнаруживается на такте 8 и после неудачной попытки нейтрализации удалением ошибочного символа (это обнаруживается на такте 9) успешно нейтрализуется заменой его на символ x (такты 10…13). Начиная с такта 14, автомат возвращается в нормальный режим работы, обнаруживает на такте 18 следующую ошибку и окончательно останавливается, поскольку достигнут конец входной цепочки.

Аналогично предыдущему варианту автомата будут формироваться и диагностические сообщения. Полный перебор вариантов нейтрализации за счет дополнительных затрат времени позволил бы уменьшить количество диагностических сообщений о сомнительных с точки зрения пользователя ошибках.

Восходящий **SLR(1)**-автомат эту цепочку будет обрабатывать так:

Такт	0	1	2	3	4	5	6	7	8	9	10	11
Вход	*(*	*x*	+	+	+	+	+	+	+	*)*	*	*
Сост.	0	4	5	4	3	4	2	4	9	7	7	4
Опер.	**S**4	**S**5	**R**1,2	**G**3	**R**1,1	**G**2	**R**1,0	**G**9	**S**7		**S**4	
Стек	0	4	5	4	3	4	2	4	9	7	7	4
		0	4	0	4	0	4	0	4	9	9	7
			0		0		0		0	4	4	9
										0	0	4
												0

Вплоть до момента обнаружения первой ошибки на такте 9 автомат работает обычным образом. На такте 10 делается попытка нейтрализации ошибки путем удаления символа *)*, однако на такте 11 при обработке следующего за удаленным символа автомат останавливается вновь. Поэтому на такте 12 состояние автомата и его стека восстанавливается и делается попытка заменить ошибочный символ идентификатором *x*.

12	13	14	15	16	17	18	19	20	21	22
x	*	*	*	*	*	*z*	▶	▶	▶	▶
7	5	7	3	7	10	8	5	8	11	7
S5	**R**1,2	**G**3	**R**1,1	**G**10	**S**8	**S**5	**R**1,2	**G**11	**R**3,1	
7	5	7	3	7	10	8	5	8	11	7
9	7	9	7	9	7	10	8	10	8	9
4	9	4	9	4	9	7	10	7	10	4
0	4	0	4	0	4	9	7	9	7	0
	0		0		0	4	9	4	9	
						0	4	0	4	
							0		0	

На такте 18 эта попытка признается успешной, и автомат возвращается к обычному режиму работы. Обнаружение следующей ошибки на такте 22 приводит к окончательному останову.

В завершение рассмотрения метода нейтрализации одиночных ошибок отметим, что его использование дает практически приемлемые результаты до тех пор, пока входные цепочки не очень сильно искажены по сравнению с правильными.

Возможно увеличение мощности этого метода за счет удаления (замены, вставки) не одного ошибочного символа, а цепочек длины 2, 3, … , начинающихся с этого символа. По существу, такой подход подводит нас к идее режима переполоха (иногда его называют методом паники), часто используемого разработчиками трансляторов и обсуждаемого в следующем разделе

4.1.2. Режим переполоха

Этот метод нейтрализации ошибок предусматривает просмотр без обработки некоторой части входной цепочки, начиная с ошибочного символа и до такого символа, после которого (возможно – включая этот символ) восстанавливается обычное функционирование автомата. Режим переполоха может включаться после неудачной попытки нейтрализации одиночной ошибки, но может в принципе использоваться и самостоятельно без предварительного применения других методов.

Существо этого метода вытекает из следующих рассуждений.

Ошибочный символ t во входной цепочке соответствует правильному символу p в подцепочке $\gamma\ \varphi$ текущего уровня восстанавливаемого дерева (символ p является предшественником φ), которая может быть выведена из некоторого нетерминала N. Другими словами, текущий уровень восстанавливаемого дерева разбора к моменту обнаружения ошибки может рассматриваться так:

$$S \Rightarrow \alpha\, N\, \beta \Rightarrow \alpha\, \gamma\ \blacktriangledown \varphi\, \beta\,,$$

где маркером ▼ обозначена точка, условно соответствующая позиции автомата (без различий между нисходящим и восходящим методами) на этом уровне.

Входное предложение в этот момент можно рассматривать следующим образом:

$$\omega = \alpha\, \gamma\, t\, \lambda\,,$$

где начало цепочки выводится из начала текущего уровня дерева (или полностью совпадает с ним), символ t – очередной символ, при обработке которого обнаружена ошибка (состоящая в том, что из $\varphi\ \beta$ не может быть выведена $t\ \lambda$).

Допустим, что цепочка $t\,\lambda$ может быть представлена в виде конкатенации двух цепочек

$$t\,\lambda = \mu\,\psi\,,$$

таких, что цепочка ψ выводится из β. Тогда, очевидно, цепочка ψ обязана начинаться с одного из терминальных предшественников цепочки β. Если заменить цепочку μ на цепочку φ, то, начиная с первого символа цепочки ψ, восстановление дерева разбора может быть продолжено в обычном режиме работы.

Теперь можно сформулировать общую идею (процедуру) метода.

1. В момент обнаружения ошибки нужно выявить какое-либо разбиение текущего уровня дерева разбора на цепочки $\alpha\ \gamma\ \varphi\ \beta$ (используя факт о существовании вывода $\alpha\ N\ \beta \Rightarrow \alpha\ \gamma\ \varphi\ \beta$). Казалось бы, что сделать это непросто, в силу того что автоматы, рассмотренные в разделе 2 и 3, не хранят дерево разбора или его уровни в явном виде. Однако, как будет показано далее, из текущего состояния стека автомата и его управляющей таблицы легко извлекается информация, необходимая для выполнения последующих шагов процедуры нейтрализации.

2. Вычислить множество предшественников цепочки β.

3. Выполнить просмотр входной цепочки до первого символа, входящего в это множество, не используя автомат синтаксического акцептора.

4. Модифицировать текущее состояние и содержимое стека автомата таким образом, как если бы им была полностью обработана цепочка, выводимая из $\gamma\ \varphi$, т. е. из нетерминала N. Для разных автоматов модификация выполняется по-своему и будет рассмотрена далее.

5. Восстановить обычное функционирование автомата.

Вместо множества предшественников цепочки β, которое всякий раз нужно вычислять в процессе обработки ошибки, можно использовать множество последователей нетерминала N (очевидно, содержащее множество предшественников β), которое все равно определяется в процессе преобразования грамматики в автомат. В этом случае увеличивается вероятность (на самом деле отличающаяся от нуля и в случае использования множества предшественников β) того, что ожидаемые символы входят в цепочку μ, предположительно выводимую из φ, и что один из таких символов встретится раньше, чем начало цепочки, действительно выводимой из β.

Другими словами, выход из режима переполоха в любом случае может быть произведен раньше, чем это действительно требуется. Со-

вершенно очевидно, что в таких случаях автомат рано или поздно вновь остановится по ошибке, причем, скорее всего, реально не существующей во входной цепочке. Необходимо предусматривать процедуры принятия решения о том, что делать в такой ситуации: считать обнаруженную ошибку самостоятельной или восстанавливать режим переполоха для продолжения нейтрализации первичной ошибки. Рассмотрение критериев для принятия такого решения выходит за рамки данного пособия.

Вернемся к вопросу о том, каким образом в момент обнаружения ошибки можно выполнить шаг 1, т. е. произвести подготовку к включению режима переполоха для различных типов автоматов.

Действительными целями этой подготовки являются:

– определение множества ожидаемых символов (либо предшественников цепочки β, либо последователей нетерминала N);

– определение состояния автомата, в котором он должен оказаться в момент выхода из режима переполоха;

– определение требуемого содержимого стека автомата к этому моменту.

Возможны различные варианты выполнения подготовки к включению режима переполоха. Вначале мы рассмотрим простейшие, а затем обсудим способы и цели их модификации.

Для нисходящего автомата с несколькими состояниями все эти сведения определяются очень просто, поскольку в любой момент его работы на верхушке стека находится номер состояния, в которое автомат должен вернуться после разбора цепочки, выводимой из нетерминала N, встретившегося в правой части какого-либо правила. Множество выбора этого состояния и есть множество ожидаемых символов. В момент выхода из режима переполоха (т. е. при появлении на входе любого из ожидаемых символов) с верхушки стека просто надо снять один номер состояния, установить его в качестве текущего и перезапустить автомат.

Для нисходящего автомата с одним состоянием в момент обнаружения ошибки необходимо просто удалить с верхушки стека один символ. Тем самым подготовлено требуемое состояние стека к моменту выхода из режима переполоха. Ожидаемыми будут символы, помечающие столбцы с непустыми клетками в той строке управляющей таблицы, которая озаглавлена символом, оказавшимся теперь на верхушке стека.

Для восходящего автомата аналогично простейшим способом подготовки является удаление одного номера состояния с верхушки стека.

В момент выхода из режима переполоха автомат должен быть установлен в состояние, номер которого находится теперь на верхушке стека. Ожидаемыми будут терминалы, помечающие столбцы с непустыми клетками в строке этого состояния управляющей таблицы.

Рассмотрим истории работы различных автоматов по обработке цепочки *(x +)) *) ((+ y* ▶, в которой ошибки не могут быть нейтрализованы ранее рассмотренным методом удаления/вставки/замены одного символа. Будем считать, что режим переполоха включается сразу же при обнаружении ошибки.

Нисходящий автомат с несколькими состояниями будет реализовывать такую историю работы:

Такт	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Вход	(	(	(	(	(	(	(	x	x	x	x	x	x	x	x	+	+	+	+	+	+	)
Сост.	0	2	11	5	15	8	19	20	2	11	5	15	8	9	22	16	6	7	12	3	13	14
Стек		1	1	12	12	16	16	16	21	21	12	12	16	16	16	12	12	12	21	21	21	21
				1	1	12	12	12	16	16	21	21	12	12	12	21	21	21	16	16	16	16
						1	1	1	12	12	16	16	21	21	21	16	16	16	12	12	12	12
									1	1	12	12	16	16	16	12	12	12	1	1	1	1
											1	1	12	12	12	1	1	1				
													1	1	1							

На такте 21 обнаруживается ошибка. Автомат входит в режим переполоха, в качестве ожидаемых символов берется множество выбора состояния 21, номер которого находится на верхушке стека. Это множество содержит единственный символ *)*. Продолжение истории работы выглядит так:

22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
)	)	*	*	*	)	(	(	+	+	+	+	y	y	y	y	y	y	y	y	▶
	21	16	6	17	18				12	3	13	14	2	11	5	15	8	9	22	16
	16	12	12	12	12				1	1	1	1	1	1	12	12	16	16	16	12
	12	1	1	1	1										1	1	12	12	12	1
	1																1	1	1	

На такте 22 именно этот символ (следующий за ошибочной закрывающей скобкой) обнаруживается во входной цепочке, поэтому начиная с такта 23, восстанавливается нормальный режим функционирования, причем в качестве текущего устанавливается состояние 21, номер которого снимается с верхушки стека. В результате применения режи-

ма переполоха входная цепочка обработана так, как если бы ее начало выводилось из *(S)**... .

На такте 27 автомат останавливается вновь. Теперь множество ожидаемых символов – это множество выбора состояния 12, содержащее терминалы *, +, *)* и ▶. На тактах 28…30 автомат пропускает входные символы вплоть до появления одного из ожидаемых.

Начиная с такта 31, восстанавливается нормальный режим работы автомата из состояния 12 (структура входной цепочки считается выводимой из *(S)+...)*. На такте 42 автомат можно остановить окончательно, поскольку новых ошибок не предвидится (достигнут конец входной цепочки), а доводить восстановление дерева до нормального завершения смысла уже нет.

В итоге для очень сильно искаженной (отличающейся от правильной) входной цепочки символов будет сформировано два диагностических сообщения:

первое будет указывать на точку, помеченную маркером

(x + ▼ *)) *) ((+ y* ▶ ,

и по смыслу (ожидается идентификатор/константа) соответствовать допущенной ошибке;
второе будет указывать на точку

*(x +)) ** ▼ *) ((+ y* ▶

и сообщать о том, что ожидается знак операции, закрывающая скобка (!) или конец файла, что никак не соответствует желаемому виду диагностики.

Нисходящий автомат с одним состоянием эту же цепочку обработает так:

Такт	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Вход	(	(	(	x	x	x	+	+	)	)	)	*	)	(	(	+	+	y	y	y	▶
Стек	S	U	V	S	U	V	W	R	S		)	W	U				R	S	U	V	W
	▶	R	W	)	R	W	R	)	)		W	R	R				▶	▶	R	W	R
		▶	R	W	)	R	)	W	W		R	▶	▶						▶	R	▶
			▶	R	W	)	W	R	R		▶									▶	
				▶	R	W	R	▶	▶												
					▶	R	▶														
						▶															

В качестве пояснения отметим только, что согласно этой истории работы остановы по ошибкам и выходы из режима переполоха произошли по тем же самым символам, что и для автомата с несколькими состояниями. Вид диагностических сообщений также будет похожим.

В этом нет ничего удивительного, поскольку использован один и тот же метод восстановления дерева – нисходящий, и автоматы построены по одной и той же грамматике.

Восходящий автомат для этой входной цепочки реализует следующую историю работы:

Такт	0	1	2	3	4	5	6	7	8	9	10	
Вход	*(*	*x*	+	+	+	+	+	+	+	*)*	*)*	*
Сост.	0	4	5	4	3	4	2	4	9	7	9	12
Опер.	**S**4	**S**5	**R**1,2	**G**3	**R**1,1	**G**2	**R**1,0	**G**9	**S**7		**S**12	**R**3,2
Стек	0	4	5	4	3	4	2	4	9	7	9	12
		0	4	0	4	0	4	0	4	9	4	9
			0		0		0		0	4	0	4
										0		0

По первой ошибке автомат останавливается на такте 9 в состоянии 7. Номер этого состояния удаляется из стека, для состояния 9 определяются ожидаемые символы (это + и *)*) и включается режим переполоха. Читается следующий символ и обнаруживается, что этот символ входит в множество ожидаемых. Поэтому на такте 10 автомат возвращается в нормальный режим, выполняет на такте 11 свертку по правилу *V* : *(S)* и продолжает работать вплоть до обнаружения следующей ошибки на такте 16:

12	13	14	15	16	17	18	19	20	21	22	23	24
*	*	*	*	*)*	*(*	*(*	+	+	+	*y*	▶	▶
0	3	0	2	8			2	0	1	7	5	7
G3	**R**1,1	**G**2	**S**8				**R**1,0	**G**1	**S**7	**S**5	**R**1,2	
0	3	0	2	8			2	0	1	7	5	7
	0		0	2			0		0	1	7	1
				0						0	1	0
											0	

После этого с верхушки стека сбрасывается номер состояния 8, при этом верхним оказывается состояние 2. Ожидаемыми символами для

этого состояния являются: *, +, *)* и ▶. Дождавшись символа +, автомат выходит из режима переполоха, выполняет такты 19…23 и на такте 21, обнаружив очередную ошибку по концу входной цепочки, останавливается окончательно.

Как видим, восходящий автомат останавливается и выходит из режима переполоха в тех же самых точках входной цепочки, что и нисходящие. Суть диагностических сообщений будет аналогичной.

Даже на этом единственном примере видно, что при обработке сильно искаженных входных цепочек возможны ситуации преждевременного выхода автоматов из режима переполоха, что влечет за собой формирование сообщений об ошибках, очень странно выглядящих с точки зрения пользователя транслятора. Для преодоления этого недостатка может быть использована следующая идея: ограничить набор терминалов, которые могут быть ожидаемыми при входе в режим переполоха. Так, например, для языков класса C/C++ при обнаружении ошибок в арифметическом выражении, находящемся в правой части оператора присваивания, имеет смысл запретить выход из режима переполоха вплоть до терминала ; , ограничивающего весь оператор присваивания.

Если разработчик транслятора определяет ограниченный набор символов, по которым осуществляется выход из режима переполоха (обозначим этот набор через $M_{огр}$), то подготовка к его включению будет заключаться в просмотре стека, начиная с верхушки, и проверке, содержит ли очередное множество ожидаемых символов хотя бы один, принадлежащий $M_{огр}$. Только в случае положительного результата такой проверки подготовка заканчивается, иначе верхний элемент удаляется из стека, и проверка продолжается.

Применение этой модификации метода переполоха позволяет уменьшить количество бессмысленных сообщений об ошибках, но может в некоторых случаях приводить к тому, что значительные по объему последовательности слов входного текста будут просматриваться транслятором в режиме переполоха без обработки.

Возможны и другие модификации программной модели конечных автоматов без памяти, направленные на нейтрализацию ошибок.

Рассмотрим теперь методы нейтрализации ошибок на уровне грамматик.

4.1.3. Включение действий в грамматику

Основная идея этого подхода состоит в том, чтобы задачу нейтрализации часто встречающихся ошибок заменить акцептом неправильных входных цепочек в соответствии с добавляемыми в грамматику правилами для типичных ошибок. В момент обнаружения необходимости применения такого правила точно так же, как и при обнаружении ошибки, должны подавляться процессы преобразования входного текста в объектный код. Поэтому правила для типичных ошибок должны содержать явное указание преобразователю грамматики в автомат на то, что применение данного правила при восстановлении дерева разбора эквивалентно обнаружению ошибки.

Для реализации этой идеи должно быть увеличено количество алфавитов порождающей грамматики $\boldsymbol{G}$ путем определения и использования так называемого алфавита транслирующих символов $\boldsymbol{A}_s$:

$$\boldsymbol{G} = \{\boldsymbol{A}_t, \boldsymbol{A}_n, \boldsymbol{A}_s, \boldsymbol{S}, \boldsymbol{P}\}\ .$$

Символы алфавита $\boldsymbol{A}_s$ могут использоваться не только для нейтрализации ошибок и, как мы увидим далее, являться не только символами, а процедурами или функциями, встраиваемыми преобразователем грамматики в автомат акцептора и превращающими его в синтаксический анализатор.

Однако на данный момент мы ограничимся добавлением алфавита, содержащего единственный символ ***error***, который не может использоваться в левой части порождающих правил. Приведем пример использования этого символа для модификации грамматики, порождающей язык арифметических выражений. Допустим, разработчик грамматики выяснил, что часто встречающимися ошибками при записи выражений являются потери скобок (как открывающей, так и закрывающей). Тогда дополнительно к правилу V : *(S)* в грамматике $\boldsymbol{G}_{a1}$ могут быть записаны еще два правила:

$$V : (\ S\ \textbf{error}$$

$$V : S\ )\ \textbf{error}$$

Следует заметить, что включение таких правил может изменить свойства грамматики, т. е. создать проблемы для преобразования грамматики в автомат. Мы не будем особо заострять на этом внимание, поскольку идея этого метода является промежуточной для развития идеи грамматики действий.

Применение любого из этих правил при восстановлении дерева разбора должно приводить к формированию диагностического сообщения об ошибке и к подавлению процессов преобразования входного текста без запуска процедур нейтрализации ошибок, рассмотренных в разделе 4.1. Эти процедуры теперь будут запускаться только по таким ошибкам, для которых нет правил с символом ***error***.

Специальный символ ***error*** можно трактовать как вызов функции *error()*, написанной разработчиком преобразователя грамматики в автомат и встроенный в его управляющую таблицу. Соответственно использование правил для типичных ошибок в грамматике влечет за собой модификацию (расширение) состава полей управляющей таблицы автомата или связанных с ней структур данных. Как минимум, должен добавляться флажок (булевское значение). Более общее решение состоит в добавлении поля указателя на функцию, вызываемую программной моделью автомата при выполнении действий, связанных с данным состоянием. Специальное значение (например, *null*) обычно указывает, что с данным состоянием автомата не связана необходимость вызова какой-либо функции.

Преобразователи грамматик в автоматы обычно предоставляют возможность записывать тексты функций, выполняющих специальные действия, прямо в правых частях правил грамматики. Для того чтобы отличать обычные терминальные и нетерминальные символы грамматики от действий (т. е. от символов транслирующего алфавита $\boldsymbol{A}_s$), действия должны записываться в соответствии с определенным синтаксисом (а также лексикой и семантикой). Способы включения действий в грамматику содержатся в документации на каждый конкретный преобразователь. Приведем условный пример, ориентированный на запись действий в соответствии с синтаксисом преобразователя YACC, формирующего программную модель **LALR**(1)-автомата на языке C/C++:

V : (S {error_message("Ожидается закрывающая скобка.");stop_convert();}
V : S) {error_message("Отсутствует открывающая скобка.");stop_convert();}

Действия (символы алфавита $\boldsymbol{A}_s$) заключаются в фигурные скобки. Такие скобки не могут использоваться в именах терминалов (токенов в YACC) и нетерминалов, что и позволяет однозначно различать симво-

лы различных алфавитов. Каждое действие в соответствии с синтаксисом языка C/C++ – это блок, превращаемый программой YACC в тело функции, имя которой формируется преобразователем и используется для связывания с состоянием, при достижении которого функция должна быть вызвана.

В рассматриваемом примере каждое действие – это вызов последовательно двух функций, возможно, определенных где-то вне грамматики. Функция *error_message()* получает в качестве параметра содержательную строку для формирования диагностического сообщения об ошибке. Функция *stop_convert()* блокирует процессы преобразования входного текста, в котором обнаружена ошибка, в объектный код.

Преобразователем обычно обеспечивается возможность доступа из вызываемых функций к другим процедурам/функциям транслятора (например, к функциям работы с информационными таблицами, в том числе с таблицами идентификаторов), к некоторым общим структурам данных (например, к значению обрабатываемого в данный момент терминального символа, т. е. к структуре, которую мы ранее называли *CurrentLexem*) и возможность определения собственных структур данных для взаимодействия между разными действиями.

Теперь в рассматриваемом нами примере обработки типовых ошибок можно отказаться от включения дополнительных правил, но записать правило *V* : *(S)* в виде

```
V : ( S {
   if ( CurrentLexem != ")" ) {
      error_message("Ожидается закрывающая скобка.");
      stop_convert();
      insert_terminal(")");
   }
)
```

Предполагается, что действие, определенное в данном правиле, вызывается в тот момент, когда завершена обработка части входной цепочки, выводимой из нетерминала *S*. Если очередным входным символом не является закрывающая скобка, то кроме формирования сообщения об ошибке и блокировки процессов преобразования вызывается *insert_terminal(")")* – функция для нейтрализации ошибки путем вставки ожидаемого символа перед ошибочным. Если же очередной входной символ есть закрывающая скобка, то кроме его проверки ничего делаться не будет. Данный пример приведен только для иллюст-

рации развиваемых идей и не может быть рекомендован для использования.

Таким образом, идея включения дополнительного алфавита символов в грамматику для обработки типичных ошибок оказалась трансформированной в идею предоставления возможности разработчику грамматики определять необходимые действия по дополнительной обработке входного текста в процессе восстановления дерева грамматического разбора.

Далее показано, что путем включения действий в грамматику можно обеспечить решение всех задач не только синтаксического, но и семантического анализа и даже определенной части задач генерации объектного кода.

4.2. Преобразование анализируемого предложения в постфиксную форму записи

Конечной целью работы транслятора является эквивалентное преобразование входного текста в объектный модуль (здесь под транслятором понимается компилятор, но отличия его от интерпретатора несущественны с точки зрения задач, стоящих перед синтаксическим анализом). Как входной текст, так и объектный модуль являются программами, но записанными на различных языках. Требование эквивалентности преобразования означает, что результаты выполнения исходной и преобразованной (объектной) программ при одинаковых обрабатываемых данных должны быть идентичными. Процессы (истории) выполнения программ, записанных на различных языках, можно рассматривать как последовательности выполнения различных по сложности операций. При формальных преобразованиях, выполняемых трансляторами, добиться эквивалентности можно только в том случае, если гарантируется, что в любой паре историй работы выполнению каждой операции исходной программы соответствует выполнение эквивалентной ей последовательности из одной или нескольких операций объектной программы.

Таким образом, транслятор рано или поздно обязан выяснить, какие операции и в какой последовательности должны выполняться согласно алгоритму обработки данных, определенному текстом исходной программы. Здесь очень важным моментом является соотношение между синтаксисом (определяющим способ записи последовательности операций) и семантикой (определяющей способ выполнения этой последовательности) для языков разного уровня. Любой язык програм-

мирования высокого уровня ориентирован на предоставление максимальных удобств разработчику программ. Поэтому, как правило, последовательность знаков операций в тексте исходной программы не совпадает с последовательностью их выполнения в истории работы программы. Приведем простейший пример. Пусть в тексте программы на языке С/С++ записан оператор присваивания $a=b*c+d$;

Последовательность появления знаков операций в тексте такова: = * + . Однако выполнение этого оператора в целом, определяемое семантикой языка, эквивалентно последовательности выполнения трех элементарных операций, эквивалентных машинным командам.

1) умножить значение b на значение c;

2) сложить полученное значение со значением d;

3) присвоить последнее полученное значение переменной a.

Следовательно, в объектной программе знаки операций должны быть записаны в последовательности * + = , поскольку семантика машинно-ориентированных языков предусматривает последовательную выборку выполняемых команд из линейно организованной памяти. Легко можно привести множество других примеров, из которых следует, что привычная для человека форма записи выражений и операторов присваивания в целом существенно отличается от того вида, в котором они должны быть представлены в объектном модуле.

Несколько другие отличия форм представления исходной и объектной программ характерны для управляющих конструкций языков программирования, таких как условные операторы, переключатели и операторы цикла. Синтаксис таких конструкций не предусматривает явной записи операций передач управления, подразумеваемых семантикой языка, и ориентирован на удобство использования человеком. Однако в процессе эквивалентного преобразования исходной программы эти операции, очевидно, должны появиться в тексте объектной программы. Например, пусть в тексте программы на языке С/С++ записан условный оператор:

```
if ( c>0 )
        a = b * c + d;
else
        a = ( d – b ) * c;
```

Смысл этой записи совершенно ясен человеку и сводится к тому, что должна быть выполнена следующая последовательность действий.

1. Вычислить результат сравнения значения *c* с нулем и получить булевское значение *true* или *false*.
2. Если результат сравнения есть *false*, то перейти к шагу 7 (к оператору присваивания $a = d - b * c;$).
3. Перемножить значения *b* и *c*.
4. Сложить полученное значение с *d*.
5. Присвоить полученное значение переменной *a*.
6. Перейти к шагу 10.
7. Вычесть значение *b* из значения *d*.
8. Умножить результат вычитания на значение *c*.
9. Присвоить полученное значение переменной *a*.
10. … (Следующий по тексту оператор программы.)

Именно в этой последовательности должны быть записаны операции в тексте объектной программы, для того чтобы процессор компьютера мог выбирать их из оперативной памяти и выполнять. В этом представлении появились операции переходов или передач управления (шаги 2 и 6), отсутствующие в явном виде в исходном тексте, но подразумеваемые семантикой входного языка.

Постфиксная запись, эквивалентная исходному оператору и содержащая близкие к машинно-ориентированному языку элементы, может выглядеть так:

$$c\ 0\ \underline{>}\ lblF\ \underline{JmpF}\ a\ b\ c\ \underline{*}\ d\ \underline{+}\ \underline{=}\ lblEnd\ \underline{Jmp}\ lblF{:}\ a\ d\ b\ \underline{-}\ c\ \underline{*}\ \underline{=}\ lblEnd{:}$$

В этой записи подчеркнуты знаки операций, в том числе операции условной (JmpF) и безусловной (Jmp) передачи управления. Операнды каждой операции записаны перед знаком этой операции. В этом примере все знаки операций, кроме безусловной передачи управления, являются бинарными (используют 2 операнда). Операция Jmp имеет единственный операнд – метку *lblEnd*. Метки, именующие некоторые операторы программы и используемые в операциях перехода, введены при преобразовании.

Особо отметим, что в постфиксной записи второго оператора присваивания отсутствуют скобки, изменяющие порядок выполнения операций в исходном операторе программы. Постфиксная форма записи уникальна тем, что:

– последовательность появления в ней знаков операций совпадает с требуемым порядком их выполнения;

– не нужны скобки для изменения порядка выполнения операций.

Казалось бы, затронутые вопросы относятся к этапам семантического анализа и генерации объектного кода. Однако задача выявления последовательности операций, эквивалентной исходной программе, хотя и определяется семантикой двух языков, но имеет глубокие внутренние связи с задачей восстановления дерева грамматического разбора и решается, как правило, на этапе синтаксического анализа.

4.2.1. Синтаксические деревья и постфиксная форма записи

Синтаксическим деревом, или деревом операций, называется такое графическое представление совокупности операций, связанных значениями обрабатываемых данных (операндами), в котором:

– узлы (вершины дерева, из которых выходят дуги, ведущие к потомкам) помечены знаками операций;

– листья (концевые вершины дерева, не имеющие потомков) помечены наименованиями операндов;

– нет вершин, помеченных какими-либо другими символами.

Синтаксическое дерево оператора присваивания $a=b*c+d$; может выглядеть так, как представлено на рис. 11, *а*. Для сравнения на рис. 11, *б* показано дерево грамматического разбора этого оператора в грамматике $\boldsymbol{G}_{a1}$, расширенной путем добавления правила $P : i = S$; для нового начального нетерминала P.

Синтаксическое дерево (рис. 11, *а*) в наглядной форме показывает зависимость операций друг от друга и может быть использовано для определения последовательности их выполнения. Ясно, что до тех пор, пока не вычислено произведение значений b и c, не может быть выполнена операция сложения.

В свою очередь, операция присваивания зависит от результата выполнения операции сложения и может быть выполнена только после нее. Далее определим процедуру обхода любого синтаксического дерева для построения требуемой последовательности операций в линейном представлении.

Дерево грамматического разбора, восстанавливаемое при проверке правильности данного оператора присваивания и показанное справа, также содержит всю необходимую информацию для решения этой задачи. Однако это дерево содержит «лишние» с точки зрения выявления последовательности операций элементы: вершины, помеченные нетерминалами и выходящими их них дугами, а также вершину, поме-

ченную ограничителем оператора присваивания (*;*) вместе со всеми дугами, ведущими к таким вершинам. В данном операторе не использовались скобки *()*, но если бы они были, то также считались бы «лишними».

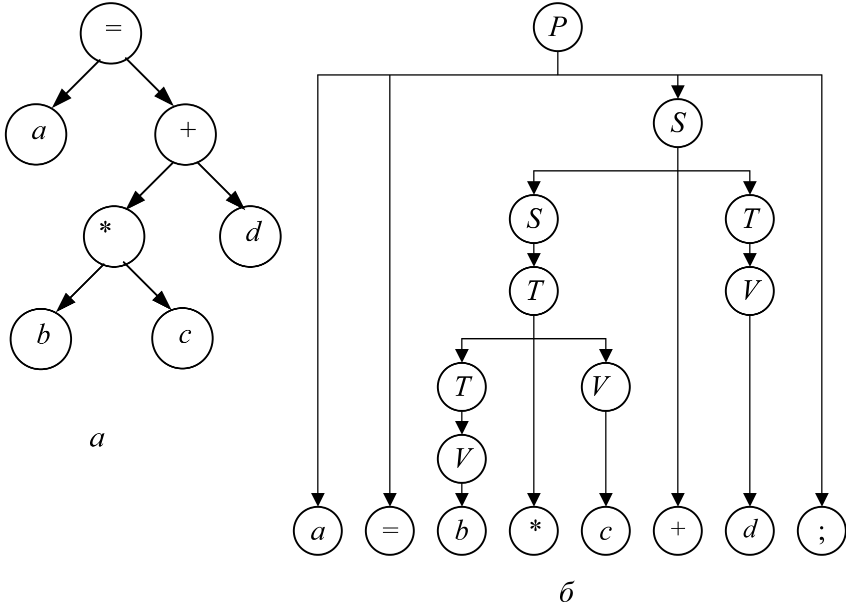


Рис. 11. Связь дерева операций и дерева разбора:

а – дерево операций; *б* – дерево грамматического разбора

Дерево грамматического разбора может быть преобразовано в дерево операций путем применения следующей процедуры.

1. Удалить все листья (вершины, помеченные терминальными символами), пометки которых не являются знаками операций и наименованиями операндов.

2. Просмотреть узлы дерева (вершины, имеющие исходящие дуги), начиная с корня. Для каждого просматриваемого узла сохранять в стеке перечень дочерних узлов. Если какая-либо из дочерних вершин помечена знаком операции, то просматриваемый узел пометить этим знаком и удалить из дерева дочернюю вершину, но только в том случае, если она является листом. Если после обработки очередного узла стек

не пуст, то перейти к обработке узла, номер которого снимается с верхушки стека. Если же стек опустел, то повторять шаг 2 до тех пор, пока состояние дерева не перестанет изменяться.

3. Для каждого листа, помеченного операндом, проверить пометку родительского узла. В том случае если родительский узел помечен нетерминалом, перенести в него наименование операнда из дочернего листа и удалить этот лист. Продолжать выполнение шага 3 до тех пор, пока состояние дерева не перестанет изменяться.

Если применить эту процедуру к приведенному выше дереву разбора, то будет получено именно такое дерево операций, которое приведено на рис. 11, *а*.

Для данной грамматики применение этой процедуры позволит получить желаемый результат из дерева разбора любого оператора присваивания. Объясняется это тем, что вся семантика, определяющая последовательность выполнения операций сложения, вычитания и присваивания, а также изменение порядка их выполнения при использовании скобок в выражении некоторым неявным образом заложена в совокупность порождающих правил, описывающих синтаксис языка операторов присваивания.

Из дерева операций легко можно получить постфиксную форму записи линейной последовательности знаков операций (с их операндами) такую, в которой порядок появления операций совпадает с требуемым порядком их выполнения.

Процедура преобразования дерева операций в постфиксную форму записи является рекурсивной и может быть определена следующим образом.

1. Взять корень дерева операций в качестве текущей вершины.

2. Если текущая вершина не является листом, перейти к шагу **3**, иначе выдать ее пометку (наименование операнда) на выход и завершить обход поддерева.

3. Обойти левое поддерево данного корня (рекурсивно вызвать шаг **2** процедуры для корня левого поддерева текущей вершины).

4. Обойти правое поддерево данного корня (рекурсивно вызвать шаг **2** процедуры для корня правого поддерева текущей вершины).

5. Выдать пометку текущей вершины (знак операции) на выход. Завершить обход поддерева.

Применение процедуры к дереву операций, построенному для оператора присваивания *a=b*c+d;* , позволит получить такую постфиксную запись:

$$a\ b\ c * d + =$$

Ее смысл (семантика) состоит в следующем:

Сначала должна быть выполнена операция умножения значений *b* и *c*, наименования которых записаны **перед** знаком *. Можно считать, что в результате выполнения операции умножения получено промежуточное значение, которое обозначим через *r*:

$$a\ r\ d + =$$

Затем аналогичным образом должна быть выполнена операция сложения значений *r* и *d* (результат сложения обозначим также: *r*), после чего запись оператора будет выглядеть так:

$$a\ r =$$

Окончательно после выполнения операции присваивания запись оператора получит вид *r* , где под *r* можно понимать (как это делается в языке C/C++) результат выполнения всего оператора присваивания.

Главной особенностью постфиксной записи по сравнению с привычной для человека смесью инфиксной (знак операции находится между наименованиями операндов, например: $a+b$) и префиксной (знак операции находится перед наименованиями своих операндов, например: *sin(x)*) форм записи является то, что порядок следования знаков операций в тексте строго совпадает с требуемым порядком их выполнения при полном отсутствии необходимости в скобках, меняющих этот порядок.

Таким образом, если построено дерево разбора правильного предложения, то известен и метод решения основной задачи синтаксического анализа: преобразование предложения в такую промежуточную форму записи, в которой положение знаков операций совпадает с требуемым порядком их выполнения.

Однако следует заметить, что:

1) в явном виде дерево разбора не строится никаким синтаксическим акцептором (во всяком случае, теми, которые были рассмотрены в разделе 2 и 3). Следовательно, прямое применение описанных процедур невозможно или требуется их модификация;

2) в простой грамматике $\boldsymbol{G}_{a1}$ нет правил, содержащих в правой части более одного знака операции (на чем и основана данная процедура). При наличии таких правил придется существенно усложнять шаг 2 процедуры преобразования дерева разбора в дерево операций;

3) для реальных языков программирования не всегда возможно построить такую грамматику, чтобы все семантические правила, определяющие требуемую последовательность (следующую из приоритетов) выполнения операций, были использованы в синтаксических порождающих правилах. В подобных случаях дерево разбора невозможно преобразовать в дерево операций с помощью описанной процедуры.

4.2.2. Включение действий в грамматику для преобразования предложения в постфиксную форму записи

Сформулируем индуктивные определения постфиксной формы записи для арифметических выражений и операторов присваивания:

1. Постфиксной записью пустой цепочки символов ε является ε.

2. Постфиксной записью идентификатора i является идентификатор i.

3. Постфиксной записью константы c является константа c.

4. Если E – произвольное выражение, то постфиксной записью выражения, взятого в скобки, т. е. *(E)* будет просто ПФЗ(E).

5. Если E – выражение вида 2, 3 или 4, то постфиксной записью выражения–E (изменение знака значения E) будет ПФЗ(E) – .

6. Если E – выражение вида 2, то постфиксной записью выражений ++E (– –E) является ПФЗ(++E) = ПФЗ(*incPre*(*&E*)) = *E & incPre* (ПФЗ(– –E) = ПФЗ(*decPre*(*&E*)) = *E & decPre*), где & – знак операции взятия адреса, а *incPre* (*decPre*) – имя функции (знак операции), увеличивающей (уменьшающей) значение своего аргумента на единицу и возвращающей измененное значение.

7. Если E – выражение вида 2, то постфиксной записью выражений E++ (E – –) является ПФЗ(E++) = ПФЗ(*incPost*(*&E*)) = *E & incPost* (ПФЗ(E– –) = ПФЗ(*decPost*(*&E*)) = *E & decPost*), где & – знак операции взятия адреса, а *incPost* (*decPost*) – имя функции (знак операции), увеличивающей (уменьшающей) значение своего аргумента на единицу и возвращающей еще не измененное значение.

8. Если E_1, E_2, … E_k – выражения вида 2…9, то постфиксной записью выражения ○ (E_1, E_2, … E_k) является ПФЗ(E_1) ПФЗ(E_2) ... ПФЗ(E_k) ○, где ○ – знак любой k-арной операции (функции с k аргументами), а ПФЗ(E_1), ПФЗ(E_2) … ПФЗ(E_k) – постфиксные записи выражений E_1, E_2 … E_k соответственно.

9. Если E_1 и E_2 – выражения вида 2…7, то постфиксной записью выражения E_1 ○ E_2 является ПФЗ(E_1) ПФЗ(E_2) ○, где ○ – знак любой

бинарной операции (присваивания, сложения, вычитания, умножения, …), а ПФЗ(E_1) и ПФЗ(E_2) – постфиксные записи выражений E_1 и E_2 соответственно.

10. Если E_1 , E_2 , E_3 – выражения вида 2…8, то постфиксной записью выражения $E_1 \circ E_2 \bullet E_3$ (где ○ и ● – знаки бинарных операций) является ПФЗ(E_1) ПФЗ(E_2) ○ ПФЗ(E_3) ●, если приоритет знака операции ○ не меньше приоритета знака операции ●, и ПФЗ(E_1) ПФЗ(E_2) ПФЗ(E_3) ● ○ – в противном случае.

11. Если E_1 – выражение вида 2, а E_2 – выражение вида 2…9, то постфиксной записью выражения $E_1 \circ E_2$; является ПФЗ(E_1) ПФЗ(E_2) ○, где ○ – знак операции присваивания (напомним, что в языке C/C++, например, существует не один, а несколько знаков операции присваивания: =, +=, *=, …).

Подобные определения можно сформулировать и для других синтаксических конструкций языка программирования, одно из которых далее рассмотрим.

Руководствуясь индуктивными определениями постфиксной формы записи, можно достаточно просто осуществить преобразование заданной грамматики в грамматику действий (см. разд. 4.1.3) таким образом, чтобы построенный на ее основе автомат обеспечивал построение постфиксной формы записи входной цепочки терминальных символов в процессе проверки ее правильности. Покажем, как это делается, на примере грамматики операторов присваивания (расширенной грамматики $\boldsymbol{G}_{a1}$).

Будем считать, что имеется функция, выполняющая занесение одного терминального символа (лексемы) в формируемое в процессе синтаксического анализа промежуточное представление программы (постфиксную запись), прототип которой выглядит так:

void PutToPFR(Lexem);

Грамматика операторов присваивания, расширенная действиями по формированию постфиксной записи, может выглядеть так:

0. *Z* : *P* ▶
1. *P* : *{ PutToPFR(CurrentLexem)*; *} i* = *S { PutToPFR(“=”)*; *}* ;
2. *S* : *S* + *T { PutToPFR(“+”)*; *}*
3. *S* : *T*
4. *T* : *T* * *V { PutToPFR(“*“)*; *}*
5. *T* : *V*
6. *V* : *(S)*
7. *V* : *{ PutToPFR(CurrentLexem)*; *} i*
8. *V* : *{ PutToPFR(CurrentLexem)*; *} c*

В правило 1 добавлены два действия.

Первое (*{PutToPFR(CurrentLexem);}*) обеспечивает преобразование идентификатора, находящегося в левой части оператора присваивания, в постфиксную форму (в соответствии с определением ПФЗ для идентификатора). Действие вставлено в правило до терминального символа *i* по той простой причине, что при функционировании любого (нисходящего или восходящего) автомата оно должно быть выполнено в тот момент, когда текущим входным символом (значением переменной *CurrentLexem*) является идентификатор из левой части оператора присваивания. При переходе нисходящего акцептора в состояние, соответствующее знаку оператора присваивания, а восходящего – в состояние, соответствующее точке между терминалами *i* и =, в этом правиле текущим входным символом уже будет слово =. Поэтому для помещения лексемы, прочитанной из входной цепочки, в постфиксную запись вызов функции *PutToPFR* должен помещаться в правило непосредственно перед терминалом, обозначающим группу слов типа идентификаторов (или констант). Аналогичные действия в тех же точках можно увидеть в правилах 7 и 8. Здесь приведено обоснование точек вставки подобных действий как для нисходящего, так и для восходящего методов синтаксического акцепта, несмотря на то что рассматриваемая грамматика годится только для восходящего.

Второе действие в правиле 1 (*{ PutToPFR("=");* *}*) находится между нетерминалом *S* и ограничителем оператора присваивания *;*. Для простоты будем предполагать, что функция *PutToPFR* способна преобразовать строку в лексему. Точка вставки этого действия строго соответствует определению постфиксной записи выражений, образованных с помощью бинарного знака операции присваивания. При этом ПФЗ выражения E_1 формируется первым действием в данном правиле, а ПФЗ выражения E_2 будет сформировано при разборе той части входной цепочки, которая выводится из нетерминала *S*. В этом процессе будут использоваться правила для этого и других нетерминалов и выполняться вставленные в них действия. В тот момент, когда вся цепочка символов, выводимая из *S* и представляющая собой правую часть оператора присваивания, будет обработана, и текущим входным символом станет ограничитель *;* , автомат выполнит второе действие и завершит тем самым формирование постфиксной записи оператора присваивания в целом.

Действия, вставленные в правые части правила 2 (*{ PutToPFR("+");* *}*) и правила 4 (*{ PutToPFR("+");* *}*), точно так же обусловлены определе-

нием постфиксной записи для выражений, образуемых с использованием бинарных знаков операций. Точки вставки этих действий точно так же обусловлены этим определением и для данной грамматики не могут быть изменены. Действия, вставленные в правила 7 и 8, уже описаны.

При построении конечного автомата на основе грамматики действий должно быть обеспечено выполнение этих действий в требуемые моменты времени. Любой преобразователь грамматик в синтаксические анализаторы это делает либо путем добавления специальных полей в управляющую таблицу автомата, либо путем формирования дополнительных структур с указателями на функции, в которые преобразуются действия.

Легко видеть, что расширенный таким образом **LALR(1)**-автомат, который можно построить по данной грамматике действий, обеспечит преобразование любого правильного оператора присваивания в постфиксную форму записи без построения в явном виде дерева грамматического разбора и преобразования его в дерево операций с последующим обходом последнего.

Для многих синтаксических конструкций языков программирования преобразование в постфиксную запись требует значительно больших усилий. Это объясняется необходимостью формирования уникальных меток и операций передач управления при выявлении линейной последовательности операций для операторов цикла, условных операторов и переключателей.

4.2.3. Преобразование управляющих конструкций языка программирования в постфиксную форму записи

Такое преобразование, как правило, связано с необходимостью решения ряда дополнительных задач. Рассмотрим источники их возникновения и один из возможных методов решения на простом примере оператора цикла *while* языка программирования C/C++.

Допустим, что синтаксис оператора *while* определен в грамматике следующим образом:

Operator : *while (Expression) Block*

Предполагается, что определены и другие операторы (присваивания, условный, … в том числе – оператор *break*, семантика которого состоит в выходе из тела цикла), что *Expression* определено как выражение, значение которого можно преобразовать в логическое значение,

и что *Block* определен как одиночный оператор либо как последовательность операторов, заключенная в фигурные скобки.

Запишем желаемый вид постфиксной записи для оператора цикла *while*, используя введенные выше определения ПФЗ для *Expression* и предполагая, что ПФЗ для *Block* также определено (на самом деле определение ПФЗ для блока операторов индуктивно зависит от определения ПФЗ оператора *while*, поскольку этот блок может содержать один или несколько операторов *while*):

ПФЗ(*while (Expression) Block*) =
Label1: ПФЗ(*Expression*) ***Label2 JmpF*** ПФЗ(*Block*) ***Label1 Jmp Label2***:

В этом определении жирным курсивом выделены:

Label1: – определение наименования первого элемента постфиксной записи вычисления значения выражения.

JmpF – бинарный знак операции условного перехода, использующий в качестве первого операнда для определения необходимости передачи управления значение, вычисляемое ПФЗ(*Expression*), в качестве второго – наименование (***Label2***) первого элемента постфиксной записи программы в целом, следующего непосредственно после ПФЗ данного оператора цикла.

Jmp – унарный знак операции безусловного перехода, использующий в качестве операнда наименование ***Label1***.

Label2: – определение наименования для оператора выхода из цикла (условный переход ***JmpF***).

Согласно этому определению выполнение оператора *while* должно протекать так: вычисляется значение выражения, если оно имеет значение *true*, то оператор ***JmpF*** не передает управление на оператор, помеченный именем ***Label2***:, выполняются действия постфиксной записи блока, и оператором ***Jmp*** управление возвращается на начало вычисления выражения (операцию, помеченную ***Label1:***); если же значение выражения есть *false*, то оператор ***JmpF*** передает управление, используя ***Label2*** в качестве адреса.

Если бы требовалось преобразовать в постфиксную запись единственный оператор цикла, то такого определения его ПФЗ было бы достаточно. Однако в тексте программы может встретиться несколько операторов *while*, в том числе и внутри блока, являющегося телом данного цикла.

Для того чтобы при преобразовании в постфиксную запись не формировались идентичные наименования разных точек переходов, что создаст проблемы при генерации объектного кода, необходимо в приведенном определении ПФЗ все метки понимать как уникальные, однозначно сопоставленные с конкретным оператором цикла. Уникальность меток можно обеспечить при реализации преобразования в постфиксную запись, т. е. при вставке действий в грамматику.

Для обеспечения уникальности меток, создаваемых для машинно-ориентированного эквивалента оператора while можно:

– используя специально определенную для этой цели переменную (счетчик), присваивать уникальный номер каждому оператору цикла, встретившемуся во входной программе при восстановлении дерева ее разбора;

– при увеличении значения счетчика сохранять его во вспомогательном стеке, доступном из действий, вставляемых в грамматику;

– использовать значение, находящееся на верхушке вспомогательного стека, для формирования наименований адресов перехода;

– удалять верхнее значение из стека в момент завершения обработки оператора цикла.

Приведем пример реализации, предполагая, что счетчик имеет целое значение и называется *WhileCount*, что для операций над вспомогательным стеком имеются функции с прототипами:

void Push(*int*); – поместить значение аргумента на верхушку стека.
int Pop(*void*); – возвратить значение, удалив его с верхушки стека.
int Top(*void*); – вернуть значение с верхушки, не удаляя его из стека.

Для краткости записи будем считать, что при написании действий можно использовать бинарную инфиксную операцию #, преобразующую свои операнды в строковые представления и возвращающую конкатенацию этих строк. Функция *PutToPFR* будет преобразовывать такие строковые аргументы в лексемы. Теперь рассматриваемое правило грамматики, расширенное действиями на основе определения ПФЗ оператора цикла *while*, будет выглядеть так:

```
Operator : while
        {Push(++WhileCount); PutToPFR("Label1" # Top() # ":");}
        ( Expression )
        { PutToPFR("Label2" # Top()); PutToPFR("JmpF");}
        Block
        { PutToPFR("Label1" # Top()); PutToPFR("Jmp");
        PutToPFR("Label2" # Pop() # ":"); }
```

Кроме операторов цикла *while* язык программирования, как правило, содержит другие формы операторов цикла (*for*, *do*, …), условные операторы, переключатели и т. д. Все метки, генерируемые при формировании постфиксной формы записи этих конструкций, должны быть уникальны в пределах одной транслируемой программной единицы. Поэтому разработчику транслятора придется решать следующие вопросы:

– требуется ли поддерживать отдельные счетчики для разных типов управляющих конструкций или для всех таких операторов использовать один счетчик?

– поддерживать ли несколько вспомогательных стеков или можно обойтись одним?

ОГЛАВЛЕНИЕ

1. ВВЕДЕНИЕ В СИНТАКСИЧЕСКИЙ АНАЛИЗ 3

1.1. Формальные грамматики, основные понятия 4

1.2. Связь между языками и грамматиками 6

1.3. Классификация формальных грамматик 10

1.4. Свойства контекстно-свободных грамматик 12

1.5. Свойства символов грамматики 16

1.6. Множества предшественников символов грамматики 19

1.7. Множества предшественников цепочек символов 22

1.8. Множества последователей символов грамматики 22

1.9. Эквивалентные преобразования грамматик 28

1.10. Постановка задачи синтаксического акцепта на основе формальной грамматики языка 35

2. НИСХОДЯЩИЕ МЕТОДЫ СИНТАКСИЧЕСКОГО АКЦЕПТА 38

2.1. Основная идея нисходящего восстановления дерева грамматического разбора 39

2.2. Множества выбора правил грамматик 44

2.3. LL(1)-грамматики 51

2.4. Процедурная реализация рекурсивного спуска 55

2.5. Автоматная реализация рекурсивного спуска Автомат с несколькими состояниями 61

2.6. Нисходящий автомат с одним состоянием 71

2.7. Оценки сравнительных характеристик различных реализаций нисходящего синтаксического акцепта 75

3. ВОСХОДЯЩИЕ МЕТОДЫ СИНТАКСИЧЕСКОГО АКЦЕПТА 79
3.1. Основная идея восходящего восстановления дерева грамматического разбора 79
3.2. Стековый автомат для реализации основной идеи восходящего анализа 82
3.3. Восходящий синтаксический акцептор с несколькими состояниями 86
3.4. Понятие конфигурации. Связь между конфигурациями, состояниями и операциями восходящего акцептора 91
3.5. Определение набора состояний восходящего акцептора 95
3.6. Преобразование таблицы конфигураций в управляющую таблицу 99
3.7. Предотвращение конфликтов путем использования множеств последователей нетерминальных символов 101
3.8. LR(0)- и SLR(1)-грамматики и автоматы 105
3.9. Ожидаемый правый контекст и LR(1)-автоматы 107
3.10. LALR(1)-грамматики и автоматы 116
3.11. Оценки сравнительных характеристик различных реализаций восходящего синтаксического акцепта 122
4. СОБСТВЕННО СИНТАКСИЧЕСКИЙ АНАЛИЗ 123
4.1. Нейтрализация синтаксических ошибок 124
4.2. Преобразование анализируемого предложения в постфиксную форму записи 144

Малявко Александр Антонович

**СИСТЕМНОЕ
ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ**

**ФОРМАЛЬНЫЕ ЯЗЫКИ
И МЕТОДЫ ТРАНСЛЯЦИИ**

Часть 2

СИНТАКСИЧЕСКИЙ АНАЛИЗ

Учебное пособие

Редактор *Л.Н. Ветчакова*
Выпускающий редактор *И.П. Брованова*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *Л.А. Веселовская*

Подписано в печать 04.05.2011. Формат 60 × 84 1/16. Бумага офсетная. Тираж 100 экз.
Уч.-изд. л. 9,3. Печ. л. 10,0. Изд. № 82. Заказ № Цена договорная

Отпечатано в типографии
Новосибирского государственного технического университета
630092, г. Новосибирск, пр. К. Маркса, 20